A Survey of the Bijective Burrows-Wheeler Transform

Hideo Bannai
M&D Data Science Center, Institute of Integrated Research, Institute of Science Tokyo, Japan

Dominik Köppl
Department of Computer Science and Engineering, University of Yamanashi, Japan

Zsuzsanna Lipták
Department of Computer Science, University of Verona, Italy

Abstract

The Bijective BWT (BBWT), conceived by Scott in 2007, later summarized in a preprint by Gil and Scott in 2009 (arXiv 2012), is a variant of the Burrows-Wheeler Transform which is bijective: every string is the BBWT of some string. Indeed, the BBWT of a string is the extended BWT [Mantaci et al., 2007] of the factors of its Lyndon factorization. The BBWT has been receiving increasing interest in recent years. In this paper, we survey existing research on the BBWT, starting with its history and motivation. We then present algorithmic topics including construction algorithms with various complexities and an index on top of the BBWT for pattern matching. We subsequently address some properties of the BBWT as a compressor, discussing robustness to operations such as reversal, edits, rotation, as well as compression power. We close with listing other bijective variants of the BWT and open problems concerning the BBWT.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Mathematics of computing → Combinatorics on words

Keywords and phrases Burrows-Wheeler Transform, compression, text indexing, repetitiveness measure, Lyndon words, index construction algorithms, bijective string transformation

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.2

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP24K02899
Dominik Köppl: JSPS KAKENHI Grant Numbers JP23H04378 and JP25K21150
Zsuzsanna Lipták: Partially funded by the Italian Ministry of University and Research (MUR) PRIN Project PINC, Pangenome INformatiCs: from Theory to Applications (Grant No. 2022YRB97K), and by the INdAM - GNCS Project CUP_E53C24001950001.

1 Introduction

The famous Burrows-Wheeler transform [17] (BWT) of a string is defined as the string obtained by concatenating the last letter of all rotations of the string, in the lexicographic order of the rotations. The BWT is often described as a reversible transform, meaning it must be injective, but clearly it is not, since all rotations of the string share the same output. Therefore, to make it injective and thus reversible, information to encode which of the strings among all rotations was the original is required; typically in the form of an explicit end-of-string symbol (often denoted by \$), or an integer indicating the position in the BWT that corresponds to the lexicographic rank of the original string among all rotations, and therefore to the last character of the original string. It is also clear by a simple counting argument, that the transform is not surjective, meaning that there exist strings which are not images of BWT. Note that this is true even with the addition of the extra information to make it injective [37]. For example, \$`ab` (assuming \$ < `a` < `b`) is not a BWT image.[1]

[1] As a famous example, `banana` is not a BWT image, nor is \$`banana`, `b`\$`anana`, . . . , `banana`\$; in other words, one cannot insert \$ anywhere for it to become a BWT image. For more on this problem, see [37].

The bijective variant of the BWT, *Bijective BWT* (BBWT, a.k.a. BWTS for *BWT Scottified*) considered in this paper was first announced by David Allen Scott in a Usenet thread in December 2007 [66], where the process of making the BWT bijective was termed the *Scottification* of the BWT. While Scott provided a working implementation together with explanations using concrete examples, a rigorous formal description of the procedure was not given, and the excitement of the discovery was unfortunately met with great skepticism towards its correctness and usefulness. In 2009, Joseph Gil, together with Scott, made available a preprint on Scott's website [68] – later uploaded to arXiv in 2012 [32] – which contained a more accessible description of the process and a proof of correctness. Also in 2009, Kufleitner [47] provided another formal description of the BBWT, along with another BWT variant (which we will not cover in this survey). Scott later mentions [67] that his paper with Gil had been submitted to the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) but was rejected, highlighting the reviewer comment:

> "Of course one can easily achieve invertability [sic!] in Burrows-Wheeler transform, by appending a special character to the input string."

To the best of our knowledge, the paper has not been subsequently published in a journal or proceedings[2].

Three advantages of BBWT over BWT are mentioned by Gil and Scott. First, it allows to get rid of the extra information mentioned above. Gil and Scott report empirical evidence that compression using BBWT is almost always better (albeit only slightly) than compression using BWT, on the Calgary corpus [9]. Second, they claim that in the context of applying encryption after compression, bijectivity is preferable, since *not* being bijective would reveal information to an attacker. Third, it is simply mathematically more elegant.

As can be seen from the reviewer's comment to their paper (which misses the point), the difference between BBWT and BWT may be quite subtle and perhaps difficult to understand. Although the immediately observable merits of BBWT may have been scarce, some recent works have started to realize and tap into the value of this transform.

In this survey, we summarize currently known results on the BBWT, concerning its construction (Section 3), its use as an index (Section 4), and its performance as a compressor (Section 5).

2 Basics

Let Σ denote the *alphabet*, i.e., the set of symbols (or characters), and Σ^* the set of (finite) strings (or texts) over Σ. We index strings from 0. We denote the empty string by ε, the length of a string T by $|T|$, and the substring from i to j by $T[i..j]$, where $T[i..j] = \varepsilon$ if $j < i$. We will also use $T[i..j)$ to denote $T[i..j-1]$. The reverse of string T is denoted T^{rev}. Let $\mathsf{rot}(T) = T[1..|T|)T[0]$. Then, $\mathsf{rot}^i(T) = T[i..|T|)T[0..i)$ is the rotation of T that starts at position i. A *cyclic string* T is a string whose first character $T[0]$ is considered as being subsequent to its last character $T[|T|-1]$. In other words, arithmetic on the positions $0, 1, \ldots, |T|-1$ of T is done modulo $|T|$.

A string T is called *primitive* if it cannot be represented as $T = U^k$ for some string U and integer $k \geq 2$; otherwise T is called a *power*. A string S is a *rotation* (or *conjugate*) of a string T if there exist $U, V \in \Sigma^*$ such that $T = UV$ and $S = VU$. Conjugacy is an

[2] An interesting parallel with the original BWT paper [17].

equivalence relation, and it is easy to see that the conjugacy class $[T] = \{VU : T = UV\}$ has cardinality $|T|$ if and only if T is primitive.

We denote the number of the maximal same symbol runs, or simply *runs*, of a string T by $\rho(T)$. For example, $\rho(\texttt{cabbbbbaa}) = 4$. Note that $\rho(T)$ is the size of the run-length encoding of T.

We assume a total order on Σ and denote by $\prec$ the *lexicographic order* on Σ^* induced by this total order. Furthermore, we define the *omega-order* (also called *infinite periodic order* [32, 64]) on Σ^* as follows: $T \prec_\omega S$ if either $T^\omega \prec S^\omega$, or $T^\omega = S^\omega$ and $|T| < |S|$. Here, X^ω denotes the infinite string obtained by concatenating string X an infinite number of times. Note that the two orders $\prec$ and $\prec_\omega$ coincide if neither of two strings is a proper prefix of the other but may differ otherwise. For example, $\texttt{ab} \prec \texttt{aba}$ but $\texttt{aba} \prec_\omega \texttt{ab}$. The omega-order can also be viewed as an order on cyclic strings.

Since all rotations of a string have the same length, the two orders $\prec$ and $\prec_\omega$ coincide on the conjugacy classes. Given a string T, let $\mathsf{minrot}(T)$ denote the smallest rotation of T in lexicographic, or equivalently in this case, in omega-order, i.e. $\mathsf{minrot}(T) = \min[T]$. If $T = \mathsf{minrot}(T)$ then T is called a *necklace*[3]; if T is in addition primitive, then it is called a *Lyndon word*. Accordingly, $\mathsf{minrot}(T)$ is also referred to as *necklace rotation* resp. *Lyndon rotation* of T. For instance, $\mathsf{minrot}(\texttt{baba}) = \texttt{abab}$ is a necklace and $\mathsf{minrot}(\texttt{baaba}) = \texttt{aabab}$ is a Lyndon word.

The *Lyndon factorization* [20] of a string T is a factorization $T = L_1^{e_1} \cdots L_{\ell(T)}^{e_{\ell(T)}}$ where for all $1 \le i \le \ell(T)$, $e_i \ge 1$ and L_i is a Lyndon word, and $L_i \succ_\omega L_{i+1}$. Note that our definition differs slightly from the usual one in that also here we are using the omega-order rather than the lexicographic order. Since on Lyndon words, the two orders coincide [27, Theorem 20], the two definitions are equivalent. We will refer to L_i above, for $1 \le i \le \ell(T)$, as the ith *Lyndon factor* of T. It is known that the Lyndon factorization of a string is unique [20] and can be computed in linear time in a left-to-right scan with Duval's algorithm [28].

The *suffix array* SA of a string T of length n is a permutation of the index set $\{0, \ldots, n-1\}$, defined as follows: $\mathsf{SA}[j] = i$ if the ith suffix $T[i..n-1]$ of T is the jth in lexicographic order of all suffixes (counting from 0). The *conjugate array* CA (also called *circular suffix array* and denoted $\mathsf{SA}_\circ$) of a string T is a permutation of the index set $\{0, \ldots, n-1\}$, defined as follows: $\mathsf{CA}[j] = i$ if the ith rotation $\mathsf{rot}^i(T)$ is the jth in lexicographic order (or, in this case equivalently, in omega-order) among all rotations of T. (If T is not primitive, then there are equal rotations, which are listed in CA in text order.) Note that SA and CA coincide if T is a Lyndon word, or if it is terminated by an end-of-string symbol, but otherwise they can differ.

Standard Sturmian words, or simply *standard words*, are strings that can be defined via a so-called *directive sequence*: this is an infinite sequence of integers $(d_i)_{i \ge 0}$ such that $d_0 \ge 0$ and for all $i > 0$, $d_i > 0$. This sequence generates a sequence of finite strings $(S_i)_{i \ge 0}$, called *standard words*, as follows: $S_0 = \texttt{b}$, $S_1 = \texttt{a}$, and $S_{i+1} = S_i^{d_{i-1}} S_{i-1}$ for $i \ge 1$. For instance, the directive sequence $1, 1, 1, 1, \ldots$ generates the well-known sequence of *finite Fibonacci words*: $F_0 = \texttt{b}, F_1 = \texttt{a}, F_2 = \texttt{ab}, F_3 = \texttt{aba}, F_4 = \texttt{abaab}, F_5 = \texttt{abaababa}, F_6 = \texttt{abaababaabaab}, F_7 = \texttt{abaababaabaababaababa}, \ldots$ For more details, see [52, Ch. 2].[4]

[3] Note that a *necklace* is sometimes defined as a conjugacy class w.r.t. rotation. The two definitions are closely connected: a necklace viewed as a string is a particular representative of a necklace viewed as a conjugacy class. In this sense, e.g. for counting arguments, the two definitions are equivalent.

[4] The indexing of Fibonacci words is not uniform in the literature, so one may find a shift by one or two in different papers. In particular, even and odd order Fibonacci words may be exchanged.

$\mathsf{CA}[i]$	$\mathsf{rot}^{\mathsf{CA}[i]}(T)$	$\mathsf{BWT}[i]$	$\pi_{\mathsf{BWT}}[i]$	i
10	aaabracadabr	r	10	0
11	aabracadabra	a	0	1
7	abraaabracad	d	9	2
0	abracadabraa	a	1	3
3	acadabraaabr	r	11	4
5	adabraaabrac	c	8	5
8	braaabracada	a	2	6
1	bracadabraaa	a	3	7
4	cadabraaabra	a	4	8
6	dabraaabraca	a	5	9
9	raaabracadab	b	6	10
2	racadabraaab	b	7	11

i	0	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	r	a	c	a	d	a	b	r	a	a

Figure 1 The BWT of the text $T =$ abracadabraa with its standard permutation $\pi_{\mathsf{BWT}}[i]$.

2.1 The BWT and the BBWT

The *Burrows-Wheeler Transform* (BWT) of a string T is a permutation of the characters of T. Consider the *multiset* of rotations of T, $\{\mathsf{rot}^i(T) : 0 \leq i < n\}$. If $T = U^k$ for a primitive string U, then each element of $[T]$ will appear k times in this multiset. Now $\mathsf{BWT}(T)$ can be obtained by concatenating the last characters of these strings when they are given in omega-order (or, equivalently in this case, in lexicographic order). For example, $\mathsf{BWT}($abracadabraa$) =$ radarcaaaabb. See also Figure 1. The number of runs of the BWT of a string T is denoted $r(T)$, i.e. $r(T) = \rho(\mathsf{BWT}(T))$. Thus, $r($abracadabraa$) = 8$.

Let S be a string of length n. The *standard permutation* of S is a permutation π_S of the index set $\{0, \ldots, n-1\}$ of S, defined as follows: $\pi_S(i) < \pi_S(j)$ if either $S[i] < S[j]$, or $S[i] = S[j]$ and $i < j$. In other words, $\pi_S(i)$ is the position of character $S[i]$ in the stable sort of the characters of S. For example, for $S =$ radarcaaaabb, $\pi_S = [10, 0, 9, 1, 11, 8, 2, 3, 4, 5, 6, 7] = (0, 10, 6, 2, 9, 5, 8, 4, 11, 7, 3, 1)$, where we give π both in one-line notation and in cycle notation. When S is the BWT of some string, then the standard permutation of S is also known as LF-mapping [29], which can be used to recover the original string up to rotation. For example, given π_S as above, we can recover the Lyndon word T with $\mathsf{BWT}(T) = S$, spelling it back-to-front, as follows: $T[n-1] = S[0] =$ r, and for $i > 1$, $T[n-i] = S[\pi^i(0)]$. We get $T =$ aaabracadabr, which is the Lyndon rotation of abracadabraa. The process can be modified to uniquely recover the input string by either adding the lexicographic rank of the rotation to be recovered, or by adding a unique end-of-string marker to the string.

Since all rotations of T have the same BWT, it is clear that the BWT is not a bijection on Σ^*. It is well known that a string S is the BWT of a primitive string if and only if π_S is cyclic. Likhomanov and Shur gave a more general characterization [50]: S is the BWT of some string if and only if the number of cycles of π_S equals the greatest common divisor of the lengths of the runs of S. This characterization encompasses powers, as well.

In fact, the number of strings of length n which are the BWT of some string (referred to as *BWT images* in [50]) equals the number of necklaces of length n. This follows from the definition of the BWT and the fact that the LF-mapping uniquely recovers the conjugacy class of the input string. In other words, the BWT is a bijection between necklaces of length

n and BWT images of length n. The number of necklaces of length n is $\sum_{d|n} Lyn(d,\sigma)$, where σ is the alphabet size and $Lyn(d,\sigma)$ is the number of Lyndon words of length d over an alphabet of size σ, or equivalently, the number of conjugacy classes of primitive words of length d. This can be seen by a counting argument on necklaces, distinguishing the case where the necklace is primitive (of which there are $Lyn(n,\sigma)$ many), and the case where it is a power U^k, with U primitive (in which case $d = |U|$ is a divisor of n). It is known [51] that $Lyn(d,\sigma) = \frac{1}{d}\sum_{d'|d} \mu(d')\sigma^{\frac{d}{d'}}$, where μ is the Möbius function defined as: $\mu(1) = 1$, $\mu(n) = (-1)^j$ if n is the product of j distinct primes, or 0 otherwise (n is divisible by a square number). Note that for all n, the number of necklaces of length n is at least σ^n/n, with equality if and only if n is prime.[5]

The extended BWT (eBWT) [54] is essentially[6] a bijection between multisets of Lyndon words of total length n and strings of length n. It is defined as the concatenation of the last characters of the rotations of the input strings, given in omega-order. Underlying the eBWT is the observation that the BWT is a special case of the inverse of the Gessel-Reutenauer bijection [31] between permutations of a given type and descent set and strings of a given type (see [21] for more details).

The eBWT is a straightforward generalization of the BWT in that for any primitive string T, $\mathsf{eBWT}(\{T\}) = \mathsf{BWT}(T)$. Given a string S, the standard permutation π_S of S (i.e., its LF-mapping) can be used to construct a multiset $\mathcal{M}$ with eBWT equal to S in the same way as before, yielding one string per cycle of π_S. For more details on the eBWT, see [18] in this volume. Note that the eBWT is surjective: every string is the eBWT of something. However, the preimage is not necessarily a string; in general, it is a multiset of strings.

The *bijective BWT* (BBWT), on the other hand, is a variant of the BWT which is a bijection from Σ^* to Σ^*: every string is the BBWT of some string (of the same length).

▶ **Definition 1** (Bijective BWT). *The Bijective BWT,* $\mathsf{BBWT}(T)$*, of a string* T *is the string obtained by concatenating the last character of the rotations of the multiset of the Lyndon factors of* T*, in* ω*-order of the rotations.*

The BBWT is thus the eBWT of the multiset of Lyndon factors of T (see Example 2). The inverse of the BBWT is constructed as follows: Take a string S, and let $\{L_1, \ldots, L_m\}$ be the multiset of Lyndon words that we get via the LF-mapping from S, i.e., one string per cycle of π_S. Now concatenate the strings L_i in non-increasing order. This yields the unique string T whose BBWT is S.

We define the *generalized conjugate array* GCA of a string T as follows: $\mathsf{GCA}[j] = i$ if the conjugate of L_k starting in position i is jth in omega-order among all conjugates of the Lyndon factors of T, where L_k is the Lyndon factor containing position i of T.

▶ **Example 2.** The BBWT of the 6th Fibonacci word is given by $\mathsf{BBWT}(\texttt{abaababaabaab}) = \texttt{bbbaababaaaaa}$. The Lyndon factorization of abaababaabaab is $(\texttt{ab})^1(\texttt{aabab})^1(\texttt{aab})^2$. See Figure 2 for a visualization. We can recover the original string as follows. The standard permutation of bbbaababaaaaa is $[8,9,10,0,1,11,2,12,3,4,5,6,7] = (8,3,0)(1,9,4)(2,10,5,11,6)(12,7)$. From this, we get: aab, aab, aabab, ab. Sorting these Lyndon words yields $\texttt{ab} \succeq_\omega \texttt{aabab} \succeq_\omega \texttt{aab} \succeq_\omega \texttt{aab}$, and thus, we get $T = \texttt{ab}\cdot\texttt{aabab}\cdot\texttt{aab}\cdot\texttt{aab}$.

[5] It is sometimes imprecisely stated, e.g. [32, 47], that $(1/n)$th of all strings are BWT images, but this is only true for n prime.

[6] The original definition of eBWT does not assume the strings to be Lyndon words, but stores information concerning the rotations of the input strings.

$\mathsf{CA}[i]$	$\mathsf{rot}^{\mathsf{CA}[i]}(T)$	$\mathsf{BWT}[i]$
7	aabaababaabab	b
2	aababaabaabab	b
10	aababaababaab	b
5	abaabaababaab	b
0	abaababaabaab	b
8	abaababaababa	a
3	ababaabaababa	a
11	ababaababaaba	a
6	baabaababaaba	a
1	baababaabaaba	a
9	baababaababaa	a
4	babaabaababaa	a
12	babaababaabaa	a

i	$\mathsf{GCA}[i]$		$\mathsf{BBWT}[i]$
0	7	aab	b
1	10	aab	b
2	2	aabab	b
3	8	aba	a
4	11	aba	a
5	5	abaab	b
6	3	ababa	a
7	0	ab	b
8	9	baa	a
9	12	baa	a
10	6	baaba	a
11	4	babaa	a
12	1	ba	a

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	a	b	a	a	b	a	b	a	a	b	a	a	b

Figure 2 The BWT (left) and BBWT (right) for the text $T = \mathtt{abaababaabaab}$, with Lyndon factorization $(\mathtt{ab})^1(\mathtt{aabab})^1(\mathtt{aab})^2$. We have $\mathsf{BWT}(T) = \mathtt{bbbbbaaaaaaaa}$ and $\mathsf{BBWT}(T) = \mathtt{bbbaababaaaaa}$.

Notice that the standard BWT can be considered as a special case of BBWT since $\mathsf{BBWT}(\mathsf{minrot}(T)) = \mathsf{BWT}(T)$. Indeed, the two transforms coincide exactly on the set of necklaces:

▶ **Proposition 3** ([12, Proposition 1]). $\mathsf{BWT}(T) = \mathsf{BBWT}(T)$ *if and only if* T *is a necklace.*

If we fix the inverse BWT to be always the necklace representative of the conjugacy class, then one can consider the BWT as a bijection from a subset of strings (necklaces) to a subset of strings (BWT images). The BBWT thus is a generalization of the BWT, since it is a bijection from Σ^* to Σ^* and coincides with the BWT on this subset.

Similarly to the BWT, we will be interested in the number of runs of the BBWT of a string T, denoted $r_B(T)$, i.e. $r_B(T) = \rho(\mathsf{BBWT}(T))$. Thus, $r_B(\mathtt{abaababaabaab}) = 6$.

In the rest of the paper, n denotes the length of the string under investigation (if no misunderstanding is possible).

3 Construction

Gil and Scott [32] sketched an algorithm based on the linear-time suffix array construction algorithm DC3 by Kärkkäinen, Sanders, and Burkhardt [41], for which they claim that the sorting "shall require $O(n \lg n)$ time". Unfortunately, the authors seem not to address the soundness of their algorithm nor provide source or pseudo code to give practical evidence that their approach indeed computes the BBWT. Mantaci et al. [54, before Proposition 10] conjectured that a generalization of DC3 should make it possible to compute the eBWT (and hence the BBWT) in linear time.

3.1 Dynamic construction

The first analysis of a BBWT construction algorithm with sound time bounds is due to Bannai et al. [6], who draw a connection to the algorithm of [14] constructing the eBWT,

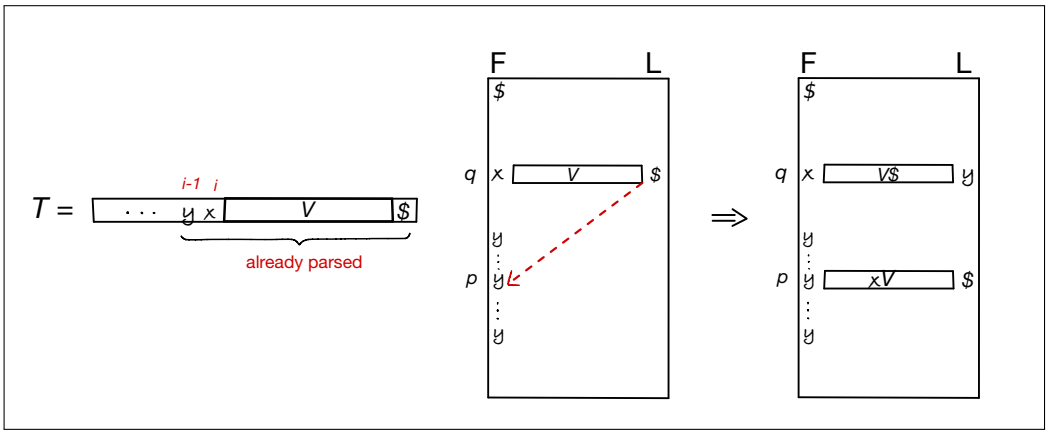

Figure 3 Online construction of the BWT on the reversed input text T described at the beginning of Section 3.1. Say that $T[i] = x$ and $T[i-1] = y$, and our task is to update the BWT of $T[i..]$ to the BWT of $T[i-1..]$. For that, it suffices to add the conjugate $yxV\$$ into the BWT matrix, where V is the suffix $T[i+1..n-2]$ after x up to the last character \$. Suppose that q is the BWT position storing \$, cf. the bottom left figure showing F and L, where L is the BWT and F the array storing the character of L sorted. A backward search step with character y from q gives a suffix starting with y and having the longest common prefix with $yxV\$$. To make the LF mapping jump from q to p, we exchange $\mathsf{BWT}[q] = y$. To close the cycle, we insert \$ at position p. This gives the L array (i.e., the BWT of $T[i-1..]$) on the bottom right. We can verify that the introduced conjugate at BWT position p is in the lexicographic order due to how the backward search works.

and were able to adapt this algorithm for computing the BBWT in $O(n \log n / \log\log n)$ time. This BBWT construction algorithm works in an *online* fashion, i.e., it builds the BBWT per Lyndon factor, starting with the first one, and then incrementally indexes each further factor until adding the last Lyndon factor. Compared to the BWT, there is no known algorithm computing the BWT in an online model that allows reading a specific prefix of the text (such as its Lyndon factors). However, it is possible to construct the BWT online by reading the text from the reverse order (e.g., [22]) – a setting for which we do not know whether a solution for the BBWT exists. The main idea of the BWT construction algorithm is that it computes the BWT of $T\$$, where \$ is a sentinel character smaller than all other characters appearing in T. The construction starts with BWT just storing \$, and at any time later, the BWT contains exactly one \$ at a specific position. By the definition of the LF mapping, going from this position backwards would give the last character of T. Now, assume that we have built the BWT of $T[i..n-1]$ and want to insert $T[i-1]$. For that, we perform a backward search step from the position of \$ for the character $T[i-1]$. Say we end up at a position p, then we exchange \$ with $T[i-1]$ but insert \$ at position p and recurse, see Figure 3 for a visualization. The same trick works without \$ for the BBWT if we process the Lyndon factors sequentially, starting from the leftmost and lexicographically largest one. This is because, when inserting a new Lyndon factor L_x, we know that it is lexicographically smaller than all other Lyndon factors. Hence, we can insert the last character c of L_x at the first position in the BBWT and backward search c to find the insertion position of the previous character in L_x. This gives the following complexities.

▶ **Theorem 4** ([6, Theorem 22]). *We can construct the BBWT of a string of length n in $O(n \log n / \log\log n)$ time in an online model, where we receive at each step one Lyndon factor of the input.*

3.2 Linear-time algorithms

In this section, we will sketch the two existing linear-time algorithms for BBWT construction. Both algorithms are non-trivial modifications of linear-time suffix array construction algorithms. The first, due to Bannai et al. [7] (2021), is based on the well-known SAIS-algorithm of Nong et al. [63]. The second, due to Olbrich et al. [64] is a modification of the algorithm GSACA by Baier [3]. In what follows, we will sketch the proofs of the following theorem:

▶ **Theorem 5** (Linear time construction of BBWT [7, 8, 15, 64]). *We can construct the BBWT of a string of length n in $O(n)$ time.*

In the case of a string that is terminated by an end-of-string character, the BWT can be computed in linear time from the suffix array, by noticing that $\mathsf{BWT}[i] = T[n-1]$ if $\mathsf{SA}[i] = 0$, and $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ otherwise. This, however, uses the fact that the last suffix is the smallest among all suffixes, which is not necessarily the case in general. In actual fact, the conjugate array CA would be needed, since for all $0 \leq i \leq n-1$, we have $\mathsf{BWT}[i] = T[\mathsf{CA}[i]-1]$, where the subtraction is modulo n. One solution is to first compute the Lyndon rotation of T because for Lyndon words, the suffix array and the conjugate array coincide. The first algorithm that directly computes the BWT of a string T without an end-of-string character, and without computing Lyndon rotations, was given in [15] (and was inspired by the SAIS-based algorithm [7] we are about to see).

For the BBWT, we need is the GCA of T, due to the following connection:

$$\mathsf{BBWT}[i] = \begin{cases} T[\mathsf{GCA}[i] + |L_k| - 1] & \text{if } \mathsf{GCA}[i] \text{ is the starting position of a Lyndon factor } L_k, \\ T[\mathsf{GCA}[i] - 1] & \text{otherwise.} \end{cases}$$

The two linear-time algorithms below take advantage of this idea by modifying known SA-construction algorithms to construct the GCA instead of the SA.

3.2.1 A SAIS-based algorithm

The first $O(n)$-time construction algorithm of the BBWT was presented in [7, 8]. It is an adaptation of the SAIS (Suffix Array by Induced Sorting) algorithm [63], a recursive linear-time algorithm. The algorithm of [7, 8] was then adapted and simplified for eBWT construction by Boucher et al. [15]. Some of the simplifications are also applicable for BBWT construction, so our presentation here is a mixture of the two.

We first give a quick recap of SAIS. The SAIS algorithm first categorizes each text position i in L- or S-type, according to whether the suffix $T[i..]$ is larger (L) or smaller (S) than the next suffix $T[i+1..]$. S-type positions following L-type positions are called LMS-positions (for "leftmost S"). The recursive step is done on a new text, where each so-called *LMS-substring* of T is replaced by a meta-character; *LMS-substrings* are minimal substrings containing two consecutive LMS-positions. The meta-characters are assigned to the LMS-substrings respecting their *LMS-order* $<_{LMS}$, a slight modification of the lexicographic order.[7] For a suffix i of T let U_i be its *LMS-prefix*, i.e., U_i is the prefix of $T[i..]$ that is a suffix of the LMS-substring in which i lies (and therefore, the full LMS-substring if i is an LMS-position). Nong et al. [63] show that if $U_i <_{LMS} U_j$ then $T[i..n) <_{\text{lex}} T[j..n)$, while if

[7] Even though the original definition is more complex, it can be shown that for two LMS-substrings U, V: $U <_{LMS} V$ if either V is a proper prefix of U, or neither is a prefix of the other, and $U <_{\text{lex}} V$.

$U_i = U_j$, then the order is determined by the subsequent LMS-substrings. This is the basis of the recursive step. The recursion terminates when all meta-characters are distinct, at which point the suffix array of the meta-text on the current recursion depth can be trivially computed. The suffix array of the meta-text implies the relative order of the suffixes starting at LMS-positions in T, which is then used to induce the SA-positions of the remaining suffixes in two linear passes over SA, the first left-to-right, inducing L-type positions, the second right-to-left, inducing S-type positions.

This algorithm has to be adapted for (a) computing the omega-order, rather than the lexicographic order, and (b) between conjugates of different strings, rather than suffixes of the same string.

Let us first assume that there are no Lyndon factors of length one. The first modification is that the position types must be defined cyclically within the Lyndon factors. This can be done in linear time, similarly to the SAIS algorithm, using the fact that the smallest rotation of Lyndon words start in their first position. Second, it can be shown that, using the cyclically defined types, the LMS-substrings and LMS-prefixes, defined cyclically, have a similar property: $U_i <_{LMS} U_j$ then $T[i..n) <_\omega T[j..n)$, while if $U_i = U_j$, then the order is determined by the subsequent LMS-substrings [15, Lemma 4]. This again enables recursion, using the additional insight that the starting positions of Lyndon factors constitute a subset of the starting positions of the LMS substrings [8, Lemma 3.4]. This implies that the meta-string's Lyndon factorization is the one induced by the Lyndon factorization of T. Finally, the inducing step (inducing the GCA using the relative order of the LMS-conjugates) can be done analogously to SAIS, because here, too, all L-type conjugates come before all S-type conjugates in the same bucket.

It can also be shown that length-1 strings are always placed between L- and S-type conjugates in the same bucket. Accordingly, the algorithm removes length-1 Lyndon factors at the beginning and places them in the correct place at the end, when the rest of the GCA has been filled in. For an example, see Example 6 and Figure 4.

Finally, Boucher et al. [15] showed that essentially the same algorithm works on *any* multiset of input strings, given in *any* order – their work targets the eBWT construction, where the input strings are not assumed to be Lyndon words, nor is it assumed that they are given in a relative order. The main algorithmic change in that case is in how the type assignment is done (scanning each string at most twice).

▶ **Example 6.** We give an example with a recursive call in Figure 4, on the string $T =$ `drabracacadabracaa`. We start on the upper left box with the classification of the text positions of T into S- and L-types, omitting positions that correspond to Lyndon factors of length 1. Each S succeeding an L (considered cyclically) gets a star ($\star$) to denote an LMS-type position. The Lyndon factorization of T is denoted by vertical bars between the characters of T. Next we replace the LMS-substrings by meta-characters `A` $<$ `B` $<$ `C` $<$ `D`, according to their rank in LMS-order, to form the meta-text based on these meta-characters. Since not all characters of T' are distinct, we recurse on T' (right side); here, all LMS conjugates are distinct such that we can immediately start the inducing step. We insert the entry for $T'[0] =$ `D` at the end of the recursive call at the end of the `D`-bucket of GCA$'$ (there being no S-type conjugates for the largest character). GCA$'$ translates back to give the relative order of the LMS-conjugates of T, which is then used to induce the entire GCA, in the usual way, except that i induces $i-1$ cyclically w.r.t. the factor it is in. Finally, the entries for $T[16] = T[17] =$ `a` are inserted at the beginning of the `a`-bucket of GCA at the end of the execution (there being no L-type conjugates for the smallest character). We give the final BBWT(T) at the bottom of the figure.

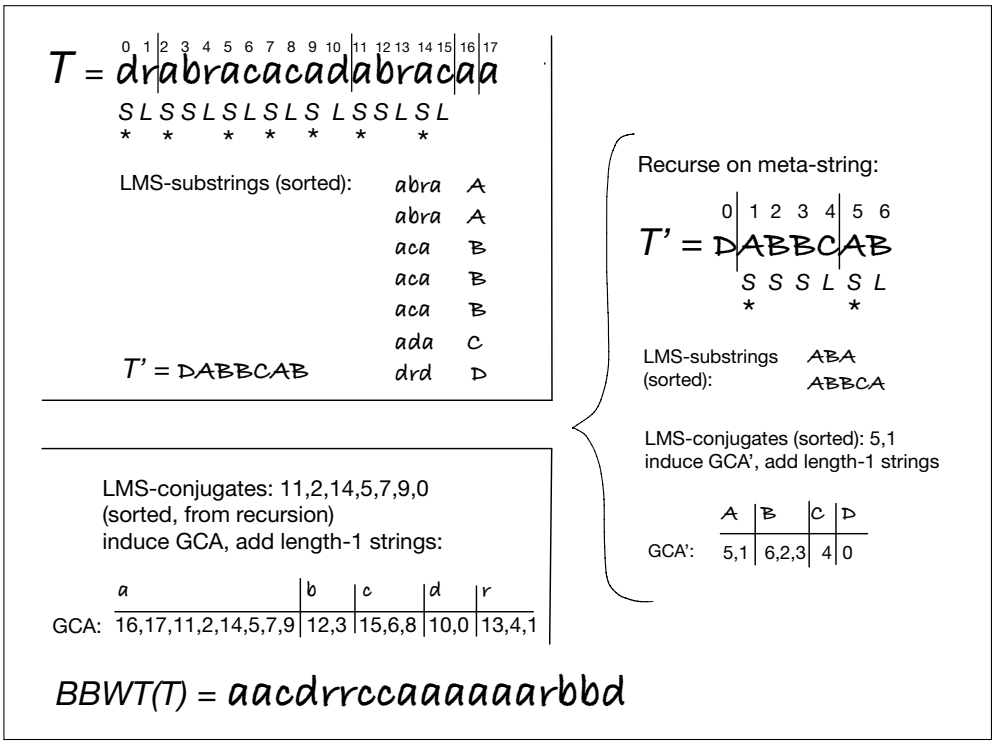

Figure 4 Construction of the GCA via a modification of the SAIS algorithm as explained in Section 3.2.1. See Example 6 for details.

3.2.2 A GSACA-based algorithm

Baier [3] in 2016 presented the first non-recursive linear-time suffix array construction algorithm, called GSACA. The algorithm came with an implementation, which was, however, not competitive with the best practical suffix construction tools. A modified version by Bertram et al. [10] later proved to be very efficient practically, as well. The original GSACA algorithm was then adapted for eBWT and BBWT construction by Olbrich et al. [64].

As was shown in [30], the GSACA algorithm uses Lyndon prefixes to compute a partial order of the suffixes. Given a string X, its *Lyndon prefix* is the longest prefix U which is a Lyndon word, which is also the first factor in its Lyndon factorization [28]. For example, the Lyndon prefix of `abracadabraa` is `abracad`. GSACA first groups the suffixes according to their Lyndon prefixes (Phase 1). This order then gets refined by sorting the suffixes within the same group (Phase 2). Since the groups appear in lexicographic order according to the groups' Lyndon prefixes, this yields the final suffix array. The crucial observation is that if U, V are the Lyndon prefixes of strings X, Y, respectively, then $U <_{\text{lex}} V$ implies $X <_{\text{lex}} Y$.

Astonishingly, the distribution of the suffixes into their groups can be done in one single right-to-left scan. First the suffixes are partitioned into groups according to their first character; these are necessarily the first characters of their Lyndon prefixes. This initial partition is then refined by removing elements from groups and extending their labels to longer prefixes, where the label of a group is a common prefix shared among all group elements. At the end of this scan, each element will be in a group whose label equals the suffix's Lyndon prefix. This is done as follows: proceeding from the highest group

towards the lowest (i.e., right to left), for every suffix i in the current group, we will take another suffix $j = prev(i)$ from a lower group, if such a j exists, and insert it into a new group right after its current group, with its label extended by the label of i's group. Here, $prev(i) = \max\{i' < i : i' \text{ is in a lower group than } i\}$. See Figure 5 for an example. This completes Phase 1.

Now, in Phase 2, the correct order within each group is induced by starting from the smallest suffix, which in the original algorithm, is the end-of-string character at position $n-1$. In the modified algorithm for computing the BBWT, we need to take care of the circular manner of inducing within Lyndon factors. Note that with the Lyndon grouping from Phase 1, we also have the Lyndon factorization of T, so we can use the first position of each Lyndon factor to start the inducing step.

▶ **Example 7.** We give an example on the same string $T =$ `drabracacadabracaa` as in Example 6, see Figure 5. In Phase 1, we first place all suffixes in a group according to their first character. We then start with the highest group, the one with label `r`: for the element 1, $prev(1) = 0$, which is in group `d`. We remove 0 from its group and move it into a new group, immediately after, with label `dr`, which is the concatenation of the label of 0's group (`d`) and the label of 1's group (`r`). Similarly, 4 and 13 cause the removal of 3 resp. 12 from their initial group `b` and are inserted into a new group, immediately after, with label `br`. Then we move to the group `dr`, whose only element 0 does not have any element to induce. 10 in group `d` induces the removal of 9 and its insertion into the new group `ad`, etc. The resulting Lyndon grouping is shown at the bottom left of the figure. In Phase 2, each group is sorted, via induced sorting with one pass from left to right. In (a) we show the result of the original algorithm constructing the SA of T\$, from which the BWT can be computed. (Note that in that case, the algorithm assumes that T has a final dollar-symbol.) In (b) we show the result if the inducing is done w.r.t. Lyndon factors, to get the GCA and thus the BBWT. Differences between the two must necessarily lie within Lyndon groups. For example, note the difference in the last group (labeled `r`), where 1 is now placed after the other two elements because it is followed by `d` in its Lyndon factor `dr` (and not by `a` as in T). We give BBWT(T) at the bottom of the figure.

3.3 In-place construction

Köppl et al. [46] discovered that the BBWT construction algorithm of Theorem 4, which incrementally builds the BBWT from the text by one Lyndon factor at a time, can be used in the setting of in-place construction, where only $O(1)$ additional working space is allowed to turn T into its BBWT. Here, the main observation is that the in-place construction algorithm of [22] for the BWT can be changed to compute the BBWT instead of the BWT. That is because we can compute the Lyndon factorization with Duval's algorithm with constant space while constructing the BBWT. In detail, we let Duval's algorithm detect the next Lyndon factor, then pause it to consume this factor, and recurse by resuming Duval's algorithm.

▶ **Theorem 8** (in-place construction [46]). *We can compute the BBWT in $O(n^2)$ time using $O(1)$ words of working space.*

For restoring the original text from the BBWT, there is a need for supporting the FL-mapping, the inverse of the LF-mapping. The FL-mapping can be implemented by a select query on BBWT, which however needs $O(n^{1+\varepsilon})$ time for a solution with constant working space [59] for a constant $\varepsilon > 0$.

▶ **Theorem 9** ([46]). *We can invert the BBWT in $O(n^{2+\varepsilon})$ time using $O(1)$ words of working space for a constant $\varepsilon > 0$.*

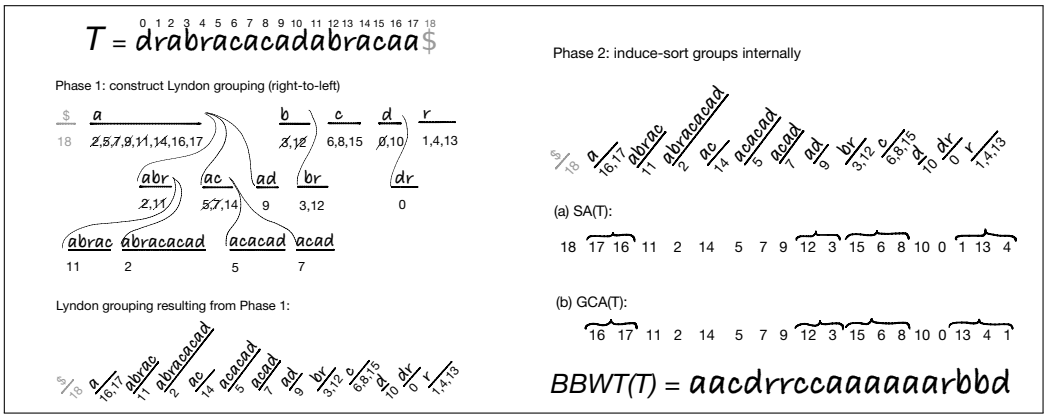

Figure 5 Construction of the BBWT using the GSACA-based algorithm, see Section 3.2.2. (a) shows the internal order of the groups we would get if we wanted to compute the SA, for the BWT of T; (b) shows the order we get for GCA, needed to compute the BBWT of T. Notice the difference in the groups labeled `a` and `r`. (As the dollar-symbol is needed only for the SA-algorithm, it and its group are marked in grey.) See Example 7 for more details.

A more involved analysis revealed that it is even possible to turn BWT into BBWT directly without inversion using $O(n^{2+\varepsilon})$ time and $O(1)$ working space. The idea is to run Duval's algorithm using FL-mappings to detect the Lyndon factorization. Since Duval's algorithm uses three pointers to text positions, we can simulate the pointers by pointers into the BWT and use the FL-mapping to advance a pointer. When a Lyndon factor has been detected, we try to move \$ such that the Lyndon factor has its own orbit in the standard permutation of the BBWT. However, this change can also break the cycle of the remaining BWT, for which additional moves have to be performed.

The authors also gave an algorithm for computing the run-length compressed BBWT online, Lyndon factor by Lyndon factor. Similar to computing the run-length compressed BWT online reading the reversed text, the authors also show how to compute the run-length compressed BBWT in $O(r_B)$ space and $O(n \log n / \log\log n)$ time.

3.4 Implementations

On the practical side, we are aware of the implementations by David A. Scott[8], Branden Brown[9], Yuta Mori in his OpenBWT library[10], and of Neal Burns[11]. While the first two are naive but easily understandable implementations calling general sorting algorithms on all conjugates to directly compute the BBWT, the implementation of Yuta Mori seems to be an adaptation of the SAIS algorithm to induce the BBWT. The implementation of Neal Burns takes an already computed suffix array SA as input, and transforms SA into GCA by shifting values to the right until they fit. Hence, the running time is based on the lengths of these shifts, which can be $O(n^2)$, but seem to be negligible in practice for common texts.

Finally, the two linear-time algorithms we presented, due to Bannai et al. [8][12] and Olbrich et al. [64][13], have both been implemented.

[8] `https://web.archive.org/web/20210615215623/http://bijective.dogma.net/bwts.zip`
[9] `https://github.com/zephyrtronium/bwst`
[10] `https://web.archive.org/web/20170306035431/https://encode.ru/attachment.php?attachmentid=959&d=1249146089`
[11] `https://github.com/NealB/Bijective-BWT`
[12] `https://github.com/mmpiatkowski/bbwt`
[13] `https://gitlab.com/qwerzuiop/lfgsaca`

4 Indexing

The eBWT is a generalization of the BWT for multiple input texts. As such, it is possible to build an FM-index-like data structure upon the eBWT. Indeed, a more advanced compressed text index built upon the eBWT, the so-called *extended r-index*, was recently presented in [16], which can efficiently answer pattern matching queries while requiring only $O(r)$ space.

The reason this kind of index is not immediately applicable for the BBWT is because, by interpreting the BBWT as the multiset of Lyndon factors of T, we would be restricted to patterns that occur within the Lyndon factors. In order to use the BBWT for an index for regular pattern matching on the original text, we must (a) remove superfluous matches that match the Lyndon factor in a cyclic sense but do not occur in the original string, and (b) find a way to detect matches that span multiple Lyndon factors, as these are not found by the standard backward search on the BBWT.

Such an index was presented in [6, 8]. We present this index under the facilitated setting that all Lyndon factors of T are distinct, i.e., $T = L_1 \cdots L_{\ell(T)}$ with $e_1 = \ldots = e_{\ell(T)} = 1$. This imposes not a restriction because we first detect the actual powers $e_1, \ldots, e_{\ell(T)}$ in a precomputation step before discarding duplicate Lyndon factors. After building the index we will introduce, we augment it such that the correct numbers are counted for each Lyndon factor [8, after Theorem 4.7]. We also limit our survey to counting queries, i.e., to return the number of occurrences of a pattern in the text.

The index we are going to present works like the FM-index. The only difference is that we need to take action whenever the LF-mapping interval contains a position corresponding to the starting position of a Lyndon factor. Otherwise, it applies the backward search step in exactly the same way as the FM-index does. This is because a Lyndon word cannot cross two Lyndon factors of the text, meaning that any pattern that is a Lyndon word cannot be the suffix of some L_k and the prefix of some L_{k+1}, since necessarily $L_k \succ L_{k+1}$. Consequently, a Lyndon pattern can be found with the same techniques as used by the FM-index since it can only occur inside a Lyndon factor of the text. For finding *all* patterns P, we use the Lyndon factorization of P-by our observation, we can be assured to match at least the last factor of P correctly.

By the above observation, we are sure that we can find the last Lyndon factor of P by the backward search without any modification. However, a backward search step of the preceding character may already give unexpected answers if this last factor is a prefix of a factor of the text. For instance, for the particular case that the pattern ends with L_k, the LF-mapping may report an occurrence of this pattern (regardless of the remaining prefix) because the LF-mapping extracts the original cyclic Lyndon words by iterative application. Hence, only at those backward search steps that read the last character of a previous Lyndon factor of the pattern, we can observe phenomena that need to be taken care of, namely:

a superfluous match: caused by matching with the same Lyndon factor L_k of the text, and

a missed match: caused by the fact that we do not check the characters starting before L_k in text order for a potential occurrence.

Both phenomena happen at most p' times, where p' is the number of Lyndon factors of P. Each time, we need to take care of at most one superfluous and one missed match. We track these individual matches by mapping them individually with the LF-mapping such that the final cost for matching P are $O(|P|p')$ rank queries. To get the final result, we note that after the longest so-called *significant suffix* [39] of P has been matched, there is only one match remaining to keep track of. Since the longest significant suffix can contain $O(\log |P|)$ different Lyndon factors, we obtain the following result.

▶ **Theorem 10** (Pattern matching using BBWT [6, 8]). *Given a text T and a pattern P of length m, we can compute all occurrences of P in T with the FM-index built on* $\mathsf{BBWT}(T)$ *with $O(m \log m)$ rank/select operations.*

The rank/select operations in Theorem 10 are not only necessary for the backward search, but for a bit vector B with rank/select support marking the positions in BBWT corresponding to the distinct Lyndon factors.

The BBWT can also be run-length compressed into $O(r_B)$ space while still supporting count queries. For that, the bit vector B can be entropy-compressed [8].

5 Compression

The BWT was originally introduced for compression: Burrows and Wheeler state that the BWT "tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially" [17]. An intuitive reason to why this is so, is that since the rotations of the string are sorted in lexicographic (or ω-)order, the BWT is a sequence of symbols that precede substrings with a similar context. It has been shown that achieving zero order entropy compression on appropriately partitioned blocks of the BWT achieves higher order entropy compression of the original string [57]. Thus, a string can be compressed by first applying BWT, and then applying some simple compression algorithms such as Move-To-Front encoding or run-length encoding.

While the analysis of the compression performance of the BWT has received much attention, the compression performance of the BBWT is less well studied. Scott [66] and Gil & Scott [32] reported the compression performance of BWT and BBWT on the Large Calgary corpus [9], often used as a benchmark dataset. On all but one of the 18 files, the BBWT resulted in a slightly better compression than the BWT: the average gain was around 1% of the original text size. These data sets are very small by today's standards. We only know of empirical evaluations on larger data sets which compare the number of runs in the BWT and BBWT for various data sets (the Calgary Corpus, Canterbury Corpus, Pizza&Chili Corpus, and Silesia Corpus), where it is reported that they are roughly equal [8].

Recently, it has become recognized that, for *highly repetitive data* that are prevalent today in many real-world applications, analyses of compressors in terms of entropy measures, i.e., on statistical properties of the data, are not useful as they do not capture the repetitiveness of the data very well [61]. Instead, a trend is to analyze the compression performance of compressors in terms of the sizes of compressed representations of the individual strings, using representations, in particular, based on *dictionary compression*, that can better model the repetitiveness of the data.

The systematic study of *repetitiveness measures* related to dictionary compression was initiated by Kempa and Prezza [44]. Dictionary compression is a family of compressed representations that are essentially based on copy and paste operations, and include representations, for example, those computed by well-known algorithms like LZ77, grammar-based representations (e.g., LZ78, RE-PAIR), run-length encoding, and run-length encoded BWT (RLBWT). See [61] for a comprehensive survey on the subject.

In the following, we introduce results focused on theoretical analyses on the compressiveness of BBWT. Recall that, for a string T, $r(T)$ denotes the number of runs of $\mathsf{BWT}(T)$ and $r_B(T)$ the number of runs of $\mathsf{BBWT}(T)$.

5.1 r_B as a measure of dictionary compression

The Run-Length compressed BWT (RLBWT) [53] is a representation of T of size $r(T)$. From its definition, it is not clear why the RLBWT can be considered as a form of dictionary compression, since the transform is applied before the run-length encoding. However, suppose for each run in the BWT, we define the reference (copying) of a symbol to point to the previous symbol in the run (except for the first one in the run). Then, if we consider all of these references on corresponding positions of the text, the text can be partitioned into at most $2r(T) + 1$ phrases, where each phrase is either a single symbol that corresponds to the beginning of a run, or, a maximal substring such that the offsets of the references are the same for all positions in the phrase [62, Theorem 9][14]. The latter phrases can then be encoded by two integers – the length and offset, resulting in a representation for T of size $O(r(T))$, known as bidirectional macro schemes (BMSs) [70], the most general (and most powerful) representation in dictionary compression.

The above relation can be shown as follows: Suppose two lexicographically adjacent rotations $\mathsf{rot}^{i+1}(T)$ and $\mathsf{rot}^{j+1}(T)$ have the same BWT symbol (i.e., $T[i] = T[j]$). Then, position j will reference position i, with offset $i - j$. Due to the property of LF mapping, $\mathsf{rot}^{i}(T)$ and $\mathsf{rot}^{j}(T)$ will also be adjacent rotations, and, if they have the same BWT symbol as well, position $j - 1$ will reference position $i - 1$, with offset $(i - 1) - (j - 1) = i - j$ and therefore have the same offset. This continues until the adjacent rotations do not have the same BWT symbol, which can happen at most $r(T) - 1$ times – at boundaries of runs. A minor point we have overlooked above is that the offsets $i - j$ and $(i - 1) - (j - 1)$ would always be the same in the cyclic sense, but they can become different in the linear sense when $i = 0$ or $j = 0$, since position -1 wraps around to position $n - 1$. This can happen at most twice. Since there are exactly $r(T)$ single symbol phrases, the total number of phrases is at most $(r(T) - 1) + 2 + r(T) = 2r(T) + 1$.

Badkobeh et al. [2] showed that for the Run-Length compressed BBWT (RLBBWT), a similar BMS of size $O(r_B(T))$ can be induced. An important observation to achieve this bound is that $r_B(T)$ can be lower bounded by the number of distinct Lyndon factors $\ell(T)$ of the Lyndon factorization of T [16, 2]. This follows from [16, Corollary 9], which states that:

▶ **Lemma 11** ([16, Corollary 9]). *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a conjugate-free set of primitive strings of total length N, r the number of runs of its eBWT. Then $m \leq r$. Moreover, if all strings T_i have the same length l, then $N/r \leq l$.*

Since, in the case of BBWT, $\mathcal{M} = \{L_1, \ldots, L_{\ell(T)}\}$ where all the Lyndon factors are distinct and primitive, we have:

▶ **Corollary 12** ([2]). *For any string T, $\ell(T) \leq r_B(T)$.*

Applying the same referencing of symbols for BBWT as BWT described above, the same arguments hold, except that there can be more instances where the references wrap around, i.e., when i or j corresponds to the beginning of a Lyndon factor. However, this is at most twice for each distinct Lyndon factor, since identical Lyndon factors occur adjacently. Therefore, by Corollary 12, the total number of phrases can be bounded by $O(r_B(T)+\ell(T)) = O(r_B(T))$.

▶ **Theorem 13** ([2, Lemma 1]). *For any string T, there exists a BMS of size $O(r_B(T))$ that represents T.*

[14] We note that [62] assumes that strings are terminated with a \$ and the proved bound is $2r(T)$. For strings that are not terminated by \$, there can be $2r(T) + 1$ phrases, e.g., with $T = ababaaaabab$.

Therefore, RLBWT and RLBBWT respectively induce BMSs of size $O(r(T))$ and $O(r_B(T))$, where all references point to a smaller (omega-order) position, and can be considered as a form of dictionary compression.

5.2 Relation to LZ77

Denote by $z(T)$ the size of the LZ77[15] factorization [48] of T. It is known that $z(T)$ is the size of a smallest BMS where all references point to a smaller (text-order) position. The question of whether $r(T)$ can be bounded in terms of $z(T)$ was answered by Kempa and Kociumaka [43, Theorem 3.2], who showed that for any string T, $r(T) = O(z(T)\log^2 n)$ holds.

Badkobeh et al. [2] showed that this proof can be extended to the case of $r_B(T)$. The term $z(T)$ appears in Kempa and Kociumaka's proof when considering, for given k, the set T_k of all substrings of T^ω of length k, where $|T_k|$ can be bounded by $O(z(T)k)$ since any substring of T^ω must have an occurrence containing the last position of an LZ77 phrase (including the last phrase). The main difference in the proof is in the definition of the set T_k, which should be modified to be the set of all substrings of length k of $\{(L_1)^\omega, \ldots, (L_{\ell(T)})^\omega\}$, rather than T^ω. This can be bounded by $O(z(T)k + \ell(T)k)$, since such a substring is either a substring of T, or, is a substring of $(L_i)^\omega$ that is not a substring of L_i. $|T_k| = O(z(T)k)$ is obtained by applying the result by Urabe et al. [71] that shows $\ell(T) \leq 4z(T)$. The rest of the proof is essentially the same, giving:

▶ **Theorem 14** ([2, Theorem 1]). *For any string T, $r_B(T) = O(z(T)\log^2 n)$.*

For BWT, a tighter bound of $r(T) = O(\delta(T)\log^2 n)$ where $\delta(T) = \max_{1\leq d\leq n}\{T_d/d\}$ follows easily, since $\delta(T) \geq |T_k|/k$, $|T_k| = O(\delta(T)k)$. Kempa and Kociumaka further show a bound of $r(T) = O(\delta \log \delta \max\{1, \log \frac{n}{\delta \log \delta}\})$ [43, Theorem 3.7]. We note that $\delta(T)$ is known to lower-bound (and can be strictly smaller than) all dictionary compression measures [45]. Tighter bounds for $r_B(T)$ are not yet known.

5.3 Robustness of r_B

The following results are known about the change that r_B may undergo if some operation is applied to T.

5.3.1 String reversal

Giuliani et al. [34] studied the robustness of r with respect to string reversal, i.e., how $r(T)$ and $r(T^{\text{rev}})$ can differ. They gave an infinite family of strings (which they called Fibonacci-plus words) for which $r(T) = O(1)$ and $r(T^{\text{rev}}) = \Theta(\log |T|)$, i.e., with a runs-ratio that is logarithmic in the length of the string. This lower bound is nearly tight, in view of the known upper bound of $r(T)/r(T^{\text{rev}}) = O(\log^2 |T|)$ [43]. Strings with higher runs-ratio than the family of [34] (albeit only experimentally) were given in [33].

Biagi et al. [11, 12] paralleled this result for the BBWT, giving a lower bound of $\Theta(\log n)$ on $r_B(T)/r_B(T^{\text{rev}})$.

▶ **Theorem 15** ([12]). *There exists an infinite family of strings such that $r_B(T) = O(1)$ and $r_B(T^{rev}) = \Theta(\log n)$.*

[15] We use the name LZ77 which is prevalent in the literature, though it has been mentioned that LZ76 is more precise [60].

The family $(T_k)_{k\in\mathbb{N}}$ provided in the proof of Theorem 15 are Lyndon rotations of Fibonacci words: On the one hand, these words have $r_B(T_k) = 2$, by Proposition 3 and Theorem 23. On the other, the authors show that $\mathsf{BBWT}(T_k^{\text{rev}})$ has exactly $2(k-2)$ runs. This is done by studying the Lyndon factorization of T_k^{rev}: the authors show that it consist of the Lyndon rotations of Fibonacci words of order 0 to $k-2$, with Lyndon rotations of even order words, in increasing order, first, followed by Lyndon rotations of odd order words, in decreasing order, with the last Lyndon factor, $\mathtt{a}$, repeated twice. Therefore, $r_B(T_k^{\text{rev}})$ equals $eBWT(\mathcal{S}_{k-2})$, where $\mathcal{S}_j = \{F_i : i = 0, \ldots, j\}$, since the BWT is invariant w.r.t. rotation and multiple occurrences of a string in a multiset do not impact on the number of runs of the eBWT [54]. The authors then give the exact form of $eBWT(\mathcal{S}_j)$, which is shown to contain exactly $2j$ runs.

In the same paper, experimental results were given both on $r_B(T)/r_B(T^{\text{rev}})$ (multiplicative difference) and on $r_B(T)-r_B(T^{\text{rev}})$ (additive difference), as well as on the connection between these two functions and the number $\ell(T)$ of distinct Lyndon factors.

5.3.2 Single character edits

A side result of [34] is that the BWT is sensitive to one-character edits, in the sense that there exist strings on which $r(T)/r(T') = \Omega(\log |T|)$, where T' results from T by appending one character to T. Akagi et al. [1] introduced the term *sensitivity* of a measure to denote the asymptotic change when a string undergoes single-character edits of different types. This was studied in detail in [35, 36] and logarithmic lower bounds were given for the multiplicative sensitivity $r(T)/r(T')$, and an $\Omega(\sqrt{n})$ bound for the additive sensitivity, $r(T) - r(T')$.

Jeon and Köppl [40] showed that this result can be translated to the BBWT with the same bounds by studying, among others, the Lyndon rotations of Fibonacci words and applying Corollary 12.

▶ **Theorem 16** ([40]). *There is a family of strings $(T_n)_{n\geq 1}$ with $r_B(T_n)/r_B(T'_n) = \Theta(\log n)$ and a family of strings $(S_n)_{n\geq 1}$ with $r_B(S_n) - r_B(S'_n) = \Theta(\sqrt{n})$ such that $|T_n| = \Theta(|S_n|) = \Theta(n)$ and T'_n and S'_n differ from T_n and S_n by one edit, respectively.*

5.4 The relationship between r and r_B

Badkobeh et al. [2] considered the difference between $r(T)$ and $r_B(T)$. Notice that a lower bound on the multiplicative difference between $r(T)$ and $r_B(T)$ is a lower bound on the multiplicative difference between $r_B(T)$ and $r_B(T')$ for any rotation T' of T due to Proposition 3, since $r(T)$ is equivalent to $r_B(\mathsf{minrot}(T))$, the RLBBWT of a specific rotation of T. A family of strings where $r_B(T)$ could be a $\Theta(\log n)$ factor larger than $r(T)$ was reported.

▶ **Theorem 17** ([2, Theorem 2]). *There exists an infinite family of strings such that $r(T) = O(1)$ and $r_B(T) = \Theta(\log n)$.*

The family for Theorem 17 is the Fibonacci words of odd order, i.e., $\{F_{2k+1} : k = 0, 1, \ldots\}$. It is well known that $r(F_k) = 2$ for any $k \geq 2$. For $r_B(F_{2k+1})$, using a result by Melançon on the Lyndon factorization of F_{2k+1} [58], it is shown that the number of Lyndon factors is $k + 1$. Then, by Corollary 12, $r_B(F_{k+1}) = \Omega(k)$ is obtained.

The opposite case was shown in [5]: that $r(T)$ can be a $\Theta(\log n)$ factor larger than $r_B(T)$.

▶ **Theorem 18** ([5, Theorem 2]). *There exists an infinite family of strings such that $r_B(T) = O(1)$ and $r(T) = \Theta(\log n)$.*

The family used in the proof of Theorem 18 was also related to Fibonacci words. It is noticed that prepending a symbol b to the Lyndon rotation of a Fibonacci word is conjugate to the string used by Giuliani et al. [34, 35, 36] for BWT, thus giving $\Theta(\log n)$ runs, while r_B is easily verifiable to be 3. An alternate (perhaps simpler) direct proof based on morphisms is also given in [5].

As for upper bounds, poly-logarithmic bounds were derived from the upper bounds $O(z(T)\log^2 n)$ of both $r(T)$ and $r_B(T)$ mentioned above, as well as other known bounds $z(T) = O(\delta(T)\log n)$ and $r(T), r_B(T) = \Omega(\delta(T))$ [44, 45, 61].

▶ **Theorem 19** ([5, Lemma 11]). *For any T and $T' \in [T]$, $r_B(T')/r_B(T) = O(\mathsf{polylog}(n))$.*

Similar to what was observed for the lower bound, Theorem 19 implies an upper bound on the multiplicative difference between $r(T)$ and $r_B(T)$, as well.

▶ **Corollary 20** ([5, Corollary 12]). *For any string T, $\max\{r(T)/r_B(T), r_B(T)/r(T)\} = O(\mathsf{polylog}(n))$.*

5.5 Rotation and BBWT

As seen in Theorem 18, rotating the string can in some cases improve the compression performance of RLBBWT by a $\Theta(\log n)$ factor. Such a rotation can be encoded as an extra integer ($\log n$ bits). Recall that in order to recover the string, BWT required extra information (an integer of $\log n$ bits[16]) to specify which rotation was the original string. Since the BWT is equivalent to BBWT for a specific rotation $\mathsf{minrot}(T)$ of T (Proposition 3), it is always possible to find an encoding of a string using RLBBWT preceded by a rotation, that uses at most the space required for RLBWT (and the extra information). It is known that $\mathsf{minrot}(T)$ can be computed in linear time [69]. Badkobeh et al. raise the question of whether the optimal rotation, i.e., $\arg\min_{0\leq i<n} \rho(\mathsf{BBWT}(\mathsf{rot}^i(T)))$ can be computed in sub-quadratic time, and give partial results towards this goal [2].

Badkobeh et al. further consider a compression scheme where multiple rotations and BBWT operations can be used. They give the following conjecture for strings with the same Parikh vector, where the Parikh vector of a string T is a vector of the length of the alphabet storing the frequencies of the characters in T.

▶ **Conjecture 21** ([2, Conjecture 1]). *Given two strings with the same Parikh vector, it is possible to transform one to the other by using only rotation and BBWT operations.*

If the conjecture is true, this implies that any string can be represented, for example, by its Parikh vector and a sequence of integers alternately representing rotations or the number of times BBWT should be applied to obtain the string.

They show that the conjecture holds for the case where the alphabet is binary, or, when each symbol occurs only once.

▶ **Theorem 22** ([2, Theorem 4]). *Given two strings of the same length with the same Parikh vector, it is possible to transform one to the other by using only rotation and BBWT operations if all symbols are distinct, or if the alphabet is binary.*

[16] Here, we do not consider the case of adding an explicit terminal symbol – using an extra explicit terminal symbol outside the alphabet may increase the encoding of each symbol by 1 bit (e.g. when $\sigma = 2^k$ for some k), which would then require an extra $n + \log(\sigma + 1)$ bits.

The theorem is proved by showing that, under the assumption of the alphabet, a string that is not the lexicographically smallest string can always be transformed, using a combination of rotations and an inverse BBWT operation, into a lexicographically smaller string. The theorem then follows from the bijectivity of rotations and BBWT.

5.6 Clustering effects

As mentioned at the beginning of Section 5, BWT is known to have a property which tends to group characters together. Other than by entropy arguments, this so-called *clustering effect* has also been analyzed with respect to the run-length encoding.

The binary strings which are clustered completely by BWT have been completely characterized by Mantaci et al. [56]:

▶ **Theorem 23** ([56, Corollary 11]). $\mathsf{BWT}(w) = 2$ *if and only if* w *is the power of a rotation of a standard Sturmian word.*

A similar result was shown for the BBWT in [12]:

▶ **Theorem 24** ([12, Theorem 2]). $\mathsf{BBWT}(T) = 2$ *if and only if* $T = \mathtt{b}^k\mathtt{a}^\ell$ *for some* $k, \ell \geq 1$, *or* T *is a power of the Lyndon rotation of a standard word.*

In the first case, $\mathsf{BBWT}(T) = \mathtt{a}^\ell\mathtt{b}^k$, while in the second case, $\mathsf{BBWT}(T) = \mathtt{b}^k\mathtt{a}^\ell$, such that T is the mth power of the Lyndon rotation of a standard word, where $m = \gcd(k, \ell)$.

Note that Fibonacci words (a subset of standard Sturmian words) have many runs: $\rho(w) = \Theta(n)$. This implies that in some cases, the BWT is able to transform, with respect to run-length encoding, a least compressible string to a most compressible string. The same holds for BBWT, again due to Proposition 3.

It is important to note that the BWT does not always make the string more compressible[17], and there exist strings T for which $r(T) > \rho(T)$. For example, $\mathsf{BWT}(\mathtt{aabb}) = \mathtt{baba}$. However, Mantaci et al. [55] showed that $r(T)$ can never be more than twice the number of runs of T:

▶ **Theorem 25** ([55, Theorem 3.3]). *For any string* T, $r(T) = \rho(\mathsf{BWT}(T)) \leq 2\rho(T)$.

Badkobeh et al. [2] showed that the result can be further generalized for BBWT.

▶ **Theorem 26** ([5, Theorem 14]). *For any string* T, $r_B(T) = \rho(\mathsf{BBWT}(T)) \leq 2\rho(T)$.

These results indicate that, with respect to run-length encoding, BWT and BBWT transform the string in an asymmetric way; while it is possible sometimes to make the string drastically compressible (Theorems 23 and 24), it will not make it much less compressible than it currently is (Theorems 25 and 26).

The examples of Theorems 23 and 24 are in some sense remarkable with respect to run-length encoding. However, the compressibility of the string with respect to dictionary compression does not change: the example only shows a compressible string (with respect to r and r_B) being transformed into a compressible string.

Bannai et al. [5] analyze how BWT and BBWT change the string with respect to other repetitiveness measures. For both BWT and BBWT, it is proved that the transforms will not increase the repetitiveness measure by more than a poly-logarithmic factor.

[17] In fact, there can exist no such lossless compression algorithm [49].

▶ **Theorem 27** ([5, Theorem 15]). $\mathfrak{m}(\mathsf{BWT}(T)) = O(\mathfrak{m}(T)\mathsf{polylog}(n))$, $\mathfrak{m}(\mathsf{BBWT}(T)) = O(\mathfrak{m}(T)\mathsf{polylog}(n))$ *hold for any repetitiveness measure* $\mathfrak{m}$ *such that* $\mathfrak{m}(T) = O(\rho(T))$ *and* $\mathfrak{m}(T) = \Omega(\delta(T))$.

The proof is based on the simple observation that since $\mathfrak{m}(T) = O(\rho(T))$, $\mathfrak{m}(\mathsf{BWT}(T)) = O(\rho(\mathsf{BWT}(T)) = O(r(T)) = O(\delta(T)\log^2 n) = O(\mathfrak{m}(T)\log^2 n)$. A poly-logarithmic bound for BBWT can be shown as well, with the use of Corollary 20. Thus, $\mathfrak{m}(\mathsf{BWT}(T))/\mathfrak{m}(T) = O(\mathsf{polylog}(n))$ and $\mathfrak{m}(\mathsf{BBWT}(T))/\mathfrak{m}(T) = O(\mathsf{polylog}(n))$.

Furthermore, it is shown that there is an infinite family of strings such that the BBWT transforms a maximally incompressible string, with respect to any measure lower-bounded by δ, to a very compressible string.

▶ **Theorem 28** ([5, Theorem 16]). *There exists an infinite family of strings such that* $\delta(T) = \Omega(n/\log n)$ *and* $\mathfrak{m}(\mathsf{BBWT}(T)) = O(1)$ *for any* $\mathfrak{m}$ *s.t.* $\mathfrak{m}(T) = O(r(T))$ *for all* T*, and* $\mathfrak{m}(\mathsf{BBWT}(T)) = O(\log n)$ *for any* $\mathfrak{m}$ *s.t.* $\mathfrak{m}(T) = O(g(T))$ *for all* T*, where* g *is the size of the smallest grammar compressed representation for* T.

The family of strings identified is again the Fibonacci words, which are very compressible, but here used as images of BBWT, i.e., T is such that $\mathsf{BBWT}(T) = F_k$ for some k. It is proved that the string T obtained by applying the inverse BBWT to Fibonacci words are not compressible by dictionary compression. Thus, BBWT transforms strings in an asymmetric way with respect to other repetitiveness measures as well. An interesting implication of this result is that in some cases, by applying BBWT in a pre-processing step before applying a dictionary compression, it is possible to transcend any dictionary compressor. It is not yet known whether analogous strings exist for BWT; the same construction using inverse BWT does not work since Fibonacci words are not always BWT images.

The proof of Theorem 28 is based on an elegant characterization of the LF mapping (or π_{F_k}) on the Fibonacci word F_k. It utilizes a bit string representation of integers based on the fact (re)discovered[18] by Zeckendorf [72], that any integer can be uniquely represented as the sum of a set of distinct and non-consecutive Fibonacci numbers. It is shown that if a position i in F_k is represented as a k-bit string s_i based on the Zeckendorf representation (for $j \in [0, k)$, the jth bit is 1, if and only if the $(j+1)$st Fibonacci number is used), the bit-string corresponding to the position $\pi_{F_k}(i)$ is a rotation of s_i. This implies that each conjugacy class of k-bit strings for positions in $[0, |F_k|)$ correspond to a cycle in the standard permutation π_{F_k} of F_k. Then, by showing that the length of cyclic strings corresponding to the cycles of π_{F_k} can be bounded by k, and, when k is prime, that the Lyndon rotation of each of the strings occur uniquely in T, it follows that $\delta(T) \geq (|F_k| - 2k + 1)/2k = \Omega(n/\log n)$.

6 A class of bijective BWTs

In some way, it would be more natural to define the BBWT as *a* bijective Burrows–Wheeler transform in the sense that it is not the only variant that is an isomorphism on Σ^n that sorts rotations of substrings of the input by the inverse of the Gessel-Reutenauer transform. In fact, we can exchange the Lyndon factorization with any factorization such that

- it is uniquely defined,
- it produces a set of primitive factors, and
- there is a unique way of restoring the text from this set of primitive strings.

[18] See `https://proofwiki.org/wiki/Zeckendorf%27s_Theorem/Historical_Note`.

The Lyndon factorization is not the only factorization that has these properties. Daykin et al. [23] introduced the term *unique maximal factorization family* (UMFF) to address a factorization that is unique and greedy in the choice of selecting the next factor. The Lyndon factorization is an example for an UMFF, but not the only one. In [25] and [24] a bijective variant of the BWT was proposed, based on the so-called V-order and B-order, respectively. For both variants, the authors gave a semantic definition similar to Lyndon words, called *V-words* and *B-words*, by switching the lexicographical order with the V-order or the B-order, respectively. Both approaches apply a factorization similar to the Lyndon factorization and sort the conjugates in the respective order. The authors give linear time constructions for both approaches based on SAIS and show how to recover the original text.

Hendrian et al. [38] proposed a bijective BWT variant based on the Galois factorization [65]. An extension to the Galois factorization is the generalized Lyndon factorization [26], which applies a generalized lexicographic order, using a different order of the characters for each text position to compare. Similarly, we can create a bijective variant of the BWT that is based on the Nyldon factorization [19] or the canonical inverse Lyndon factorization. For the latter, in [13] so-called anti-Lyndon words and inverse Lyndon words were proposed. Anti-Lyndon words are Lyndon words for the inverse order of the characters. An inverse Lyndon word is the lexicographically largest among all its suffixes [13, Definition 3.1]. (Inverse Lyndon words can also be defined on properties based on their rotations [13, Proposition 3.1], which we here skip since these properties need further definitions.) In particular, an inverse Lyndon word can have a border, while an anti-Lyndon word cannot. A class of factorizations were studied in [13], which factorize a string into inverse Lyndon words. The authors also gave one canonical factorization that uniquely creates inverse Lyndon words, where the inverse Lyndon factors $F_1, \ldots, F_m$ are in lexicographically increasing order, i.e., $F_1 \prec_{lex} \ldots \prec_{lex} F_m$. However, an obstacle to define a bijective BWT variant based on inverse Lyndon factors is that a inverse Lyndon word is not necessarily primitive. It would be interesting to study whether any type of these factorizations can form a bijective variant of the BWT which can be used for pattern matching.

7 Conclusion

We presented the BBWT, a bijective variant of the BWT which is mathematically elegant in the sense that it is designed to be invertible without requiring additional information such as end-of-string markers or marking a dedicated position in the BWT. We have surveyed the following topics.

Construction. We studied the history of construction algorithms, starting with $O(n \log n)$ time, leading to current $O(n)$ time construction algorithms. It is also possible to construct the BBWT in-place in $O(n^2)$ time. All approaches have in common that they adapt algorithms for suffix array or BWT construction.

Indexing and pattern matching. The BBWT can be used for text indexing, similar to BWT-based or eBWT-based FM-indexing. For that, a slight modification of the FM-index suffices, which takes individual measures whenever the search interval matches a prefix of a Lyndon factor of the pattern with the starting position of a Lyndon factor of the text. This additional handling can slow down the computation by requiring up to a multiplicative factor of $O(\log m)$ additional queries to a rank/select data structure built on the BBWT, where m is the pattern length.

Compression performance. Like the BWT, the BBWT tends to group similar characters together, making it effective for run-length encoding. However, this grouping can be affected by rotating the input text and by the number of Lyndon factors of the input, which serves as a lower bound on the number of character runs. We draw connections to dictionary compression techniques like LZ77 and macro schemes and give pointers to studies that show that the BBWT can achieve comparable or better compression ratios than the BWT in specific cases.

Open problems

- Can we construct the BBWT in $O(n/\log_\sigma n)$ time? It is possible to do so for constructing the BWT [42] and the Lyndon array [4], from which we can extract the Lyndon factorization.
- Can we match a pattern of length m with the techniques of Theorem 10 with $O(m)$ rank operations? The multiplicative term $\log m$ seems too pessimistic: we are confident that a refined analysis could improve the time complexity.
- Reachability: Can we express any primitive string T by its Parikh vector p and a sequence of BBWT applications and rotations such that applying this sequence of transforms on the lexicographically smallest string having p as its Parikh vector leads to T? Given the Parikh vector is $\vec{x} = (x_1, \ldots, x_\sigma)$ for an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$, the lexicographically smallest string having $\vec{x}$ as its Parikh vector is $S = c_1^{x_1} \cdots c_\sigma^{x_\sigma}$. Cyclic rotations and the BBWT are bijective transformations, which we can apply in an arbitrary amount and order sequentially, and it seems that there is always such a sequence of transforms that turn S into T, cf. Conjecture 21.
- Can we find the cyclic rotation of any primitive T that leads to the minimum number of runs in the BBWT in $o(n^2)$ time? (Cf. Section 5.5)
- Can we compute the smallest integer $x \geq 1$ such that $\mathsf{BBWT}^{(x)}(T) = T$ in $o(nx)$ time for any string T of length n?
- How far can GCA and SA differ, for instance with respect to Hamming distance? This problem is relevant for the analysis of the worst-case complexity of the (practical) algorithm computing GCA from SA by shifting.

References

1 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023. doi:10.1016/J.IC.2022.104999.

2 Golnaz Badkobeh, Hideo Bannai, and Dominik Köppl. Bijective BWT based compression schemes. In Zsuzsanna Lipták, Edleno Silva de Moura, Karina Figueroa, and Ricardo Baeza-Yates, editors, *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 16–25. Springer, 2024. doi:10.1007/978-3-031-72200-4_2.

3 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.CPM.2016.23.

4 Hideo Bannai and Jonas Ellert. Lyndon arrays in sublinear time. In *Proc. ESA*, volume 274 of *LIPIcs*, pages 14:1–14:16, 2023. doi:10.4230/LIPIcs.ESA.2023.14.

5 Hideo Bannai, Tomohiro I, and Yuto Nakashima. On the compressiveness of the Burrows-Wheeler transform. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium*

on Combinatorial Pattern Matching, CPM 2025, June 17-19, 2025, Milan, Italy, volume 331 of *LIPIcs*, pages 17:1–17:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.CPM.2025.17.

6 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 17:1–17:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.17.

7 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the bijective and the extended Burrows-Wheeler Transform in linear time. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.7.

8 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and indexing the bijective and extended Burrows-Wheeler transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.

9 Timothy C. Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989. doi:10.1145/76894.76896.

10 Nico Bertram, Jonas Ellert, and Johannes Fischer. Lyndon words accelerate suffix sorting. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ESA.2021.15.

11 Elena Biagi, Davide Cenzato, Zsuzsanna Lipták, and Giuseppe Romana. On the number of equal-letter runs of the Bijective Burrows-Wheeler Transform. In Giuseppa Castiglione and Marinella Sciortino, editors, *Proceedings of the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13-15, 2023*, volume 3587 of *CEUR Workshop Proceedings*, pages 129–142. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3587/4564.pdf.

12 Elena Biagi, Davide Cenzato, Zsuzsanna Lipták, and Giuseppe Romana. On the number of equal-letter runs of the Bijective Burrows-Wheeler Transform. *Theoretical Computer Science*, page 115004, 2024. doi:10.1016/j.tcs.2024.115004.

13 Paola Bonizzoni, Clelia De Felice, Rocco Zaccagnino, and Rosalba Zizza. Inverse Lyndon words and inverse Lyndon factorizations of words. *Adv. Appl. Math.*, 101:281–319, 2018. doi:10.1016/j.aam.2018.08.005.

14 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014. doi:10.1142/S0129054114400309.

15 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.

16 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.

17 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, Systems Reseach Center, 1994. SRC Research Report 124.

18 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. In Paolo Ferragina, Travis Gagie, and Gonzalo Navarro, editors,

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday, volume 131 of *OASIcs*, pages 3:1–3:29, 2025.

19 Émilie Charlier, Manon Philibert, and Manon Stipulanti. Nyldon words. *J. Comb. Theory, Ser. A*, 167:60–90, 2019. doi:10.1016/j.jcta.2019.04.002.

20 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958. URL: http://www.jstor.org/stable/1970044.

21 Maxime Crochemore, Jacques Désarménien, and Dominique Perrin. A note on the Burrows-Wheeler transformation. *Theor. Comput. Sci.*, 332(1-3):567–572, 2005. doi:10.1016/J.TCS.2004.11.014.

22 Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the Burrows–Wheeler transform in place and in small space. *J. Discrete Algorithms*, 32:44–52, 2015. doi:10.1016/j.jda.2015.01.004.

23 David E. Daykin, Jacqueline W. Daykin, and William F. Smyth. Combinatorics of unique maximal factorization families (umffs). *Fundam. Informaticae*, 97(3):295–309, 2009. doi:10.3233/FI-2009-202.

24 Jacqueline W. Daykin, Richard Groult, Yannick Guesnet, Thierry Lecroq, Arnaud Lefebvre, Martine Léonard, and Élise Prieur-Gaston. Binary block order rouen transform. *Theor. Comput. Sci.*, 656:118–134, 2016. doi:10.1016/J.TCS.2016.05.028.

25 Jacqueline W. Daykin and William F. Smyth. A bijective variant of the Burrows-Wheeler Transform using V-order. *Theor. Comput. Sci.*, 531:77–89, 2014. doi:10.1016/J.TCS.2014.03.014.

26 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019. doi:10.1016/j.tcs.2018.12.015.

27 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. Some variations on Lyndon words (invited talk). In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 2:1–2:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.2.

28 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

29 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

30 Frantisek Franek, Michael Liut, and William F. Smyth. On Baier's sort of maximal Lyndon substrings. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2018, Prague, Czech Republic, August 27-28, 2018*, pages 63–78. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2018. URL: http://www.stringology.org/event/2018/p07.html.

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

32 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012. arXiv:1201.3077.

33 Sara Giuliani. *Sensitivity of the Burrows-Wheeler Transform to Small Modifications, and Other Problems on String Compressors in Bioinformatics*. PhD thesis, University of Verona, 2023.

34 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello. Novel results on the number of runs of the Burrows–Wheeler-transform. In *Proc. SOFSEM*, volume 12607 of *LNCS*, pages 249–262, 2021. doi:10.1007/978-3-030-67731-2_18.

35 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit catastrophes for the Burrows-Wheeler Transform. In Frank Drewes and Mikhail Volkov, editors, *Developments in Language Theory - 27th International Conference,*

DLT 2023, Umeå, Sweden, June 12-16, 2023, Proceedings, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023. doi:10.1007/978-3-031-33264-7_8.

36 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit Catastrophes for the Burrows-Wheeler Transform. *Theory of Computing Systems*, 2025. to appear.

37 Sara Giuliani, Zsuzsanna Lipták, Francesco Masillo, and Romeo Rizzi. When a dollar makes a BWT. *Theor. Comput. Sci.*, 857:123–146, 2021. doi:10.1016/J.TCS.2021.01.008.

38 Diptarama Hendrian, Dominik Köppl, Ryo Yoshinaka, and Ayumi Shinohara. Algorithms for Galois words: Detection, factorization, and rotation. In *Proc. CPM*, volume 296 of *LIPIcs*, pages 18:1–18:16, 2024. doi:10.4230/LIPICS.CPM.2024.18.

39 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/J.TCS.2016.03.005.

40 Hyodam Jeon and Dominik Köppl. Compression sensitivity of the Burrows–Wheeler transform and its bijective variant. *Mathematics*, 13(7)(1070):1–46, March 2025. doi:10.3390/math13071070.

41 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

42 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. STOC*, pages 756–767, 2019. doi:10.1145/3313276.3316368.

43 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Commun. ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.

44 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

45 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Trans. Inf. Theory*, 69(4):2074–2092, 2023. doi:10.1109/TIT.2022.3224382.

46 Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara. In-place bijective Burrows-Wheeler Transforms. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CPM.2020.21.

47 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler Transform. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 65–79. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009. URL: http://www.stringology.org/event/2009/p07.html.

48 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.

49 Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition*. Texts in Computer Science. Springer, 2008. doi:10.1007/978-0-387-49820-1.

50 Konstantin M. Likhomanov and Arseny M. Shur. Two combinatorial criteria for BWT images. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *Computer Science - Theory and Applications - 6th International Computer Science Symposium in Russia, CSR 2011, St. Petersburg, Russia, June 14-18, 2011. Proceedings*, volume 6651 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2011. doi:10.1007/978-3-642-20712-9_30.

51 M. Lothaire. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, 2 edition, 1997.

52 M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

53 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

54 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

55 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017. doi:10.1016/J.TCS.2017.07.015.

56 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.

57 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.

58 Guy Melançon. Lyndon words and singular factors of Sturmian words. *Theor. Comput. Sci.*, 218(1):41–59, 1999. doi:10.1016/S0304-3975(98)00249-7.

59 J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996. doi:10.1016/0304-3975(95)00225-1.

60 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

61 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

62 Gonzalo Navarro, Carlos Ochoa, and Nicola Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021. doi:10.1109/TIT.2020.3042746.

63 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.

64 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. Generic non-recursive suffix array construction. *ACM Trans. Algorithms*, 20(2):18, 2024. doi:10.1145/3641854.

65 Christophe Reutenauer. Mots de Lyndon généralisés. *Séminaire Lotharingien de Combinatoire*, 54(B54h):1–16, 2005.

66 David A. Scott. A truely BIJECTIVE BWT is here! https://groups.google.com/g/comp.compression/c/SDTLJypCWvc/m/ElbLTWJbnH8J, December 2007. Usenet thread, accessed Mar. 11, 2025.

67 David A. Scott. BIJECTIVE BWTS PAPER. https://groups.google.com/g/comp.compression/c/IN4XwmstgHk/m/X1RHMF2lyTgJ, September 2009. Usenet thread, accessed Mar. 11, 2025.

68 David A. Scott. The paper "A Bijective String Sorting Transform". https://groups.google.com/g/comp.compression/c/hmMqcFOlCn8/m/AUQ7jasIiaoJ, July 2009. Usenet thread, accessed Mar. 11, 2025.

69 Yossi Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. doi:10.1016/0196-6774(81)90013-4.

70 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

71 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On the size of overlapping Lempel-Ziv and Lyndon factorizations. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 29:1–29:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.29.

72 Edouard Zeckendorf. Representations des nombres naturels par une somme de nombres de Fibonacci on de nombres de Lucas. *Bulletin de La Society Royale des Sciences de Liege*, pages 179–182, 1972. URL: https://cir.nii.ac.jp/crid/1570009749187075840.