The Burrows-Wheeler Transform of an Elastic-Degenerate String

Lapo Cioni[1,*,†], Veronica Guerrini[1,*,†] and Giovanna Rosone[1,*,†]

[1]*Department of Computer Science, University of Pisa, Italy*

Abstract

Degenerate strings (DS) and elastic degenerate strings (EDS) are a way to represent, in a compact form, strings that contain a high degree of similarity. They can be particularly useful in some fields, such as text processing or the study of DNA mutations in computational biology, where it is necessary to efficiently manage several variations of a sequence. In practice, a degenerate string is a string whose symbols, called *degenerate*, can have several alternatives (hence a degenerate symbol is a set). In the literature different constraints have been imposed on degenerate string symbols. For example, the symbol can only be *i)* a set of letters of the alphabet, *ii)* a set of strings of the same length, or *iii)* a set of strings of variable length (including the empty string). We consider the latter in its most general form, which is known as *elastic degenerate strings*. Our contribution is the introduction of the Burrows-Wheeler transform of an elastic-degenerate string (EDS-BWT). We show that EDS-BWT is reversible and that it can be used to solve the pattern matching problem, i.e., the problem of finding a standard string pattern within an EDS, by adapting the inner properties of the classical Burrows-Wheeler transform. Finally, we implemented the EDS-BWT encoding/decoding and the prototype `edsBWTSearch` to experimentally compare our pattern matching approach to other existing tools managing elastic degenerate strings.

Keywords

Burrows-Wheeler Transform, elastic degenerate strings, backward search, pattern matching problem

1. Introduction

The Burrows–Wheeler transform (BWT) was introduced in [1] as a method for compressing a single string, and then, it was shown to be effective in many other areas, with applications spanning beyond its original purpose [2]. For instance, it has been successfully used for compact text indexing [3, 4, 5, 6] and for bioinformatics applications, *e.g.*, for sequence alignment [7], phylogenetic analysis [8], genome assembly [9] as well as for sequencing data compression [10].

Roughly, the BWT performs a permutation of the letters of an input string T. First, the cyclic rotations of T are sorted in lexicographic order, and then the $\mathrm{bwt}(T)$ is obtained by concatenating the last letters of the (sorted) cyclic rotations of T. The BWT can also be defined by sorting the suffixes of $T\$$ [1], where $\$$ is an end-marker symbol that does not appear in T itself and it is lexicographically smaller than any of the symbols in T.

Shifting the focus from a string to a collection of strings, the BWT of a string collection can be

ICTCS'24: Italian Conference on Theoretical Computer Science, September 11–13, 2024, Torino, Italy

*Corresponding author.

†These authors contributed equally.

lapo.cioni@di.unipi.it (L. Cioni); veronica.guerrini@unipi.it (V. Guerrini); giovanna.rosone@unipi.it (G. Rosone)

0000-0002-4605-8473 (L. Cioni); 0000-0001-8888-9243 (V. Guerrini); 0000-0001-5075-1214 (G. Rosone)

defined analogously. That is, either by sorting the cyclic rotations of all strings in the collection[1], as in [11], or by sorting their suffixes, as in [12]. In the latter case, a distinct end-marker symbol is appended to each string, giving an ordered collection of strings. The BWT and its extension to a collection of strings allow to define a map from symbols occurring in the transformed string L to the string F of lexicographically sorted symbols of L. This mapping is known as *LF-mapping*, and it allows both the reversibility of the transform and the search for patterns (called *backward search*). The LF-mapping and the backward search are the key machinery in the FM-index [13].

Degenerate strings (DS) and elastic degenerate strings (EDS) have been introduced as a way to efficiently represent sequences that contain a high degree of similarity. For instance, in bioinformatics, they are used to represent pangenomes, which are collections of closely-related genomic sequences that one needs to analyze together [14].

A degenerate string (also known as indeterminate string) over an alphabet Σ is a sequence $D = Y_1 \cdots Y_k$ where each Y_i is a subset of Σ. It represents any string that can be obtained by selecting one letter from each subset from left to right. In 2017, Iliopoulos et al. [15] defined a more general notion of degenerate strings: an *elastic-degenerate string* (EDS) over Σ is a sequence $\mathcal{D} = X_1 \cdots X_k$ of non-empty subsets X_i of strings over Σ, where each X_i is called *degenerate symbol*. If instead each X_i contains strings of the same length, $\mathcal{D}$ is called *general degenerate string* [16].

Here is an example of an EDS, which we use throughout this paper:

$$\mathcal{D} = \left\{ \text{ATTGCT} \right\} \left\{ \begin{matrix} CTA \\ TA \\ A \end{matrix} \right\} \left\{ \text{CTACGGACT} \right\} \left\{ \begin{matrix} \text{A} \\ \epsilon \end{matrix} \right\} \left\{ \text{CTGT} \right\} \tag{1}$$

We remark that Eq. (1) is a compact way to represent all the strings that are obtained by taking an element from each set and concatenating them in order.

Elastic-degenerate strings and their variants have been much studied in the literature in recent years, mainly for the pattern matching problem, which consists of finding the occurrences of a pattern in an ED string [17, 16, 18, 19, 20]. For instance, the pattern $TTACT$ occurs in $\mathcal{D}$, across the first three degenerate symbols. A variety of methods and data structures have been used for the pattern matching problem on very similar strings. The following partial list gives a few examples [21, 22, 23, 24, 25, 26, 27] (see also references therein).

1.1. Our contribution

In this paper, we introduce the Burrows-Wheeler transform of an elastic degenerate string (EDS-BWT), which applies the EBWT to a sequence of ordered collections of strings of any length, and show that this transform is reversible. The core of our method is a mix between the classical BWT [1] and the EBWT [12], together with a mapping between strings belonging to consecutive degenerate symbols. We also present an algorithm for solving the pattern matching problem on an elastic degenerate string. Specifically, we are able to return the number of starting positions of pattern occurrences and, for each pattern occurrence, the index of the degenerate symbol and the string position at which the occurrence starts. For instance, searching pattern

[1] In this case, one needs to use the ω-order defined in [11].

$P = TTAC$ in the string $\mathcal{D}$ in Eq. (1), we return that there is at least one occurrence that starts in the first degenerate symbol at the last letter of the only string in it.

To the best of our knowledge, no other work does it, although some authors introduced the BWT for degenerate strings [27] and for closely-related sequences [28, 29, 5, 30, 31, 32] (see also references therein).

2. Background and Notation

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet Σ with $c_1 < c_2 < \ldots < c_\sigma$, where $<$ denotes the standard lexicographic order, and let ϵ be the empty string. Let S be a string (or word) of length n on Σ and $S[i]$ its i-th letter (or symbol). A *substring* $S[i, j]$ of S coincides with $S[i] \cdot S[i + 1] \cdots S[j]$, where $\cdot$ is the concatenation operator. For any $1 \leq j \leq n$, the substring $S[1, j]$ is called a *prefix* of S and $S[j, n]$ a *suffix* of S.

The Burrows-Wheeler Transform (BWT) [1]. The BWT is a well-known reversible transformation defined on a string S that permutes its letters. By appending an end-marker symbol \$ to S and by sorting all the suffixes of $S\$$ in lexicographic order, the output of the BWT is a string $\text{bwt}(S)$ of length $n + 1$ obtained by concatenating the letters (circularly) preceding each suffix in the list of sorted suffixes. More precisely, for each i, $\text{bwt}(S)[i]$ is the letter preceding the i-th lexicographically smallest suffix of the string $S\$$, except for the suffix $S\$$, where the preceding letter is set to be \$.

The BWT of a string collection [11, 12]. The BWT extended to a string collection $\mathcal{S}$, known as EBWT, is a reversible transformation that produces a string $\text{ebwt}(\mathcal{S})$ that is a permutation of all the symbols of all strings in $\mathcal{S}$.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be a collection of k strings on the alphabet Σ. We append to each string $S_i \in \mathcal{S}$ a different end-marker symbol $\$_i$, not belonging to Σ and lexicographically smaller than any other symbol in Σ, by setting $\$_i < \$_j$ for each $i < j$[2]. That is, if S_i has length n_i, we define $S_i[n_i + 1] = \$_i$. In the following, we will always use S_i to refer to a string of length $n_i + 1$ terminating with the end-marker symbol $\$_i$. Finally, let $N = \sum_{i=1}^{k} n_i + k$ be the number of letters of all strings in $\mathcal{S}$ (including their end-marker symbol).

Note that, after appending a distinct end-marker symbol to each string in $\mathcal{S}$, the collection $\mathcal{S}$ becomes an ordered collection. Exclusively for implementation purposes, a unique end-marker symbol \$ is used for all strings in $\mathcal{S}$, even if each \$ implicitly carries the index of the string to which it was appended.

LF-mapping. Let L be the string $\text{bwt}(S)$ and F the string obtained by sorting all the symbols of S lexicographically. The reversibility of the BWT (as well as of the EBWT) is based on the following two properties stated in [1]:

- For all $i = 1, \ldots, n + 1$, the symbol $F[i]$ circularly follows the symbol $L[i]$ in the string S;
- For each symbol $c \in \Sigma$, the j-th occurrence of c in L corresponds to the j-th occurrence of c in F.

[2]We note that in [11] the EBWT is defined without appending end-marker symbols to the strings: cyclic rotations of the strings in $\mathcal{S}$ are sorted by means of an order relation, called ω-order, on infinite strings.

From the second property, it follows that given a position i in L such that $L[i] = c$, the position in F corresponding to that occurrence of c is given by $C[L[i]] + \texttt{rank}_L(i, L[i])$, where C is an array storing for any $c \in \Sigma$ the total number of symbols in S that are smaller than c, and $\texttt{rank}_L(i, L[i])$ is the number of occurrences of $L[i]$ in the prefix $L[1, i]$. This mapping gives a correspondence between symbol occurrences in L and symbol occurrences in F, and is known as *LF-mapping* [3, 13].

Bit vectors. A string of zeros and ones is called *bitvector*. Given a bitvector b of length n, the operators $\texttt{rank}_b$ and $\texttt{select}_b$ are defined as follows, for any $1 \leq i \leq n$ and $c \in \{0, 1\}$:

$$\texttt{rank}_b(i, c) = |\{j \mid 1 \leq j \leq i \text{ and } b[j] = c\}|$$
$$\texttt{select}_b(i, c) = j, \text{ with } b[j] = c \text{ and } \texttt{rank}_b(j, c) = i, \text{ if such } j \text{ exists.}$$

In other words, $\texttt{rank}_b(i, c)$ gives the number of occurrences of the bit c in the prefix $b[1, i]$, while $\texttt{select}_b(i, c)$ returns the index of the i-th occurrence of c in b (if it exists). A bitvector b can be preprocessed in order to support $\texttt{rank}_b$ and $\texttt{select}_b$ queries in constant time [33].

Elastic Degenerate string. An *elastic degenerate string* (see [15, 17]) (or ED string) is a sequence of k finite nonempty sets of strings (including ϵ) $X_1 \cdots X_k$ of combined total length $\overline{N}$. Each X_i is called *degenerate symbol*.

Given an elastic degenerate string $\mathcal{D} = X_1 \cdots X_k$, for each $i = 1, \ldots, k$, we denote the strings in X_i by $w_{i,1}, \ldots, w_{i,\ell_i}$, where $|X_i| = \ell_i \geq 1$.

Essentially, an elastic degenerate string represents all possible strings that can be constructed by taking an element from each degenerate symbol and concatenating them. For example, $\{AT, C\}\{C, G\}\{TC\}$ represent the strings $ATCTC$, $ATGTC$, $CCTC$, $CGTC$.

3. The BWT of an elastic degenerate string

In this section we define EDS-BWT, which is the BWT of an elastic degenerate string $\mathcal{D} = X_1 \cdots X_k$, where $X_i = \{w_{i,1}, \ldots, w_{i,\ell_i}\}$, $|X_i| = \ell_i \geq 1$ and $\ell = \sum_{i=1}^{k} \ell_i$. We need to append to each $w_{i,j}$ an end-marker symbol $\$_r \notin \Sigma$, with $r = j + \sum_{s=1}^{i-1} \ell_s$. In this way, denoting $w_{i,j}\$_r$ by S_r, we obtain a single ordered collection (of strings) $\mathcal{S} = \{S_1, S_2, \ldots, S_{\ell_1}, S_{\ell_1+1}, \ldots, S_{\ell_1+\ell_2}, \ldots, S_\ell\}$. Roughly, we are appending an end-marker symbol at the end of each string of each degenerate symbol, proceeding left-to-right among the symbols and giving an arbitrary order to the strings within the degenerate symbols. Note that when $w_{i,j} = \epsilon$, the resulting string is $\$_r$, with r as above.

Definition 3.1. Given an elastic degenerate string $\mathcal{D} = X_1 \cdots X_k$ the Burrows-Wheeler transform of $\mathcal{D}$ (called EDS-BWT) is defined as the pair $\text{edsbwt}(\mathcal{D}) = (\text{ebwt}(\mathcal{S}), \mathcal{L}_{ED})$ where $\mathcal{S}$ is the ordered collection $\mathcal{S} = \{S_1, \ldots, S_\ell\}$ and $\mathcal{L}_{ED}$ is defined as follows:
For $q = 1, \ldots, \ell$, let p be the position of $\$_q$ in $\text{ebwt}(\mathcal{S})$, and let the associated string S_q belong to the degenerate symbol X_t, for some $1 \leq t \leq k$.
Then, if $t > 1$, we define $\mathcal{L}_{ED}(p)$ to be the interval of positions $[b, e]$ in $\text{ebwt}(\mathcal{S})$ such that $X_{t-1} = \{S_b, \ldots, S_e\}$. Otherwise, $\mathcal{L}_{ED}(p)$ circularly gives the interval of positions $[b, e]$ such that $X_k = \{S_b, \ldots, S_e\}$.

row	$\mathcal{L}_{ED}$	ebwt($\mathcal{S}$)	sorted suffixes
1		T	$\$_1$
2		A	$\$_2$
3		A	$\$_3$
4		A	$\$_4$
5		T	$\$_5$
6		A	$\$_6$
7	$[5,5]$	$\$_7$	$\$_7$
8		T	$\$_8$
9		T	A$\$_2$
10		T	A$\$_3$
11	$[1,1]$	$\$_4$	A$\$_4$
12	$[5,5]$	$\$_6$	A$\$_6$
13		T	ACGGACT$\$_5$
14		G	ACT$\$_5$
15	$[8,8]$	$\$_1$	ATTGCT$\$_1$
16		A	CGGACT$\$_5$
17		G	CT$\$_1$
18		A	CT$\$_5$
19	$[1,1]$	$\$_2$	CTA$\$_2$
20	$[2,4]$	$\$_5$	CTACGGACT$\$_5$
21	$[6,7]$	$\$_8$	CTGT$\$_8$
22		G	GACT$\$_5$
23		T	GCT$\$_1$
24		C	GGACT$\$_5$
25		T	GT$\$_8$
26		C	T$\$_1$
27		C	T$\$_5$
28		G	T$\$_8$
29		C	TA$\$_2$
30	$[1,1]$	$\$_3$	TA$\$_3$
31		C	TACGGACT$\$_5$
32		T	TGCT$\$_1$
33		C	TGT$\$_8$
34		A	TTGCT$\$_1$

Table 1
The EDS-BWT of our running example Eq. (1). For clarity, we also represent the sorted list of the suffixes.

Positions in L	Positions in F
1	26
2	9
3	10
4	11
5	27
6	12
7	**[5,5]**
8	28
9	29
10	30
11	**[1,1]**
12	**[5,5]**
13	31
14	22
15	**[8,8]**
16	13
17	23
18	14
19	**[1,1]**
20	**[2,4]**
21	**[6,7]**
22	24
23	32
24	16
25	33
26	17
27	18
28	25
29	19
30	**[1,1]**
31	20
32	34
33	21
34	15

Table 2
LF_{ED}.

row	F	L
1	$\$_1$	T
2	$\$_2$	A
3	$\$_3$	A
4	$\$_4$	A
5	$\$_5$	T
6	$\$_6$	A
7	$\$_7$	$\$_7$
8	$\$_8$	T
9	A	T
10	A	T
11	A	$\$_4$
12	A	$\$_6$
13	A	T
14	A	G
15	A	$\$_1$
16	C	A
17	C	G
18	C	A
19	C	$\$_2$
20	C	$\$_5$
21	C	$\$_8$
22	G	G
23	G	T
24	G	C
25	G	T
26	T	C
27	T	C
28	T	G
29	T	C
30	T	$\$_3$
31	T	C
32	T	T
33	T	C
34	T	A

Table 3
LF-mapping and backward search for TAC.

The ED string of our running example Eq. (1) gives the ordered collection

$$\mathcal{S} = \{ATTGCT\$_1,\ CTA\$_2, TA\$_3, A\$_4,\ CTACGGACT\$_5,\ A\$_6, \epsilon\$_7,\ CTGT\$_8\}.$$

Remark 3.2. As in [12] for the computation of the EBWT, we can use the same end-marker for all strings, so that the size of the alphabet increases by just one. However, each end-marker has a different (implicit) index determined by the order of the strings in the collection $\mathcal{S}$. In this way, during the construction of ebwt($\mathcal{S}$), a list containing the position p in ebwt($\mathcal{S}$) of each $\$_q$ together with its index q can be returned.

Note that, in the definition of $\mathcal{L}_{ED}$, $b = (\ell_1 + \cdots + \ell_{t-2}) + 1$ and $e = \ell_1 + \cdots + \ell_{t-1}$ for each $t > 1$. Also, the next remark follows:

Remark 3.3. Let p and p' be two different positions in ebwt($\mathcal{S}$) such that ebwt$[p] = \$_q$ and ebwt$[p'] = \$_{q'}$. If the associated strings S_q and $S_{q'}$ belong to the same degenerate symbol X_t, then $\mathcal{L}_{ED}(p) = \mathcal{L}_{ED}(p')$ by definition.

Reversibility To show that the EDS-BWT is reversible, we describe how LF-mapping works for this new transform. Note that LF-mapping for the EDS-BWT is different from that for the classical EBWT. Indeed, in the EBWT, the strings are independent from each other. Conversely, in our EDS-BWT, the strings in $\mathcal{S}$ belonging to consecutive degenerate symbols need to be "linked" to reconstruct the elastic degenerate string $\mathcal{D}$.

Table 1 shows the transform of our running example. For instance, $\mathsf{edsbwt}(\mathcal{S})[20] = \$_5$, and its associated string S_5 belongs to X_3, so $\mathcal{L}_{ED}(20) = [2, 4]$, which is an interval of three positions. Thus the previous degenerate symbol X_2 is a set of three strings. Specifically, it is $X_2 = \{CTA, TA, T\}$.

The following proposition states the properties of the LF-mapping for an ED string. The proof follows immediately from the original LF-mapping properties and the definition of $\mathcal{L}_{ED}$.

Proposition 3.1. *Let $\mathcal{S}$ be the collection of strings associated to an ED string $\mathcal{D}$ and let $\mathsf{edsbwt}(\mathcal{D}) = (\mathsf{ebwt}(\mathcal{S}), \mathcal{L}_{ED})$. Let $L = \mathsf{ebwt}(\mathcal{S})$ and F be the sequence of the lexicographically sorted letters of L. The following conditions hold:*

- *For all $p = 1, \ldots, N$, the letter $L[p]$ is circularly followed by the letter $F[p]$ in its associated string S_y;*
- *For each letter $c \in \Sigma$, its occurrences in L appear in the same order as in F, i.e. the α-th occurrence of c in L corresponds to the α-th occurrence of c in F.*
- *For all $p = 1, \ldots, N$ such that $L[p] = \$_q$ for some q, the letter $F[p]$ is preceded (in the ED string) by all the symbols in $L[b, e]$, where $\mathcal{L}_{ED}(p) = [b, e]$. If any end-marker symbol appears in $L[b, e]$, then $F[p]$ is preceded (in the ED string) by the empty string.*

Proposition 3.1 allows us to define the following permutation LF_{ED} that maps each position of L to F and allows us to link the first symbol of a string in a degenerate symbol to the last symbol of any string within the previous degenerate symbol.

$$LF_{ED}[p] = \begin{cases} C[L[p]] + rank_L(p, L[p]) & \text{if } L[p] \neq \$_q \\ \mathcal{L}_{ED}(p) & \text{if } L[p] = \$_q \end{cases} \tag{2}$$

The LF_{ED} of our running example is shown in Table 2.

By using this modified LF-mapping, we are able to reconstruct the ED string, as follows. We begin from the first end-marker symbol, which is $\$_1$, in position 15. Using $\mathcal{L}_{ED}$ we can find the positions $[8, 8]$ of the last letters in the final degenerate symbol, which in this case is $L[8] = T$. Note that, in this example, we have a single string in both the first and last degenerate symbols. In the general case, by Definition 3.3, $\mathcal{L}_{ED}$ applied to any end-marker symbol in X_1 gives the interval containing the positions of the last letters of all strings in X_k. Then, using the LF-mapping described in Eq. (2), we find that T is preceded by $L[28] = G$, and continue in this way until we have reconstructed the string $S_8 = CTGT$. This is correct since $X_5 = \{CTGT\}$. This step stops at position 21, which is such that $L[21] = \$_8$. Thus, we use $\mathcal{L}_{ED}$ to obtain the positions corresponding to the last letters in the previous degenerate symbol X_4, $\mathcal{L}_{ED}(21) = [6, 7]$. Since we have two positions now, we reconstruct two different strings using the same strategy as before; the first is A, ending at position 12, while the second terminates immediately, since $L(7) = \$_7$, indicating that S_7 is the empty string. This is correct since $X_4 = \{A, \epsilon\}$.

We do not need to compute both $\mathcal{L}_{ED}(7)$ and $\mathcal{L}_{ED}(12)$, since by Definition 3.3 they give the same result of $[5, 5]$. The reconstruction continues by alternating between the parallel reconstruction of all the strings in a degenerate symbol and the linking to the strings in the previous degenerate symbol. This process ends when it reaches the position of first end-marker, since that was the starting position.

Implementation of $\mathcal{L}_{ED}$ using bitvectors. Let $\mathcal{D} = X_1 \cdots X_k$ be an ED string. We define the bitvector $bv(\mathcal{D})$ associated to $\mathcal{D}$ as the concatenation of $bv(X_1)$, ..., $bv(X_k)$, where the bitvector $bv(X_i)$ of length $|X_i|$ has all zeroes except for its first bit, which is one. For instance, the bitvector of our running example is $bv(\mathcal{D}) = 1\ 100\ 1\ 10\ 1$. An analogous definition of a bitvector to represent the underlying structure of a degenerate string appears in [34].

The following proposition shows how, using rank and select on $bv(\mathcal{D})$, we can compute $\mathcal{L}_{ED}(p)$ in constant time, provided that the index q such that $L[p] = \$_q$ is given.

Proposition 3.2. *Let $\mathcal{D} = X_1 \cdots X_k$ be an elastic degenerate string, and let $\mathcal{S} = S_1, \ldots, S_\ell$ the ordered collection of strings contained in the degenerate symbols and let $1 \leq q \leq \ell$. For $t > 1$, let b, e be the indexes such that, if $S_q \in X_t$, then $X_{t-1} = \{S_b, \ldots, S_e\}$. If $t = 1$, let b, e such that $X_k = \{S_b, \ldots, S_e\}$. Then b and e can be computed in constant time using just q and $bv(\mathcal{D})$. Specifically, for $t > 1$, thus $\mathtt{rank}_{bv(\mathcal{D})}(q, 1) > 1$, then*

$$b = \mathtt{select}_{bv(\mathcal{D})}(\mathtt{rank}_{bv(\mathcal{D})}(q, 1) - 1, 1),$$
$$e = \mathtt{select}_{bv(\mathcal{D})}(\mathtt{rank}_{bv(\mathcal{D})}(q, 1), 1) - 1.$$

For $t = 1$, thus $\mathtt{rank}_{bv(\mathcal{D})}(q, 1) = 1$, then

$$b = \mathtt{select}_{bv(\mathcal{D})}(\mathtt{rank}_{bv(\mathcal{D})}(\ell, 1), 1), \quad e = \ell.$$

Proof. Since $bv(\mathcal{D})$ is the concatenation of the $bv(X_t)$'s, and since each $bv(X_t)$ contains exactly one 1, then $\mathtt{rank}_{bv(\mathcal{D})}(q, 1)$ gives the t such that $S_q \in X_t$, for each t. Now observe that, for $t > 1$, b and e are the indexes of the first and last string of X_{t-1}, so $bv(\mathcal{D})[b] = 1$ and $bv(\mathcal{D})[e + 1] = 1$. In particular, $bv(\mathcal{D})[b]$ is the $(t - 1)$-th 1, while e is the position preceding the t-th 1. On the other hand, for $t = 1$, b and e are the indexes of the first and last string of X_k.

We can now compute b and e using `select`. Finally, `rank` and `select` can be performed in constant time on a bitvector, for example using the `sdsl-lite` library [33]. □

Note that we need the index q for computing $\mathcal{L}_{ED}(p)$. The mapping from p to q can be obtained during the construction of the ebwt($\mathcal{S}$) (see Definition 3.2). In our implementation, to compute q in constant time, we use a bitvector v such that $v[p] = 1$ if and only if ebwt($\mathcal{S}$)$[p] = \$_j$ for some $1 \leq j \leq \ell$, and an associated array A of length ℓ such that $A[i] = j$, if $\mathtt{rank}_v(p, 1) = i$, for each $1 \leq i \leq \ell$. Hence $q = A[\mathtt{rank}_v(p, 1)]$.

We conclude this section by encapsulating the previous observations in Algorithm 1. For simplicity, when describing the algorithms in the following, we just write $\$_q \leftarrow L[p]$ to mean that we obtain the index q from position p, if such q exists, and assign 0 to q otherwise.

Algorithm 1: $\texttt{link}(p, bv(\mathcal{D}))$

$\$_q \leftarrow L[p]$;
if $q = 0$ **then**
  return $\emptyset$
if $\$_q \notin X_1$ **then**
  $b \leftarrow \texttt{select}_{bv(\mathcal{D})}(\texttt{rank}_{bv(\mathcal{D})}(q, 1) - 1, 1)$;
  $e \leftarrow \texttt{select}_{bv(\mathcal{D})}(\texttt{rank}_{bv(\mathcal{D})}(q, 1), 1) - 1$;
else
  $b \leftarrow \texttt{select}_{bv(\mathcal{D})}(\texttt{rank}_{bv(\mathcal{D})}(\ell, 1), 1)$;
  $e \leftarrow size(bv(D))$;
return $[b, e]$;

4. Exact pattern matching problem

In this section we show how the EDS-BWT can be used to resolve the exact pattern matching problem for an elastic degenerate string $\mathcal{D} = X_1 \cdots X_k$.

In the classical problem, a string $T \in \Sigma^*$ *contains the pattern* $P \in \Sigma^*$ if and only if P is a substring of T. This concept is generalized to (elastic) degenerate strings in the natural way.

Definition 4.1. Let $P \in \Sigma^*$, and let $\mathcal{D} = X_1 \cdots X_k$ be an ED string over Σ, with $X_i = \{w_{i,1}, \ldots, w_{i,\ell_i}\}$ for each i. We say that $\mathcal{D}$ *contains the pattern* P if there exists a collection of strings $w_{s,j_s} \in X_s, \ldots, w_{s+t,j_{s+t}} \in X_{s+t}$, such that P is a pattern of $w_{s,j_s} \cdots w_{s+t,j_{s+t}}$, for some s, t such that $0 \leq t \leq k-1$ and $1 \leq s \leq k-t$.

In this case, we say that $\mathcal{D}$ contains an *occurrence* of P at position (i, j_i, r) if P begins at position r in w_{i,j_i}.

Note that more than one occurrence of P can start (and end) at the same starting (and ending) position, since they can select different strings in the degenerate symbols. For example the pattern CAT in $\mathcal{D} = \{T, G, TTTT\mathbf{C}\}\{A, NG, \epsilon\}\{A, CT, \epsilon\}\{\mathbf{T}\}$ can be obtained in two different ways starting at position $(1, 3, 5)$ and ending at position $(4, 1, 1)$ (in bold), taking either the red or the blue strings in the degenerate symbols X_2 and X_3. Conversely, there are no occurrences of TG, because T and G belong to the same degenerate symbol.

In [3], the authors showed how to search a pattern $P = P[1, m]$ backwards by the output of the classical BWT. A backward search algorithm first searches for the $P[m]$-interval (*i.e.* the interval in L of the symbols associated to suffixes starting with $P[m]$), then for the $(P[m-1]P[m])$-interval (*i.e.* the interval in L of the symbols associated to suffixes starting with $P[m-1]P[m]$), and so on, until the whole pattern $P[1, m]$ is found, if there is one. Specifically, given an interval $[b, e]$ such that $L[b, e]$ are all the letters followed by $P[j, m]$ in the original string, then the letters followed by $P[j-1, m]$ are in the interval $[b', e']$, with

$$b' = C[c] + rank_L(b-1, c) + 1, \qquad e' = C[c] + rank_L(e, c).$$

We adapt the backward search algorithm defined in [3] in order to apply it to $\text{edsbwt}(\mathcal{D})$ and solve the pattern matching problem.

Algorithm 2: `edsBWTSearch`(edsbwt($\mathcal{D}$), $C, bv(\mathcal{D}), P[1,m]$)

```
1  c = P[m], i = m − 1;
2  b = C[c] + 1, e = C[c + 1];
3  Intervals = {[b, e]} ;
4  while Intervals ≠ ∅ and i > 0 do
       // Compute L_ED on every interval
5      for [b, e] ∈ Intervals do
6          for p ∈ [b, e] do
7              Intervals ← Merge(Intervals, link(p, bv(D))) ;
       // Search P[i] in the intervals
8      Intervals' = ∅ ;
9      c = P[i];
10     for [b, e] ∈ Intervals do
11         b' = C[c] + rank_L(b − 1, c) + 1 ;
12         e' = C[c] + rank_L(e, c) ;
13         if b' ≤ e' then
14             Intervals' ← Merge(Intervals', {[b', e']}) ;
15     i = i − 1 ;
16     Intervals ← Intervals';
17 if Intervals ≠ ∅ then
18     return "Found";
19 else
20     return "Not found";
```

Algorithm 2 (`edsBWTSearch`) outlines the steps of the algorithm. `edsBWTSearch` loops over the symbols of P, starting from the last, and for each iteration updates a list *Intervals* of the intervals of L which have a positive match for the current symbol. The update happens in two steps:

- the first step calls `link` (Algorithm 1). The algorithm finds all the end-marker symbols contained in the current intervals, and compute $\mathcal{L}_{ED}$ for each of those positions, adding new intervals to the list. We note the following:
  - the newly added intervals are checked again, since they could also contain an end-marker symbol, meaning that its corresponding string is the empty string;
  - the algorithm actually uses a non-circular version of `link`, because the pattern matching problem is not circular. Thus, when an end-marker symbol belonging to the first degenerate symbol is considered, the output of `link` is the empty interval (differently from lines 8-9 in Algorithm 1);
  - the `Merge` called in line 7 (and again in line 14) is a modified version of the union operation. It adds the interval I obtained from `link` to the list of intervals, but

first checks if I is consecutive to or included in other intervals in *Intervals*. If this happens, a new interval is instead created by combining the two into one;

- the second step applies the modified backward search to each interval in the list *Intervals*. Note that `Merge` is applied also in this step.

Since `link` and the LF-mapping are called for each interval, then the `Merge` operation allows for a faster search by reducing the number of intervals (see Section 5 for an upper bound on the number of intervals at each iteration).

Table 3 shows the backward search of pattern TAC on our running example. The search begins from interval $[16, 21]$, which is the interval of positions corresponding to letter $P[3] = C$ (marked in blue in the table). For readability, we will not color every interval in the table at every step, but we will just mark some "branchings" that visually show the `link` and update parts. Nonetheless, we fully describe the search in the following.

Since there are three end-marker symbols in $L[16, 21]$, `link` is recursively called on each of them, giving four additional intervals $[1, 1]$, $[2, 4]$ (in orange in the table), $[5, 5]$, $[6, 7]$. Note that $[5, 5]$ is obtained from the recursive call on $[6, 7]$, since $L[7] = \$_7$, which corresponds to the empty string in X_4 of Eq. (1). The intervals are thus merged into one: $[1, 7]$. Thus $Intervals = [[1, 7], [16, 21]]$.

The next step is to update each interval, searching for $P[2] = A$. Interval $[1, 7]$ gives $[9, 12]$ and interval $[16, 21]$ gives $[13, 14]$. Thus $Intervals = [[9, 14]]$.

Again, each end-marker symbol in the intervals is checked, giving $[1, 1]$ (in green) and $[5, 5]$. Finally, $P[1] = T$ is searched, obtaining five occurrences of TAC, starting at positions 1, 5, 9, 10 and 13 of L.

The positions in the original ED string can be recovered using the LF-mapping on each of the resulting positions until reaching an end-marker symbol. This gives the string index, the degenerate symbol index and the position in the string, which are $(1, 1, 6)$, $(2, 1, 2)$, $(2, 2, 1)$, $(3, 1, 2)$, $(3, 1, 9)$.

Experiments. In order to evaluate our backward search applied to edsbwt$(\mathcal{D})$, we implemented[3] a prototype in C++ that builds $\mathcal{L}_{ED}$ and then solves the pattern matching problem on an EDS. Since the EBWT is a well-known structure in string algorithms, we used existing tools to build the ebwt$(\mathcal{S})$. Note that the efficient construction of the EBWT has been the subject of extensive research (see for instance [12, 35, 36, 37, 38, 39]), that is beyond the goal of this paper. Therefore, the time needed to build the ebwt$(\mathcal{S})$ depends on the tool used, and the resources available. Moreover, since our pattern matching strategy is *off-line* (it builds edsbwt$(\mathcal{D})$), the cost to pre-process $\mathcal{D}$, which however took a couple of seconds for the dataset with the longest EDS in our experiments, can be amortized for all the pattern searches.

In order to show the effectiveness of our pattern matching method, we have considered two existing tools, `edsm` [17] and `eds_search` [20], that solve the elastic-degenerate string matching problem in an *on-line* manner by taking as input an ED string on a biologic alphabet. In particular, `edsm` takes an ED string and a solid pattern P, it builds the suffix tree of P and scan the ED string left-to-right to return the indices of the degenerate symbols at which each

[3] https://github.com/giovannarosone/EDS-BWT

occurrence of P ends; while `eds_search`, given as input an ED string and a solid pattern P, performs an on-line backward pattern matching by combining the traditional algorithms BNDM and SHIFT-AND, and returns the number of positions in which an occurrence of P starts.

Note that there exist other tools for the pattern matching on closely-related sequences, but they encode the similar strings with data structures different from ED strings, see for instance [31, 29, 28, 40].

The experiments were conducted on a DELL PowerEdge R630 machine, 24-core, with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GB of shared memory. The system is Ubuntu 14.04.2 LTS.

We tested our tool on the synthetic elastic degenerate strings used in [17], which are 5 randomly generated strings of lengths 100000, 200000, 400000, 800000 and 1600000. We searched 40 patterns of lengths 8, 16, 32 and 64[4] (ten patterns each). The patterns were obtained by selecting random parts of the ED string and extracting a substring of the desired length, so that each pattern would occur at least once. This is important, since the absence of a pattern stops prematurely the execution of the algorithm, giving distorted timings.

Table 4 shows that the performance for all datasets of `edsBWTSearch` is comparable to the one of `edsm`. The tool `eds_search` is always the fastest, but it does not return the positions of the pattern occurrences. Excluding the pre-processing time needed to build edsbwt($\mathcal{D}$), `edsBWTSearch` is always faster than `edsm`; however, the pre-processing time for the largest dataset is 4.72 seconds that, if added to the `edsBWTSearch` time of 2.69 seconds, gives 7.41 seconds. Finally, we note that `edsBWTSearch` is implemented in semi-external memory and stores on disk part of the index edsbwt($\mathcal{D}$). In this way, `edsBWTSearch` is the tool that uses the least RAM on the largest dataset.

5. Conclusion, discussion and further work

In this work, we introduced a new transform EDS-BWT inspired by the BWT of a string and the EBWT of a string collection. The EDS-BWT permutes the letters of the strings in $\mathcal{S}$ associated with an ED string by exploiting the $ebwt(\mathcal{S})$ and also returns a function $\mathcal{L}_{ED}$ that allows to link the strings of a degenerate symbol to the strings of the previous one. The introduced transform works over any alphabet and it does not concatenate strings in $\mathcal{S}$, but keeps track of links between strings of consecutive degenerate symbols. Moreover, it allows, like the BWT on a string, to build an index on which to perform pattern searches.

We observe that the time and space usage for computing EDS-BWT depends mainly on the construction of ebwt($\mathcal{S}$), because the construction of $\mathcal{L}_{ED}$ can be achieved by simply reading $\mathcal{D}$ and producing the bit vector $bv(\mathcal{D})$. Since there are several tools for building ebwt($\mathcal{S}$), one can choose the implementation that best suits their own resources.

The pattern matching algorithm involves m iterations, where m is the length of the pattern P. It is easy to see that the first iteration may obtain at most $k+1$ intervals, because, as for the classical backward search, `edsBWTSearch` produces a single interval for the LF-mapping of letter $P[m]$, while $\mathcal{L}_{ED}$ can return at most one interval for each degenerate symbol (see Definition 3.3). In the same fashion, at most k intervals can be added at each subsequent

[4] `eds_search` does not support patterns of length greater than 32.

		edsBWTSearch		edsm		eds_search	
N	m	Wall clock (sec)	RAM (kb)	Wall clock (sec)	RAM (kb)	Wall clock (sec)	RAM (kb)
100000	8	0.27	4404	1.24	14064	0.09	2796
	16	0.28	4324	1.20	14048	0.09	2836
	32	0.28	4348	1.17	14056	0.09	2884
	64	0.28	4396	1.22	14040	-	-
200000	8	0.53	4676	1.65	14084	0.13	3212
	16	0.54	4692	1.67	14052	0.14	3252
	32	0.54	4744	1.74	14048	0.13	3256
	64	0.54	4712	1.84	14092	-	-
400000	8	0.68	5448	2.50	14056	0.18	4948
	16	0.77	5388	2.63	14060	0.19	4948
	32	0.68	5380	2.80	14044	0.19	4908
	64	0.79	5388	3.15	13988	-	-
800000	8	1.34	7224	4.15	14060	0.28	8360
	16	1.41	7044	4.61	14060	0.29	8360
	32	1.59	7364	5.01	14080	0.27	8360
	64	1.54	7028	5.63	14008	-	-
1600000	8	2.61	10260	7.47	14020	0.44	15116
	16	2.46	10880	8.37	14044	0.46	15116
	32	2.57	10200	9.20	14004	0.46	15160
	64	2.69	10500	10.67	14052	-	-

Table 4
The table shows the total running time and RAM used by the three tools `edsBWTSearch`, `edsm` and `eds_search` for searching 10 patterns of size m in 5 different ED strings of total length N.

iteration, one for each degenerate symbol of the EDS. Therefore the worst case is to add k intervals at each iteration, for a total of $km + 1$. However, this is a theoretical upper bound that do not take into account the interval merging. Indeed, in the above case, the k intervals of each iteration would be merged into one, giving just m intervals, which is far from the worst case. Accounting for merging, the worst case occurs when all the intervals are non consecutive. Thus, the maximum amount of intervals added at each iteration is $\lceil \frac{k}{2} \rceil$, one every two degenerate symbols. Since we have m iterations, this sums up to $\lceil \frac{k}{2} \rceil m + 1$ total maximum intervals.

During our experiments, we observed the number of intervals to be strictly decreasing after the first iteration. We believe this not to be a coincidence and plan to investigate the matter in subsequent work. Finally, we note that it is possible to reduce the number of intervals by building a partial index for patterns of small lengths.

Another future direction of work is to show that, like the EBWT, the EDS-BWT is dynamic, meaning that we can add/remove a string to a degenerate symbol without needing to rebuild the entire transform, but suitably adding/removing the symbols and updating $\mathcal{L}_{ED}$.

Moreover, we aim to show that EDS-BWT also allows us to search for multiple patterns at the same time.

Acknowledgments

Work partially supported by the MUR PRIN 2022YRB97K PINC, by INdAM - GNCS Project, codice CUP_E53C23001670001, “Compressione, indicizzazione, analisi e confronto di dati biologici”, and by PNRR - M4C2 - Investimento 1.5, Ecosistema dell’Innovazione ECS00000017 - “THE - Tuscany Health Ecosystem” - Spoke 6 “Precision medicine & personalized healthcare”, funded

by the European Commission under the NextGeneration EU programme.

References

[1] M. Burrows, D. J. Wheeler, A Block Sorting Lossless Data Compression Algorithm, Technical Report 124, Digital Equipment Corporation, 1994.

[2] G. Rosone, M. Sciortino, The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words, in: CiE, volume 7921 LNCS of *LNCS*, Springer, 2013, pp. 353–364.

[3] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: FOCS, IEEE Computer Society, 2000, pp. 390–398. doi:10.1109/SFCS.2000.892127.

[4] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nordic J. of Computing 12 (2005) 40–66.

[5] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, J. Comput. Biol. 17 (2010) 281–308.

[6] T. Gagie, G. Navarro, N. Prezza, Fully functional suffix trees and optimal text searching in bwt-runs bounded space, J. ACM 67 (2020). doi:10.1145/3375890.

[7] H. Li, R. Durbin, Fast and accurate long-read alignment with Burrows-Wheeler transform, Bioinformatics 26 (2010) 589–595. doi:10.1093/bioinformatics/btp698.

[8] V. Guerrini, A. Conte, R. Grossi, G. Liti, G. Rosone, L. Tattini, phyBWT2: phylogeny reconstruction via eBWT positional clustering, Algorithms Mol. Biol. 18 (2023) 11. doi:10.1186/S13015-023-00232-4.

[9] J. T. Simpson, R. Durbin, Efficient construction of an assembly string graph using the FM-index, Bioinform. 26 (2010) 367–373.

[10] V. Guerrini, F. A. Louza, G. Rosone, Parallel lossy compression for large fastq files, in: Biomedical Engineering Systems and Technologies, Springer Nature Switzerland, Cham, 2023, pp. 97–120.

[11] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows-Wheeler Transform, Theor. Comput. Sci. 387 (2007) 298–312. doi:10.1016/j.tcs.2007.07.014.

[12] M. J. Bauer, A. J. Cox, G. Rosone, Lightweight algorithms for constructing and inverting the BWT of string collections, Theor. Comput. Sci. 483 (2013) 134 – 148. Source code: https://github.com/BEETL/BEETL.

[13] P. Ferragina, G. Manzini, An experimental study of a compressed index, Information Sciences 135 (2001) 13–28. doi:10.1016/S0020-0255(01)00098-6.

[14] T. C. P.-G. Consortium, Computational pan-genomics: status, promises and challenges, Briefings in Bioinformatics 19 (2016) 118–135. doi:10.1093/bib/bbw089.

[15] C. S. Iliopoulos, R. Kundu, S. P. Pissis, Efficient pattern matching in elastic-degenerate texts, in: 11th International Conference on Language and Automata Theory and Applications (LATA), volume 10168 of *Springer LNCS*, 2017, pp. 131–142.

[16] M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone, Comparing degenerate strings, Fundam. Informaticae 175 (2020) 41–58. doi:10.3233/FI-2020-1947.

[17] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani,

L. Versari, et al., On-line pattern matching on similar texts, in: Proceedings of 28th Annual Symposium on Combinatorial Pattern Matching (CPM), volume 78, Schloss Dagstuhl–Leibniz-Zentrum für Informatik GmbH, 2017, p. 1.

[18] C. S. Iliopoulos, R. Kundu, S. P. Pissis, Efficient pattern matching in elastic-degenerate strings, Information and Computation 279 (2021) 104616. doi:10.1016/j.ic.2020.104616.

[19] G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, G. Rosone, Elastic-degenerate string matching via fast matrix multiplication, SIAM Journal on Computing 51 (2022) 549–576. doi:10.1137/20M1368033.

[20] P. Procházka, O. Cvacho, L. Krčál, J. Holub, Backward pattern matching on elastic-degenerate strings, SN Computer Science 4 (2023) 442.

[21] H. Soldano, A. Viari, M. Champesme, Searching for flexible repeated patterns using a non-transitive similarity relation, Pattern Recognition Letters 16 (1995) 233–246.

[22] N. Pisanti, H. Soldano, M. Carpentier, Incremental inference of relational motifs with a degenerate alphabet, in: CPM, volume 3537 of *Springer LNCS*, 2005, pp. 229–240.

[23] N. Pisanti, H. Soldano, M. Carpentier, J. Pothier, A relational extension of the notion of motifs: Application to the common 3d protein substructures searching problem, Journal of Computational Biology 16 (2009) 1635–1660.

[24] K. Abrahamson, Generalized string matching, SIAM Journal of Computing 16 (1987) 1039–1051.

[25] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, T. Walen, Covering problems for partial words and for indeterminate strings, Theoretical Computer Science 698 (2017) 25–39.

[26] C. S. Iliopoulos, J. Radoszewski, Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties, in: 27th Annual Symposium on Combinatorial Pattern Matching (CPM), volume 54 of *LIPIcs*, 2016, pp. 8:1–8:12.

[27] J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur, B. W. Watson, Efficient pattern matching in degenerate strings with the Burrows–Wheeler transform, Information Processing Letters 147 (2019) 82–87. doi:10.1016/j.ipl.2019.03.003.

[28] J. C. Na, H. Kim, H. Park, T. Lecroq, M. Léonard, L. Mouchard, K. Park, FM-index of alignment: A compressed index for similar strings, Theoretical Computer Science 638 (2016) 159–170.

[29] S. Maciuca, C. del Ojo Elias, G. McVean, Z. Iqbal, A Natural Encoding of Genetic Variation in a Burrows-Wheeler Transform to Enable Mapping and Genome Inference, in: Algorithms in Bioinformatics, Springer International Publishing, Cham, 2016, pp. 222–233.

[30] S. Huang, T. W. Lam, W. K. Sung, S. L. Tam, S. M. Yiu, Indexing similar DNA sequences, in: Algorithmic Aspects in Information and Management, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 180–190.

[31] L. Huang, V. Popic, S. Batzoglou, Short read alignment with populations of genomes, Bioinformatics 29 (2013) i361–i370. doi:10.1093/bioinformatics/btt215.

[32] T. Büchler, E. Ohlebusch, An improved encoding of genetic variation in a Burrows–Wheeler transform, Bioinformatics 36 (2019) 1413–1419. doi:10.1093/bioinformatics/btz782.

[33] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: Plug and play with succinct data structures, in: 13th International Symposium on Experimental Algorithms, (SEA 2014), 2014, pp. 326–337. doi:10.1007/978-3-319-07959-2_28, source code: https://github.com/simongog/sdsl-lite.

[34] P. Bille, I. L. Gørtz, T. Stordalen, Rank and select on degenerate strings, in: 2024 Data Compression Conference (DCC), IEEE, 2024, pp. 283–292.

[35] L. Egidi, F. A. Louza, G. Manzini, G. P. Telles, External memory BWT and LCP computation for sequence collections with applications, Algorithms Mol. Biol. 14 (2019) 6:1–6:15. doi:10.1186/s13015-019-0140-0.

[36] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, R. Rizzi, Multithread Multistring Burrows-Wheeler Transform and Longest Common Prefix Array, Journal of computational biology 26 (2019) 948—961. doi:10.1089/cmb.2018.0230.

[37] F. A. Louza, G. P. Telles, S. Gog, N. Prezza, G. Rosone, gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections, Algorithms Mol. Biol. 15 (2020) 18. doi:10.1186/s13015-020-00177-y.

[38] N. Prezza, G. Rosone, Space-efficient construction of compressed suffix trees, Theoretical Computer Science 852 (2021) 138 – 156. doi:10.1016/j.tcs.2020.11.024.

[39] D. Díaz-Domínguez, G. Navarro, Efficient Construction of the BWT for Repetitive Text Using String Compression, in: CPM 2022, volume 223 of *LIPIcs*, 2022, pp. 29:1–29:18.

[40] J. Sirén, Indexing variation graphs, in: Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX), 2017, pp. 13–27. doi:10.1137/1.9781611974768.2.