Computing the original eBWT faster, simpler, and with less memory

Christina Boucher
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, United States
c.boucher@cise.ufl.edu

Davide Cenzato
Department of Computer Science, University of Verona, Verona, Italy
davide.cenzato@univr.it

Zsuzsanna Lipták
Department of Computer Science, University of Verona, Verona, Italy
zsuzsanna.liptak@univr.it

Massimiliano Rossi
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, United States
rossi.m@ufl.edu

Marinella Sciortino
Department of Computer Science, University of Palermo, Palermo, Italy
marinella.sciortino@unipa.it

Abstract

Given an input string, the *Burrows-Wheeler Transform* (BWT) can be seen as a reversible permutation of it that allows efficient compression and fast substring queries. Due to these properties, it has been widely applied in the analysis of genomic sequence data, enabling important tasks such as read alignment. Mantaci et al. [TCS2007] extended the notion of the BWT to a collection of strings by defining the *extended Burrows-Wheeler Transform* (eBWT). This definition requires no modification of the input collection, and has the property that the output is independent of the order of the strings in the collection. However, over the years, the term eBWT has been used more generally to describe any BWT of a collection of strings. The fundamental property of the original definition (i.e., the independence from the input order) is frequently disregarded. In this paper, we propose a simple linear-time algorithm for the construction of the original eBWT, which does not require the preprocessing of Bannai et al. [CPM 2021]. As a byproduct, we obtain the first linear-time algorithm for computing the BWT of a single string that uses neither an end-of-string symbol nor Lyndon rotations.

We also combine our new eBWT construction with a variation of *prefix-free parsing* (PFP) [WABI 2019] to allow for construction of the eBWT on large collections of genomic sequences. We implement this combined algorithm (`pfpebwt`) and evaluate it on a collection of human chromosomes 19 from the 1,000 Genomes Project, on a collection of Salmonella genomes from GenomeTrakr, and on a collection of SARS-CoV2 genomes from EBI's COVID-19 data portal. We demonstrate that `pfpebwt` is the fastest method for all collections, with a maximum speedup of 7.6x on the second best method. The peak memory is at most 2x larger than the second best method. Comparing with methods that are also, as our algorithm, able to report suffix array samples, we obtain a 57.1x improvement in peak memory. The source code is publicly available at `https://github.com/davidecenzato/PFP-eBWT`.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Pattern matching; Theory of computation → Data structures design and analysis

Keywords and phrases extended BWT, prefix-free parsing, SAIS algorithm, omega-order

1 Introduction

In the last several decades, the number of sequenced human genomes has been growing at unprecedented pace. In 2015 the number of sequenced genomes was doubling every 7 months [40] – a pace that has not slowed into the current decade. The plethora of resulting sequencing data has expanded our knowledge of the biomarkers responsible for human disease and phenotypes [5, 43, 42], the evolutionary history between and among species [38], and will eventually help realize the personalization of healthcare [3]. However, the amount of data for any individual species is large enough that it poses challenges with respect to storage and analysis. One of the most well-known and widely-used methods for compressing and indexing data that has been applied in bioinformatics is the Burrows-Wheeler Transform (BWT), which is a text transformation that compresses the input in a manner that also allows for efficient substring queries. Not only can it be constructed in linear-time in the length of the input, it is also reversible – meaning the original input can be constructed from its compressed form. The BWT is formally defined over a single input string; thus, in order to define and construct it for one or more strings, the input strings need to be concatenated or modified in some way. In 2007 Mantaci et al. [30] presented a formal definition of the BWT for a multiset of strings, which they call the *extended Burrows-Wheeler Transform* (eBWT). It is a bijective transformation that sorts the cyclic rotations of the strings of the multiset according to the ω-order relation, an order, defined by considering infinite iterations of each string, which is different from the lexicographic order.

Since its introduction several algorithms have been developed that construct the BWT of collection of strings for various types of biological data including short sequence reads [6, 4, 11, 14, 27, 13, 14, 1, 19, 36, 37], protein sequences [44], metagenomic data [18] and longer DNA sequences such as long sequence reads and whole chromosomes [25]. However, we note that in the development of some of these methods the underlying definition of eBWT was loosened. For example, `ropebwt2` [25] tackles a similar problem of building what they describe as the FM-index for a multiset of long sequence reads, however, they do not construct the suffix array (SA) or SA samples, and also, require that the sequences are delimited by separator symbols. Similarly, `gsufsort` [27] and `egap` [14] construct the BWT for a collection of strings but do not construct the eBWT according to its original definition. `gsufsort` [27] requires the collection of strings to be concatenated in a manner that the strings are deliminated by separator symbols that have an augmented relative order among them. `egap` [14], which was developed to construct the BWT and LCP for a collection of strings in external memory, uses the `gSACA-K` algorithm to construct the suffix array of the concatenated input using an additional $O(\alpha + 1) \log n$ bits, and then constructs the BWT for the collection from the resulting suffix array. Lastly, we note that there exists a number of methods for construction of the BWT for a collection of short sequence reads, including `ble` [6], `BCR` [4], `G2BWT` [13], `egsa` [28]; however, these methods make implicit or explicit use of end-of-string symbols appended to strings in the collection. For an example of the effects of these manipulations, see Section 2, and [10] for a more detailed study.

We present an efficient algorithm for constructing the eBWT that preserves the original definition of Mantaci et al. [30]—thus, it does not impose any ordering of the input strings or delimiter symbols. It is an adaptation of the well-known Suffix Array Induced Sorting (SAIS) algorithm of Nong et al. [33], which computes the suffix array of a single string T ending with an end-of-string character \$. Our adaptation is similar to the algorithm proposed by Bannai et al. [2] for computing the BBWT, which can also be used for computing the eBWT, after linear-time preprocessing of the input strings. The key change in our approach

is based on the insight that the properties necessary for applying Induced Sorting are valid also for the ω-order between different strings. As a result, is it not necessary that the input be Lyndon words, or that their relative order be known at the beginning. Furthermore, our algorithmic strategy, when applied to a single string, provides the first linear-time algorithm for computing the BWT of the string that uses neither an end-of-string symbol nor Lyndon rotations.

We then combine our new eBWT construction with a variation of a preprocessing technique called *prefix free parsing* (PFP). PFP was introduced by Boucher et al. [8] for building the (run length encoded) BWT of large and highly repetitive input text. Since its original introduction, it has been extended to construct the r-index [24], been applied as a preprocessing step for building grammars [15], and used as a data structure itself [7]. Briefly, PFP is a one-pass algorithm that divides the input into overlapping variable length phrases with delimiting prefixes and suffixes; which in effect, leads to the construction of what is referred to as the dictionary and parse of the input. It follows that the BWT can be constructed in the space that is proportional to the size of the dictionary and parse, which is expected to be significantly smaller than linear for repetitive text.

In our approach, prefix-free parsing is applied to obtain a parse that is a multiset of cyclic strings (*cyclic prefix-free parse*) on which our eBWT construction is applied. We implement our approach (called `pfpebwt`), measure the time and memory required to build the eBWT for sets of increasing size of chromosome 19, *Salmonella*, and SARS-CoV2 genomes, and compare this to that required by `gsufsort`, `ropebwt2`, and `egap`. We show that `pfpebwt` is consistently faster and uses less memory than `gsufsort` and `egap` on reasonably large input ($\geq$ 4 copies of chromosome 19, $\geq$ 50 *Salmonella* genomes, and $\geq$ 25,000 SARS-CoV2 genomes). Although `ropebwt2` uses less memory than `pfpebwt` on large input, `pfpebwt` is 7x more efficient in terms of wall clock time, and 2.8x in terms of CPU time. Moreover, `pfpebwt` is capable of reporting SA samples in addition to the eBWT with a negligible increase in time and memory [24], whereas `ropebwt2` does not have that ability. If we compare `pfpebwt` only with methods that are able to report SA samples in addition to the eBWT (e.g., `egap` and `gsufsort`), we obtain a 57.1x improvement in peak memory.

2 Preliminaries

A string $T = T[1..n]$ is a sequence of characters $T[1] \cdots T[n]$ drawn from an ordered alphabet Σ of size σ. We denote by $|T|$ the length n of T, and by ε the empty string, the only string of length 0. Given two integers $1 \leq i, j \leq n$, we denote by $T[i..j]$ the string $T[i] \cdots T[j]$, if $i \leq j$, while $T[i..j] = \varepsilon$ if $i > j$. We refer to $T[i..j]$ as a *substring* (or *factor*) of T, to $T[1..j]$ as the j-th *prefix* of T, and to $T[i..n] = T[i..]$ as the i-th *suffix* of T. A substring S of T is called *proper* if $T \neq S$. Given two strings S and T, we denote by $\texttt{lcp}(S, T)$ the length of the *longest common prefix* of S and T, i.e., $\texttt{lcp}(S, T) = \max\{i \mid S[1..i] = T[1..i]\}$.

Given a string $T = T[1..n]$ and an integer k, we denote by T^k the kn-length string $TT \cdots T$ (k-fold concatenation of T), and by T^ω the infinite string $TT \cdots$ obtained by concatenating an infinite number of copies of T. A string T is called *primitive* if $T = S^k$ implies $T = S$ and $k = 1$. For any string T, there exists a unique primitive word S and a unique integer k such that $T = S^k$. We refer to $S = S[1..\frac{n}{k}]$ as $\text{root}(T)$ and to k as $\exp(T)$. Thus, $T = \text{root}(T)^{\exp(T)}$.

We denote by $<_{\text{lex}}$ the lexicographic order: for two strings $S[1..n]$ and $T[1..m]$, $S <_{\text{lex}} T$ if S is a proper prefix of T, or there exists an index $1 \leq i \leq n, m$ such that $S[1..i-1] = T[1..i-1]$ and $S[i] < T[i]$. Given a string $T[1..n]$, the *suffix array* [29], denoted by $\text{SA} = \text{SA}_T$, is the

permutation of $\{1,\ldots,n\}$ such that $T[\mathrm{SA}[i]..]$ is the i-th lexicographically smallest suffix of T.

We denote by $\prec_\omega$ the ω-order [16, 30], defined as follows: for two strings S and T, $S \prec_\omega T$ if $\mathrm{root}(S) = \mathrm{root}(T)$ and $\exp(S) < \exp(T)$, or $S^\omega <_{\mathrm{lex}} T^\omega$ (this implies $\mathrm{root}(S) \neq \mathrm{root}(T)$). One can verify that the ω-order relation is different from the lexicographic one. For instance, $CG <_{\mathrm{lex}} CGA$ but $CGA \prec_\omega CG$.

The string S is a *conjugate* of the string T if $S = T[i..n]T[1..i-1]$, for some $i \in \{1,\ldots,n\}$ (also called the *i-th rotation* of T). The conjugate S is also denoted $\mathrm{conj}_i(T)$. It is easy to see that T is primitive if and only if it has n distinct conjugates. A *Lyndon word* is a primitive string which is lexicographically smaller than all of its conjugates. For a string T, the *conjugate array*[1] $\mathrm{CA} = \mathrm{CA}_T$ of T is the permutation of $\{1,\ldots,n\}$ such that $\mathrm{CA}[i] = j$ if $\mathrm{conj}_j(T)$ is the i-th conjugate of T with respect to the lexicographic order, with ties broken according to string order, i.e. if $\mathrm{CA}[i] = j$ and $\mathrm{CA}[i'] = j'$ for some $i < i'$, then either $\mathrm{conj}_j(T) <_{\mathrm{lex}} \mathrm{conj}_{j'}(T)$, or $\mathrm{conj}_j(T) = \mathrm{conj}_{j'}(T)$ and $j < j'$. Note that if T is a Lyndon word, then $\mathrm{CA}[i] = \mathrm{SA}[i]$ for all $1 \le i \le n$ [17].

Given a string T, U a *circular* or *cyclic substring* of T if it is a factor of TT of length at most $|T|$, or equivalently, if it is the prefix of some conjugate of T. For instance, ATA is a cyclic substring of $AGCAT$. It is sometimes also convenient to regard a given string $T[1..n]$ itself as *circular* (or *cyclic*); in this case we set $T[0] = T[n]$ and $T[n+1] = T[1]$.

2.1 Burrows-Wheeler-Transform

The *Burrows-Wheeler Transform* [9] of T, denoted BWT, is a reversible transformation extensively used in data compression. Given a string T, $\mathrm{BWT}(T)$ is a permutation of the letters of T which equals the last column of the matrix of the lexicographically sorted conjugates of T. The mapping $T \mapsto \mathrm{BWT}(T)$ is reversible, up to rotation. It can be made uniquely reversible by adding to $\mathrm{BWT}(T)$ and index indicating the rank of T in the lexicographic order of all of its conjugates. Given $\mathrm{BWT}(T)$ and an index i, the original string T can be computed in linear time [9]. The BWT itself can be computed from the conjugate array, since for all $i = 1,\ldots,n$, $\mathrm{BWT}(T)[i] = T[\mathrm{CA}[i]-1]$, where T is considered to be cyclic.

It should be noted that in many applications, it is assumed that an end-of-string-character (usually denoted \$), which is not element of Σ, is appended to the string; this character is assumed to be smaller than all characters from Σ. Since $T\$$ has exactly one occurrence of \$, $\mathrm{BWT}(T\$)$ is now uniquely reversible, without the need for the additional index i, since $T\$$ is the unique conjugate ending in \$. Moreover, adding a final \$ makes the string primitive, and $\$T$ is a Lyndon word. Therefore, computing the conjugate array becomes equivalent to computing the suffix array, since $\mathrm{CA}_{T\$}[i] = \mathrm{SA}_{T\$}[i]$. Thus, applying one of the linear-time suffix-array computation algorithms [32] leads to linear-time computation of the BWT.

When no \$-character is appended to the string, the situation is slightly more complex. For primitive strings T, first the Lyndon conjugate of T has to be computed (in linear time, [39]) and then a linear-time suffix array algorithm can be employed [17]. For strings T which are not primitive, one can take advantage of the following well-known property of the BWT: let $T = S^k$ and $\mathrm{BWT}(S) = U[1..m]$, then $\mathrm{BWT}(T) = U[1]^k U[2]^k \cdots U[m]^k$ (Prop. 2 in [31]). Thus, it suffices to compute the BWT of $\mathrm{root}(T)$. The root of T can be found by computing

[1] Our conjugate array CA is called *circular suffix array* and denoted $\mathrm{SA}_\circ$ in [20, 2], and *BW-array* in [23, 35], but in both cases defined for primitive strings only.

the border array $\mathbf{b}$ of T: T is a power if and only if $n/(n - \mathbf{b}[n])$ is an integer, which is then also the length of root(T). The border array can be computed, for example, by the preprocessing phase of the KMP-algorithm for pattern matching [21], in linear time in the length of T.

2.2 Generalized Conjugate Array and Extended Burrows-Wheeler Transform

Given a multiset of strings $\mathcal{M} = \{T_1[1..n_1], \ldots, T_m[1..n_m]\}$, the *generalized conjugate array* of $\mathcal{M}$, denoted by $\mathrm{GCA}_{\mathcal{M}}$ or just by GCA, contains the list of the conjugates of all strings in $\mathcal{M}$, sorted according to the ω-order relation. More formally, $\mathrm{GCA}[i] = (j, d)$ if $\mathrm{conj}_j(T_d)$ is the i-th string in the $\preceq_\omega$-sorted list of the conjugates of all strings of $\mathcal{M}$, with ties broken first w.r.t. the index of the string (in case of identical strings), and then w.r.t. the index in the string itself.

The *extended Burrows-Wheeler Transform* (eBWT) is an extension of the BWT to a multiset of strings [30]. It is a bijective transformation that, given a multiset of strings $\mathcal{M} = \{T_1, \ldots, T_m\}$, produces a permutation of the characters on the strings in the multiset $\mathcal{M}$. Formally, $\mathrm{eBWT}(\mathcal{M})$ can be computed by sorting all the conjugates of the strings in the multiset according to the $\preceq_\omega$-order, and the output is the string obtained by concatenating the last character of each conjugate in the sorted list, together with the set of indices representing the positions of the original strings of $\mathcal{M}$ in the list. Similarly to the BWT, the eBWT is thus uniquely reversible. The $\mathrm{eBWT}(\mathcal{M})$ can be computed from the generalized conjugate array of $\mathcal{M}$ in linear time, since $\mathrm{eBWT}(\mathcal{M})[i] = T_d[j-1]$ if $\mathrm{GCA}[i] = (j, d)$, where again, the strings in $\mathcal{M}$ are considered to be cyclic. It is easy to see that when $\mathcal{M}$ consists of only one string, i.e. $\mathcal{M} = \{T\}$, then $\mathrm{eBWT}(\mathcal{M}) = \mathrm{BWT}(T)$.

▶ **Example 1.** Let $\mathcal{M} = \{GTACAACG, CGGCACACACGT, C\}$. Then $\mathrm{GCA}(\mathcal{M})$ is as follows, where we give the pair (j, d) vertically, i.e. the first row contains the position in the string, and the second row the index of the string:

5	3	5	7	6	9	4	4	6	8	1	1	7	10	3	2	8	1	11	2	12
1	1	2	2	1	2	1	2	2	2	3	2	1	2	2	2	1	1	2	1	2

From the GCA we can compute $\mathrm{eBWT}(\mathcal{M}) = CTCCACAGAACTAAGCCGCGG$, with index set $\{11, 12, 18\}$. Note that e.g. the conjugate $\mathrm{conj}_8(T_2)$ comes before $\mathrm{conj}_1(T_3)$, since $CACGTCGGCACA \prec_\omega C$, because $(CACGTCGGCACA)^\omega <_{\mathrm{lex}} C^\omega = CCCC\ldots$ holds. The full list of conjugates is in Appendix A.

▶ Remark 2. Note that if end-of-string symbols are appended to the string of the collection the output of eBWT could be quite different. For instance, if $\mathcal{M} = \{GTACAACG\$_1, CGGCACACACGT\$_2, C\$_3\}$, $\mathrm{eBWT}(\mathcal{M}) = GTCCTCCAC\$_3AGAAA\$_2ACGCC\$_1GG$.

Note that while in the original definition of eBWT [30], the multiset $\mathcal{M}$ was assumed to contain only primitive strings, our definition is more general and allows also for non-primitive strings. For example, $\mathrm{eBWT}(\{ATA,\ TATA\}) = \mathbf{TATTAAA}$, with index set $\{2, 6\}$, while $\mathrm{eBWT}(\{ATA, TA, TA\}) = \mathbf{TATTAAA}$, with index set $\{2, 6, 7\}$. Also the linear-time algorithm for recovering the original multiset can be straightforwardly extended.

The following lemma shows how to construct the generalized conjugate array $\mathrm{GCA}_{\mathcal{M}}$ of a multiset $\mathcal{M}$ of strings (not necessarily primitive), once we know the generalized conjugate array $\mathrm{GCA}_{\mathcal{R}}$ of the multiset $\mathcal{R}$ of the roots of the strings in $\mathcal{M}$. It follows straightforwardly from the fact that equal conjugates will end up consecutively in the GCA.

▶ **Lemma 3.** *Let $\mathcal{M} = \{T_1, \ldots, T_m\}$ be a multiset of strings and let $\mathcal{R}$ the multiset of the roots of the strings in $\mathcal{M}$, i.e. $\mathcal{R} = \{S_1, \ldots, S_m\}$, where $T_i = (S_i^{r_i})$, with $r_i \geq 1$ for $1 \leq i \leq m$. Let $\mathrm{GCA}_{\mathcal{R}}[1..K] = [(j_1, i_1), (j_2, i_2), \ldots, (j_K, i_K)]$, where $K = \sum_{i=1}^{m} |S_i|$. The generalized conjugate array is then given by*

$$\begin{aligned}\mathrm{GCA}_{\mathcal{M}}[1..N] = [&(j_1, i_1), (j_1 + |S_{i_1}|, i_1), \ldots, (j_1 + (r_{i_1} - 1) \cdot |S_{i_1}|, i_1),\\ &(j_2, i_2), (j_2 + |S_{i_2}|, i_2), \ldots, (j_2 + (r_{i_2} - 1) \cdot |S_{i_2}|, i_2),\\ &\ldots\\ &(j_K, i_K), (j_K + |S_{i_K}|, i_K), \ldots, (j_K + (r_{i_K} - 1) \cdot |S_{i_K}|, i_K)],\end{aligned}$$

with $N = \sum_{i=1}^{m} |S_i| \cdot r_i$.

From now on we will assume that the multiset $\mathcal{M} = \{T_1, \ldots, T_m\}$ consists of m primitive strings.

3 A simpler algorithm for computing the eBWT and GCA

In this section, we describe our algorithm to compute the eBWT of a multiset of strings $\mathcal{M}$. We will assume that all strings in $\mathcal{M}$ are primitive, since we can use Lemma 3 to compute the eBWT of $\mathcal{M}$ otherwise. Our algorithm is an adaptation of the well-known SAIS algorithm of Nong et al. [33], which computes the suffix array of a single string T ending with an end-of-string character \$. Our adaptation is similar to that of Bannai et al. [2] for computing the BBWT, which can also be used for computing the eBWT. Even though our algorithm does not improve the latter asymptotically (both are linear time), it is significantly simpler, since it does not require first computing and sorting the Lyndon rotations of the input strings.

In the following, we assume some familiarity with the SAIS algorithm, focusing on the differences between our algorithm and the original SAIS. Detailed explanations of SAIS can be found in the original paper [33], or in the books [34, 26].

The main differences between our algorithm and the original SAIS algorithm are: (1) we are comparing conjugates rather than suffixes, (2) we have a multiset of strings rather than just one string, (3) the comparison is done w.r.t. the omega-order rather than the lexicographic order, and (4) the strings are not terminated by an end-of-string symbol.

We need the following definition, which is the cyclic version of the definition in [33] (where S stands for smaller, L for larger, and *LMS* for leftmost-S):

▶ **Definition 4** (Cyclic types, *LMS*-substrings)**.** *Let T be a primitive string of length at least 2, and $1 \leq i \leq |T|$. Position i of T is called* (cyclic) S-type *if $conj_i(T) <_{\mathrm{lex}} conj_{i+1}(T)$, and* (cyclic) L-type *if $conj_i(T) >_{\mathrm{lex}} conj_{i+1}(T)$. An S-type position i is called* (cyclic) LMS *if $i - 1$ is L-type (where we view T as a cyclic string). An* LMS-substring *is a cyclic substring $T[i, j]$ of T such that both i and j are LMS-positions, but there is no LMS-position between i and j. Given a conjugate $conj_i(T)$, its* LMS-prefix *is the cyclic substring from i to the first LMS-position strictly greater than i (viewed cyclically).*

Since T is primitive, no two conjugates are equal, and in particular, no two adjacent conjugates are equal. Therefore, the type of every position of T is defined.

▶ **Example 5.** Continuing Example 1,

G	T	A	C	A	A	C	G		C	G	G	C	A	C	A	C	A	C	G	T
S	L	S	L	S	S	S	S		S	L	L	L	S	L	S	L	S	S	S	L
		$*$		$*$					$*$				$*$		$*$		$*$			

where we mark *LMS*-positions with a $*$. The *LMS*-substrings are ACA, $AACGGTA$, $CGGCA$, and $ACGTC$. The *LMS*-prefix of the conjugate $\text{conj}_7(T_1) = CGGTACAA$ is $CGGTA$.

▶ **Lemma 6** (Cyclic type properties). *Let T be primitive string of length at least 2. Let a_1 be the smallest and a_σ the largest character of the alphabet. Then the following hold, where T is viewed cyclically:*

1. *if $T[i] < T[i+1]$, then i is of type S, and if $T[i] > T[i+1]$, then i is of type L,*
2. *if $T[i] = T[i+1]$, then the type of i is the same as the type of $i+1$,*
3. *i is of type S iff $T[i'] > T[i]$, where $i' = \min\{j \mid T[j] \neq T[i]\}$,*
4. *if $T[i] = a_1$, then i is of type S, and if $T[i] = a_\sigma$, then i is of type L.*

Proof. 1. follows from the fact that for all $b, c \in \Sigma$, if $b < c$ then for all $U, V \in \Sigma^*$, $bU \prec_\omega cV$; 2. follows by induction from the fact that for all $U, V \in \Sigma^*$, if $U \prec_\omega V$, then $cU \prec_\omega cV$; 3. and 4. follow from 2. by induction. ◀

▶ **Corollary 7** (Linear-time cyclic type assignment). *Let T be a primitive string of length at least 2. Then all positions can be assigned a type in altogether at most $2|T|$ steps.*

Proof. Once the type of one position is known, then the assignment can be done in one cyclic pass over T from right to left, by Lemma 6. Therefore, it suffices to find the type of one single position. Any position of character a_1 or of character a_σ will do; alternatively, any position i such that $T[i+1] \neq T[i]$, again by Lemma 6. Since T is primitive and has length at least 2, the latter must exist and can be found in at most one pass over T. ◀

Let N be the total length of the strings in $\mathcal{M}$. The algorithm constructs an initially empty array A of size N, which, at termination, will contain the GCA of $\mathcal{M}$. The algorithm also returns the set $\mathcal{I}$ containing the set of indices in A representing the positions of the strings of $\mathcal{M}$. The overall procedure consists of the following steps:

Algorithm `SAIS-for-`eBWT

Step 1 remove strings of length 1 from $\mathcal{M}$ (these will be added back at the end)
Step 2 assign cyclic types to all positions of strings from $\mathcal{M}$
Step 3 use procedure `Induced Sorting` to sort cyclic *LMS*-substrings
Step 4 assign names to cyclic *LMS*-substrings; if all distinct, go to Step 6
Step 5 recurse on new string multiset $\mathcal{M}'$, returning array A', map A' back to A
Step 6 use procedure `Induced Sorting` to sort all positions in $\mathcal{M}$, add length-1 strings in their respective positions, return $(A, \mathcal{I})$

At the heart of the algorithm is the procedure `Induced Sorting` of [33] (Algorithms 3.3 and 3.4), which is used once to sort the *LMS*-substrings (Step 3), and once to induce the order of all conjugates from the correct order of the *LMS*-positions (Step 6), as in the original SAIS. Before sketching this procedure, we need to define the order according to which the *LMS*-substrings are sorted in Step 2. Note that our definition of *LMS*-order is an extension of the *LMS*-order defined in [33], to *LMS*-prefixes. It can be proved that these definitions coincide for *LMS*-substrings.

▶ **Definition 8** (*LMS*-order). *Given two strings S and T, let U resp. V be their LMS-prefixes. We define $U <_{LMS} V$ if either V is a proper prefix of U, or neither is a proper prefix of the other and $U <_{\text{lex}} V$.*

The procedure `Induced Sorting` for the conjugates of the multiset is analogous to the original one, except that strings are viewed cyclically. First, the array A is subdivided into so-called *buckets*, one for each character. For $c \in \Sigma$, let n_c denote the total number of occurrences of the character c in the strings in $\mathcal{M}$. Then the buckets are $[1, n_{a_1}], [n_{a_1} + 1, n_{a_1} + n_{a_2}], \ldots, [N - n_{a_\sigma} + 1, N]$, i.e., the k-th bucket will contain all conjugates starting with character a_k. The procedure `Induced Sorting` first inserts all *LMS*-positions at the end of their respective buckets, then induces the L-type positions in a left-to-right scan of A, and finally, induces the S-type positions in a right-to-left scan of A, possibly overwriting previously inserted positions. We need two pointers for each bucket $\mathbf{b}$, *head*($\mathbf{b}$) and *tail*($\mathbf{b}$), pointing to the current first resp. last free position of the bucket.

Procedure `Induced Sorting` [33]

1. insert all *LMS*-positions at the end of their respective buckets; initialize *head*($\mathbf{b}$), *tail*($\mathbf{b}$) to the first resp. last position of the bucket, for all buckets $\mathbf{b}$
2. induce the L-type positions in a left-to-right scan of A: for i from 1 to $N-1$, if $A[i] = (j, d)$ then $A[\mathit{head}(\mathit{bucket}(T_d[j-1]))] \leftarrow (j-1, d)$; increment *head*(*bucket*($T_d[j-1]$))
3. induce the S-type positions in a right-to-left scan of A: for i from N to 2, if $A[i] = (j, d)$ then $A[\mathit{tail}(\mathit{bucket}(T_d[j-1]))] \leftarrow (j-1, d)$; decrement *tail*(*bucket*($T_d[j-1]$))

At the end of this procedure, the *LMS*-substrings are listed in correct relative *LMS*-order (see Lemma 10), and they can be named according to their rank. For the recursive step, we define, for $i = 1, \ldots, m$, a new string T'_i, where each *LMS*-substring of T_i is replaced by its rank. The algorithm is called recursively on $\mathcal{M}' = \{T'_1, \ldots, T'_m\}$ (Step 5).

Finally (Step 6), the array $A' = \text{GCA}(\mathcal{M}')$ from the recursive step is mapped back into the original array, resulting in the placement of the *LMS*-substrings in their correct relative order. This is then used to induce the full array A. All length-1 strings T_i which were removed in Step 1 can now be inserted between the L- and S-type positions in their bucket (Lemma 9). See Figure 1 for a full example.

3.1 Correctness and running time

The following lemma shows that the individual steps of `Induced Sorting` are applicable for the ω-order on conjugates of a multiset (part 1), that L-type conjugates (of all strings) come before the S-type conjugates within the same bucket (part 2), and that length-1 strings are placed between S-type and L-type conjugates (part 3). The second property was originally proved for the lexicographic order between suffixes in [22]:

▶ **Lemma 9** (Induced sorting for multisets). *Let $U, V \in \Sigma^*$.*

1. *If $U \prec_\omega V$, then for all $c \in \Sigma$, $cU \prec_\omega cV$.*
2. *If $U[i] = V[j]$, i is an L-type position, and j an S-type position, then $\mathit{conj}_i(U) \prec_\omega \mathit{conj}_j(V)$.*
3. *If $U[i] = V[j] = c$, i is an L-type position, and j an S-type position, then $\mathit{conj}_i(U) \prec_\omega c \prec_\omega \mathit{conj}_j(V)$.*

Proof. *1.* follows directly from the definition of ω-order. *3.* implies *2.* For *3.*, let i' be the nearest character following i in U such that $U[i'] \neq c$. By Lemma 6, $U[i'] < c$, and thus

Step 1 - remove strings of length 1 from $\mathcal{M}$

Step 2 - assign cyclic types to all positions of strings from $\mathcal{M}$

	T_1								T_2												T_3
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	1
$\mathcal{M} = \{$	G	T	A	C	A	A	C	G ,	C	G	G	C	A	C	A	C	A	C	G	T ,	C $\}$
	S	L	S	L	S	S	S	S	S	L	L	L	S	L	S	L	S	S	S	L	
			*		*				*				*		*		*				

Step 3 - use procedure `Induced Sorting` to sort cyclic *LMS*-substrings

	A						C					G				T	
S*		5	7	9	3	5					1						
		2	2	2	1	1					2						
L							4	6	8	4		3	2			2	12
$\longrightarrow$							2	2	2	1		2	2			1	2
S	5	5	7	3	6	9					1	7	10		8	1	11
$\longleftarrow$	1	2	2	1	1	2					2	1	2		1	1	2

	A						C					G				T	
	5	5	7	3	6	9	4	6	8	4	1	7	10	3	2	8	1	11	2	12
	1	2	2	1	1	2	2	2	2	1	2	1	2	2	2	1	1	2	1	2
	*	*	*	*		*					*									

Step 4 - Assign names to cyclic *LMS*-substrings

A A C G G T A	a
A C A	b
A C G T C	c
C G G C A	d

$T'_1 = b\ \ a$
$T'_2 = d\ \ b\ \ b\ \ a$

Step 5 - recurse on new string multiset $\mathcal{M}'$

	T'_1		T'_2				
	1	2	1	2	3	4	
$\mathcal{M}' = \{$	b	a ,	d	b	b	c $\}$	
	L	S	L	S	S	S	
		*		*			

	a	b		c	d
S*	2		2		
	1		2		
L		1			1
$\longrightarrow$		1			2
S	2		2	3	4
$\longleftarrow$	1		2	2	2

A'	2	1	2	3	4	1
	1	1	2	2	2	2

Step 6 - use procedure `Induced Sorting` to sort cyclic *LMS*-substrings, add length-1 strings in their respective positions

	A					C					G				T	
S*		5	3	5	7	9				1						
		1	1	2	2	2				2						
L						4	4	6	8		3	2			2	12
$\longrightarrow$						1	2	2	2		2	2			1	2
S	5	3	5	7	6	9			1	1	7	10		8	1	11
$\longleftarrow$	1	1	2	2	1	2			3	2	1	2		1	1	2

T_3

Generalized conjugate array of $\mathcal{M}$

GCA	5	3	5	7	6	9	4	4	6	8	**1**	**1**	7	10	3	2	8	**1**	11	2	12	
	1	1	2	2	1	2	1	2	2	2	**3**	**2**	1	2	2	2	1	**1**	2	1	2	
eBWT	*C*	*T*	*C*	*C*	*A*	*C*	*A*	*G*	*A*	*A*	*C*	*T*	*A*	A	*G*	*C*	*C*	*G*	C	*G*	G	

Figure 1 The algorithm `SAIS-for-`eBWT on Example 1. Start positions of input strings are marked in bold.

$\text{conj}_i(U) <_{\text{lex}} c^{|U|}$, and therefore, $conj_i(U) \prec_\omega c$. Analogously, if j' is the next character in V s.t. $V[j'] \neq c$, then by Lemma 6, $V[j'] > c$, and therefore, $c \prec_\omega conj_j(V)$. ◀

Next, we show that after applying procedure `Induced Sorting`, the conjugates will appear in A such that they are correctly sorted w.r.t. to the *LMS*-order of their *LMS*-prefixes, while the order in which conjugates with identical *LMS*-prefixes appear in A is determined by the input order of the *LMS*-positions.

▶ **Lemma 10** (Extension of Thm. 3.12 of [33]). *Let $T_1, T_2 \in \mathcal{M}$, let U be the LMS-prefix of $conj_i(T_1)$, with i' the last position of U; let V be the LMS-prefix of $conj_j(T_2)$, and j' the last position of V. Let k_1 be the position of $conj_i(T_1)$ in array A after the procedure* `Induced Sorting`*, and k_2 that of $conj_j(T_2)$.*

1. *If $U <_{LMS} V$, then $k_1 < k_2$.*
2. *If $U = V$, then $k_1 < k_2$ if and only if $conj_{i'}(T_1)$ was placed before $conj_{j'}(T_2)$ at the start of the procedure.*

Proof. Both claims follow from Lemma 9, and the fact that from one *LMS*-position to the previous one, there is exactly one run of L-type positions, preceded by one run of S-type positions. ◀

The next lemma shows that the *LMS*-order of the *LMS*-prefixes respects the ω-order.

▶ **Lemma 11.** *Let $S, T \in \Sigma^*$, let U be the LMS-prefix of S and V the LMS-prefix of T. If $U <_{LMS} V$ then $S \prec_\omega T$.*

Proof. If neither U nor V is a proper prefix one of the other, then there exists an index i s.t. $S[i] = U[i] < V[i] = T[i]$, and therefore, $S \prec_\omega T$. Otherwise, V is a proper prefix of U. Let $i = |V|$ and $c = V[i]$. Since both U and V are *LMS*-prefixes, with i being the last position of V but not of U, this implies that $V[i] = T[i]$ is of type S, while $U[i] = S[i]$ is of type L. Let j be the next character in S s.t. $S[j] \neq c$, and k be the next character in T s.t. $T[k] \neq c$. By Lemma 6, $S[j] < c$, $T[k] > c$, and by definition of j, k all characters inbetween equal c. Then for $i' = \min(j, k)$, we have $S[i'] < T[i']$, with i' being the first position where S and T differ. Therefore, $S \prec_\omega T$. ◀

▶ **Theorem 12.** *Algorithm* `SAIS-for-`eBWT *correctly computes the* GCA *and* eBWT *of a multiset of strings $\mathcal{M}$ in time $O(N)$, where N is the total length of the strings in $\mathcal{M}$.*

Proof. By Lemma 6, Step 2 correctly assigns the types. Step 3 correctly sorts the *LMS*-substrings by Lemma 10. It follows from Lemma 11 that the order of the conjugates of the new strings T'_i coincides with the relative order of the *LMS*-conjugates. In Step 6, the *LMS*-conjugates are placed in A in correct relative order from the recursion; by Lemmas 10 and 11, this results in the correct placement of all conjugates of strings of length > 1, while the positioning of the length-1 strings is given by Lemma 9.

For the running time, note that Step 1 takes time at most $2N$. The `Induced Sorting` procedure also runs in linear time $O(N)$. Finally, since no two *LMS*-positions are consecutive, and we remove strings of length 1, the problem size in the recursion step is reduced to at most $N/2$. ◀

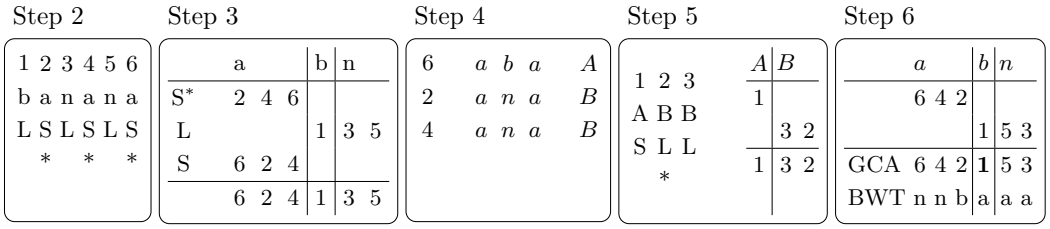


Figure 2 Example for computing the BWT for one string, start index marked in bold.

3.2 Computing the BWT for one single string

The special case where $\mathcal{M}$ consists of one single string leads to a new algorithm for computing the BWT, since for a singleton set, the eBWT coincides with the BWT. To the best of our knowledge, this is the first linear-time algorithm for computing the BWT *of a string without an end-of-string character* that uses neither Lyndon rotations nor end-of-string characters.

We demonstrate the algorithm on a well-known example, $T = banana$. We get the following types, from left to right: $LSLSLS$, and all three S-type positions are *LMS*. We insert $2, 4, 6$ into the array A; after the left-to-right pass, indices are in the order $2, 4, 6, 1, 3, 5$, and after the right-to-left pass, in the order $6, 2, 4, 1, 3, 5$. The *LMS*-substring *aba* (pos. 6) gets the name A, and the *LMS*-substring *ana* (pos. 2,4) gets the name B. In the recursive step, the new string $T' = ABB$, with types SLL and only one *LMS*-position 1, the GCA gets induced in just one pass: $1, 3, 2$. This maps back to the original string: $6, 2, 4$, and one more pass over the array A results in $6, 4, 2, 1, 5, 3$ and the BWT *nnbaaa*. See Figure 2.

4 eBWT and prefix-free parsing

In this section, we show how to extend the prefix-free parsing to build the eBWT. We define the *cyclic prefix-free parse* for a multiset of strings $\mathcal{M} = \{T_1, T_2, \ldots, T_m\}$ (with $|T_i| = n_i$, $1 \leq i \leq m$) as the multiset of parses $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ with dictionary D, where we consider T_i as circular, and P_i is the parse of T_i. We denote by p_i the length of the parse P_i.

Next, given a positive integer w, let E be a set of strings of length w called *trigger strings*. We assume that each string $T_h \in \mathcal{M}$ has length at least w and at least one cyclic factor in E.

We divide each string $T_h \in \mathcal{M}$ into overlapping phrases as follows: a phrase is a circular factor of T_h of length $> w$ that starts and ends with a trigger string and has no internal occurrences of a trigger string. The set of phrases obtained from strings in $\mathcal{M}$ is the dictionary D. The parse P_h can be computed from the string T_h by replacing each occurrence of a phrase in T_h with its lexicographic rank in D.

▶ **Example 13.** Let $\mathcal{M} = \{T_1 : CACGTGCTAT, T_2 : CCACTTGCTAGA, T_3 : CACTTGCTAT\}$ and let $E = \{AC, GC\}$. The dictionary D of the multiset of parses $\mathcal{P}$ of $\mathcal{M}$ is $D = \{ACCAC, ACGTGC, ACTTGC, GCTAGAC, GCTATCAC\}$ and $\mathcal{P} = \{2\ 5, 3\ 4\ 1, 3\ 5\}$, where $P_2 = 2\ 5$ means that the parsing of T_2 is given by the second and fifth phrases of the dictionary. Note that the string T_2 has a trigger string AC that spans the first position of T_2.

We denote by $\mathcal{S}$ the set of suffixes of D having length greater than w. The first important property of the dictionary D is that the set $\mathcal{S}$ *prefix-free*, i.e., no string in $\mathcal{S}$ is prefix of another string of $\mathcal{S}$. This follows directly from [8].

▶ **Example 14.** Continuing Example 13, we have that

$$\mathcal{S} = \{ACCAC, ACGTGC, ACTTGC, AGAC, ATCAC, CAC, CCAC, CGTGC, \\ CTAGAC, CTATCAC, CTTGC, GAC, GCTAGAC, GCTATCAC, GTGC, \\ TAGAC, TATCAC, TCAC, TGC, TTGC\}$$

The computation of eBWT from the prefix-free parse consists of three steps: computing the cyclic prefix-free parse of $\mathcal{M}$ (denoted as $\mathcal{P}$), computing the eBWT of $\mathcal{P}$ by using the algorithm described in Section 3; and lastly, computing the eBWT of $\mathcal{M}$ from the eBWT of $\mathcal{P}$ using the lexicographically sorted dictionary $D = \{D_1, D_2, \ldots, D_{|D|}\}$ and its prefix-free suffix set $\mathcal{S}$. We now describe the last step as follows. We define δ as the function that uniquely maps each character of $T_h[j]$ to the pair (i, k), where with $1 \leq i \leq p_h$, $k > w$, and $T_h[j]$ corresponds to the k-th character of the $P_h[i]$-th phrase of D. We call i and k the *position* and the *offset* of $T_h[j]$, respectively. Furthermore, we define α as the function that uniquely associates to each conjugate $conj_j(T_h)$ the element $s \in \mathcal{S}$ such that s is the k-th suffix of the $P_h[i]$-th element of D, where $(i, k) = \delta(T_h[j])$. By extension, i and k are also called the *position* and the *offset* of the suffix $\alpha(\text{conj}_j(T_h))$.

▶ **Example 15.** In Example 13, $\delta(T_2[4]) = (1, 2)$ since $T_2[4]$ is the second character (offset 2) of the phrase $ACTTGC$, which is the first phrase (position 1) of P_2. Moreover, $\alpha(\text{conj}_4(T_2)) = CTTGC$ since $CTTGC$ is the suffix of D_3, which is prefix of $\text{conj}_4(T_2) = CTTGCTAGACCA$.

▶ **Lemma 16.** *Given two strings $T_g, T_h \in \mathcal{M}$, if $\alpha(conj_i(T_g)) <_{\text{lex}} \alpha(conj_i(T_h))$ it follows that $conj_i(T_g) \prec_\omega conj_j(T_h)$.*

Proof. It follows from the definition of α that $\alpha(conj_i(T_g))$ and $\alpha(conj_j(T_h))$ are prefixes of $conj_i(T_g)$ and $conj_j(T_h)$, respectively. ◀

▶ **Proposition 17.** *Given two strings $T_g, T_h \in \mathcal{M}$. Let $conj_i(T_g)$ and $conj_j(T_h)$ be the i-th and j-th conjugates of T_g and T_h, respectively, and let $(i', g') = \delta(T_g[i])$ and $(j', h') = \delta(T_h[j])$. Then $conj_i(T_g) \prec_\omega conj_j(T_h)$ if and only if either $\alpha(conj_i(T_g)) <_{\text{lex}} \alpha(conj_j(T_h))$, or $conj_{i'+1}(P_g) \prec_\omega conj_{j'+1}(P_h)$, i.e., $P_g[i']$ precedes $P_h[j']$ in* eBWT($\mathcal{P}$).

Proof. By definition of α, $\text{conj}_i(T_g) = \alpha(\text{conj}_i(T_g))T_g[i + g'']T_g[i + g'' + 1] \ldots T_g[i - 1]$ and $\text{conj}_j(T_h) = \alpha(\text{conj}_j(T_h))T_h[j + h'']T_h[j + h'' + 1] \ldots T_h[j - 1]$, where $g'' = |\alpha(\text{conj}_i(T_g))|$ and $h'' = |\alpha(\text{conj}_j(T_h))|$, respectively. Moreover, $\text{conj}_i(T_g) \prec_\omega \text{conj}_j(T_h)$ if and only if either $\alpha(\text{conj}_j(T_h)) <_{\text{lex}} \alpha(\text{conj}_j(T_h))$ or $\text{conj}_{i+g''-w}(T_g) \prec_\omega \text{conj}_{j+h''-w}(T_h)$, where w is the length of trigger strings. It is easy to verify that the position of $T_g[i + g'' - w]$ and $T_h[j + h'' - w]$ is $i' + 1$ and $j' + 1$, respectively. Moreover, since $T_g[i + g'' - w]$ and $T_h[j + h'' - w]$ are the first character of a phrase, we have that $\text{conj}_{i+g''-w}(T_g) \prec_\omega \text{conj}_{j+h''-w}(T_h)$ if and only if $\text{conj}_{i'+1}(P_g) \prec_\omega \text{conj}_{j'+1}(P_h)$. ◀

Next, using Proposition 17, we define how to build the eBWT of the multiset of strings $\mathcal{M}$ from $\mathcal{P}$ and D. First, we note that we will iterate through all the suffixes in $\mathcal{S}$ in lexicographic order, and build the eBWT of $\mathcal{M}$ in blocks corresponding to the suffixes in $\mathcal{S}$. Hence, it follows that we only need to describe how to build an eBWT block corresponding to a suffix $s \in \mathcal{S}$. Given $s \in \mathcal{S}$, we let $\mathcal{S}_s$ be the set of the lexicographic ranks of the phrases of D that have s as a suffix, i.e., $\mathcal{S}_s = \{i \mid 1 \leq i \leq |D|, s \text{ is a suffix of } D_i \in D\}$. Moreover, given the string $T_h \in \mathcal{M}$, we let $\text{conj}_i(T_h)$ be the i-th conjugate of T_h, let j and k be the position and offset of $T_h[i]$, and lastly, let p be the position of $P_h[j]$ in eBWT($\mathcal{P}$). We define $f(p, k) = D_{P_h[j]}[k - 1]$ if $k > 1$, otherwise $f(p, k) = D_{P_h[j-1]}[|D_{P_h[j-1]}| - w]$ where we view P_h as a cyclic string.

▶ **Example 18.** In Example 13, eBWT($\mathcal{P}$) = *4 5 1 5 3 2 3*. Let us consider $\text{conj}_4(T_2)$ and $\text{conj}_3(T_3)$ that are both mapped to the suffix CTT by the function α. By using Example 15, the position and the offset of $T_2[4]$ are 1 and 2, respectively. The position of $P_2[1] = 3$ in eBWT($\mathcal{P}$) is 5, because $\text{conj}_2(P_2) \prec_\omega \text{conj}_2(P_3)$. This implies that $\text{conj}_4(T_2) \prec_\omega \text{conj}_3(T_3)$ by Proposition 17. Furthermore, $f(5, 2) = T_2[3] = A$.

Finally, we let $\mathcal{O}_s$ be the set of pairs (p, c) such that for all $d \in \mathcal{S}_s$, p is the position of an occurrence of d in eBWT($\mathcal{P}$), and c is the character resulting the application of the f function considering as k the offset of s in D_d, i.e., $c = f(p, |D_d| - |s| + 1)$. Formally, $\mathcal{O}_s = \{(p, f(p, |D_{\text{eBWT}(\mathcal{P})[p]}| - |s| + 1) \mid \text{eBWT}(\mathcal{P})[p] \in \mathcal{S}_s\}$.

▶ **Example 19.** In Example 13, if $s = CAC \in \mathcal{S}$ and $\mathcal{S}_s = \{1, 5\}$, where *1* : $ACCAC$ and *5* : $GCTATCAC$, then it follows that $\mathcal{O}_s = \{(3, C), (2, T), (4, T)\}$ since the phrase *1* is in position 3 in the eBWT($\mathcal{P}$) and the suffix CAC starts in position 3 of D_1, the character preceding the occurrences of CAC corresponding to the phrase *1* is C. Analogously, the phrase *5* is in positions 2 and 4 in the eBWT($\mathcal{P}$) and the suffix CAC starts in position 6 of D_5, hence the character preceding the occurrences of CAC corresponding to the phrase *5* is T.

To build the eBWT block corresponding to $s \in \mathcal{S}$, we scan the set $\mathcal{O}_s$ in increasing order of the first element of the pair, i.e., the position of the occurrence in eBWT($\mathcal{P}$), and concatenate the values of the second element of the pair, i.e., the character preceding the occurrence of s in T_h.Note that if all the occurrences in $\mathcal{O}_s$ are preceded by the same character c, we do not need to iterate through all the occurrences but rather concatenate $|\mathcal{O}_s|$ copies of the character c.

▶ **Example 20.** In Example 13, eBWT($\mathcal{M}$) = $GCCCTTT\underline{TCT}AAGGGAAATTTCCCCAATGTCC$, where the block of the eBWT corresponding to the suffix $s = CAC \in \mathcal{S}$ is underlined. Given $\mathcal{O}_s = \{(3, C), (2, T), (4, T)\}$, we generate the block by sorting $\mathcal{O}_s$ by the first element of each pair – resulting in $\mathcal{O}_s = \{(2, T), (3, C), (4, T)\}$ – and concatenating the second element of each pair obtaining TCT.

Keeping track of the first rotations.

So far, we showed how to compute the first component of the eBWT. Now we show how to compute the second component of the eBWT i.e., the set of indices marking the first rotation of each string. The idea is to keep track of the starting positions of each text in the parse, by marking the offset of the first position of each string in the last phrase of the corresponding parse. We propagate this information during the computation of the eBWT of the parse. When scanning the suffixes of $\mathcal{S}$, we check if one of the phrases sharing the same suffix $s \in \mathcal{S}$ is marked as a phrase containing a starting position, and if the offset of the starting position coincides with the offset of the suffix. If so, when generating the elements of $\mathcal{O}_s$, we mark the element corresponding to the occurrence of the first rotation of a string, and we output the index of the eBWT when that element is processed.

Implementation notes.

In practice, as in [8], we implicitly select the set of trigger strings E, by rolling a Karp-Rabin hash over consecutive windows of size w and take as a trigger strings of length w all windows such that their hash value is congruent 0 modulo a parameter p. In our version of the PFP, we also need to ensure that there is at least one trigger string on each sequence of the

collection. Hence, we change the way we select the trigger strings as follows. We define a set $\mathcal{D}$ of remainders and we select a window of length w as a trigger string with hash value congruent d modulo p if $d \in \mathcal{D}$. Note that if we set $\mathcal{D} = \{0\}$ we obtain the same set of trigger strings as in the original definition. We choose the set $\mathcal{D}$ in a greedy way. We start with $\mathcal{D} = \{0\}$ by scanning the set of sequences and checking if the current sequence has a trigger string according to the current $\mathcal{D}$. As soon as we find one, we move to the next sequence. If we don't find any trigger string, we take the reminder of the last window we checked, and we include it in the set $\mathcal{D}$.

We note that we consider $\mathcal{S}$ to be the set of suffixes of the phrases of D such that $s \in \mathcal{S}$ is not a phrase in D nor it has length smaller than w in the implementation. This allows us to compute f more efficiently since we can compute the preceding character of all the occurrences of a suffix in $\mathcal{S}$ from its corresponding phrase in D. Moreover, as in [8], for each phrase in D, we keep an ordered list of their occurrences in the eBWT of the parse. For a given suffix $s \in \mathcal{S}$, we do not generate $\mathcal{O}_s$ all at once and sort it – but rather, we visit the elements of $\mathcal{O}_s$ in order using a min-heap as we merge the ordered lists of the occurrences in the eBWT of the parse of the phrases that share the same suffix s.

5 Experimental results

We implemented the algorithm for building the eBWT and measured its performance on real biological data. We performed the experiments on a server with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with 16 cores and 62 gigabytes of RAM running Ubuntu 16.04 (64bit, kernel 4.4.0). The compiler was `g++` version 9.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. We recorded the runtime and memory usage using the wall clock time, CPU time, and maximum resident set size from `/usr/bin/time`. The source code is available online at: `https://github.com/davidecenzato/PFP-eBWT`.

We compared our method (`pfpebwt`) with the BCR algorithm implementation of [25] (`ropebwt2`), `gsufsort` [27], and `egap` [14]. We did not compare against `G2BWT` [13], `lba` [6], and `BCR` [4] since they are currently implemented only for short reads[2]. We did not compare against `egsa` [28] since it is the predecessor of `egap` or against methods that construct the BWT of a multiset of strings using one of the methods we evaluated against, i.e., `LiME` [18], `BEETL` [11], `metaBEETL` [1], and `ebwt2snp` [36, 37].

5.1 Datasets

We evaluated our method using 2,048 copies of human chromosomes 19 from the 1000 Genomes Project [42]; 10,000 *Salmonella* genomes taken from the GenomeTrakr project [41], and 400,000 SARS-CoV2 genomes from EBI's COVID-19 data portal [12]. The sequence data for the *Salmonella* genomes were assembled, and the assembled sequences that had length less than 500 bp were removed. In addition, we note that we replaced all degenerate bases in the SARS-CoV2 genomes with N's and filtered all sequences with more than 95% N's. A brief description of the datasets is reported in Table 1. We used 12 sets of variants of human chromosome 19 (`chr19`), containing 2^i variants for $i = 0, \ldots, 11$ respectively. We used 6 collections of *Salmonella* genomes (`salmonella`) containing 50, 100, 500, 1,000, 5,000, and 10,000 genomes respectively. We used 5 sets of SARS-CoV2 genomes (`sars-cov2`) containing

[2] `G2BWT` crashed and `BCR` did not terminate within 48 hours with the smallest of each dataset; `lba` works only with sequences of length up to 255

Name	Description	σ	$n/10^6$	n/r
`chr19`	Human chromosome 19	5	121,086.62	2199.21
`salmonella`	*Salmonella* genomes	4	48,791.75	112.72
`sars-cov2`	SARS-CoV2 genomes	5	11,930.96	1424.65

Table 1 Datasets used in the experiments. We give the alphabet size in column 3. We report the length of the file and the ratio of the length to the number of runs in the eBWT in columns 4 and 5, respectively.

25,000, 50,000, 100,000, 200,000, 400,000 genomes respectively. Each collection is a superset of the previous one.

5.2 Setup

We run `pfpebwt` and `ropebwt2` with 16 threads, and `gsufsort` and `egap` with a single thread since they do not support multi-threading. Using `pfpebwt`, we set $w = 10$ and $p = 100$. Furthermore, for `pfpebwt` on the `salmonella` dataset, we used up to three different remainders to build the eBWT. We used `ropebwt2` with the `-R` flag to exclude the reverse complement of the sequences from the computation of the BWT. All other methods were run with default parameters.

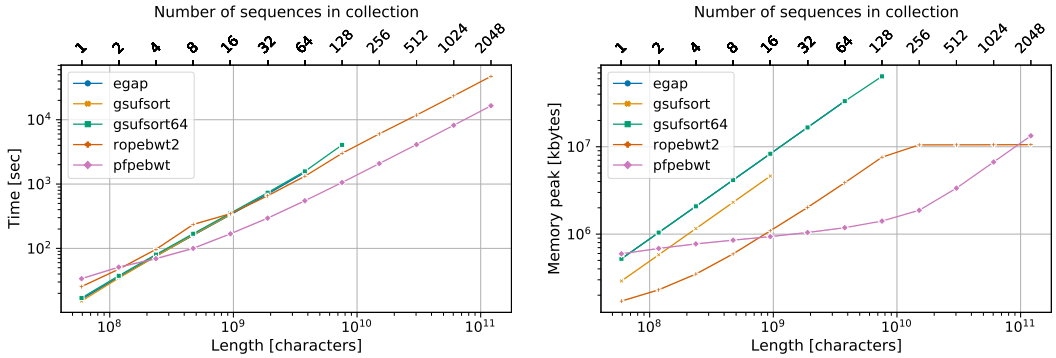
We repeated each experiment five times, and report the average CPU time and peak memory for the set of chromosomes 19 up to 64 distinct variants, for *Salmonella* up to 1,000 sequences, and for all SARS-CoV2. The experiments that exceeded 48 hours of wall clock time or exceeded 62 GB of memory were omitted for further consideration, e.g., 128 sequences of `chr19`, 5000 sequences of `salmonella` and 400,000 sequences of `sars-cov2` for `egap`. Furthermore, `gsufsort` failed to successfully build the eBWT for 256 sequences of `chr19`, 5000 sequences of `salmonella`, and 400,000 sequences of `sars-cov2` or more, because it exceeded the 62GB memory limit.

5.3 Results

In Figures 3, 4, and 5 we illustrate the construction time and memory usage to build the eBWT and the BWT of collections of strings for the chromosome 19 dataset, the *Salmonella* dataset, and the SARS-CoV2 dataset, respectively.

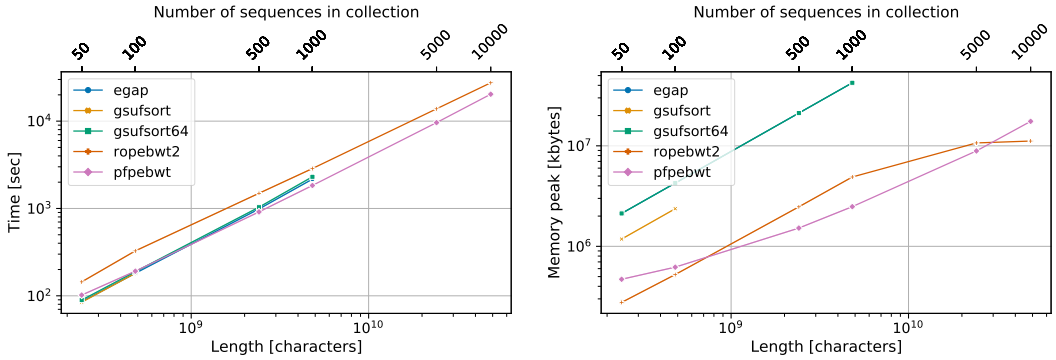
`pfpebwt` was the fastest method to build the eBWT of 4 or more sequences of chromosome 19, with a maximum speedup of 7.6x of wall-clock time and 2.9x of CPU time over `ropebwt2` on 256 sequences of chromosomes 19, 2.7x of CPU time over `egap` on 64 sequences, and 3.8x of CPU time over `gsufsort` on 128 sequences. On *Salmonella* sequences, `pfpebwt` was always the fastest method, except for 10,000 sequences where `ropebwt2` was the fastest method on wall-clock time. `pfpebwt` had a maximum speedup of 3.0x of wall-clock time over `ropebwt2` on 100 sequences of `salmonella`. Considering the CPU time, `pfpebwt` was the fastest for $\geq$ 500 sequences with a maximum speedup of 1.7x over `ropebwt2` on 100 sequences and 1.2x over `gsufsort` and `egap` on 1,000 sequences. On SARS-CoV2 sequences, `pfpebwt` was always the fastest method, with a maximum speedup of 2.4x of wall-clock time over `ropebwt2` while a maximum speedup of 1.3x of CPU time over `ropebwt2` on 400,000 sequences, 2.9x over `gsufsort` and 2.7x over `egap` on 200,000 sequences of SARS-CoV2.

Considering the peak memory, on the chromosomes 19 dataset, `ropebwt2` used the smallest amount of memory for 1, 2, 4, 8, and 2,048 sequences, while `pfpebwt` used the smallest amount of memory in all other cases. `pfpebwt` used a maximum of 5.6x less memory than `ropebwt2`



(a) Construction time. **(b)** Peak memory.

Figure 3 Chromosome 19 dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

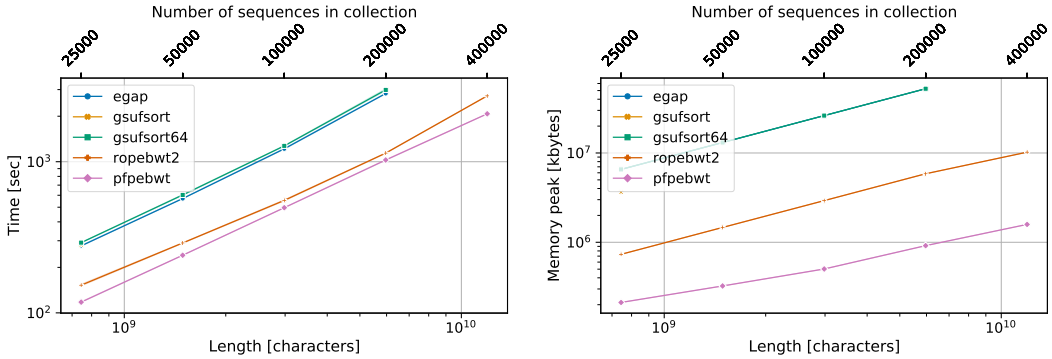


(a) Construction time. **(b)** Peak memory.

Figure 4 *Salmonella* dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

on 256 sequences of chromosomes 19, 28.0x less than `egap` on 64 sequences, and 45.3x less than `gsufsort` on 128 sequences. On *Salmonella* sequences, `pfpebwt` used more memory than `ropebwt2` for 50, 100, and 10,000 sequences, while `pfpebwt` used the smallest amount of memory on all other cases. The largest gap between `ropebwt2` and `pfpebwt` memory peak is of 1.7x on 50 sequences. On the other hand, `pfpebwt` used a maximum of 17.0x less memory than `egap` and `gsufsort` on 1,000 sequences. On SARS-CoV2 sequences, `pfpebwt` always used the smallest amount of memory, with a maximum of 6.4x less memory than `ropebwt2` on 25,000 sequences of SARS-CoV2, 57.1x over `gsufsort` and `egap` on 200,000 sequences.

The memory peak of `ropebwt2` is given by the default buffer size of 10 GB, and the size of the run-length encoded BWT stored in the rope data structure. This explains the memory plateau on 10.5 GB of `ropebwt2` on the chromosomes 19 dataset. However, `ropebwt2` is able only to produce the BWT of the input sequence collection, while `pfpebwt` can be trivially extended to produce also the samples of the conjugate array at the run boundaries with negligible additional costs in terms of time and peak memory.



(a) Construction time.

(b) Peak memory.

Figure 5 SARS-CoV2 dataset construction CPU time and peak memory usage. We compare `pfpebwt` with `ropebwt2`, `gsufsort`, and `egap`.

6 Conclusion

We described the first linear-time algorithm for building the eBWT of a collection of strings that does not require the manipulation of the input sequence, i.e., neither the addition of an end-of-string character, nor computing and sorting the Lyndon rotations of the input strings. We also combined our algorithm with an extension of the prefix-free parsing to enable scalable construction of the eBWT. We demonstrated `pfpebwt` was efficient with respect to both memory and time when the input is highly repetitive. Lastly, we curated a novel dataset of 400,000 SARS-CoV2 genomes from EBI's COVID-19 data portal, which we believe will be important for future benchmarking of data structures that have potential use in bioinformatics.

References

1 C. Ander, O.B. Schulz-Trieglaff, J. Stoye, and A.J. Cox. metaBEETL: high-throughput analysis of heterogeneous microbial populations from shotgun DNA sequences. *BMC Bioinf*, 14(5):S2, 2013.

2 H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Constructing the bijective and the extended Burrows-Wheeler-Transform in linear time. In *Proc. of CPM*, 2021.

3 J.B. Barwell, R.B.G. O'Sullivan, L.K. Mansbridge, J.M. Lowry, and H.R. Dorkins. Challenges in implementing genomic medicine: the 100,000 Genomes Project. *J Transl Genet Genome*, 2(13), 2018.

4 M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor Comput Sci*, 483:134–148, 2013.

5 A.M. Berner, G.J. Morrissey, and N. Murugaesu. Clinical analysis of whole genome sequencing in cancer patients. *Curr Genet Med Rep*, 7:136–143, 2019.

6 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theor. Comput. Sci.*, 862:42–58, 2021.

7 Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. of the Symposium on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 60–72. SIAM, 2021.

8 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big bwts. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.

9 M. Burrows and D.J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

10 Davide Cenzato and Zsuzsanna Lipták. On different variants of the extended Burrows-Wheeler-Transform. Unpublished manuscript, 2021.

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

12 The COVID-19 Data Portal. Available at `https://www.covid19dataportal.org/`. Accessed 17-05-2021.

13 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the extended BWT from grammar-compressed DNA sequencing reads. *CoRR*, abs/2102.03961, 2021.

14 L. Egidi, F. Louza, G. Manzini, and G.P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol Biol*, 14(1):1–15, 2019.

15 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling repair with rsync. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *26th International Symposium on String Processing and Information Retrieval (SPIRE 2019)*, volume 11811 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2019.

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

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

18 V. Guerrini, F.A. Louza, and G. Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinfo*, 21(299), 2020.

19 V. Guerrini and G. Rosone. Lightweight Metagenomic Classification via eBWT. In *Proc of WABI*, pages 112–124, 2019.

20 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient Algorithm for Circular Burrows-Wheeler Transform. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2012.

21 D. Knuth, J.H. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, 1977.

22 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.

23 G. Kucherov, L. Tóthmérész, and S. Vialette. On the combinatorics of suffix arrays. *Inf Process Lett*, 113(22-24):915–920, 2013.

24 A. Kuhnle et al. Efficient construction of a complete index for pan-genomics read alignment. In *Proc. of RECOMB*, pages 158–173, 2019.

25 H. Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

26 F. Louza, S. Gog, and G. P. Telles. *Construction of Fundamental Data Structures for Strings*. Springer International Publishing, 2020.

27 F.A. Louza, G.P. Telles, S. Gog, N. Prezza, and G. Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol Biol*, 15(1):1–5, 2020.

28 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26:1–26:16, 2017.

29 U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J Comput*, 22(5):935–948, 1993.

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

31 S. Mantaci, A. Restivo, and M. Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf Process Lett*, 86(5):241–246, 2003.

32 G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

33 G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans Comput*, 60(10):1471–1484, 2011.

34 E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

35 D. Perrin and A. Restivo. Enumerative combinatorics on words. In *Handbook of Enumerative Combinatorics, ed. by Miklos Bona*. 2015.

36 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms Mol Biol*, 14(1):1–13, 2019.

37 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler Transform. *BMC Bioinform*, 21-S(8):260, 2020.

38 A. Rhie et al. Towards complete and error-free genome assemblies of all vertebrate species. *Nature*, 592:737–0746, 2021.

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

40 Z. D. Stephens et al. Big Data: Astronomical or Genomical? *PLOS Biology*, 13(7):e1002195, 2015.

41 E.L. Stevens et al. The public health impact of a publically available, environmental database of microbial genomes. *Front Microbiol*, 8:808, 2017.

42 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

43 C. Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the NHS. *Br Med J*, 361, 2018.

44 L. Yang, X. Zhang, and T. Wang. The Burrows–Wheeler similarity distribution between biological sequences based on Burrows–Wheeler transform. *J Theor Biol*, 262(4):742–749, 2010.

A eBWT missing examples

Full conjugate table for Example 1: $\mathcal{M} = \{\text{GTACAACG,CGGCACACACGT,C}\}$.

		GCA	$\preceq_\omega$-sorted conjugates
	1	(5,1)	AACGGTA**C**
	2	(3,1)	ACAACGG**T**
	3	(5,2)	ACACACGTCGG**C**
	4	(7,2)	ACACGTCGGCA**C**
	5	(6,1)	ACGGTAC**A**
	6	(9,2)	ACGTCGGCACA**C**
	7	(4,1)	CAACGGT**A**
	8	(4,2)	CACACACGTCG**G**
	9	(6,2)	CACACGTCGGC**A**
	10	(8,2)	CACGTCGGCAC**A**
→	11	(1,3)	**C**
→	12	(1,2)	CGGCACACACG**T**
	13	(7,1)	CGGTACA**A**
	14	(10,2)	CGTCGGCACAC**A**
	15	(3,2)	GCACACACGTC**G**
	16	(2,2)	GGCACACACGT**C**
	17	(8,1)	GGTACAA**C**
→	18	(1,1)	GTACAAC**G**
	19	(11,2)	GTCGGCACACA**C**
	20	(2,1)	TACAACG**G**
	21	(12,2)	TCGGCACACAC**G**

$$\text{eBWT}(\{\text{GTACAACG,CGGCACACACGT,C}\}) = \text{CTCCACAGAACTAAGCCGCGG}$$