Fast and Memory-Efficient BWT Construction of Repetitive Texts Using Lyndon Grammars

Jannik Olbrich
Ulm University, Germany

Abstract

The Burrows-Wheeler Transform (BWT) serves as the basis for many important sequence indexes. On very large datasets (e.g. genomic databases), classical BWT construction algorithms are often infeasible because they usually need to have the entire dataset in main memory. Fortunately, such large datasets are often highly repetitive. It can thus be beneficial to compute the BWT from a compressed representation. We propose an algorithm for computing the BWT via the Lyndon straight-line program, a grammar based on the standard factorization of Lyndon words. Our algorithm can also be used to compute the extended BWT (eBWT) of a multiset of sequences. We empirically evaluate our implementation and find that we can compute the BWT and eBWT of very large datasets faster and/or with less memory than competing methods.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Shared memory algorithms; Theory of computation → Data compression; Theory of computation → Sorting and searching; Mathematics of computing → Combinatorics on words

Keywords and phrases Burrows-Wheeler Transform, Grammar compression

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.60

Supplementary Material *Software (Source Code)*: `https://gitlab.com/qwerzuiop/lyndongrammar` [38], archived at `swh:1:dir:0274cae4abb893b49bc035ef3a57986a418afe40`

1 Introduction

The *Burrows-Wheeler Transform* (BWT) [13] is one of the most important data structures in sequence analysis and is applied in a wide range of fields from data compression to bioinformatics. It is obtained by assigning the ith symbol of the BWT to the last character of the ith lexicographically smallest conjugate of the input text. The BWT can be computed in linear time and space. The run-length compressed BWT (RLBWT) uses the fact that the number of equal-character runs in the BWT is small compared to the text length when the text is repetitive [13]. Mantaci et al. [32] extended the notion of the BWT to string collections: the *extended* BWT (eBWT) of a string collection consists of the last characters of the strings in $\mathcal{M}$ arranged in infinite periodic order (see Section 2 for definitions).

The BWT serves as the basis of several *self-indexes*, i.e., data structures supporting access to the text as well as pattern matching queries. One of the most successful such indexes is the FM-index [20], which is used e.g. in the important bioinformatics programs BWA [30] and Bowtie [28]. However, the FM-index uses space proportional to the size of the dataset and is thus not applicable for huge datasets that vastly exceed the size of main memory. Such datasets often contain data from thousands or millions of individuals from the same species and are therefore extremely repetitive. The *r-index* [22] is based on the RLBWT and is the first text index of size proportional to the number r of runs in the BWT. It can be constructed in space proportional to r from the RLBWT, which raises the problem of constructing the RLBWT of huge datasets with small working memory. For this reason, various tools have emerged that exploit the repetitiveness of these datasets.

The Lyndon grammar of a text is a straight-line program (SLP), i.e., a context-free grammar in Chomsky normal form that generates a single text, where each symbol corresponds to a node in the Lyndon tree. It was introduced in [25] as the basis for a self-index. In this

paper, we describe a method for efficiently computing several BWT variants of a text or a text collection via the input's Lyndon grammar and show that our method surpasses other tools with respect to time or memory used, especially when using multiple threads.

1.1 Related work

Boucher et al. [11] developed *prefix-free parsing (PFP)*, a dictionary-compression technique that results in a dictionary and parse from which the (RL)BWT can be constructed in space proportional to their total size and in time linear to the size of the dataset. While the dictionary typically grows slowly as the size of the dataset increases, the parse grows linearly with the input size (with a small constant chosen at runtime). Consequently, for extremely large and repetitive datasets, "the parse is more burdensome than the size of the dictionary" [41]. In an effort to remedy this, Oliva et al. [41] developed *recursive prefix-free parsing*, which applies PFP to the parse. PFP has also been used to construct the eBWT [10]. The currently fastest implementation for computing the suffix array SA (and BWT) for general texts is `libsais` and is based on the linear-time *suffix array induced sorting (SAIS)* algorithm [37, 40]. `ropebwt3` uses `libsais` to compute the BWT of chunks of the input and successively merges these BWTs [29]. Díaz-Domínguez and Navarro used grammar compression to maintain the intermediate data in the SAIS algorithm with low main memory usage, resulting in a linear-time semi-external algorithm that computes a variant of the eBWT [14, 17]. Specifically for a collection of sequences where each is similar to a given reference (e.g. human chromosomes), Masillo introduced `CMS-BWT`, which computes the BWT via *matching statistics* [35].

Lyndon words have previously been used in the construction of SA and play an important role in the original eBWT. In [33, 34], a strategy for computing SA was presented where "local suffixes" (i.e., substrings ending at boundaries between Lyndon factors) can be processed separately in each Lyndon factor. Later, Baier effectively generalized the underlying idea by showing that sorting the suffixes by their longest Lyndon prefixes can be used for linear time SA construction [2]. The *bijective* BWT (BBWT) is the eBWT of the text's Lyndon factors [23, 27, 32]. Bonomo et al. showed that it is possible to compute the eBWT via the BBWT and gave an $\mathcal{O}(N \log N / \log\log N)$ construction algorithm based on properties of Lyndon words [8, 9]. Later, Bannai et al. modified the SAIS algorithm to compute the BBWT in linear time [3]. Since publication, Baier's algorithm was improved in terms of time and memory usage [6, 39] and further generalized such that it can be used to compute eBWT and BBWT in linear time [40]. The principles used this generalized algorithm lie at the base of our algorithm for computing the BWT, BBWT and eBWT variants from a Lyndon grammar.

1.2 Our contributions

Let N be the size of a text or text collection, $n_{\max}$ the maximum size of a string in the text collection, and g the size of the Lyndon SLP of the text or text collection. We give two online in-memory algorithms for constructing the Lyndon SLP with $\mathcal{O}(Ng)$ and $\mathcal{O}\big(N \log g + g \log^2 g\big)$ worst-case time complexity, both using $\mathcal{O}(g)$ words of memory. Notably, the text or text collection is streamed from right to left and does not have to reside in main memory. Additionally, we give an expected linear-time algorithm that uses $\mathcal{O}(g + n_{\max})$ words of memory. Furthermore, we give an $\mathcal{O}(g)$ algorithm for sorting the symbols of a Lyndon SLP lexicographically by their generated strings, and an $\mathcal{O}(N)$ algorithm for constructing the (run-length compressed) BWT, BBWT, or eBWT from a Lyndon SLP. We implemented our algorithms and demonstrate empirically that we can construct the BWT, BBWT or eBWT of repetitive texts or text collections faster and/or with less space than competing methods.

2 Preliminaries

For $i, j \in \mathbb{N}_0$ we denote the set $\{k \in \mathbb{N}_0 : i \leq k \leq j\}$ by the interval notations $[i\,..\,j] = [i\,..\,j+1) = (i-1\,..\,j] = (i-1\,..\,j+1)$. For an array A we analogously denote the *subarray* from i to j by $\mathsf{A}[i\,..\,j] = \mathsf{A}[i\,..\,j+1) = \mathsf{A}(i-1\,..\,j] = \mathsf{A}(i-1\,..\,j+1) = \mathsf{A}[i]\mathsf{A}[i+1]\ldots\mathsf{A}[j]$. We use zero-based indexing, i.e., the first entry of the array A is $\mathsf{A}[0]$.

A *string* S of *length* n over an *alphabet* Σ is a sequence of n characters from Σ. We denote the length n of S by $|S|$ and the ith symbol of S by $S[i-1]$, i.e., strings are zero-indexed. In this paper we assume any string S of length n to be over a totally ordered and linearly sortable alphabet (i.e., the characters in S can be sorted in $\mathcal{O}(n)$). Analogous to arrays we denote the *substring* from i to j by $S[i\,..\,j] = S[i\,..\,j+1) = S(i-1\,..\,j] = S(i-1\,..\,j+1) = S[i]S[i+1]\ldots S[j]$. For $j > i$ we let $S[i\,..\,j]$ be the *empty string* ε. For two strings u and v and an integer $k \geq 0$ we let uv be the concatenation of u and v and denote the k-times concatenation of u by u^k. A string S is *primitive* if it is non-periodic, i.e., $S = w^k$ implies $w = S$ and $k = 1$. The *suffix* i of a string S of length n is the substring $S[i\,..\,n)$ and is denoted by $\mathrm{suf}_i(S)$. Similarly, the substring $S[0\,..\,i]$ is a *prefix* of S. A suffix (prefix) is *proper* if $i > 0$ ($i+1 < n$). For two possibly empty strings u and v, uv is a conjugate of vu.

We assume totally ordered alphabets. This induces a total order on strings. Specifically, we say a string S of length n is *lexicographically smaller* than another string T of length m if and only if there is some $\ell \leq \min\{n, m\}$ such that $S[0\,..\,\ell) = T[0\,..\,\ell)$ and either $n = \ell < m$ or $\ell < \min\{n, m\}$ and $S[\ell] < T[\ell]$, and write $S <_{lex} T$ in this case. A non-empty string S is in its *canonical form* if and only if it is lexicographically minimal among its conjugates. If S is additionally strictly lexicographically smaller than all of its other conjugates, S is a *Lyndon word*. Equivalently, S is a Lyndon word if and only if S is lexicographically smaller than all its proper suffixes [18].

In the following, we use `abbabcbcabb` as our running example.

▶ **Theorem 1** (Chen-Fox-Lyndon theorem [15]). *Any non-empty string S has a unique* Lyndon factorization*, that is, there is a unique sequence of Lyndon words (*Lyndon factors*) $v_1 \geq_{lex} \ldots \geq_{lex} v_k$ with $S = v_1 \ldots v_k$.*

The Lyndon factorization of our running example is $\texttt{abbabcbc}, \texttt{abb}$.

▶ **Definition 2** (Standard Factorization [15]). *The* standard factorization *of a Lyndon word w of length $|w| \geq 2$ is the tuple (u, v) where $w = uv$ and v is the longest proper suffix of w that is Lyndon. The standard factorization (u, v) of a Lyndon word w with $|w| \geq 2$ always exists and both u and v are Lyndon.*

The standard factorization of the first Lyndon factor of our running example is $(\texttt{abb}, \texttt{abcbc})$ as `abcbc` is Lyndon and no longer proper suffix is Lyndon.

▶ **Lemma 3** ([40, Lemma 3.20]). *Any Lyndon word w with $|w| \geq 2$ is of the form $w[0]w_{c_1} \cdots w_{c_k}$, where $w_{c_1} \geq_{lex} w_{c_2} \geq_{lex} \ldots \geq_{lex} w_{c_k}$ are the Lyndon factors of $\mathrm{suf}_1(w)$.*

Figure 1 illustrates the following definitions.

▶ **Definition 4** (next smaller suffix array, nss). *Let the* next smaller suffix *array* nss *of a string S be such that $\mathrm{nss}[i] = \min\{j \in (i\,..\,|S|] \mid \mathrm{suf}_j(S) <_{lex} \mathrm{suf}_i(S)\}$.*

▶ **Definition 5** (Infinite Periodic Order). *We write $S <_\omega T$ if and only if the infinite concatenation $S^\infty = SS\ldots$ is lexicographically smaller than the infinite concatenation $T^\infty = TT\ldots$.*

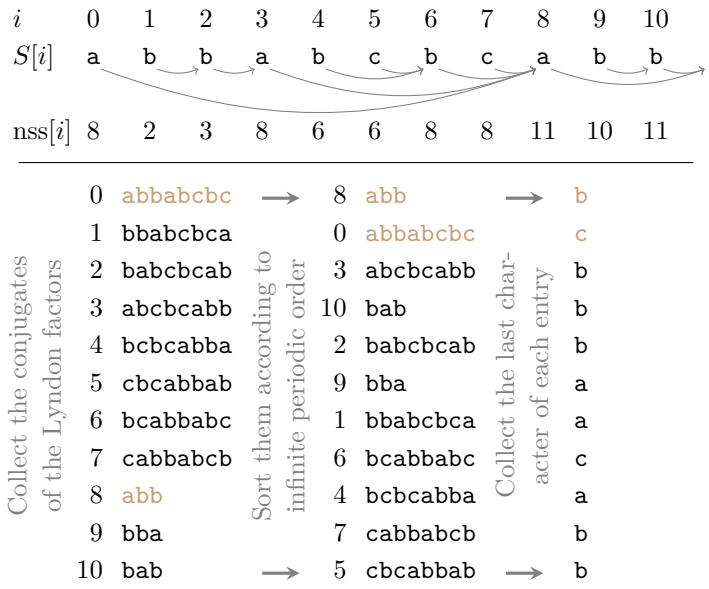


Figure 1 Next smaller suffix array nss (top) and BBWT (bottom) of the running example $S =$ `abbabcbcabb`. Each arrow points from i to nss[i]. Lyndon factors are coloured (■).

For instance, `ab` $<_{lex}$ `aba` $<_{lex}$ `abb` and `abb` $>_{\omega}$ `ab` $>_{\omega}$ `aba` since `abb`$\ldots >_{lex}$ `abab`$\ldots >_{lex}$ `abaab`$\ldots$.

▶ **Definition 6** (Bijective Burrows-Wheeler Transform (BBWT)). *The* bijective Burrows-Wheeler Transform (BBWT) *of a string S is the string obtained by taking the last character of each conjugate of the Lyndon factors of S arranged in infinite periodic order.*

▶ **Definition 7** (Extended Burrows-Wheeler Transform (eBWT)). *The* extended Burrows-Wheeler Transform (eBWT) *of a multiset $\mathcal{M}$ of strings is the string obtained by taking the last character of each conjugate of the strings in $\mathcal{M}$ arranged in infinite periodic order.*

Note that, by definition, the BBWT of a string S is the eBWT of the Lyndon factors of S. Similarly, the eBWT of a multiset $\mathcal{M}$ of strings is the BBWT of the concatenation of the canonical forms of the strings in $\mathcal{M}$ arranged in lexicographically decreasing order [3] (the Lyndon factors of the resulting string are conjugates of the roots of the strings in $\mathcal{M}$).[1]

For a primitive string S, its BWT, the eBWT of $\{S\}$, and the BBWT of the smallest conjugate of S are identical. Consequently, the $-BWT of a string S (i.e., BWT(S\$), where \$ is smaller than all characters in S) commonly computed via the suffix array of S, is identical to BWT(\$$S$) = BBWT(\$$S$) = eBWT($\{$\$$S\}$).

2.1 Lyndon Grammar

▶ **Definition 8** (Lyndon Tree/Forest [4, 25]). *The* Lyndon tree *of a Lyndon word w – denoted by LTree(w) – is the ordered full binary tree defined recursively as follows: If $|w| = 1$, then LTree(w) consists of a single node labelled by w, and if $|w| \geq 2$ and w has the standard factorization (u, v), the root of LTree(w) is labelled by w, the left child of the root is LTree(u), and the right child of the root is LTree(v). For a non-Lyndon word, we let the* Lyndon Forest *be the sequence of Lyndon trees of the Lyndon factors.*

[1] The eBWT was originally defined for sets of primitive strings [32]. This limitation is unnecessary [10, 40].

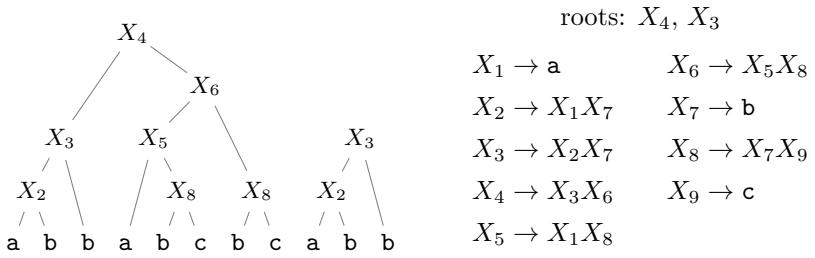


Figure 2 Lyndon Forest (left) and Lyndon SLP (right) for the running example $S = \texttt{abbabcbcabb}$. For clarity, instead of the node labels the corresponding grammar symbols are shown. Note the structural similarity between the Lyndon forest and the arrows indicating nss in Figure 1.

The Lyndon Forest of the running example is shown in Figure 2. Note that the Lyndon Forest of a string S is closely related to the Lyndon array λ of S [25], where $\lambda[i]$ is the length of the longest prefix of $\mathrm{suf}_i(S)$ that is Lyndon (equivalently, $\lambda[i] = \mathrm{nss}[i] - i$ [7], cf. Figure 1). A succinct representation of λ occupies $2|S|$ bits and can be computed in linear time directly from the text [7, 16, 31].

▶ **Definition 9** (Lyndon straight-line program [25]). *A* Lyndon straight-line program *(SLP) is a context-free grammar $\mathcal{G}$ over an alphabet Σ in Chomsky normal form, where $[X_i]$ is a Lyndon word for each symbol X_i and each rule $X_i \to X_aX_b$ is such that the standard factorization of $[X_i]$ is $([X_a], [X_b])$, where $[X_i]$ denotes the unique string generated by X_i. A symbol X_i is called* terminal symbol *if there is a rule of the form $X_i \to c$ ($c \in \Sigma$), and* non-terminal symbol *otherwise. When $r_1, \ldots, r_k$ are the root symbols of $\mathcal{G}$,[2] $\mathcal{G}$ generates $[\mathcal{G}] = [r_1] \ldots [r_k]$, and $[r_1] \geq \cdots \geq [r_k]$ is the Lyndon factorization of $[\mathcal{G}]$. The* size $|\mathcal{G}|$ *of $\mathcal{G}$ is the number of production rules plus the number of start symbols. The* derivation tree *of the SLP $\mathcal{G}$ is a labelled ordered tree where the root node has the children $r_1, \ldots, r_k$. We assume that distinct symbols generate distinct words.*

A Lyndon SLP arises from renaming the nodes of a Lyndon Forest such that two nodes with isomorphic subtrees (i.e., representing the same string) are assigned the same symbol. The Lyndon Forest and Lyndon SLP of our running example can be seen in Figure 2.

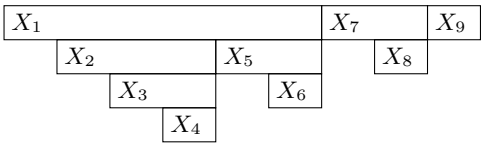
Clearly, the size of a Lyndon SLP $\mathcal{G}$ is bounded by $\mathcal{O}(|[\mathcal{G}]|)$. This bound is tight and there is no non-trivial bound in terms of e.g. the number of runs in the BWT, the size γ of the smallest string attractor [26] or the size of the smallest SLP: consider $S = \texttt{a}^{n-1}\texttt{b}$ for $n > 1$. The smallest SLP generating S has size $\mathcal{O}(\log n)$, its BWT has two runs, and the smallest string attractor has size $\gamma = |\{0, n-1\}| = 2$, but every suffix of S is Lyndon and thus the Lyndon SLP generating S has $\Theta(n)$ symbols.

In the following, we assume the set of symbols V of an SLP $\mathcal{G}$ to be numbered consecutively, i.e., $V = \{X_1, \ldots, X_{|V|}\}$.

3 Properties of sorted Lyndon Grammars

Consider a Lyndon grammar $\mathcal{G}$ where the symbols are sorted lexicographically, i.e., $[X_i] <_{lex} [X_j] \iff i < j$. In this section we examine such Lyndon SLPs and find properties that enable sorting any Lyndon SLP lexicographically in linear time (Section 4) and give rise to an online algorithm for constructing Lyndon SLPs (Section 6.3). Note that the grammar shown in Figure 2 is lexicographically sorted.

[2] In slight deviation of the usual definition of SLPs, we allow multiple root/start symbols. The reason for this is that we want to represent arbitrary strings with the Lyndon SLP, not just Lyndon words.



$X_1 \to \mathtt{a}$ $X_7 \to \mathtt{b}$ $X_9 \to \mathtt{c}$
$X_2 \to X_1X_7$ $X_5 \to X_1X_8$ $X_8 \to X_7X_9$
$X_3 \to X_2X_7$ $X_6 \to X_5X_8$
$X_4 \to X_3X_6$

Figure 3 First-symbol forest of the (lex. sorted) Lyndon SLP of the running example (cf. Figure 2).

▶ **Definition 10** (Prefix- and Suffix-symbols). *Define $\mathcal{P}_L(X_i)$ ($\mathcal{P}_R(X_i)$) as the set of symbols occurring on the leftmost (rightmost) path of X_i's derivation tree. More formally, if X_i is a terminal symbol define $\mathcal{P}_L(X_i) = \mathcal{P}_R(X_i) = \{X_i\}$, and if $X_i \to X_aX_b$ is a non-terminal symbol define $\mathcal{P}_L(X_i) = \{X_i\} \cup \mathcal{P}_L(X_a)$ and $\mathcal{P}_R(X_i) = \{X_i\} \cup \mathcal{P}_R(X_b)$. We correspondingly define $\mathcal{P}_L^{-1}(X_i)$ ($\mathcal{P}_R^{-1}(X_i)$) to be the set of symbols on whose leftmost (rightmost) path of the derivation tree X_i occurs, i.e. $\mathcal{P}_L^{-1}(X_i) = \{X_j \mid X_i \in \mathcal{P}_L(X_j)\}$ and $\mathcal{P}_R^{-1}(X_i) = \{X_j \mid X_i \in \mathcal{P}_R(X_j)\}$.*

For example, we have $\mathcal{P}_L(X_6) = \{X_6, X_5, X_1\}$, $\mathcal{P}_R(X_6) = \{X_6, X_8, X_9\}$ and $\mathcal{P}_L^{-1}(X_1) = \{X_1, X_2, X_3, X_4, X_5, X_6\}$ for the grammar shown in Figure 2.

▶ **Definition 11.** *Define $\mathcal{D}(X_i)$ as the set of symbols occurring in X_i's derivation tree. More formally, if X_i is a terminal symbol define $\mathcal{D}(X_i) = \{X_i\}$, and if $X_i \to X_aX_b$ is a non-terminal symbol define $\mathcal{D}(X_i) = \{X_i\} \cup \mathcal{D}(X_a) \cup \mathcal{D}(X_b)$.*

Let $r(X_i)$ be the largest symbol whose derivation tree has X_i on the leftmost path, i.e., $r(X_i) = \max\big(\mathcal{P}_L^{-1}(X_i)\big)$. Similarly, let $l(X_i)$ be the smallest symbol whose derivation tree has X_i on the leftmost path, i.e. $l(X_i) = \min\big(\mathcal{P}_L^{-1}(X_i)\big)$. For example, $l(X_1) = X_1$ and $r(X_1) = X_6$ in the grammar from Figure 2. In the remainder of this section, we show that all symbols X_j in the interval $[l(X_i)\,..\,r(X_i)]$ generate strings with $[X_i]$ as prefix and even satisfy $X_j \in \mathcal{P}_L(X_i)$. This then implies that these intervals form a tree-structure.

▶ **Lemma 12.** *$l(X_i) = X_i$ for all i.*

Proof. For $X_j \in \mathcal{P}_L^{-1}(X_i)$ we have $[X_i] \leq_{lex} [X_j]$ as $[X_i]$ is prefix of $[X_j]$ by definition. ◀

The following lemma follows from basic properties of Lyndon words and the definition of the Lyndon grammar/tree. Its proof is omitted due to space constraints.

▶ **Lemma 13.** *For all $X_j \in [l(X_i)\,..\,r(X_i)]$ it holds $X_i \in \mathcal{P}_L(X_j)$, i.e., X_i occurs on the leftmost path of X_j's derivation tree. Equivalently, $[l(X_i)\,..\,r(X_i)] = \mathcal{P}_L^{-1}(X_i)$.*

▶ **Corollary 14.** *Any two intervals $[l(X_i), r(X_i)]$ and $[l(X_j), r(X_j)]$ either do not intersect or one is fully contained within the other.*

Proof. Let X_i and X_j be different such that $l(X_i) < l(X_j) \leq r(X_i)$ (Lemma 3 implies $l(X_i) \neq l(X_j)$). This implies $X_j \in \mathcal{P}_L(X_i)$ by Lemma 13 and thus $\mathcal{P}_L(X_j) \subset \mathcal{P}_L(X_i)$ and $r(X_j) \leq r(X_i)$. ◀

▶ **Corollary 15.** *The intervals $[l(X_i), r(X_i)]$ induce a forest.*

The following definition defines this forest. An example can be seen in Figure 3.

▶ **Definition 16.** *We say an interval* $[l(X_j), r(X_j)]$ *is* embedded *in an interval* $[l(X_i), r(X_i)]$ *if it is a subinterval of* $[l(X_i), r(X_i)]$*, i.e.* $l(X_i) < l(X_j) \leq r(X_j) \leq r(X_i)$*. If* $[l(X_j), r(X_j)]$ *is embedded in* $[l(X_i), r(X_i)]$ *and there is no interval embedded in* $[l(X_i), r(X_i)]$ *in which* $[l(X_j), r(X_j)]$ *is embedded,* $[l(X_j), r(X_j)]$ *is a* child interval *of* $[l(X_i), r(X_i)]$*.*

The first-symbol forest *is a rooted forest, where each symbol* X_i *of the SLP corresponds to a node* u_i*, and* u_j *is a child of* u_i *if and only if* $[l(X_j), r(X_j)]$ *is a child interval of* $[l(X_i), r(X_i)]$*. Note that the parent of* $X_i \to X_a X_b$ *is* X_a*. Also, the terminal symbols correspond to the roots of the trees in the forest.*

▶ **Lemma 17.** *Let* X_i *be a symbol of a Lyndon SLP. We have* $\mathcal{P}_L(X_i) = \{X_{\ell_0}, \ldots, X_{\ell_k}\}$ *where* $X_{\ell_j} \to X_{\ell_{j-1}} X_{c_j}$ *for all* $j \in [1\,..\,k]$*,* X_{ℓ_0} *is a terminal symbol,* $X_{\ell_k} = X_i$*, and* $[X_{c_1}] \geq \cdots \geq [X_{c_k}]$ *is the Lyndon factorization of* $\mathrm{suf}_1([X_i])$*.*

Proof. Follows immediately from Lemma 3. ◀

4 Lexicographically sorting a Lyndon Grammar

Given a Lyndon SLP $\mathcal{G}$ with roots $r_1, \ldots, r_k$, we want to rename the symbols such that $[X_i] <_{lex} [X_j]$ holds if and only if $i < j$. This is possible in $\mathcal{O}(|\mathcal{G}|)$ using a similar principle to what is used for computing the Lyndon grouping in [2, 40]: consider a symbol X_i. By Lemma 17 and simple properties of Lyndon words, the symbols in $\mathcal{D}(X_i) \setminus \mathcal{P}_L(X_i)$ are lexicographically greater than X_i. Therefore, when considering the symbols in lexicographically decreasing order we can "induce" the position of a symbol $X_i \to X_a X_b$ upon encountering X_b: the symbols in $\mathcal{P}_L^{-1}(X_i)$ must be the lexicographically largest symbols with X_a as prefix symbol that have not yet been induced.

Algorithm 1 shows the procedure. Throughout, $\mathsf{A}[i]$ is either the ith symbol in the lexicographical order or $\perp$, and for each inserted X_i, $\mathsf{G}[i]$ is the index in A of the smallest child of X_i that has been inserted into A (or $r(X_i) + 1$ if no child has been inserted yet). In the first for-loop, all terminal symbols are inserted into A, and in the second for-loop, the remaining symbols are induced. Consequently, after the second for-loop, A contains the lexicographic order of the symbols, and G is (almost) it's inverse, i.e. $\mathsf{G}[\mathsf{A}[i]] = i + 1$.

There are two operations in Algorithm 1 that are not immediately obvious, namely finding $\left|\mathcal{P}_L^{-1}(X_i)\right|$ and iterating over all X_j with $X_j \to X_a X_{\mathsf{A}[i]}$ for a given $\mathsf{A}[i]$. For the first, note that the values $\left|\mathcal{P}_L^{-1}(X_i)\right|$ can be trivially computed in linear-time using the recurrence $\left|\mathcal{P}_L^{-1}(X_a)\right| = 1 + \sum_{X_i \to X_a X_b} \left|\mathcal{P}_L^{-1}(X_i)\right|$. Secondly, iterating efficiently over all X_j with $X_j \to X_a X_{\mathsf{A}[i]}$ can be achieved using a linear-time preprocessing step where we collect for each symbol X_i the symbols which have X_i as second symbol on the right-hand side.

The following theorem follows. Its proof is omitted due to space constraints.

▶ **Theorem 18.** *Algorithm 1 lexicographically sorts a Lyndon SLP in linear time.*

5 BWT construction

In this section, we describe an algorithm that derives the BBWT from the Lyndon SLP. Specifically, combining the second phase of `GSACA` [2, 40] with run-length encoding on the lexicographically sorted Lyndon SLP results in a very efficient algorithm on real data.

The following lemmas and corollary establish a relationship between the lexicographical order of suffixes and the lexicographical order of certain symbols of the Lyndon grammar.

Algorithm 1 Lexicographically sorting a Lyndon SLP.

$(\mathsf{A}, s) \leftarrow (\perp^n, 0)$;
for $X_i \to c$ *with* $c \in \Sigma$ **do** /* in increasing lexicographical order of c */
  $\mathsf{A}[s] \leftarrow i$; /* There are s symbols X with $[X][0] < c$ */
  $s \leftarrow s + \left|\mathcal{P}_L^{-1}(X_i)\right|$; /* There are $\left|\mathcal{P}_L^{-1}(X_i)\right|$ symbols X with $[X][0] = c$ */
  $\mathsf{G}[i] \leftarrow s$; /* X_i is the lex. smallest symbol in $\mathcal{P}_L^{-1}(X_i)$ */
end
for $i = n - 1 \to 0$ **do**
  for $X_j \to X_a X_{\mathsf{A}[i]}$ **do**
    // The symbols in $\mathcal{P}_L^{-1}(X_j)$ must be in $A\big[G[a] - \left|\mathcal{P}_L^{-1}(X_j)\right| .. G[a]\big)$
    $\mathsf{G}[j] \leftarrow \mathsf{G}[a]$;
    $\mathsf{G}[a] \leftarrow \mathsf{G}[a] - \left|\mathcal{P}_L^{-1}(X_j)\right|$;
    $\mathsf{A}[\mathsf{G}[a]] \leftarrow j$; /* X_j is the lex. smallest symbol in $\mathcal{P}_L^{-1}(X_j)$ */
  end
end

▶ **Lemma 19** ([40, Lemma 3.3]). *Let $\mathcal{L}_i$ be the longest prefix of $\mathrm{suf}_i(S)$ that is Lyndon. Then, $\mathcal{L}_i <_{lex} \mathcal{L}_j$ implies $\mathrm{suf}_i(S) <_{lex} \mathrm{suf}_j(S)$.*

▶ **Lemma 20.** *Consider an occurrence of $X_i \to X_a X_b$ at position j in the text S (this implies $S[j\,..\,j + |[X_i]|] = [X_i]$). Then, $[X_b]$ is the longest prefix of $\mathrm{suf}_{j+|[X_a]|}(S)$ that is Lyndon.*

Proof. Follows immediately from the relationship between Lyndon forest and Lyndon grammar [25] and [21, Lemma 15]. ◀

▶ **Corollary 21.** *Consider occurrences of $X_i \to X_a X_b$ at position j and of $X_{i'} \to X_{a'} X_{b'}$ at position j'. It holds $\mathrm{suf}_{j+|[X_a]|}(S) <_{lex} \mathrm{suf}_{j'+|[X_{a'}]|}(S)$ if $[X_b] <_{lex} [X_{b'}]$.*

To see why Corollary 21 is useful, consider a symbol $X_i \to X_a X_b$. Each occurrence of X_i in the derivation tree corresponds to a suffix of the text with prefix $[X_b]$, and each such suffix introduces an occurrence of the last character of $[X_a]$ to the BWT in the SA-interval of $[X_b]$. In our running example (cf. Figure 2), each occurrence of $X_8 \to X_7 X_9$ introduces a b in the SA-interval of $[X_9] = \mathsf{c}$.

In [40], an array $\mathsf{SA}'_\circ$[3] is computed such that $\mathsf{BBWT}[i] = S[\mathsf{SA}'_\circ[i]]$ holds for all i. This array arises from sorting the conjugates of the Lyndon factors of S and replacing each start position of a Lyndon word with the end position of the Lyndon word in S. It was shown that Algorithm 2 correctly computes the array $\mathsf{SA}'_\circ$ in linear time [40].[4] Basically, Lemma 19 implies that the positions in $\mathsf{SA}'_\circ$ are grouped by the longest Lyndon prefixes of the corresponding suffixes. The longest Lyndon prefix of $\mathrm{suf}_i(S)$ is $S[i\,..\,\mathrm{nss}[i])$. Because $\mathrm{suf}_{\mathrm{nss}[i]}(S) <_{lex} \mathrm{suf}_i(S)$ by definition, we can proceed by induction from lexicographically small to large.[5] Consider for instance the suffixes starting with c at indices 5 and 7 in our

[3] $\mathsf{SA}'_\circ$ is essentially the analogue to SA for conjugates of Lyndon factors and the infinite periodic order. A precise definition is omitted because only the relationship between $SA'_\circ$ and BBWT is relevant for this paper.
[4] In [40], Lyndon factors (roots of the Lyndon forest) come after every non-root representing the same Lyndon word. To reflect this, we use $\mathsf{L}[2(i-1)+1]$ if an occurrence of X_i is a root and $\mathsf{L}[2(i-1)]$ otherwise.
[5] There is a slight detail omitted here for clarity and brevity. Namely, that for the BBWT we are concerned with the *next smaller conjugate* of the respective Lyndon factor instead of the next smaller suffix [40].

Algorithm 2 Computing $\mathsf{SA}'_\circ$ from the lexicographically sorted Lyndon grammar [40, Algorithms 2 and 3].

```
(s, SA'_∘, L) ← (n, [], []^n);
for i = k → 1 do /* insert the end positions of the roots r_1, ..., r_k of
 the Lyndon SLP                                                        */
    L[2(r_i − 1) + 1].append(s);
    s ← s − |[X_{r_i}]|;
end
for m = 0 → 2g − 1 do
    foreach i in L[m] do
        SA'_∘.append(i);
        for j with nss[j] = i do
            Find X_k such that [X_k] = S[j .. nss[j]);
            L[2(k − 1)].append(j);
        end
    end
end
```

running example (cf. Figure 2). By Lemma 19, they are in the same "Lyndon group" [2] in the sorted list of suffixes because $\mathtt{c}$ is their longest Lyndon prefix. We have $\mathrm{nss}[5] = 5+|\mathtt{c}| = 6$ and $\mathrm{nss}[7] = 7+|\mathtt{c}| = 8$. When processing these Lyndon groups in lexicographically increasing order, the relative order of suffixes 6 and 8 is known at the point in time when the Lyndon group $\mathtt{c}$ is considered (because they are lexicographically smaller), and can therefore determine the lexicographical order of the suffixes 5 and 7.

We are now going to transform Algorithm 2 such that it outputs the BBWT instead of $\mathsf{SA}'_\circ$ and works without nss and the text indices. First, consider a text index $i \in [0\,..\,N)$ and let u_i be the highest node in the Lyndon forest at position i. (Equivalently, u_i corresponds to the longest Lyndon word starting at i.) Now consider the parent v_i of u_i. By its definition, it is clear that u_i is the right child of v_i. Let w_i be the left child of v_i. We can observe that, starting from w_i, the nodes on the rightmost path (excluding w_i) correspond exactly to the set $\{j \in [0\,..\,i) \mid \mathrm{nss}[j] = i\}$. For instance, in our running example there is a node labelled with $[X_8] = \mathtt{bc}$ at position 6 (cf. Figures 1 and 2). This node is the right child of a node labelled with $[X_6]$, which has a node labelled with $[X_5]$ as left child. There are two nodes on the rightmost path from this latter node, namely one at position 4 labelled with $[X_8] = \mathtt{bc}$ and one at position 5 labelled with $[X_9] = \mathtt{c}$. We consequently have $\mathrm{nss}[4] = \mathrm{nss}[5] = 6$.

Second, for $\mathsf{SA}'_\circ[k] = i$ we have $\mathsf{BBWT}[k] = S[i-1]$ and it follows that $\mathsf{BBWT}[k]$ is the last character of the string corresponding to v_i's left child w_i. Because of these two observations, we can replace the insertion of i in Algorithm 2 with the insertion of w_i. Consequently, we can reformulate Algorithm 2 to Algorithm 3.

▶ **Theorem 22.** *Algorithm 3 correctly computes* BBWT *from the sorted Lyndon SLP in* $\mathcal{O}(N)$ *time.*

Proof. The linear worst-case time complexity trivially follows from the fact that each iteration of the inner loop appends at least one character to BBWT. A proof of correctness is omitted due to space constraints. ◀

Note that runs in a list in L compound: If there is a run of symbol X_i, there is also a run of at least the same length of X_ℓ in X_r for each non-terminal $X_s \in \mathcal{P}_R(X_i)$ with $X_s \to X_\ell X_r$. Thus, Algorithm 3 often requires much fewer than N iterations.

Algorithm 3 Deriving the BBWT from the lexicographically sorted Lyndon grammar.

```
L ← []^{2g};                    /* initialize L[i] as an empty RLE list ∀i ∈ [0..2g) */
for i = 1 → k do A[2(r_i − 1) + 1].append(r_i); /* insert the roots r_1, ..., r_k */
for i = 0 → 2g − 1 do
    for (s, count) in L[i] do
        BBWT.append(last_char[s], count);
        while X_s is a non-terminal do /* walk the rightmost path from X_s */
            Let X_s → X_a X_b;
            A[2(b − 1)].append(a, count);
            s ← b;
        end
    end
end
```

5.1 Computing BWT and eBWT

When the text S is a Lyndon word, the BBWT of S is equal to the original BWT of S [3]. Since the BWT is independent of the rotation of the input and $\$S$ is a Lyndon word, we have $\mathsf{BBWT}(\$S) = \mathsf{BWT}(S\$)$. The latter is the \$-BWT commonly computed via the suffix array. Therefore, we can simply compute the \$-BWT by prepending a sentinel character to the text. Note that this is trivial and can be done after building the Lyndon grammar of S.

The eBWT of a set $\mathcal{S}$ of strings is the same as the BBWT of the string $\mathcal{S}^c$ that arises from arranging the canonical forms of the strings in $\mathcal{S}$ in lexicographically decreasing order [40]. In particular, the Lyndon factors of $\mathcal{S}^c$ are exactly the canonical forms of the input strings (assuming that the input strings are primitive). Note that, in Algorithm 3, sorting the canonical forms of the input strings is done implicitly in the first for-loop.

Note that the Lyndon grammars of the input strings are independent of each other (as long as equal Lyndon words are assigned equal grammar symbols). Therefore, we can also construct their Lyndon grammars independently of each other while using the same dictionary. As a consequence, it is easy to use our algorithms for parallel construction of the Lyndon grammar of a collection of sequences. Although our algorithm for deriving the BWT from the grammar is not parallelized, parallel construction of the Lyndon grammar leads to a substantial reduction in wall-clock-time because it is by far the most time-consuming part of the pipeline (see Section 7).

5.2 Other BWT variants for string collections

Besides the original eBWT, several other BWT variants for string collections have been proposed (see [14] for an overview). Several of the variants are especially suited to be computed with our algorithm, because there the input sequences $\mathcal{S} = \{S_1, \ldots, S_n\}$ can be parsed independently (and in parallel) like for the eBWT: The *dollar*-eBWT $\mathsf{dolEBWT}(\mathcal{S}) = \mathsf{eBWT}(\{S\$ \mid S \in \mathcal{S}\})$, the *multidollar* BWT $\mathsf{mdolBWT}(\mathcal{S}) = \mathsf{BWT}(S_1\$_1 \ldots S_n\$_n)$, and the *concatenated* BWT $\mathsf{concBWT}(\mathcal{S}) = \mathsf{BWT}(S_1\$ \ldots S_n\$\#)$, where $\# < \$$ and $\$_1 < \cdots < \$_n$ are

smaller than all characters in $\mathcal{S}$. This is done by constructing the Lyndon SLP of each S_i (with a shared dictionary) and then applying post-processing steps such that the desired eBWT variant can be derived from the resulting SLP. This is possible because the \$'s separate the input strings in the sense that no Lyndon word starting inside some S_i can contain a \$.

More specifically, for the doleBWT, no post-processing is required and we can just apply our eBWT construction algorithm, except that finding the canonical forms of the strings is trivial because $\mathsf{eBWT}(\{S\$ \mid S \in \mathcal{S}\}) = \mathsf{eBWT}(\{\$S \mid S \in \mathcal{S}\})$ and $\$S$ is a Lyndon word. For the mdolBWT and concBWT, we replace each S_i in $S_1\$_1 \dots S_n\$_n$ and $S_1\$ \dots \$S_n\$\#$, respectively, with the roots of the grammar generating S_i and compute the grammar of a rotation of the resulting string using Algorithm 3: both $\$_1S_2\$_2 \dots S_n\$_nS_1$ and $\#S_1\$ \dots \$S_n\$$ are Lyndon. Therefore, $\mathsf{BWT}(S_1\$_1 \dots S_n\$_n) = \mathsf{BBWT}(\$_1S_2\$_2 \dots S_n\$_nS_1)$ and $\mathsf{BWT}(S_1\$ \dots \$S_n\$\#) = \mathsf{BBWT}(\#S_1\$ \dots \$S_n\$)$ and we can apply our BBWT construction algorithm to these rotations.

Note that all other BWT variants for string collections given in [14] can be simulated with the multidollar BWT by using different relative orders of the separator symbols.

6 Practical Lyndon grammar construction

As shown in [1], the simple folklore algorithm for constructing the Lyndon array can be used to construct the Lyndon forest. With a fitting lookup data structure, we can trivially construct the Lyndon SLP instead. We essentially use Algorithm 4.

Algorithm 4 Prepending $S[i]$ [1].

```
Find c with [X_c] = S[i];                    /*Set c to the symbol generating S[i]*/
while stack is not empty do
    t ← stack.top();
    if [X_t] ≤_lex [X_c] then break;
    else
        stack.pop();
        Find X_c' with X_c' → X_c X_t;
        c ← c';
    end
end
stack.push(c);
```

There are two steps that have to be explained: First, one must be able to compare $[X_t]$ and $[X_c]$ lexicographically, and secondly, one must be able to find $X_{c'}$ with $X_{c'} \to X_cX_t$ (this is also called the *naming function*). For the former, we store for each symbol on the stack a fixed-length prefix of the generated string. In many cases, this is sufficient for determining the lexicographical order. Possibilities to handle the other cases are explained in Sections 6.1, 6.2 and 6.3. The following paragraphs will deal with the naming function.

For the naming function, we use a hash table to find the symbol names. This provides (expected) constant time lookup/insertion.[6] In practice however, N hash table lookups are unnecessarily slow, especially because we assume the input to be repetitive. In particular, having fewer but longer keys is cache friendlier and thus faster on modern computer architectures.

[6] Note that this is also possible in $\mathcal{O}(\log\log g)$ deterministic time with at most $2g + g\log g + o(g\log g))$ bits of memory, where g is the number of symbols of the grammar [42].

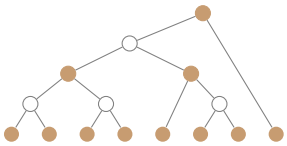


Figure 4 Partial representation of a Lyndon Forest with $n_{thres} = 2$. Heavy nodes are coloured in (■) while light nodes are not.

Specifically, we decrease the number of hash table accesses by partitioning the nodes of the Lyndon forest into *heavy* and *light* nodes and querying a second hash table H for each heavy node. The naming function is then only used when such a query to H does not return an answer. Leaves of the Lyndon forest are considered heavy. An inner node v is considered heavy, if and only if its number of *immediate heavy descendants* (heavy descendants for which only light nodes occur on the path to v) exceeds a fixed constant $n_{thres} > 1$ (see Figure 4).[7]

For a heavy node v of the Lyndon forest we query H with the sequence of immediate heavy descendants $v_1, \ldots, v_k$ of v. This means that we only store heavy nodes on the stack, for a light node we instead store all immediate heavy descendants.

Different occurrences of the same symbol represent the exact same Lyndon tree and therefore correspond to the exact same sequence of immediate heavy descendants. Consequently, if a subtree isomorphic to the subtree rooted at a heavy node has occurred previously, the query to H is sufficient to resolve it. Otherwise, we use the naming function to find the correct symbol and insert the sequence of immediate heavy descendants with it into H.

Note that each heavy node has at most $2n_{thres}$ immediate heavy descendants because it has exactly two children (or zero), each of which is either light (and thus has at most n_{thres} immediate heavy descendants) or heavy itself. This implies that the total memory usage increases by a factor of at most $\mathcal{O}(n_{thres})$.

6.1 Constant time suffix comparisons

Because the Lyndon forest (and thus the Lyndon SLP) is closely related to the Lyndon array, we can use the Lyndon array to decide whether $[X_t] \leq_{lex} [X_c]$ holds in constant time. Specifically, when considering $S[i]$ in Algorithm 3, we have $[X_c] <_{lex} [X_t]$ while $\lambda[i] > |[X_c]|$.

Unfortunately, as far as we know, all linear-time algorithms for constructing the Lyndon array directly from the text need random access to the text [1, 7, 19] and thus the text should reside in main memory.

For a BWT of a real-world string collection $\mathcal{S}$ however, this is typically not a problem because, as noted in Section 5.1, the SLPs of the strings in $\mathcal{S}$ can be computed independently (with a shared dictionary). Consequently, it suffices to have only the string (and its (succinct) Lyndon array) in main memory whose Lyndon SLP is currently computed. This increases the RAM usage by $n_{\max}(\lceil \log_2 |\Sigma| \rceil + 2)$ bits, where $n_{\max}$ is the length of the longest string in $\mathcal{S}$. For real biological data, $n_{\max}$ is generally small compared to the size of the entire input.

6.2 Naïve suffix comparisons using the Lyndon grammar

In this section, we describe a simple method for comparing $[X_t]$ and $[X_c]$ lexicographically which requires constant extra memory and works well in practice. We assume that $a, b < i$ holds for each rule $X_i \to X_a X_b$. This is clearly the case for SLPs constructed with Algorithm 4 if we assign increasing indices to new symbols.

[7] In our experiments, $n_{thres} = 31$.

First note that, because we assign equal Lyndon words to equal symbols, $[X_t] = [X_c]$ if and only if $t = c$. The key to our algorithm is that we find the longest common symbol $X_\ell \in \mathcal{P}_L(X_t) \cap \mathcal{P}_L(X_c)$. Additionally, we determine $X_{t'} \in \mathcal{P}_L(X_t)$ and $X_{c'} \in \mathcal{P}_L(X_c)$ with $X_{t'} \to X_\ell X_r$ and $X_{c'} \to X_\ell X_{r'}$. Note that, by definition, $r \neq r'$ (otherwise, $c' = t'$, contradicting the choice of ℓ) and therefore the lexicographical order of $[X_r]$ and $[X_{r'}]$ is the same as the lexicographical order of $[X_t]$ and $[X_c]$. Formally, $[X_t] <_{lex} [X_c] \iff [X_r] <_{lex} [X_{r'}]$. If such a tuple (ℓ, t', c') does not exist, either $[X_t]$ and $[X_c]$ do not share a non-empty prefix (i.e., $[X_t][0] \neq [X_c][0]$), or one of X_t and X_c is a prefix of the other.

Note that the indices of the elements in $\mathcal{P}_L(X_i)$ have the same relative order as the lengths of the generated strings in the sense that for all $a, b \in \mathcal{P}_L(X_i)$ we have $a < b$ if and only if $|[X_a]| < |[X_b]|$. For this reason, we can proceed with a two-pointer search to find ℓ (and t' and c'). More specifically, assuming that the desired $\ell \in \mathcal{P}_L(X_t) \cap \mathcal{P}_L(X_c)$ exists, we have $\ell \in \mathcal{P}_L(X_a)$ if $|[X_t]| < |[X_c]|$ for $X_c \to X_a X_b$, and vice versa. A concrete implementation is omitted due to space constraints.

Because in each step, at least one of the symbol indices decreases, the time complexity for a comparison is at most linear in the size of the SLP. In fact, because in each step we go from a symbol to one of its children, the time complexity is actually bounded by the *height* of the SLP. This in turn implies a time complexity of $\mathcal{O}(Ng)$ when using Algorithm 4 with the described method for constructing the Lyndon SLP.

Note that this worst-case time complexity is tight, e.g. on the string $a^k b a^k$ ($k \in \mathbb{N}$). However, the Lyndon forests of random strings have expected height proportional to the logarithm of the input size [36] and our approach works very well in practice (see Section 7).

6.3 Construction in $\mathcal{O}\left(N \log g + g \log^2 g\right)$

In this Section, we describe an algorithm that is able to compute the Lyndon grammar online in $\mathcal{O}\left(N \log g + g \log^2 g\right)$ deterministic time from right to left in a streaming fashion using $\mathcal{O}(g)$ words of extra memory.

Basically, this is done by maintaining the grammar's set of symbols $\mathcal{S}$ in an ordered sequence, lexicographically sorted by their respective generated strings. Using e.g. a B-Tree [5], one can then find the rank of any symbol in $\mathcal{O}(\log|\mathcal{S}|)$ time. Thus, determining the lexicographical order of two symbols in $\mathcal{S}$ can also be done in $\mathcal{O}(\log|\mathcal{S}|)$ time. What remains to be shown is how the symbols can be maintained in this sorted arrangement.

As shown in Section 3, a lexicographically sorted Lyndon grammar has a forest-structure (cf. Figure 3). This first-symbol forest can be represented using a *balanced parenthesis sequence (BPS)* of length $2|\mathcal{S}|$ [24], which can be obtained using a depth-first traversal of the first-symbol forest (starting at the roots) by writing an opening parenthesis `'('` when visiting a node for the first time and a closing parenthesis `')'` when all subtrees of a node have been visited [24]. Each parenthesis pair corresponds to a symbol in the grammar, where the ith opening parenthesis corresponds to the ith smallest symbol (by lexicographical order). For a grammar with symbol set $\mathcal{S}$, let this BPS be $\mathcal{B}_\mathcal{S}$. We represent $\mathcal{B}_\mathcal{S}$ as an ordered sequence $\mathcal{T}_\mathcal{S}$, which contains two markers $(_i$ and $)_i$ for each symbol X_i in $\mathcal{S}$, such that the ranks of $(_i$ and $)_i$ are the indices of X_i's opening and closing parenthesis in $\mathcal{B}_\mathcal{S}$, respectively. For instance, the sequence $\mathcal{T}_\mathcal{S}$ for the lexicographically sorted Lyndon grammar of our running example (see Figure 3) would be $(_1(_2(_3(_4)_4)_3)_2(_5(_6)_6)_5)_1(_7(_8)_8)_7(_9)_9$.

Now consider adding a new symbol $X_i \to X_a X_b$, where X_a and X_b are in $\mathcal{T}$ (i.e. $X_i \notin \mathcal{S}$, $X_a, X_b \in \mathcal{S}$). By Lemma 15, the parent of X_i's parenthesis pair in $\mathcal{B}_{\mathcal{S}\cup\{X_i\}}$ must be X_a's parenthesis pair. Therefore, it suffices to determine X_i's lexicographically smallest "sibling" $X_j \to X_a X_c$ ($X_j \in \mathcal{S}$) with $[X_c] >_{lex} [X_b]$; X_i's parenthesis pair must appear immediately

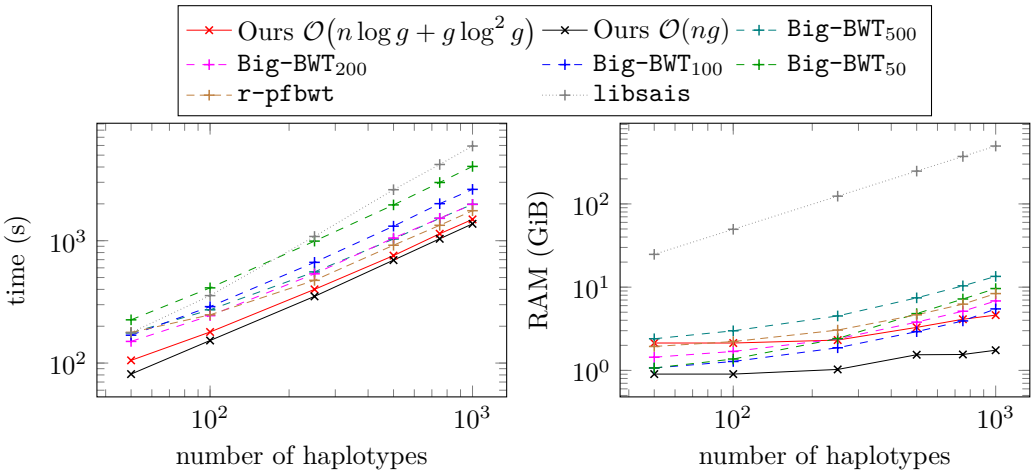


Figure 5 Wall clock time and maximum resident memory of BWT construction algorithms on chromosome 19 sequences. The sequences were concatenated and all programs used only one thread.

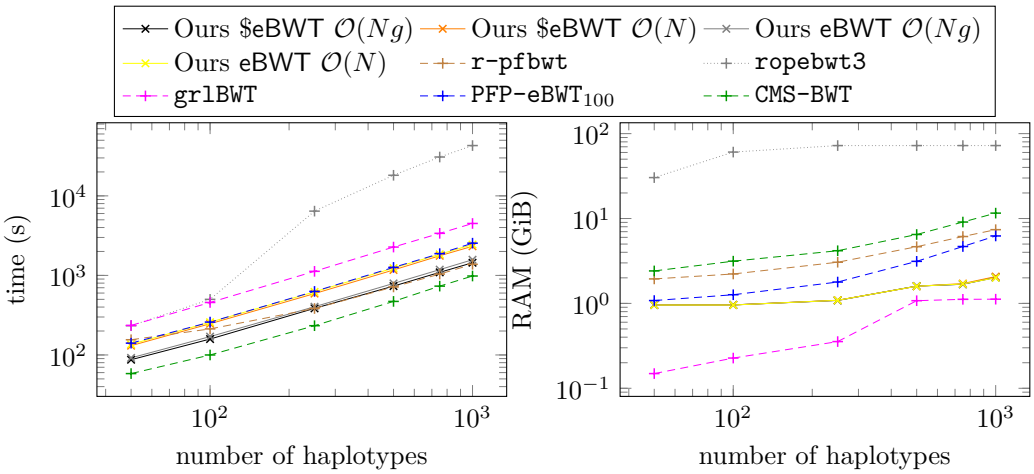


Figure 6 Wall clock time and maximum resident memory of tools for constructing BWT variants for string collection on chromosome 19 sequences. All programs used only one thread.

in front of $(_j$ in $\mathcal{T}_{\mathcal{S}\cup\{X_i\}}$. If there is no such sibling, X_i is (currently) the largest child of X_a and thus X_i's parenthesis pair must be immediately in front of $)_a$ instead. Note that comparing $[X_b]$ and $[X_c]$ is possible in $\mathcal{O}(\log|\mathcal{S}|)$ because both X_b and X_c are in $\mathcal{T}_{\mathcal{S}}$.

In order to be able to find the correct sibling of X_i as described, we additionally maintain for each X_a an ordered sequence $\mathcal{T}_a$ containing the symbols $\mathcal{S}_a = \{X_k \to X_a X_b \mid X_k, X_b \in \mathcal{S}\}$ in lexicographical order. Inserting $X_i \to X_a X_b$ into $\mathcal{T}_a$ can then be accomplished with $\mathcal{O}(\log|\mathcal{S}_a|) \subseteq \mathcal{O}(\log|\mathcal{S}|)$ comparisons, each of which is possible in $\mathcal{O}(\log|\mathcal{S}|)$ via $\mathcal{T}_{\mathcal{S}}$.

In total, we obtain a time complexity of $\mathcal{O}\big(|\mathcal{S}|\log^2|\mathcal{S}|\big)$ for maintaining the grammar's symbols lexicographically sorted, and $\mathcal{O}(N\log|\mathcal{S}|)$ for constructing the Lyndon forest.

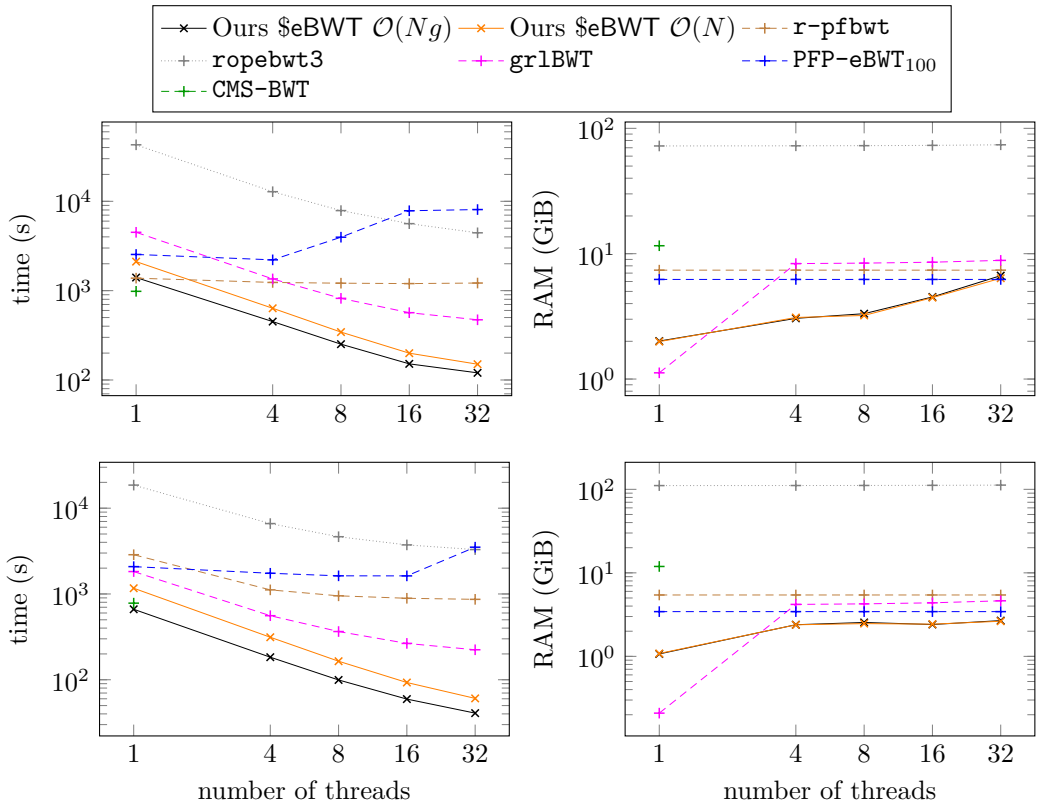


Figure 7 Wall clock time and maximum resident memory of tools for constructing BWT variants on 1000 Chromosome 19 sequences (top) and 10^6 SARS-CoV-2 sequences (bottom) using multiple threads. Note that `CMS-BWT` does not support multithreading. For the SARS-CoV-2 data, `PFP-eBWT` required the `--reads` flag.

7 Experiments

The source code of our implementation is publicly available.[8]

We compare our BWT algorithms for single texts with the programs `Big-BWT`[9] [11] and `r-pfbwt`[10] [41], as well as `libsais`.[11] The latter uses a modified version of the Suffix-Array Induced Sorting (SAIS) algorithm [37] to compute the BWT and, since it is based on the currently fastest SA construction implementation for general real-world data [40], can be viewed as a lower bound for algorithms using the suffix array to compute the BWT.

For the BWT of text collections, we compare with `PFP-eBWT`[12] [10], `r-pfbwt` [41], `ropebwt3`[13] [29], `grlBWT`[14] [17] and `CMS-BWT`[15] [35] (for `CMS-BWT` we used the first sequence in the collection as reference). All tool except the last one support multi-threading. Note

[8] `https://gitlab.com/qwerzuiop/lyndongrammar`
[9] `https://gitlab.com/manzai/Big-BWT`, last accessed: 22.04.2025, git hash `944cb27`
[10] `https://github.com/marco-oliva/r-pfbwt`, last accessed: 22.04.2025, git hash `1fea5c3`
[11] `https://github.com/IlyaGrebnov/libsais`, last accessed: 22.04.2025, git hash `a138159`
[12] `https://github.com/davidecenzato/PFP-eBWT`, last accessed: 22.04.2025, git hash `4ca75ce`
[13] `https://github.com/lh3/ropebwt3`, last accessed: 22.04.2025, git hash `36a6411`
[14] `https://github.com/ddiazdom/grlBWT`, last accessed: 22.04.2025, git hash `f09e7fa`
[15] `https://github.com/fmasillo/cms-bwt`, last accessed: 22.04.2025, git hash `1099d07`. Note that the speed and memory usage of `CMS-BWT` has improved massively since its publication in [35].

that not all of these tools compute the same BWT variant [14]. Also note that all algorithms based on PFP as well as `grlBWT` are semi-external, i.e., write/read some temporary data to/from disk.

As test data, we use up to 1000 human Chromosome 19 haplotypes from [12] ($\approx 6 \cdot 10^{10}$bp) and 10^6 SARS-CoV-2 sequences ($\approx 3 \cdot 10^{10}$bp).[16] All experiments were conducted on a Linux-6.8.0 machine with an Intel Xeon Gold 6338 CPU and 512 GB of RAM. All programs were compiled with GCC 13.3.0. Before each test, the test file was scanned once to ensure it is cached by the kernel. The results can be seen in Figures 5, 6 and 7. The subscripts for the PFP-based algorithms indicate the used modulus.

For our programs, computing the doleBWT using the suffix comparison method from Section 6.2 is generally the fastest. Computing the original eBWT is slightly slower because we need to find the smallest rotation of each input string. In the single-threaded case, for both the 1000 Chromosome 19 haplotypes and 10^6 SARS-CoV-2 sequences, over 98% of the time was spent constructing the grammar (for our fastest algorithm). Using the linear-time algorithm for constructing the Lyndon array from [7] to ensure expected linear time complexity slows our programs down by up to 60%. As expected, the suffix comparison method from Section 6.3 is much slower than our other methods. The increase in memory consumption of our program in the multithreaded case is due to the use of thread-safe hash tables and multiple sequences and stacks residing in main memory.

For the Chromosome 19 collections and a single thread, `CMS-BWT` is the fastest program (at the cost of high memory usage), followed by `r-pfbwt` (for larger cases) and our algorithms. For more threads, our program is always the fastest. Regarding the memory consumption, `grlBWT` uses the least amount of main memory for single-threaded processing, followed by our programs. For the SARS-CoV-2 sequences, our program is the fastest, especially with multiple threads, and for more than one thread also the most memory efficient.

8 Conclusion and Further Work

We described an algorithm to compute the BBWT – and by extension the common \$-BWT and various versions of the eBWT – from the lexicographically sorted Lyndon grammar of a text or text collection. Furthermore, we gave an algorithm that lexicographically sorts a Lyndon grammar and discussed approaches to efficiently compute the Lyndon grammar of a text or text collection. We implemented the described algorithms and found that they outperform other current state-of-the art programs in terms of time or memory consumed (often both).

References

1 Golnaz Badkobeh, Maxime Crochemore, Jonas Ellert, and Cyril Nicaud. Back-to-front online lyndon forest construction. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching*, volume 223 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.CPM.2022.13`.

2 Uwe Baier. Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching*, volume 54 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CPM.2016.23`.

[16] Downloaded from `https://www.covid19dataportal.org` on 28.08.2024.

3 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piątkowski. Constructing the Bijective and the Extended Burrows-Wheeler Transform in Linear Time. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching*, volume 191 of *Leibniz International Proceedings in Informatics*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.7.

4 Hélène Barcelo. On the action of the symmetric group on the free lie algebra and the partition lattice. *Journal of Combinatorial Theory, Series A*, 55(1):93–129, 1990. doi:10.1016/0097-3165(90)90050-7.

5 Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970. doi:10.1145/1734663.1734671.

6 Nico Bertram, Jonas Ellert, and Johannes Fischer. Lyndon Words Accelerate Suffix Sorting. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms*, volume 204 of *Leibniz International Proceedings in Informatics*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.15.

7 Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. Space Efficient Construction of Lyndon Arrays in Linear Time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming*, volume 168 of *Leibniz International Proceedings in Informatics*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.14.

8 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffixes, Conjugates and Lyndon Words. In Marie-Pierre Béal and Olivier Carton, editors, *Developments in Language Theory*, pages 131–142, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38771-5_13.

9 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *International Journal of Foundations of Computer Science*, 25(08):1161–1175, 2014. doi:10.1142/S0129054114400309.

10 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *International Symposium on String Processing and Information Retrieval*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.

11 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14:1–15, 2019. doi:10.1186/s13015-019-0148-5.

12 Christina Boucher, Travis Gagie, I Tomohiro, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *2021 Data Compression Conference*, pages 193–202. IEEE, 2021. doi:10.1109/dcc50243.2021.00027.

13 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report*, 124, 1994.

14 Davide Cenzato and Zsuzsanna Lipták. A survey of BWT variants for string collections. *Bioinformatics*, 40(7):btae333, 2024. doi:10.1093/bioinformatics/btae333.

15 Kuo Tsai Chen, Ralph H. Fox, and Roger C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958. doi:10.2307/1970044.

16 Maxime Crochemore and Luís M.S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806, 2020. doi:10.1016/j.tcs.2018.08.011.

17 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. *Information and Computation*, 294:105088, 2023. doi:10.1016/j.ic.2023.105088.

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

19 Jonas Ellert. Lyndon Arrays Simplified. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms*, volume 244 of *Leibniz International Proceedings in Informatics*, pages 48:1–48:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2022.48.

20 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

21 Frantisek Franek, ASM Sohidull Islam, M Sohel Rahman, and William F Smyth. Algorithms to Compute the Lyndon Array. In Jan Holub and Jan Žďárek, editors, *Prague Stringology Conference*, pages 172–184, 2016. URL: http://www.stringology.org/event/2016/p15.html.

22 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

23 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform, 2012. doi:10.48550/arXiv.1201.3077.

24 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Los Alamitos, CA, USA, 1989. IEEE Computer Society. doi:10.1109/sfcs.1989.63533.

25 Tsuruta Kazuya, Köppl Dominik, Nakashima Yuto, Inenaga Shunsuke, Bannai Hideo, Takeda Masayuki, Tsuruta Kazuya, Köppl Dominik, Nakashima Yuto, Inenaga Shunsuke, Bannai Hideo, and Takeda Masayuki. Grammar-compressed Self-index with Lyndon Words. *Information Processing Society of Japan*, 13:84–92, 2020. URL: https://ipsj.ixsq.nii.ac.jp/records/206700.

26 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, Stoc 2018, pages 827–840, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3188745.3188814.

27 Manfred Kufleitner. On bijective variants of the burrows-wheeler transform. In Jan Holub and Jan Žďárek, editors, *Prague Stringology Conference*, pages 65–79, Czech Technical University in Prague, Czech Republic, 2009. URL: http://www.stringology.org/event/2009/p07.html.

28 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.

29 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40(12):btae717, 2024. doi:10.1093/bioinformatics/btae717.

30 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

31 Felipe A. Louza, Sabrina Mantaci, Giovanni Manzini, Marinella Sciortino, and Guilherme P. Telles. Inducing the Lyndon Array. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 138–151, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32686-9_10.

32 Sabrina Mantaci, Antonio Restivo, G. Rosone, and Marinella Sciortino. An Extension of the Burrows Wheeler Transform and Applications to Sequence Comparison and Data Compression. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *16th Annual Symposium on Combinatorial Pattern Matching*, pages 178–189. Springer, 2005. doi:10.1007/11496656_16.

33 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting Suffixes of a Text via its Lyndon Factorization. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference*, pages 119–127, 2013. URL: http://www.stringology.org/event/2013/p11.html.

34 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffix array and Lyndon factorization of a text. *Journal of Discrete Algorithms*, 28:2–8, 2014. doi:10.1016/j.jda.2014.06.001.

35 Francesco Masillo. Matching Statistics Speed up BWT Construction. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms*, volume 274 of *Leibniz International Proceedings in Informatics*, pages 83:1–83:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.83.

36 Lucas Mercier and Philippe Chassaing. The height of the Lyndon tree. *Discrete Mathematics & Theoretical Computer Science*, DMTCS Proceedings vol. AS, 25th International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC 2013), January 2013. doi:10.46298/dmtcs.2357.

37 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *2009 Data Compression Conference*, pages 193–202, 2009. doi:10.1109/dcc.2009.42.

38 Jannik Olbrich. lg. Software, swhId: swh:1:dir:0274cae4abb893b49bc035ef3a57986a418afe40 (visited on 2025-09-04). URL: https://gitlab.com/qwerzuiop/lyndongrammar, doi:10.4230/artifacts.24673.

39 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. On the Optimisation of the GSACA Suffix Array Construction Algorithm. In Diego Arroyuelo and Barbara Poblete, editors, *String Processing and Information Retrieval*, pages 99–113, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-20643-6_8.

40 Jannik Olbrich, Enno Ohlebusch, and Thomas Büchler. Generic Non-recursive Suffix Array Construction. *ACM Transactions on Algorithms*, 20(2):18, 2024. doi:10.1145/3641854.

41 Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big BWTs. In *2023 data compression conference*, pages 62–70. IEEE, 2023. doi:10.1109/DCC55655.2023.00014.

42 Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A Space-Optimal Grammar Compression. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67:1–67:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2017.67.