Vol.2019-AL-175 No.19
2019/11/29

Constructing the Bijective BWT

Hideo Bannai[1,a)] Juha Kärkkäinen[3,b)] Dominik Köppl[1,2,c)]
Marcin Piątkowski[4,d)]

Abstract: The Burrows-Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* (BBWT) is a bijective variant of it. Although it is known that the BWT can be constructed in linear time for integer alphabets by using a linear time suffix array construction algorithm, it was up to now only conjectured that the BBWT can also be constructed in linear time. We confirm this conjecture by proposing a construction algorithm that is based on SAIS, improving the best known result of $\mathcal{O}(n \lg n / \lg\lg n)$ time to linear.

1. Introduction

The Burrows-Wheeler transform (BWT) [4] is a transformation permuting all symbols of a given string $T\$$, where $\$$ is a symbol that is strictly smaller than all symbols occurring it T. The i-th entry of the BWT of $T\$$ is the character preceding the i-th lexicographically smallest suffix of $T\$$, or $\$$ if this suffix is $T\$$ itself. Strictly speaking, the BWT is not a bijection since its output contains $\$$ at an arbitrary position while it requests the input T to have $\$$ as a delimiter symbol at its end in order to restore T. A variant, called the bijective BWT [15], is a *bijective* transformation, which does not require the artificial delimiter $\$$. It is based on the Lyndon factorization [5] of T. In this variant, the output consists of the last symbols of the lexicographically sorted cyclic rotations of all factors composing the Lyndon factorization of T.

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. It is well known that the traditional BWT has many applications in data compression [1] and text indexing [8–10]. Recently, such a text index was adapted to work with the bijective BWT [2].

Related Work. In what follows, we focus on a text T of length n whose characters are drawn from an integer alphabet. Thanks to linear time suffix array construction algorithms [14, 18], we can construct the traditional BWT based on the relation $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$ in linear time. Considering the bijective BWT, Gil and Scott [11] postulated that it can be built in linear time, but did not give a construction algorithm. It is clear that the time is upper bounded by the total length of all conjugates [17, after Example 9], which is $\mathcal{O}(n^2)$. Mantaci et al. [17] also introduced the *extended* BWT, a generalization of the BBWT in that it is a BWT based on a set $\mathcal{S}$ of *primitive* strings, i.e., strings that are not periodic. Hon et al. [12] provided an algorithm building the extended BWT in $\mathcal{O}(n \lg n)$ time. Their idea is to construct the *circular suffix array* $\mathsf{SA}_\circ$ such that the i-th position of the extended BWT is given by $T[\mathsf{SA}_\circ[i] - 1]$, where T is the concatenation of all strings in $\mathcal{S}$. Bonomo et al. [3] presented the most recent algorithm building the bijective BWT online in $\mathcal{O}(n \lg n / \lg\lg n)$ time. In [3, Sect. 6], they also gave a linear time reduction from computing the extended BWT to computing the BBWT. Knowing that an irreducible word has exactly one conjugate being a Lyndon word, the reduction is done by exchanging each element of the set of irreducible strings $\mathcal{S}$ by the conjugate being a Lyndon word, and concatenating these Lyndon words after sorting them in descending order. Consequently, a linear-time BBWT construction algorithm can be used to compute the extended BWT in linear time.

Our Result. In this article, we present a linear time algorithm computing the BBWT. The main idea is to adapt the suffix array construction algorithm SAIS [18] to compute the circular suffix array of the Lyndon factors. We obtain linear running time by exploiting some facts based on the nature of the Lyndon factorization.

2. Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. In this article, we study strings on an *integer* alphabet Σ with size $\sigma = n^{\mathcal{O}(1)}$:

[1] Department of Informatics, Kyushu University, Japan
[2] Japan Society for Promotion of Science
[3] Helsinki Institute of Information Technology (HIIT), Finland
[4] Nicolaus Copernicus University, Toruń, Poland
[a)] bannai@inf.kyushu-u.ac.jp
[b)] juha.karkkainen@cs.helsinki.fi
[c)] dominik.koeppl@inf.kyushu-u.ac.jp
[d)] marcin.piatkowski@mat.umk.pl

2.1 Strings

We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer j with $1 \le j \le |T|$, we access the j-th character of T with $T[j]$. Concatenating a string $T \in \Sigma^*$ k times is abbreviated by T^k. When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X, Y, and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. A prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. For two integers i, j with $1 \le i \le j \le |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. In particular, the suffix starting at position j of T is denoted with $T[j..n]$.

Orders on Strings. We denote the *lexicographic order* with $\prec_{\mathrm{lex}}$. Given two string S and T, then $S \prec_{\mathrm{lex}} T$ if S is a proper prefix of T or there exists an integer ℓ with $1 \le \ell \le \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] < T[\ell]$. We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS\cdots$ is lexicographically smaller than $T^\omega := TTT\cdots$. For instance, $\mathtt{ab} \prec_{\mathrm{lex}} \mathtt{aba}$ but $\mathtt{aba} \prec_\omega \mathtt{ab}$.

2.2 Lyndon Words

Given a string $T = T[1..n]$, its i-th *conjugate* $\mathsf{conj}_i(T)$ is defined as $T[i+1..n]T[1..i]$ for an integer $i \in [0..n-1]$. We say that T and every of its conjugates belongs to the *conjugate class* $\mathsf{conj}(T) := \{\mathsf{conj}_0(T), \ldots, \mathsf{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [16]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec S$ for every proper suffix S of T. A consequence is that a Lyndon word is *border-free*, i.e., there is no Lyndon word $T = SUS$ with $S \in \Sigma^+$ and $U \in \Sigma^*$.

The *Lyndon factorization* [5] of $T \in \Sigma^+$ is the factorization of T into a sequence of lexicographically non-increasing Lyndon words $T_1 \cdots T_t$, where (a) each $T_x \in \Sigma^+$ is a Lyndon word, and (b) $T_x \succ_{\mathrm{lex}} T_{x+1}$ for each $1 \le x < t$.

Lemma 2.1 ([7, Algo. 2.1])**.** The Lyndon-factorization of a string can be computed in linear time.

Each Lyndon word T_x is called a *Lyndon factor*. We denote the multiset of T's Lyndon factors by $\mathsf{LynF}(T) := \{T_1, \ldots, T_t\}$. For what follows, we fix a string $T[1..n]$ over an alphabet Σ of size σ. We use the string $T := \mathtt{cbbcacbbcadacbadacba}$ as our running example. Its Lyndon factorization is $\mathsf{LynF}(T) = \{\mathtt{c}, \mathtt{bbc}, \mathtt{acbbcad}, \mathtt{acbad}, \mathtt{acb}, \mathtt{a}\}$.

2.3 Bijective Burrows-Wheeler transform

We denote the bijective BWT of T by BBWT, where $\mathsf{BBWT}[i]$ is the last character of the i-th string in the list storing the conjugates of all Lyndon factors $T_1, \ldots, T_t$ of T sorted with respect to $\prec_\omega$. Figure 1 shows the BBWT of our running example.

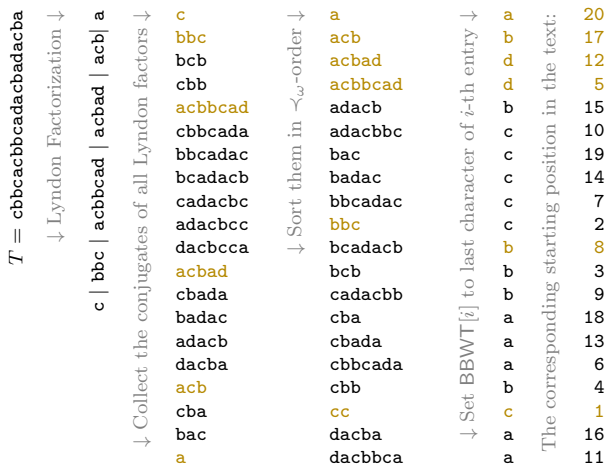

Fig. 1 Constructing BBWT of $T = \mathtt{cbbcacbbcadacbadacba}$. The Lyndon factors are colored in dark yellow. Reading the characters of the penultimate column top-down yields BBWT. The last column shows in its i-th row the starting position of the i-th smallest conjugate of a Lyndon factor in the text. It is the circular suffix array studied later in Sect. 3. Note that $\mathtt{cbb} \prec_{\mathrm{lex}} \mathtt{cbbcada}$, but $\mathtt{cbbcada} \prec_\omega \mathtt{cbb}$.

3. Linear-Time Construction of BBWT

In a pre-computation step, we want to facilitate the computation by removing all identical Lyndon factors from T yielding a reduced string R. We want to remove them since a naive character-wise comparison of the same string in the $\prec_\omega$-order does not terminate. Consequently, the first step is to show that we can obtain the BBWT of T from the circular suffix array of R:

The *composed Lyndon factorization* of T is given by $T_1^{\tau_1} \cdots T_t^{\tau_t} = T$ with $T_1 \succ_{\mathrm{lex}} \ldots \succ_{\mathrm{lex}} T_t$ and $\tau_x \ge 1$ for $x \in [1..t]$. Let $R := T_1 \cdots T_t$ denote the text, in which all duplicate Lyndon factors are removed. Obviously, the Lyndon factorization of R is $\mathsf{LynF}(R) = \{T_1, \ldots, T_t\}$. Let $\mathsf{b}(T_x)$ and $\mathsf{e}(T_x)$ denote the starting and ending position of the x-th Lyndon factor in R, i.e., $T[\mathsf{b}(T_x)..\mathsf{e}(T_x)]$ is the x-th Lyndon factor T_x of R.

Our aim is to compute the $\prec_\omega$-order of all conjugates of all Lyndon factors of R, which are given by the set $\mathcal{S} := \bigcup_{x \in [1..t]} \mathsf{conj}(T_x)$. Like Hon et al. [13], we present this order in the so-called *circular suffix array* $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, i.e., an array of length $|R|$ with $\mathsf{SA}_\circ[k] = i$ if $R[i..\mathsf{e}(T_x)]R[\mathsf{b}(T_x)..i-1]$ is the k-th smallest string in $\mathcal{S}$ with respect to $\prec_\omega$, where $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. The length of $\mathsf{SA}_\circ$ is $|R|$ since we can associate each text position $\mathsf{SA}_\circ[k]$ in R with a conjugate starting with $T[\mathsf{SA}_\circ[k]]$.

Having the circular suffix array $\mathsf{SA}_\circ$ of $\{T_1, \ldots, T_t\}$, we can compute the BBWT of T by reading $\mathsf{SA}_\circ[k]$ for $k \in [1..|R|]$ from left to right: Given $\mathsf{SA}_\circ[k] = i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$, we append $T_x[j]$ τ_x-times to BBWT, where $j := (|T_x| + i - \mathsf{b}(T_x) - 1) \mod |T_x| + \mathsf{b}(T_x)$ is $i-1$ or $\mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$.

3.1 Reviewing SAIS

Our idea is to adapt SAIS to compute $\mathsf{SA}_\circ$ instead of the suffix array. To explain this adaptation, we briefly review SAIS. First, SAIS assigns each suffix a type, which is either $\mathtt{L}$ or $\mathtt{S}$:

- $R[i..|R|]$ is type $\mathtt{L}$ if $R[i..|R|] \succ_{\mathrm{lex}} R[i+1..|R|]$, or

- $R[i..|R|]$ is type S otherwise, i.e., $R[i..|R|] \prec_{\text{lex}} R[i+1..|R|]$.

Since it is not possible that $R[i..|R|] = R[i+1..|R|]$, SAIS assigns each suffix a type. An S suffix $R[i..|R|]$ is additionally type $\mathtt{S}^*$ if $i = 1$ or $R[i-1..|R|]$ is type L. The substring between two succeeding $\mathtt{S}^*$ suffixes is called an *LMS substring*. In other words, $R[i..j]$ is an LMS substring if and only if $R[i..|R|]$ and $R[j..|R|]$ are type $\mathtt{S}^*$ and there is no $k \in (i..j)$ such that $R[k..|R|]$ is type $\mathtt{S}^*$. The types for the suffixes of our running example are given in Fig. 2.

Next, SAIS gives the LMS substrings a rank based on the *substring order* [18, Def. 3.3]: Given two LMS substrings S and U with $S \neq U$, we write $S \prec_{\text{LMS}} U$ if and only if (a) $S[i] < U[i]$ or (b) $S[i]$ is type L and $U[i]$ is type S or $\mathtt{S}^*$, for the smallest position i where (a) $S[i] \neq U[i]$ or (b) the types of $S[i]$ and $U[i]$ differ. This order is on the left side of Fig. 3 for the LMS suffixes of the left side of Fig. 2.

Having the $\prec_{\text{LMS}}$-order of all LMS substrings, we can assign each LMS substring its $\prec_{\text{LMS}}$-rank, and replace the LMS substrings in R by the respective ranks, keeping the last character during a replacement remaining if this character is the first character of the subsequent LMS substring. See the right side of Fig. 2 for our running example. We recursively call SAIS on this text of ranks until all LMS substrings have a different rank, since then these ranks determine the order of the $\mathtt{S}^*$ suffixes of R. The order of the $\mathtt{S}^*$ suffixes of our running example are given in Fig. 3 on the right side. Having the order of the $\mathtt{S}^*$ suffixes, we allocate space for the suffix array, and divide the suffix array into buckets, grouping each suffix with the same starting character and same type (either L or S) into one bucket. Putting $\mathtt{S}^*$ suffixes in their respective buckets according to their order (smallest elements are the leftmost elements in the buckets), we can induce the L suffixes, as these precede either L or $\mathtt{S}^*$ suffixes. Since an L suffix immediately preceding an $\mathtt{S}^*$ suffix is smaller than an L suffix immediately preceding two or more L suffixes, we can induce all L suffixes by a scan of the suffix array from left to right: When accessing the entry $\mathsf{SA}_\circ[k] = i$, write $i-1$ to the L type bucket with the character $R[i-1]$ if $R[i-1..|R|]$ is type L. Finally, we can induce those S suffixes that are not type $\mathtt{S}^*$ by scanning the suffix array from right to left: When accessing the entry $\mathsf{SA}_\circ[k] = i$, write $i-1$ to the S type bucket with the character $R[i-1]$ if $R[i-1..|R|]$ is type S. We conduct these steps for our running example in Fig. 4.

In total, the induction takes $\mathcal{O}(|R|)$ time. The recursion step takes also $\mathcal{O}(|R|)$ time since there are at most $|R|/2$ LMS substrings (there are no two text position $R[i]$ and $R[i+1]$ with type $\mathtt{S}^*$ for $i \in [1..n-1]$).

However, with SAIS we cannot obtain $\mathsf{SA}_\circ$ ad-hoc, since we need to exchange $\prec_{\text{lex}}$ with $\prec_\omega$. Although these orders are the same for Lyndon words, they differ for LMS substrings as can be seen in Fig. 5. Hence, we need to come up with an idea to modify SAIS in such way to compute $\mathsf{SA}_\circ$.

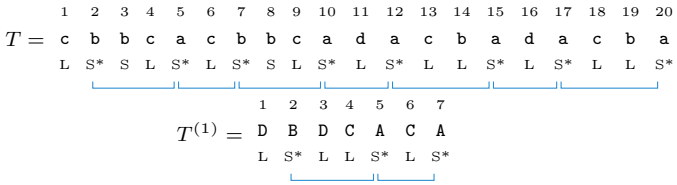

Fig. 2 Splitting T and $T^{(1)}$ into LMS substrings. The rectangular brackets below the types represent the LMS substrings. $T^{(1)}$ is T after the replacement of its LMS substrings with their corresponding ranks defined in Sect. 3.3 and on the left of Fig. 3.

LMS Substring	Contents	Non-Terminal
$T[2..5]$	bbca	D
$T[5..7]$	acb	B
$T[7..10]$	bbca	D
$T[10..12]$	ada	C
$T[12..15]$	acba	A
$T[15..17]$	ada	C
$T[17..20]$	acba	A

$\mathtt{S}^*$ Suffix	Contents
$T[20]$	a
$T[17..20]$	acba
$T[12..20]$	acbadacba
$T[5..20]$	acbbcadacbadacba
$T[15..20]$	adacba
$T[10..20]$	adacbadacba
$T[2..20]$	bbcacbbcadacbadacba
$T[7..20]$	bbcadacbadacba

Fig. 3 Ranking of the LMS substrings and the $\mathtt{S}^*$ suffixes of the text T given in Sect. 3.3 and Fig. 2. *Top*: LMS substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\text{LMS}}$-order. *Bottom*: $\mathtt{S}^*$ suffixes of T sorted in $\prec_{\text{lex}}$-order. Note that $T[5..7] = \mathtt{acb} \prec_{\text{lex}} \mathtt{acba} = T[12..15] = T[17..20]$, but $\mathtt{acba} \prec_{\text{LMS}} \mathtt{acb}$.

3.2 Our Adaptation

We want SAIS to sort Lyndon conjugates in $\prec_\omega$-order instead of suffixes in $\prec_{\text{lex}}$-order. For that, we introduce the notion of inf-suffixes, replacing the suffixes as the elements to sort in SAIS: Let $R[i..]$ denote the infinite string $R[i..\mathsf{e}(T_x)]T_xT_x\cdots = \mathsf{conj}_{i-1}(T_x)\mathsf{conj}_{i-1}(T_x)\cdots$ with x such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. We say that $R[i..]$ is an *inf-suffix*. The *factorization borders* are between $R[\mathsf{e}(F_x)]R[\mathsf{b}(F_{x+1})]$ for $x \in [1..t-1]$. Like in SAIS, we distinguish between L and S inf-suffixes:

- $R[i..]$ is type L if $R[i..] \succ_{\text{lex}} R[j..]$, and
- $R[i..]$ is type S if $R[i..] \prec_{\text{lex}} R[j..]$,

where $j := (i - \mathsf{b}(T_x) + 1) \mod |T_x| + \mathsf{b}(T_x)$ is either $i+1$ or $\mathsf{b}(T_x)$ if $i = \mathsf{e}(T_x)$, and x is given such that $i \in [\mathsf{b}(T_x)..\mathsf{e}(T_x)]$. When speaking about types, we do not distinguish between an inf-suffix and its starting position in R. This definition assigns all positions of R a type except those belonging to a Lyndon factor of length one. In all other cases, thanks to the Lyndon factorization, this definition matches the definitions of L and S suffixes of the SAIS algorithm. That is because of two facts:

- A Lyndon factor T_x of length at least two starts with the smallest character among all other characters of T_x. Since a Lyndon word is border-free, $R[\mathsf{b}(T_x)]$ is type S.
- Due to the Lyndon factorization, $R[\mathsf{b}(T_x)..|R|] \succ_{\text{lex}} R[\mathsf{b}(T_{x+1})..|R|]$ for $x \in [1..t-1]$. Hence, the suffix

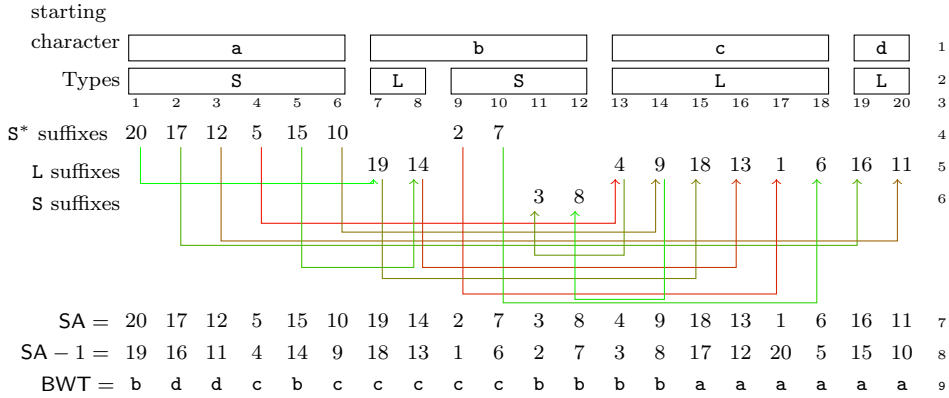

Fig. 4 Inducing L and S suffixes from the $\prec_{\mathrm{lex}}$-order of the S* suffixes given in Fig. 2. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective suffixes, and second by the types L and S. Row 4 is SA after inserting the S* suffixes according to their $\prec_{\mathrm{lex}}$-order rank obtained from the right of Fig. 3. The S* (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). Putting all together yields SA in Row 7. In the penultimate row SA − 1, each text position stored in SA is decremented by one. The last row shows $T[(\mathsf{SA}-1)[i]] = \mathsf{BWT}[i]$ in its i-th column, which is the BWT of T. This BWT is not reversible since the input is not terminated with a unique character like $. To obtain the BWT of $T$$, we first write $T[\mathsf{SA}[1]] = T[20] =$ a, and then BWT, but exchanging $\mathsf{BWT}[\mathsf{SA}^{-1}[1]] = \mathsf{BWT}[17] =$ a with $, i.e., abddcbcccccbbbbaa$aaa.

U	V	$\prec_{\mathrm{lex}}$	$\prec_{\omega}$	$\prec_{\mathrm{LMS}}$
aba	aca	<	<	<
adc	adcb	<	>	>
acb	acba	<	<	>

Fig. 5 Comparison of the three orders studied in this paper applied to LMS substrings. Assume that U and V are substrings of the text, neighbored by a character d such that the first and the last character of both U and V start with an S* suffix. We mark with the signs < and > whether U is smaller or respectively larger than V according to the corresponding order. The orders can differ only when one string is the prefix of another string, as this is the case in the last two rows.

$R[\mathsf{e}(T_x)..|R|]$ starting at $R[\mathsf{e}(T_x)]$ has to be lexicographically larger than the suffix $R[\mathsf{e}(T_x)+1..|R|]$, otherwise we could extend the Lyndon factor T_x.

To give all positions a type, we introduce the type S*, which is handled like a special case of type S:

- If $R[i..]$ is type S, it is further type S* if $R[j..]$ is type L with $j := (|T_x|+i-\mathsf{b}(T_x)-1) \mod |T_x|+\mathsf{b}(T_x)$ being either $i-1$ or $\mathsf{e}(T_x)$ if $i = \mathsf{b}(T_x)$.
- $R[\mathsf{b}(T_x)]$ is type S* for every $x \in [1..t]$.

If T_x and T_{x+1} are longer than one, then the types of all positions of $R[\mathsf{b}(T_x)+1..\mathsf{e}(T_{x+1})]$ match the original SAIS types.[*1] That is because the second condition comes into play only in the case that $|T_x| = 1$, since otherwise the last character $R[\mathsf{e}(T_x)]$ with $R[\mathsf{e}(T_x)] > R[\mathsf{b}(T_x)]$ is type L, Further, since $R[\mathsf{e}(T_x)..|R|] \succ_{\mathrm{lex}} R[\mathsf{b}(T_x)..|R|] \succ_{\mathrm{lex}} R[\mathsf{b}(T_{x+1})..|R|]$, the suffix $R[\mathsf{e}(T_x)..|R|]$ is an L suffix.

Next, we define the equivalent to the LMS substrings for the inf-suffixes, which we call *LMS inf-suffixes*. We want the LMS inf-suffixes to be contained inside the Lyndon factors since the $\prec_{\omega}$-order of a conjugate depends only on the order of its characters, and not, unlike suffixes, on all succeeding characters in the text. To obtain this property, we only have to change the original definition of the LMS substrings slightly: Stipulating that $T_x[|T_x|+1] = T_x[1]$, for $1 \le i < j \le |T_x|+1$, the substring $T_x[i..j]$ is an *LMS inf-substring* if $T_x[i]$ and $T_x[j]$ are type S* and there is no $k \in (i..j)$ such that $T_x[k]$ is type S*. This definition differs from the original LMS substrings only for the last LMS inf-substring of each Lyndon factor. Here, we append $T_x[1]$ instead of $T_{x+1}[1]$ to the suffix starting with the last type S* position of T_x. If T_x has length one, it is possible that $R[\mathsf{b}(T_x)..|R|]$ is not an S* suffix, while $R[\mathsf{b}(T_x)..]$ is always an S* inf-suffix with the associated LMS inf-substring $R[\mathsf{b}(T_x)]R[\mathsf{b}(T_x)]$.

Exactly as in the SAIS recursion step, we map the LMS inf-substrings to meta-characters having its $\prec_{\mathrm{LMS}}$-rank assigned. Since the Lyndon factorization of the string based on the meta-characters has the same factorization borders as the original string, we can continue with our deviation of SAIS by building LMS inf-substrings of the text based on the meta-characters while keeping the same factorization borders.

By doing so, we compute the $\prec_{\omega}$ -order of all conjugates of the Lyndon factors of R (instead of the lexicographic order of all suffixes of R). The correctness follows by construction: Instead of partitioning the suffixes into LMS substrings, we partition the Lyndon factors whose factorization borders happens to coincide with some borders of the LMS substrings. We use the same trick of the LMS substring partitioning, since we can obtain the $\prec_{\mathrm{lex}}$ of the S* inf-suffixes from the $\prec_{\mathrm{lex}}$-order of the LMS inf-substrings in the same way as obtaining the $\prec_{\mathrm{lex}}$ of the S* suffixes from the $\prec_{\mathrm{lex}}$-order of the LMS substrings. Finally, the induction steps can be conducted in the same way as in SAIS when taking care of the Lyndon boundaries, i.e., moving to the end of a

[*1] If $T[\mathsf{b}(T_{x-1})] = T[\mathsf{b}(T_x)]$ and $|T_{x-1}| = 1$, then $T[\mathsf{b}(T_x)..|R|]$ is not an S* suffix by the original definition.

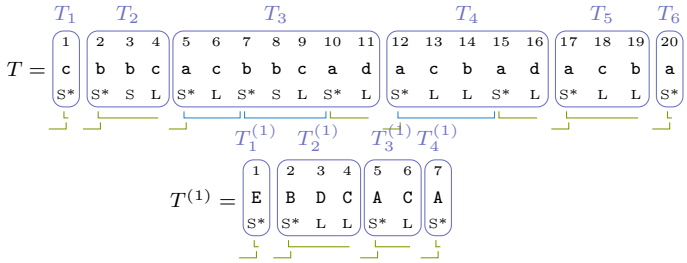

Fig. 6 Splitting T and $T^{(1)}$ into LMS inf-substrings. The rectangular brackets below the types represent the LMS inf-substrings. Broken brackets denote that the corresponding LMS inf-substring ends with the first character of the Lyndon factor in which it is contained. $T^{(1)}$ is T after the replacement of its LMS inf-substrings with their corresponding ranks defined in Sect. 3.3 and on the left of Fig. 7.

LMS Inf-Substring	Contents	Non-Terminal
$T[1]T[1]$	cc	-
$T[2..4]T[2]$	bbcb	E
$T[5..7]$	acb	B
$T[7..10]$	bbca	D
$T[10..11]T[10]$	ada	C
$T[12..15]$	acba	A
$T[15..16]T[12]$	ada	C
$T[17..19]T[17]$	acba	A
$T[20]T[20]$	aa	-

S* Inf-Suffix	Contents
$T[20..]$	a...
$T[17..]$	acb...
$T[12..]$	acbad...
$T[5..]$	acbbcad...
$T[15..]$	adacb...
$T[10..]$	adacbbc...
$T[7..]$	bbcadac...
$T[2..]$	bbc...
$T[1..]$	c...

Fig. 7 Ranking of the LMS inf-substrings and the S* suffixes of the text T given in Sect. 3.3 and Fig. 6. *Top*: LMS inf-substrings assigned with non-terminals reflecting their corresponding rank in $\prec_{\mathrm{LMS}}$-order. The first and the last LMS substring do not receive a non-terminal since their lengths are one. *Bottom*: S* inf-suffixes of T sorted in $\prec_{\mathrm{lex}}$-order, which corresponds to the $\prec_\omega$ of the conjugate starting with this inf-suffix. Compared with Fig. 3, the suffixes $T[2..20]$ and $T[7..20]$ in the $\prec_{\mathrm{lex}}$-order are order differently than their respective inf-suffixes $T[2..]$ and $T[7..]$ in the $\prec_{\mathrm{lex}}$-order.

Lyndon factor instead of moving from its first position one position backwards.

However, there is a problem with the time bounds: Since two positions $R[i]$ and $R[i+1]$ are type S* if $R[i]$ belongs to a Lyndon factor of length one, we cannot bound the maximum number of all S* inf-suffixes by $n/2$. In fact, the situation is worse, since we keep the Lyndon factorization in all levels of the recursive call, such that we can have $\Theta(n)$ LMS inf-suffixes on all levels. In the following, we omit the Lyndon factors of length one to restore the upper bound on the number of all S* inf-suffixes. To omit them, we need to think about their order such that we can reinsert them after the recursive call at the right position: Suppose that there is a Lyndon factor consisting of a single character b (the following holds if $\mathtt{b} \in \Sigma$ or if b is a rank of an LMS substring considered in the recursive call). All LMS inf-substrings larger than one starting with b are larger than bb in the $\prec_\omega$-order because such an LMS inf-substring starting with $R[i]$ having type S* is lexicographically smaller than $R[i+1..]$. Consequently, $\mathtt{bb}\cdots \prec_{\mathrm{lex}} R[i..] = \mathtt{b}R[i+1..]$ since $\mathtt{b}\cdots \prec_{\mathrm{lex}} R[i+1..]$. Thus, the Lyndon factor consisting of the single character b does not have to be tracked further in the recursive call since we know that its rank precedes the ranks of all other LMS inf-substrings starting with b. After the recursion, we can simply insert all omitted LMS inf-substrings into the order returned by the recursive call by a linear scan. Overall, by omitting the single character LMS inf-substrings, we retain the $\mathcal{O}(n)$ time of SAIS.

3.3 Elaborated Example

The LMS inf-substrings of our running example $T :=$ cbbcacbbcadacbadacba with $R = T$ are given in Fig. 6. Their $\prec_{\mathrm{LMS}}$-ranking is given on the left side of Fig. 7, where we associate each LMS inf-substring, except those consisting of a single letter, with a non-terminal reflecting its rank. By replacing the LMS inf-substrings by their $\prec_{\mathrm{LMS}}$-ranks in the text while discarding the single letter Lyndon factors, we obtain the string $T^{(1)} :=$ DCFBFABA, whose LMS inf-substrings are given on the right side of Fig. 6. Among these LMS inf-substrings, only CFC, BFB, and ABA are interesting. Finding their $\prec_{\mathrm{LMS}}$-ranks gives us the $\prec_\omega$-order of the S* inf-suffixes as shown on the right side of Fig. 7. It is left to induce the L and S suffixes, which is done exactly as in the SAIS algorithm. We conduct these steps in Fig. 8, which finally lead us to $\mathsf{SA}_\circ$.

Open Problems

The BBWT is bijective in the sense that it transforms a string of Σ^n into another string of Σ^n while preserving distinctness. Consequently, given a string of length n, there is an integer $k \geq 1$ with $\mathsf{BBWT}^k(T) = \mathsf{BBWT}^{k-1}(\mathsf{BBWT}(T)) = T$. With our presented algorithm we can compute the smallest such number k in $\mathcal{O}(n^k)$ time. However, we wonder whether we can compute this number faster, possible by scanning only the text in $\mathcal{O}(n)$ time independent of k.

We also wonder whether we can define the BBWT for the generalized Lyndon factorization [6]. Contrary to the Lyndon factorization, the generalized Lyndon factorization uses a different order, called the *generalized lexicographic order* $\prec_{\mathrm{gen}}$. In this order, two strings $S, T \in \Sigma^*$ are compared character-wise like in the lexicographic order. However, the generalized lexicographic order $\prec_{\mathrm{gen}}$ can use different orders $<_1, <_2, \ldots$ for each text position, i.e., $S \prec_{\mathrm{gen}} T$ if and only if S is a proper prefix of T or there is an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] <_\ell T[\ell]$.

Acknowledgments This work is funded by the JSPS KAKENHI Grant Number JP18F18120.

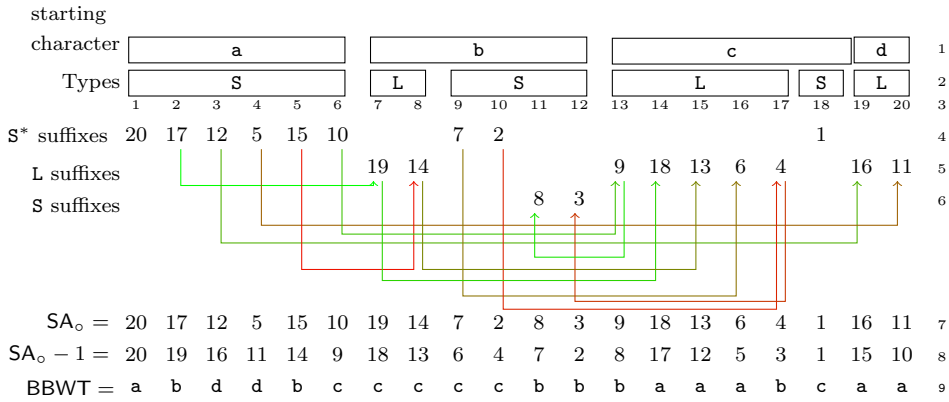

Fig. 8 Inducing L and S inf-suffixes from the $\prec_{\text{lex}}$-order of the S* inf-suffixes given in Fig. 6. Rows 1 and 2 show the partitioning of $\mathsf{SA}_\circ$ into buckets, first divided by the starting characters of the respective inf-suffixes, and second by the types L and S. Row 4 is $\mathsf{SA}_\circ$ after inserting the S* inf-suffixes according to their $\prec_{\text{lex}}$-order rank obtained from the right of Fig. 7.. The S* (resp. L) inf-suffixes induce the L (resp. S) inf-suffixes in Row 5 (resp. Row 6). Putting all together yields $\mathsf{SA}_\circ$ in Row 7. In the penultimate row $\mathsf{SA}_\circ - 1$, each text position stored in $\mathsf{SA}_\circ$is decremented by one, wrapping around a Lyndon factor if necessary (for instance, $(\mathsf{SA}_\circ - 1)[2] = 19 = \mathsf{e}(T_5)$ since $\mathsf{SA}_\circ[2] = 17 = \mathsf{b}(T_5)$). The last row shows $T[(\mathsf{SA}_\circ - 1)[i]]$ in its i-th column, which is the BBWT of T as given in Fig. 1.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019.

[3] S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.

[4] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[5] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

[6] F. Dolce, A. Restivo, and C. Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777: 232–242, 2019.

[7] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[9] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[10] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[11] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012.

[12] W. Hon, C. Lu, R. Shah, and S. V. Thankachan. Succinct indexes for circular patterns. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 673–682, 2011.

[13] W. Hon, T. Ku, C. Lu, R. Shah, and S. V. Thankachan. Efficient algorithm for circular Burrows-Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.

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

[15] M. Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC*, pages 65–79, 2009.

[16] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2): 202–215, 1954.

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

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