Lyndon Factorization of Grammar Compressed Texts Revisited

Isamu Furuya
Graduate School of IST, Hokkaido University, Japan
furuya@ist.hokudai.ac.jp

Yuto Nakashima
Department of Informatics, Kyushu University, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Tomohiro I
Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp
https://orcid.org/0000-0001-9106-6192

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai
Department of Informatics, Kyushu University, Japan
RIKEN Center for Advanced Intelligence Project, Japan
bannai@inf.kyushu-u.ac.jp
https://orcid.org/0000-0002-6856-5185

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

We revisit the problem of computing the Lyndon factorization of a string w of length N which is given as a straight line program (SLP) of size n. For this problem, we show a new algorithm which runs in $O(P(n,N) + Q(n,N)n \log\log N)$ time and $O(n \log N + S(n,N))$ space where $P(n,N)$, $S(n,N)$, $Q(n,N)$ are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. Our algorithm improves the algorithm proposed by I et al. (TCS '17), and can be more efficient than the $O(N)$-time solution by Duval (J. Algorithms '83) when w is highly compressible.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases Lyndon word, Lyndon factorization, Straight line program

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.24

Funding This work was supported by JSPS KAKENHI Grant Numbers JP17H06923 (YN), JP16K16009 (TI), JP17H01697 (SI), JP16H02783 (HB), and JP25240003 (MT).

1 Introduction

A string w is said to be a Lyndon word if w is lexicographically smaller than any of its proper suffixes. For instance, `abb` is a Lyndon word, but `bba` and `aba` are not. The Lyndon factorization of a string w is the sequence of strings $\ell_1^{p_1}, \ldots, \ell_m^{p_m}$ such that $w = \ell_1^{p_1} \cdots \ell_m^{p_m}$, ℓ_i

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 24; pp. 24:1–24:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is a Lyndon word, $p_i \geq 1 (1 \leq i \leq m)$, and $\ell_i \succ \ell_{i+1} (1 \leq i < m)$ [4]. Lyndon factorizations are used, for example, in a bijective variant of Burrows-Wheeler transform [13, 7] and an algorithm to check digital convexity [2].

Let LF_w denote the Lyndon factorization of a string w. Given a string w of length N, LF_w can be computed on-line in $O(N)$ time [6]. When the length N of the string w is huge, even the $O(N)$-time solution may not be efficient enough. I et al. [10] showed an efficient Lyndon factorization algorithm when the string w is given as a *straight line program (SLP)*, which is a compressed representation of the string based on a context free grammar that derives only w. The algorithm runs in $O(n^2 + P(n, N) + Q(n, N)n \log n)$ time and $O(n^2 + S(n, N))$ space where $P(n, N)$, $S(n, N)$, $Q(n, N)$ are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. This algorithm can be more efficient than the $O(N)$-time solution when w is highly compressible.

In this paper, we revisit the Lyndon factorization problem on SLPs and give a more efficient solution. Given an SLP $\mathcal{S}$ of size n representing a string w of length N, our new algorithm runs in $O(P(n, N) + Q(n, N)n \log \log N)$ time and $O(n \log N + S(n, N))$ space. If we use the LCE data structure of [8], we can compute the Lyndon factorization in $O(n \log N \log \log N)$ time and $O(n \log N)$ space. This improves the previous algorithm since $\log N \leq n$ holds.

We note that the previous algorithm [10] computes the Lyndon factorization in a bottom-up manner, which requires us to store the Lyndon factorization for *every* variable of a given SLP. This implies that we must use $\Omega(n^2)$ space (and thus time) in total because the size of each Lyndon factorization can be $\Omega(n)$. We show that the Lyndon factorization of w can be computed without computing the Lyndon factorization of each variable.

2 Preliminaries

2.1 Strings and model of computation

Let Σ be an ordered finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x of w is called a *proper prefix* of w if $x \neq w$. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. For any string w let $w^1 = w$, and for any integer $k \geq 2$ let $w^k = ww^{k-1}$, i.e., w^k is a k-time repetition of w.

If character a is lexicographically smaller than another character b, then we write $a \prec b$. For any strings x, y, let $lcp(x, y)$ be the length of the longest common prefix of x and y. We write $x \prec y$ iff either $x[lcp(x, y) + 1] \prec y[lcp(x, y) + 1]$ or x is a proper prefix of y.

Our model of computation is the word RAM. We assume the computer word size is at least $\lceil \log_2 |w| \rceil$, and hence, standard operations on values representing lengths and positions of string w can be manipulated in $O(1)$ time. Space complexities will be determined by the number of computer words (not bits).

2.2 Lyndon words and Lyndon factorization of strings

Two strings x and y are *conjugates*, if $x = uv$ and $y = vu$ for some strings u and v. A string w is said to be a *Lyndon word*, if w is lexicographically strictly smaller than all of its conjugates. Namely, w is a Lyndon word, if for any factorization $w = uv$, it holds that

$uv \prec vu$. An equivalent definition of Lyndon words is: a string w is a *Lyndon word*, if $w \prec v$ for any non-empty proper suffix v of w.

The *Lyndon factorization* of a string w, denoted LF_w, is the factorization $\ell_1^{p_1}, \ldots, \ell_m^{p_m}$ of w, such that each $\ell_i \in \Sigma^+$ is a Lyndon word, $p_i \geq 1$, and $\ell_i \succ \ell_{i+1}$ for all $1 \leq i < m$. The size of LF_w is m and denoted by $|LF_w|$. LF_w can be represented by the sequence $(|\ell_1|, p_1), \ldots, (|\ell_m|, p_m)$ of integer pairs, where each pair $(|\ell_i|, p_i)$ represents the i-th Lyndon factor $\ell_i^{p_i}$ of w. Note that this representation requires $O(m)$ space.

In the literature, the Lyndon factorization is sometimes defined to be a sequence of lexicographically non-increasing Lyndon words, namely, each Lyndon factor ℓ^p is decomposed into a sequence of p ℓ's. In this paper, each Lyndon word ℓ in the Lyndon factor ℓ^p is called a *decomposed Lyndon factor*.

For any string w, let $LF_w = \ell_1^{p_1}, \ldots, \ell_m^{p_m}$. Let $\mathit{lfb}_w(i)$ denote the position where the i-th Lyndon factor begins in w, i.e., $\mathit{lfb}_w(1) = 1$ and $\mathit{lfb}_w(i) = \mathit{lfb}_w(i-1) + |\ell_{i-1}^{p_{i-1}}|$ for any $2 \leq i \leq m$. For any $1 \leq i \leq m$, let $\mathit{lfs}_w(i) = \ell_i^{p_i}\ell_{i+1}^{p_{i+1}} \cdots \ell_m^{p_m}$ and $\mathit{lfp}_w(i) = \ell_1^{p_1}\ell_2^{p_2} \cdots \ell_i^{p_i}$. For convenience, let $\mathit{lfs}_w(m+1) = \mathit{lfp}_w(0) = \varepsilon$.

▶ **Example 1.** Let $w = \texttt{abcabcabababcbababababcababa}$. Then,

- $LF_w = (\texttt{abc})^2, \texttt{abababcb}, \texttt{abababc}, (\texttt{ab})^2, \texttt{a}$;
- the decomposed Lyndon factorization of w is $\texttt{abc}, \texttt{abc}, \texttt{abababcb}, \texttt{abababc}, \texttt{ab}, \texttt{ab}, \texttt{a}$.

Moreover, $\mathit{lfb}_w(2) = 7$, $\mathit{lfs}_w(3) = \texttt{abababcababa}$, and $\mathit{lfp}_w(2) = \texttt{abcabcabababcb}$.

The following is a useful lemma concerning Lyndon factorizations.

▶ **Lemma 2** (Lemma 4 of [11]). *Let $LF_w = \ell_1^{p_1}, \ldots, \ell_m^{p_m}$ and $1 \leq i, j \leq m$. Assume that $\ell_i^{p_i} \cdots \ell_j^{p_j}$ has an occurrence to the left in w. Then,*

1. *the leftmost occurrence of $\ell_i^{p_i} \cdots \ell_j^{p_j}$ is a prefix of ℓ_k for some $k < i$;*
2. *$\ell_i^{p_i} \cdots \ell_j^{p_j}$ is a prefix of every ℓ_h with $k \leq h < i$.*

2.3 Straight line programs (SLPs)

A *straight line program* (*SLP*) is a set of productions $\mathcal{S} = \{X_i \to \mathit{expr}_i\}_{i=1}^n$, where each X_i is a variable and each expr_i is an expression of the form $\mathit{expr}_i = a$ ($a \in \Sigma$), or $\mathit{expr}_i = X_l X_r$ ($i > l, r$). Let $\mathit{val}(X_i)$ denote the string derived by X_i. Also let $\mathit{val}(a) = a$ for $a \in \Sigma$. We will sometimes associate $\mathit{val}(X_i)$ with X_i and denote $|\mathit{val}(X_i)|$ as $|X_i|$. An SLP $\mathcal{S}$ *represents* the string $w = \mathit{val}(X_n)$. The *size* of the program $\mathcal{S}$ is the number n of productions in $\mathcal{S}$. If N is the length of the string represented by SLP $\mathcal{S}$, then N can be as large as 2^{n-1}.

The derivation tree $T_{\mathcal{S}}$ of SLP $\mathcal{S}$ is a labeled ordered tree obtained by recursively applying the productions of variable X_i, starting from X_n, i.e., the root node has label X_n, and for each internal node labeled X_i, if $X_i \to X_l X_r$, then its left child is labeled X_l and its right child is labeled X_r, if $X_i \to a$, then its single child is labeled a.

The height of SLP $\mathcal{S}$ is the height of $T_{\mathcal{S}}$. We associate to each leaf of $T_{\mathcal{S}}$ the corresponding position in string $w = \mathit{val}(X_n)$. An example of the derivation tree of an SLP is shown in Figure 1.

It is known that the size of the Lyndon factorization of w is a lower bound of the size of smallest SLP which derives w.

▶ **Lemma 3** (Lemma 17 of [10]). *For any string w, let $m = |LF_w|$, and n be the size of an SLP which derives w. Then $m \leq n$ holds.*

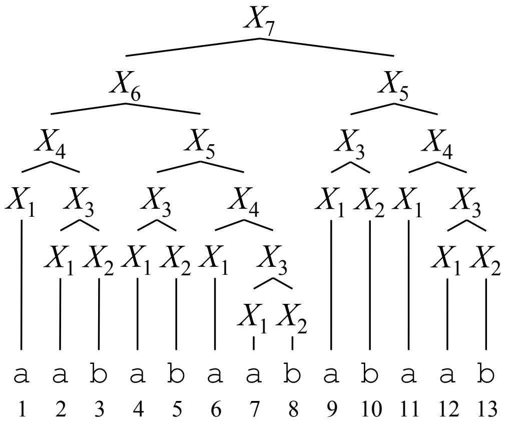
X_7
X_6
X_5
X_4
X_5
X_3
X_4
X_1 X_3 X_3 X_4 X_1 X_2 X_1 X_3
X_1 X_2 X_1 X_2 X_1 X_3 X_1 X_2
X_1 X_2
a a b a b a a b a b a a b
1 2 3 4 5 6 7 8 9 10 11 12 13

Figure 1 The derivation tree of SLP $\mathcal{S} = \{X_1 \to \texttt{a},\ X_2 \to \texttt{b},\ X_3 \to X_1X_2,\ X_4 \to X_1X_3,\ X_5 \to X_3X_4,\ X_6 \to X_4X_5,\ X_7 \to X_6X_5\}$, representing string $w = val(X_7) = \texttt{aababaababaab}$.

2.4 Longest common extension problem on SLPs

The longest common extension (LCE) problem on SLPs is to preprocess an SLP so that we can efficiently answer LCE queries that ask to compute $lcp(val(X_i)[k_1..|X_i|], val(X_i)[k_2..|X_i|])$ for any variable X_i and $1 \leq k_1, k_2 \leq |X_i|$. Currently, the best known deterministic solution to this problem is the following.

▶ **Lemma 4** (Theorem 2 of [8]). *Given an SLP of size n representing a string of length N, we can preprocess in $O(n \log(N/n))$ time and $O(n + t \log(N/t))$ space to support LCE queries in $O(\log N)$ time where t is the size of non-overlapping LZ77 factorization.*

In order to describe the complexity of our algorithm independent from the choice of LCE data structures, the preprocessing time, space and query time of the chosen LCE data structure are denoted by $P(n, N)$, $S(n, N)$ and $Q(n, N)$, respectively.

3 Lyndon factorization algorithm for SLP

In this paper, we propose a new Lyndon factorization algorithm for an SLP compressed text. More formally, we are given an SLP $\mathcal{S}$ which derives a string w, and we compute LF_w.

Firstly, we explain the idea of our algorithm. We compute the Lyndon factorization of w from left to right based on *G-factorization* defined as follows. The G-factorization of a string w that is derived by an SLP $\mathcal{S}$ is defined by the *Partial Parse Tree* of $\mathcal{S}$.

▶ **Definition 5** (Partial Parse Tree [14]). The partial parse tree of an SLP $\mathcal{S}$ is a subtree of the derivation tree of $\mathcal{S}$ such that each variable occurs exactly once as a label of an internal node and the occurrence is the leftmost possible.

▶ **Definition 6** (G-factorization [14]). The G-factorization of a string w that is derived by an SLP $\mathcal{S}$ is $GF_{(w,\mathcal{S})} = val(leaf_1), \ldots, val(leaf_g)$, where $leaf_1, leaf_2, \ldots, leaf_g$ is the sequence of leaf labels of the partial parse tree of an SLP $\mathcal{S}$.

Figure 2 shows the partial parse tree and the G-factorization of SLP $\mathcal{S}$ which was shown in Figure 1. Since the number of internal nodes of the partial parse tree of $\mathcal{S}$ is exactly the same as the number of variables of $\mathcal{S}$, it is clear that $|GF_{(w,\mathcal{S})}|$ as well as the size of the partial parse tree is $O(n)$, where n is the size of SLP $\mathcal{S}$.

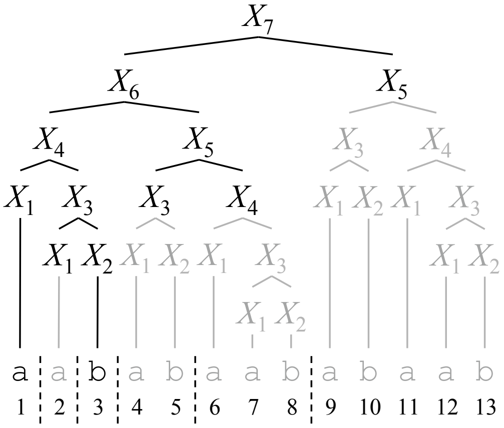

Figure 2 The partial parse tree of SLP $\mathcal{S}$ which was shown in Figure 1. The G-factorization of this string is shown by dash lines on the string.

Let $GF_{(w,\mathcal{S})} = \mathit{val}(\mathit{leaf}_1), \ldots, \mathit{val}(\mathit{leaf}_g)$ and $w_j = \mathit{val}(\mathit{leaf}_1) \cdots \mathit{val}(\mathit{leaf}_j)$ for any $1 \leq j \leq g$. Our algorithm consists of the following two parts.

1. Compute the set of significant suffixes of X_i for all $1 \leq i \leq n$.
2. Compute the Lyndon factorization and the significant suffixes of w_{j+1}.

Here, *significant suffixes* of a string are suffixes of the string, as defined in [9, 10] (also used in [12]), closely related to Lyndon factorizations, and will be explained in detail in Section 3.1.

The second part of our algorithm consists of g steps: In the $(j+1)$-th step of our algorithm, we compute $LF_{w_{j+1}}$ by using information computed for w_j and leaf_i. More precisely, we compute $LF_{w_{j+1}}$ by using:

- the Lyndon factorization of w_j,
- the significant suffixes of w_j, and
- the significant suffixes of leaf_i.

The rest of this section is organized as follows. In Section 3.1, we explain what significant suffixes are. We also show properties on significant suffixes which are used in our algorithm. In Section 3.2 and 3.3, we describe respectively, the first part and the second part of our algorithm.

3.1 Significant suffix

Assume that $LF_u = u_1^{p_1}, \ldots, u_m^{p_m}$. $\mathit{lfs}_u(i)$ is said to be a significant suffix of u if $\mathit{lfs}_u(i+1)$ is a prefix of $\mathit{lfs}_u(i)$ for any $1 \leq i \leq m$. It is clear that $\mathit{lfs}_u(m)$ (i.e., the last Lyndon factor) is always a significant suffix of u.

▶ **Lemma 7.** *Assume that $\mathit{lfs}_u(i+1)$ is a prefix of $\mathit{lfs}_u(i)$ for some $1 \leq i \leq m$. Then $\mathit{lfs}_u(j+1)$ is a prefix of $\mathit{lfs}_u(j)$ for any $i < j \leq m$.*

Proof. Assume that $\mathit{lfs}_u(i+1)$ is a prefix of $\mathit{lfs}_u(i)$ for some $1 \leq i \leq m$. Let $i < j \leq m$. By the definition, $\mathit{lfs}_u(j)$ and $\mathit{lfs}_u(j+1)$ are suffixes of $\mathit{lfs}_u(i+1)$. Thus, $\mathit{lfs}_u(j)$ and $\mathit{lfs}_u(j+1)$ are proper substrings of $\mathit{lfs}_u(i)$. By Lemma 2, $\mathit{lfs}_u(j)$ and $\mathit{lfs}_u(j+1)$ have to be proper prefixes of $\mathit{lfs}_u(i)$. Since $|\mathit{lfs}_u(j)| > |\mathit{lfs}_u(j+1)|$, then $\mathit{lfs}_u(j+1)$ is a prefix of $\mathit{lfs}_u(j)$. ◀

Let λ_u be the minimum integer such that $\mathit{lfs}_u(i+1)$ is a prefix of u_i for any $\lambda_u \leq i \leq m$. We define the set of significant suffixes Λ_u of u as $\Lambda_u = \{\mathit{lfs}_u(i) \mid \lambda_u \leq i \leq m\}$.

▶ **Example 8.** Let $w = \texttt{abcabcabababcbababababcababa}$ (same as Example 1). Then, $\lambda_w = 3$ and $\Lambda_w = \{\texttt{abababcababa}, \texttt{ababa}, \texttt{a}\}$ since $\mathit{lfs}_w(3)$ is not a prefix of $\mathit{lfs}_w(2)$, but $\mathit{lfs}_w(4)$ is a prefix of $\mathit{lfs}_w(3)$.

It is clear from the definition of Lyndon factorization and λ_u that for any $\lambda_u \leq i \leq m$, $u_i = \mathit{lfs}_u(i+1)y_i$ for some non-empty string y_i. We will represent Λ_u by the sequence $(\mathit{lfb}_u(\lambda_u), p_{\lambda_u}), \ldots, (\mathit{lfb}_u(m), p_m)$ of integer pairs. Note that this representation requires $O(\log |u|)$ space by the following lemma.

▶ **Lemma 9** (Lemma 12 of [10])**.** *For any string u, $|\Lambda_u| = O(\log |u|)$.*

3.2 Computing significant suffixes

For any strings u, v, let $LF_u = u_1^{p_1}, \ldots, u_m^{p_m} = U_1, \ldots, U_m$ where $U_i = u_i^{p_i}$ and $LF_v = v_1^{q_1}, \ldots, v_{m'}^{q_{m'}} = V_1, \ldots, V_{m'}$ where $V_i = v_i^{q_i}$. Our idea of computing significant suffixes is based on the following lemma used in [10].

▶ **Lemma 10** ([1, 5])**.** *$LF_{uv} = U_1, \ldots, U_c, z^k, V_{c'}, \ldots V_{m'}$ for some $0 \leq c \leq m$, $1 \leq c' \leq m'+1$ and $LF_{\mathit{lfs}_u(c+1)\mathit{lfp}_v(c'-1)} = z^k$.*

This lemma says that LF_{uv} can be obtained from LF_u and LF_v by computing the medial Lyndon factor z^k since the other Lyndon factors remain unchanged in uv.

Let $X_i = X_\ell X_r (1 \leq \ell, r < i \leq n)$. Assume that we have computed Λ_{X_ℓ} and Λ_{X_r}. Then we compute Λ_{X_i} from this information. The following lemmas are useful for our algorithm.

▶ **Lemma 11** (Lemma 16 of [10])**.** *$\lambda_u \leq c+1$.*

▶ **Lemma 12.** *$\mathit{lfb}_{uv}(\lambda_{uv}) \in \{\mathit{lfb}_u(i) \mid \lambda_u \leq i \leq c+1\} \cup \{|u| + \mathit{lfb}_v(\max\{c', \lambda_v\})\}$.*

Proof. By Lemma 10, V_j is a Lyndon factor of uv for any $c' \leq j \leq |LF_v|$. Hence, $\mathit{lfs}_v(k)$ is a significant suffix of uv for any $\max\{c', \lambda_v\} \leq k \leq m$. Let $1 \leq j < \lambda_u$. By Lemma 11, U_j is a Lyndon factor of uv. By the definition of significant suffix and Lemma 7, $\mathit{lfs}_u(j+1)$ is not a prefix of $\mathit{lfs}_u(j)$. From this fact, it is easy to see that $\mathit{lfs}_u(j+1)v$ is not a prefix of $\mathit{lfs}_u(j)v$, i.e., $\mathit{lfs}_{uv}(j+1)$ is not a prefix of $\mathit{lfs}_{uv}(j)$. Thus, $\mathit{lfb}_{uv}(\lambda_{uv}) \geq \mathit{lfb}_u(\lambda_u)$. Therefore, this lemma holds. ◀

From Lemma 10 and the definition of Lyndon factorization, there exists exact one z which begins in u and ends in v (if $u_m \prec v_1$). We refer to this z as *crossing factor*. By the next lemma, we can determine the lexicographic order between u_m and v_1 using a single LCE query $\mathit{lcp}(u_m v, v)$. We remark that we do not have to know $|v_1|$ as well as the Lyndon factorization of v.

▶ **Lemma 13.** *Let $\alpha = \mathit{lcp}(u_m v, v)$. Then,*

1. *$u_m \succ v_1$ if $\alpha < |u_m|$ and $u_m v \succ v$;*
2. *$u_m = v_1$ if $\alpha \geq |u_m|$ and $u_m v \succ v$;*
3. *$u_m \prec v_1$ if $u_m v \prec v$.*

Proof.

1. If $\alpha < |u_m|$ and $u_m v \succ v$, then the prefix of v of length α is not a prefix of any Lyndon words. This implies that the longest prefix of v which is a Lyndon word is shorter than α. Thus, $u_m \succ v_1$ holds.
2. If $\alpha \geq |u_m|$ and $u_m v \succ v$, then the prefix of v of length $\alpha + 1$ can be represented as $u_m^i u_m' c$ such that $i \geq 1$, u_m' is a prefix of u_m, and $u_m[|u_m'|+1] \succ c \in \Sigma$. This implies that u_m is the longest prefix of v which is a Lyndon word. Thus, $u_m = v_1$ holds.

3. If $\alpha \geq |u_m|$ and $u_m v \prec v$, then the prefix of v of length $\alpha + 1$ can be represented as $u_m^i u'_m c$ such that $i \geq 1$, u'_m is a prefix of u_m, and $u_m[|u'_m| + 1] \prec c \in \Sigma$. This implies that the prefix of v of length $\alpha + 1$ is a Lyndon word which has u_m as a prefix. Thus, $u_m \prec v_1$. If $\alpha < |u_m|$ and $u_m v \prec v$, then the prefix of v of length $\alpha + 1$ is a Lyndon word which is lexicographically larger than u_m. Thus, $u_m \prec v_1$. ◀

Due to the following lemma, we can compute the crossing factor efficiently.

▶ **Lemma 14** (Lemma 16 of [10]). *Assume that $u_m \prec v_1$. Let i, j be the beginning position and the ending position of the crossing factor z, respectively. Then*

- *$i = \mathit{lfb}_u(i')$ for some $\lambda_u \leq i' \leq m$,*
- *$\mathit{lfs}_u(1)v \succ \ldots \succ \mathit{lfs}_u(i')v \prec \ldots \prec \mathit{lfs}_u(m+1)v$,*
- *$j = \mathit{lfb}_u(j')$ such that $\mathit{lfs}_v(j'-1) \succ \mathit{lfs}_u(i')v \succ \mathit{lfs}_v(j')$.*

The following lemma is the main result of this section.

▶ **Lemma 15.** *We can compute all significant suffixes for each variable of an SLP $\mathcal{S}$ by $O(n \log\log N)$ lexicographical string comparisons.*

Proof. Let $i \leq n$. Assume that we have computed all significant suffixes of variable X_j for any $j < i$. We show how to compute all significant suffixes of variable $X_i = X_\ell X_r (\ell, r < i)$. Let $LF_{X_\ell} = U_1, \ldots, U_m$ and $LF_{X_r} = V_1, \ldots, V_{m'}$. Firstly, we compute the lexicographic order between u_m and v_1 by Lemma 13. This can be done by one LCE query.

Suppose that $u_m \succ v_1$. Then $LF_{X_i} = U_1, \ldots, U_m, V_1, \ldots, V_{m'}$ by the definition of Lyndon factorization. Since we have all significant suffixes of X_ℓ and X_r, we can compute all significant suffixes of X_i by $O(\log\log|\mathit{val}(X_\ell)| + \log\log|\mathit{val}(X_r)|)$ lexicographical string comparisons from Lemmas 7 and 12. It is clear that the last decomposed Lyndon factor of X_i is the same as X_r.

Suppose that $u_m = v_1$. Then $LF_{X_i} = U_1, \ldots, U_{m-1}, U_m V_1, V_2, \ldots, V_{m'}$ by the definition of Lyndon factorization. We can compute all significant suffixes of X_i in a similar way.

Suppose that $u_m \prec v_1$. We can compute the beginning position of the crossing factor z by $O(\log|\Lambda_u|)$ lexicographic string comparisons from Lemma 14. Let $b = \mathit{lfb}_u(j)$ be this position. Next, we check whether b is also the beginning position of z^k or not. We can do this with one LCE query as follows. If $j = \lambda_u$, then b is the beginning position of z^k (since u_{j-1} does not have $\mathit{lfs}_u(j)$ as a prefix). If the length of the longest common prefix between u_{j-1} and $\mathit{lfs}_u(j)v$ is $|u_{j-1}|$, then $u_{j-1} = z$ and $\mathit{lfb}_u(j-1)$ is the beginning position of z^k. Suppose that $u_{j-1} = z$. Then we can compute z^k by a constant number of lexicographic string comparisons. Thus we can also compute all significant suffixes by Lemma 12 from significant suffixes of X_ℓ and X_r. Suppose that $u_{j-1} \neq z$. Firstly, we check whether the ending position of z which begins in u and ends in v is larger than $|u| + \mathit{lfb}_v(\lambda_v)$ or not. We can do this by $O(\log|\Lambda_v|)$ lexicographic string comparisons from Lemma 14. If so, by Lemmas 7 and 12, we can compute all significant suffixes of X_i by additional $O(\log|\Lambda_u|)$ lexicographic string comparisons. Otherwise, $\Lambda_{uv} \subseteq \Lambda_v$ holds.

Therefore, we can compute all significant suffixes of X_i by $O(\log\log N)$ lexicographical string comparisons. ◀

3.3 Computing Lyndon factorization

Let $GF_{(w,\mathcal{S})} = \mathit{val}(\mathit{leaf}_1), \ldots, \mathit{val}(\mathit{leaf}_g)$ and $w_j = \mathit{val}(\mathit{leaf}_1) \cdots \mathit{val}(\mathit{leaf}_j)$ for any $1 \leq j \leq g$. We consider computing the Lyndon factorization and the significant suffixes of w_{j+1} assuming that we have computed the Lyndon factorization and the significant suffixes of w_j, and also computed the significant suffixes for each variable of $\mathcal{S}$ by Section 3.2. Notice that for $w_{j+1} = w_j \mathit{val}(\mathit{leaf}_{j+1})$, $\mathit{val}(\mathit{leaf}_{j+1})$ has already occurred in w_j at least once if leaf_{j+1} is a variable. The following lemma is very useful for our algorithm.

▶ **Lemma 16** (Lemma 21 of [10]). *Let w be non-empty string such that $w = xvyvz$ with $v \in \Sigma^+$ and $x, y, z \in \Sigma^*$. If $|xvy| < \mathit{lfb}_w(k) \leq |xvyv|$ for some k, then $\mathit{lfb}_w(k) \in \{|xvy| + \mathit{lfb}_v(j) \mid \lambda_v \leq j \leq m'\}$, where $m' = |LF_v|$.*

This lemma implies that the ending position of z^k is restricted by significant suffixes of v. Below, we change this lemma for the ending position of the crossing factor rather than the ending position of z^k.

▶ **Lemma 17.** *Let $i - 1$ be the ending position of the crossing factor. Assume that v is a substring of u. Then $i \in \{|u| + \mathit{lfb}_v(j) \mid \lambda_v \leq j \leq m'\}$, where $m' = |LF_v|$.*

Proof. Assume for a contradiction that $i < |u| + \mathit{lfb}_v(\lambda_v)$. From Lemma 16, $i = \mathit{lfb}_v(\lambda_v - 1)$ and $v_{\lambda_v - 1} = z$ holds. Moreover, $V_{\lambda_v}, \ldots, V_{m'}$ are also Lyndon factors of uv. Since v is a substring of u, $V_{\lambda_v} \cdots V_{m'}$ has a left occurrence. By Lemma 2, $v_{\lambda_v - 1} = z$ has $V_{\lambda_v} \cdots V_{m'}$ as a prefix. This contradicts that $\mathit{lfs}_v(\lambda_v - 1)$ is not a significant suffix of v. ◀

Thus, we can compute the ending position of the crossing factor by significant suffixes of an added string v (i.e., we do not need the whole Lyndon factorization of v).

▶ **Lemma 18.** *Given an SLP $\mathcal{S}$ of size n representing string w of length N, we can compute LF_w in $O(n \log\log N)$ lexicographic string comparisons.*

Proof. Let $GF_{(w,\mathcal{S})} = \mathit{val}(\mathit{leaf}_1), \ldots, \mathit{val}(\mathit{leaf}_g)$ and $w_j = \mathit{val}(\mathit{leaf}_1) \cdots \mathit{val}(\mathit{leaf}_j)$ for any $1 \leq j \leq g$.

Firstly we compute significant suffixes for all variables in $\mathcal{S}$. By Lemma 15, it can be done in $O(n \log\log N)$ lexicographical string comparisons.

Next we consider computing $LF_{w_{j+1}}$ and significant suffixes of w_{j+1} assuming that we have computed $LF_{w_j} = U_1, \ldots, U_m$. Suppose now that leaf_{j+1} is a variable (otherwise $\mathit{leaf}_{j+1} \in \Sigma$). Let $\mathit{leaf}_{j+1} = X_i$ and $LF_{X_i} = V_1, \ldots, V_{m'}$ (remark that we do not actually have LF_{X_i}). According to the lexicographic order between U_m and V_1, which can be checked using a single LCE query by Lemma 13, we proceed as follows:

- $U_m \succ V_1$. This implies that $LF_{w_{j+1}} = U_1, \ldots, U_m, V_1, \ldots, V_{m'}$. By Lemma 16, $\lambda_{X_i} = 1$ holds, which means that the significant suffixes of X_i hold the whole information of LF_{X_i}. Thus we can get $LF_{w_{j+1}}$ without string comparisons. Finally, we can compute significant suffixes of w_{j+1} by $O(\log|\Lambda_{w_j}|)$ string comparisons (same as Lemma 15).
- $U_m = V_1$. This implies that $LF_{X_i} = U_1, \ldots, U_{m-1}, U_m V_1, V_2, \ldots, V_{m'}$. We can compute $LF_{w_{j+1}}$ without string comparisons in a similar way to the previous case. We can also compute $\Lambda_{w_{j+1}}$ by $O(\log|\Lambda_{w_j}|)$ string comparisons in a similar way to the previous case.
- $U_m \prec V_1$. Firstly, we compute the crossing factor z of $w_j X_i$ by $O(\log|\Lambda_{w_j}| + \log|\Lambda_{X_i}|)$ string comparisons from Lemmas 14 and 17. Next we compute z^k by checking consecutive Lyndon factors, which can be done using two LCE queries. Then we can obtain $LF_{w_{j+1}} = U_1, \ldots, U_c, z^k, V_{c'}, \ldots, V_{m'}$ because $V_{c'}, \ldots, V_{m'}$ are part of the significant suffixes of X_i due to Lemma 17. Finally, we can compute significant suffixes of w_{j+1} by $O(\log|\Lambda_{w_j}|)$ string comparisons (same as Lemma 15).

In either case, we can compute $LF_{w_{j+1}}$ and the significant suffixes of w_{j+1} by $O(\log|\Lambda_{w_j}| + \log|\Lambda_{X_i}|) = O(\log\log N)$ string comparisons from Lemma 9.

Now suppose that $\mathit{leaf}_{j+1} \in \Sigma$. Since $\mathit{leaf}_{j+1} = c$ is a new character that does not appear in w_j, the situation does not directly match with the condition of Lemma 17. Still it is easy to see that $LF_c = c$, and thus, we can compute $LF_{w_{j+1}}$ and the significant suffixes of w_{j+1} in a similar way to the case that leaf_{j+1} is a variable.

Note that we do not have to "rebuild" the whole Lyndon factorization of w_{j+1} (which would take $O(n)$ time) because each Lyndon factor of $LF_{w_{j+1}}$ whose beginning position is in $[\mathit{lfb}_{w_j}(\lambda_{w_j}), \mathit{lfb}_{w_{j+1}}(\lambda_{w_{j+1}}))$ remains as a Lyndon factor while appending strings to it, and thus, it is a Lyndon factor of LF_w to output. Hence, we can compute LF_w while treating only the last $O(\log N)$ Lyndon factors that are corresponding to the significant suffixes of the current w_{j+1}'s.

Therefore, we can compute $LF_w = LF_{w_g}$ by $O(n \log\log N)$ string comparisons. ◀

Here, we analyze the space requirement. We need $O(n \log N)$ space for all significant suffixes. In each step, the size of the Lyndon factorization is less than n by Lemma 3. Thus we need $O(n)$ space for storing the Lyndon factorization. Finally, we get the following result by using an LCE data structure for string comparisons.

▶ **Theorem 19.** *Given an SLP of size n representing string w of length N, we can compute LF_w in $O(P(n,N) + Q(n,N)n \log\log N)$ time and $O(n \log N + S(n,N))$ space.*

When we use an LCE data structure of Lemma 4, we can get the following (since the size of LZ factorization is a lower bound on the smallest grammar [3]).

▶ **Corollary 20.** *Given an SLP of size n representing string w of length N, we can compute LF_w in $O(n \log N \log\log N)$ time and $O(n \log N)$ space.*

4 Conclusion

We revisited the problem of computing the Lyndon factorization on SLPs. Given an SLP G of size n representing a string w of length N, our new algorithm runs in $O(P(n,N) + Q(n,N)n \log\log N)$ time and $O(n \log N + S(n,N))$ space where $P(n,N)$, $S(n,N)$, $Q(n,N)$ are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. If we use the LCE data structure of [8], we can compute the Lyndon factorization in $O(n \log N \log\log N)$ time and $O(n \log N)$ space.

The paper [10] also proposed an algorithm to compute in $O(s \log s)$ time and space the Lyndon factorization of a string that is compressed by LZ78 in s size. Future work would include improving this result and/or deriving new algorithms working on other compression schemes.

References

1 Alberto Apostolico and Maxime Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995.

2 Srecko Brlek, Jacques-Olivier Lachaud, Xavier Provençal, and Christophe Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42(10):2239–2246, 2009.

3 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

4 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

5 Jacqueline W. Daykin, Costas S. Iliopoulos, and William F. Smyth. Parallel RAM algorithms for factorizing words. *Theor. Comput. Sci.*, 127(1):53–67, 1994.

6 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

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

8 Tomohiro I. Longest common extensions with recompression. In *Proc. CPM 2017*, pages 18:1–18:15, 2017.

9 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Lyndon factorization of grammar compressed text. In *Proc. CPM 2013*, pages 153–164, 2013.

10 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016.

11 Juha Kärkkäinen, Dominik Kempa, Yuto Nakashima, Simon J. Puglisi, and Arseny M. Shur. On the size of Lempel-Ziv and Lyndon factorizations. In *Proc. STACS 2017*, pages 45:1–45:13, 2017.

12 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *Proc. CPM 2016*, pages 28:1–28:12, 2016.

13 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC 2009*, pages 65–79, 2009.

14 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. `doi:10.1016/S0304-3975(02)00777-6`.