Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Grammar-compressed Self-index with Lyndon Words

Kazuya Tsuruta[1,a)] Dominik Köppl[1,3,b)] Yuto Nakashima[1,c)] Shunsuke Inenaga[1,4,d)]
Hideo Bannai[2,e)] Masayuki Takeda[1,f)]

Received: March 4, 2016, Accepted: August 1, 2016

Abstract: We introduce a new class of straight-line programs (SLPs), named the *Lyndon SLP*, inspired by the Lyndon trees (Barcelo, 1990). Based on this SLP, we propose a self-index data structure of $O(g)$ words of space that can be built from a string T in $O(n \lg n)$ expected time, retrieving the starting positions of all occurrences of a pattern P of length m in $O(m + \lg m \lg n + \mathrm{occ} \lg g)$ time, where n is the length of T, g is the size of the Lyndon SLP for T, and occ is the number of occurrences of P in T.

Keywords: Grammar Compression, Lyndon words, Self-Index

1. Introduction

Although highly-repetitive texts are perceived as a common problem instance nowadays due to use cases such as the collection of DNA sequences with a very narrow taxonomy group or different versions under revision control systems, this problem is difficult to handle due to usual large amount of data. Compressed text indexes have become the standard tool for tackling this problem if full-text search queries like locating all occurrences of a pattern are of importance. One area of research in this field is devoted to grammar compression since a grammar can eliminate repetitions while exhibiting powerful query properties. A grammar index is a *self-index*, i.e., a data structure that supports queries on the underlying text without storing the text in its plain form. **?** gave a grammar index based on a *straight-line program (SLP)*, i.e., a context-free grammar representing a single string in the Chomsky normal form.

A *self-index* is a data structure that is a full-text index, i.e., supports various pattern matching queries on the text, and also provides random access to the text, usually without explicitly holding the text itself. Examples are the compressed suffix array [21], [22], [31], the compressed compact suffix array [35], and the FM index [17].[*1] These self-indexes are, however, unable to fully exploit the redundancy of highly repetitive strings.

To exploit such repetitiveness, Claude and Navarro [12] proposed the first self-index based on grammar-based compression. The method is based on a *straight-line program (SLP)*, a context-free grammar representing a single string in the Chomsky normal form. Plenty of grammar-based self-indexes have already been proposed (e.g., [13], [43], [50], [51]).

In this paper, we first introduce a new class of SLPs, named the *Lyndon SLP*, inspired by the Lyndon tree [4]. We then propose a self-index structure of $O(g)$ words of space that can be built from a string T in $O(n \lg n)$ expected time. The proposed self-index can find the starting positions of all occurrences of a pattern P of length m in $O(m + \lg m \lg n + \mathrm{occ} \lg g)$ time, where n is the length of T, g is the size of the Lyndon SLP for T, σ is the alphabet size, w is the computer word length and occ is the number of occurrences of P in T.

1.1 Related work

The *smallest grammar problem* is, given a string T, to find the context-free grammar G representing T with the smallest possible size, where the *size* of G is the total length of the right-hand sides of the production rules in G. Since the smallest grammar problem is NP-hard [49], many attempts have been made to develop small-sized context-free grammars representing a given string T. LZ78 [53], LZW [52], Sequitur [42], Sequential [29], Longest-Match [29], Re-Pair [32], and Bisection [28] are grammars based on simple greedy heuristics. Among them Re-Pair is known for achieving high compression ratios in practice.

Approximations for the smallest grammar have also been proposed. The AVL grammars [44] and the α-balanced grammars [9] can be computed in linear time and achieve the currently best approximation ratio of $O(\lg(|T|/g_T^*))$ by using the LZ77 factorization and the balanced binary grammars, where g_T^* denotes the smallest grammar size for T. Other grammars with linear-time algorithms achieving the approximation $O(\lg(|T|/g_T^*))$ are LevelwiseRePair [46] and Recompression [24]. They basically replace di-grams with a new variable in a bottom-up manner similar to Re-Pair, but use different mechanisms to select the di-grams. On the other hand, LCA [47] and its variants [37], [38], [48] are

1 Department of Informatics, Kyushu University, Nishi-Ku, Fukuoka 819–0395, Japan
2 M&D Data Science Center, Tokyo Medical and Dental University, Chiyoda, 101–0062, Japan
3 Japan Society for Promotion of Science
4 Japan Science and Technology Agency
a) kazuya.tsuruta@inf.kyushu-u.ac.jp
b) dominik.koeppl@inf.kyushu-u.ac.jp
c) yuto.nakashima@inf.kyushu-u.ac.jp
d) inenaga@inf.kyushu-u.ac.jp
e) hdbn.dsc@tmd.ac.jp
f) takeda@inf.kyushu-u.ac.jp

*1 Navarro and Mäkinen [40] published an excellent survey on this topic.

known as scalable practical approximation algorithms. The core idea of LCA is the *edit-sensitive parsing (ESP)* [14], a parsing algorithm developed for approximately computing the edit distance with moves. The *locally-consistent-parsing (LCP)* [45] is a generalization of ESP. The *signature encoding (SE)* [39], developed for equality testing on a dynamic set of strings, is based on LCP and can be used as a grammar-transform method. The ESP index [50], [51] and the SE index [43] are grammar-based self-indexes based on ESP and SE, respectively.

While our experimental section (Sect. 3.3) serves as a practical comparison between the sizes of some of the above mentioned grammars, **Table 1** gives a comparison with some theoretically appealing index data structures based on grammar compression. There, we chose the indexes of Claude and Navarro [13], Gagie et al. [18], and Christiansen et al. [11] because these indexes have non-trivial time complexities for answering queries. We observe that our proposed index has the fastest construction among the chosen grammar indexes, and is competitively small if $g = o(\gamma \lg(n/\gamma))$ while being clearly faster than the first two approaches for long patterns. It is worth pointing out that the grammar index of Christiansen et al. [11] achieves a grammar size whose upper bound $O(\gamma \lg(n/\gamma))$ matches the upper bound of the size g^*_T of the smallest possible grammar. Unfortunately, we do not know how to compare our result within these terms in general.

2. Preliminaries

2.1 Notation

Let Σ be an ordered finite *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is the string of length 0. For a string $S = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of S, respectively. A prefix (resp. suffix) x of S is called a *proper prefix* (resp. suffix) of S if $x \neq S$. S^ℓ denotes the ℓ times concatenation of the string S. The i-th character of a string S is denoted by $S[i]$, where $i \in [1..|S|]$. For a string S and two integers i and j with $1 \leq i \leq j \leq |S|$, let $S[i..j]$ denote the substring of S that begins at position i and ends at position j. For convenience, let $S[i..j] = \varepsilon$ when $i > j$.

2.2 Lyndon words and Lyndon trees

Let $\preceq$ denote some total order on Σ that induces the lexicographic order $\preceq$ on Σ^*. We write $u \prec v$ to imply $u \preceq v$ and $u \neq v$ for any $u, v \in \Sigma^*$.

Definition 2.1 (Lyndon Word [34])**.** A non-empty string $w \in \Sigma^+$ is said to be a *Lyndon word* with respect to $\prec$ if $w \prec u$ for every non-empty proper suffix u of w.

By this definition, all characters ($\in \Sigma^1$) are Lyndon words.

Definition 2.2 (Standard Factorization [10], [33])**.** The *standard factorization* of a Lyndon word w with $|w| \geq 2$ is an ordered pair (u, v) of strings u, v such that $w = uv$ and v is the longest proper suffix of w that is also a Lyndon word.

Lemma 2.3 ([5], [33])**.** For a Lyndon word w with $|w| > 1$, the standard factorization (u, v) of w always exists, and the strings u and v are Lyndon words.

The Lyndon tree of a Lyndon word w, defined below, is the full binary tree induced by recursively applying the standard factorization on w.

Definition 2.4 (Lyndon Tree [4])**.** The *Lyndon tree* of a Lyndon word w, denoted by $LTree(w)$, is an ordered full binary tree defined recursively as follows:

- if $|w| = 1$, then $LTree(w)$ consists of a single node labeled by w;
- if $|w| \geq 2$, then the root of $LTree(w)$, labeled by w, has the left child $LTree(u)$ and the right child $LTree(v)$, where (u, v) is the standard factorization of w.

Fig. 1 shows an example of a Lyndon tree for the Lyndon word `aababaababb`.

2.3 Admissible grammars and straight-line programs (SLPs)

An admissible grammar [29] is a context-free grammar that generates a language consisting only of a single string. Formally, an *admissible grammar (AG)* is a set of production rules $\mathcal{G}_{\mathrm{AG}} = \{X_i \to \mathrm{expr}_i\}_{i=1}^{g}$, where X_i is a *variable* and expr_i is a non-empty string over $\Sigma \cup \{X_1, \ldots, X_{i-1}\}$, called an *expression*. The variable X_g is called the *start symbol*. We denote by $val(X_i)$ the string derived by X_i. We say that an admissible grammar $\mathcal{G}_{\mathrm{AG}}$ *represents* a string T if $T = val(X_g)$. To ease notation, we sometimes associate $val(X_i)$ with X_i. The *size* of $\mathcal{G}_{\mathrm{AG}}$ is the total length of all expressions expr_i. We assume that any admissible grammar has no useless symbols.

It should be stated that the above definition of admissible grammar is different with but equivalent to the original definition in [29], which defines an admissible grammar to be a context-free grammar G satisfying the conditions: (1) G is deterministic, i.e., for every variable A there is exactly one production rule of the form $A \to \gamma$, where γ is a non-empty string consisting of variables and characters; (2) G has no production rule of the form $A \to \varepsilon$; (3) The language $L(G)$ of G is not empty; and (4) G has no useless symbols, i.e., every symbol appears in some derivation that begins with the start symbol and ends with a string consisting only of characters.

A *straight-line program* (*SLP*) is an admissible grammar in the Chomsky normal form, namely, each production rule is either of the form $X_i \to a$ for some $a \in \Sigma$ or $X_i \to X_{i_\mathrm{L}} X_{i_\mathrm{R}}$ with $i > i_\mathrm{L}, i_\mathrm{R}$. Note that $\mathcal{G}_{\mathrm{SLP}}$ can derive a string up to length $O(2^g)$. This can be seen by the example string $T = \mathtt{a} \cdots \mathtt{a}$ consisting of $n = 2^\ell$ a's, where the smallest SLP $\{X_1 \to \mathtt{a}\} \cup \bigcup_{j=2}^{\ell+1} \{X_j \to X_{j-1} X_{j-1}\}$ has size $2\ell + 1$.

The derivation tree $\mathcal{T}_{\mathcal{G}_{\mathrm{SLP}}}$ of $\mathcal{G}_{\mathrm{SLP}}$ is a labeled ordered binary tree, where each internal node is labeled with a variable in $\{X_1, \ldots, X_g\}$, and each leaf is labeled with a character in Σ. The root node has the start symbol X_g as label. An example of the derivation tree of an SLP is shown in **Fig. 2**.

2.4 Grammar irreducibility

An admissible grammar is said to be *irreducible* if it satisfies the following conditions:

C-1. Every variable other than the start symbol is used more than once (**rule utility**);

Table 1 Complexity bounds of self-indexes based on grammar compression.

Construction space (in words) and time

Index	Space	Time
[13]	$O(n)$	$O(n + \hat{g} \lg \hat{g})$
[11]	$O(n)$	$O(n \lg n)$ expected
[11]	$O(n)$	$O(n \lg n)$ expected
This paper	$O(n)$	$O(n \lg n)$ expected

Needed space (in words) and query time for a pattern of length m

Index	Space	Locate Time
[13]	$O(\hat{g})$	$O(m^2 \lg \lg_{\hat{g}} n + (m + \mathrm{occ}) \lg \hat{g})$
[18]	$O(\hat{g} + z \lg \lg z)$	$O(m^2 + (m + \mathrm{occ}) \lg \lg n)$
[11]	$O(\gamma \lg(n/\gamma))$	$O(m + \lg^{\epsilon} \gamma + \mathrm{occ} \lg^{\epsilon}(\gamma \lg(n/\gamma)))$
[11]	$O(\gamma \lg(n/\gamma) \lg^{\epsilon}(\gamma \lg(n/\gamma)))$	$O(m + \mathrm{occ})$
Theorem 4.19	$O(g)$	$O(m + \lg m \lg n + \mathrm{occ} \lg g)$

n is the length of T, z is the number of LZ77 [54] phrases of T, γ is the size of the smallest string attractor [27] of T, g is the size of the Lyndon SLP of T, $\hat{g}$ is the size of a given admissible grammar, $\epsilon > 0$ is a constant, m is the length of a pattern P, and occ is the number of occurrences of P in T.

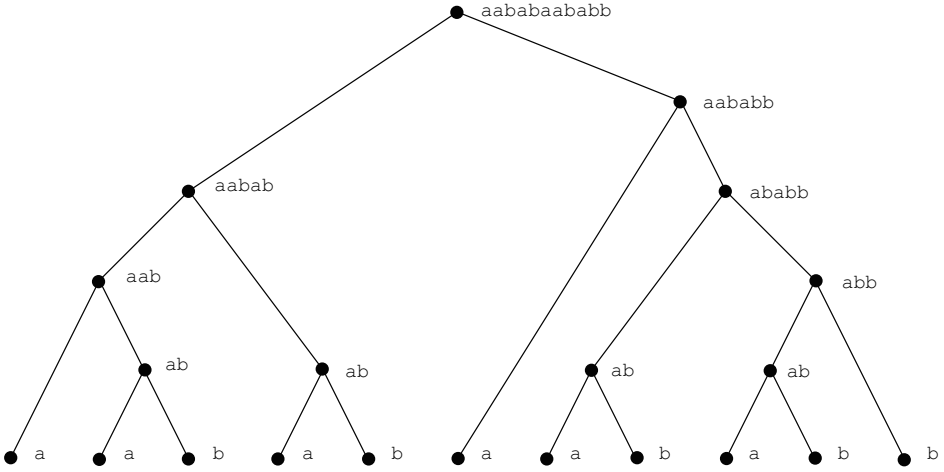


Fig. 1 The Lyndon tree for the Lyndon word `aababaababb` with respect to the order `a` $\prec$ `b`, where each node is accompanied by its derived string to its right.

C-2. All pairs of symbols have at most one non-overlapping occurrence in the right-hand sides of the production rules (**digram uniqueness**); and

C-3. Distinct variables derive different strings.

Grammar-based compression is a combination of

(1) the *grammar transform*, i.e., the computation of an admissible grammar G representing the input string T, and

(2) the *grammar encoding*, i.e., an encoding for G.

Kieffer and Yang [29] showed that a combination of an irreducible grammar transform and a zero order arithmetic code is universal, where a grammar transform is said to be *irreducible* if the resulting grammars are irreducible.

If an admissible grammar G is not irreducible, we can apply at least one of the following reduction rules [29] to make G irreducible:

R-1. Replace each variable X_i occurring only once in the right-hand sides of the production rules with expr_i and remove the rule $X_i \to \mathrm{expr}_i$. We also remove all production rules with useless symbols.

R-2. Given there are at least two non-overlapping occurrences of a string γ of symbols with $|\gamma| \geq 2$ in the right-hand sides of the production rules, replace each of the occurrences of γ with the variable X_i, where $X_i \to \gamma$ is an existing or newly created production rule. Recurse until no such γ longer exists.

R-3. For each two distinct variables X_i and X_j deriving an identical string, (a) replace all occurrences of X_j with X_i in the right-hand sides of the production rules, and (b) remove the production rule $X_j \to \mathrm{expr}_j$ and discard the variable X_j. Consequently, there are no two distinct variables X_i and X_j with $val(X_i) = val(X_j)$. This operation possibly makes some variables useless; the production rules with such variables will be removed by R-1.

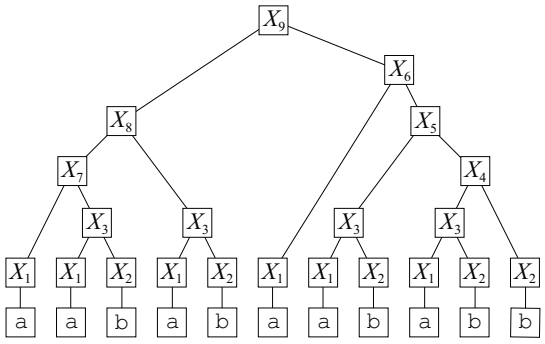
3. Lyndon SLP

In what follows, we propose a new SLP, called Lyndon SLP. A *Lyndon SLP* is an SLP $\mathcal{G}_{\mathrm{LYN}} = \{X_i \to \mathrm{expr}_i\}_{i=1}^{g}$ representing a Lyndon word, and satisfies the following properties:

- The strings $val(X_i)$ are Lyndon words for all variables X_i.
- The standard factorization of the string $val(X_i)$ is $(val(X_{i_{\mathrm{L}}}), val(X_{i_{\mathrm{R}}}))$ for every rule $X_i \to X_{i_{\mathrm{L}}} X_{i_{\mathrm{R}}}$.
- No pair of distinct variables X_i and X_j satisfies $val(X_i) = val(X_j)$.

The derivation tree (when excluding its leaves) of $\mathcal{T}_{\mathcal{G}_{\mathrm{LYN}}}$ is isomorphic to the Lyndon tree of T (cf. Fig. 2).

The rest of this article is devoted to algorithmic aspects regarding the Lyndon SLP. We study its construction (Sect. 3.1), practically evaluate its size (Sect. 3.3), and propose an index data structure on it (Sect. 4). For that, we work in the word RAM model supporting packing characters of sufficiently small bit widths into a single machine word. Let w denote the machine word size in bits.



$X_1 \rightarrow$ a
$X_2 \rightarrow$ b
$X_3 \rightarrow X_1X_2$
$X_4 \rightarrow X_3X_2$
$X_5 \rightarrow X_3X_4$
$X_6 \rightarrow X_1X_5$
$X_7 \rightarrow X_1X_3$
$X_8 \rightarrow X_7X_3$
$X_9 \rightarrow X_8X_6$

Fig. 2 *Left*: The derivation tree of the Lyndon SLP $\mathcal{G}_{\mathrm{LYN}}$ with $g = 9$ representing the Lyndon word $T =$ aababaababb. *Right*: The production rules of $\mathcal{G}_{\mathrm{LYN}}$.

We fix a text $T[1..n]$ over an integer alphabet Σ with size $\sigma = n^{O(1)}$. If T is not a Lyndon word, we prepend T with a character smaller than all other characters appearing in T. Let g denote the size $|\mathcal{G}_{\mathrm{LYN}}|$ of the Lyndon SLP $\mathcal{G}_{\mathrm{LYN}}$ of T.

Lemma 3.1 (Algo. 1 of [3])**.** We can construct the Lyndon tree of T in $O(n)$ time.

3.1 Constructing Lyndon SLPs

The algorithm of Bannai et al. [3], Algo. 1 builds the Lyndon tree *online* from right to left. We can modify this algorithm to create the Lyndon SLP of T by storing a dictionary for the rules and a reverse dictionary for looking up rules: Whenever the algorithm creates a new node u, we query the reverse dictionary with u's two children v and w for an existing rule $X \rightarrow X_vX_w$, where X_v and X_w are the variables representing v and w. If such a rule exists, we assign u the variable X, otherwise we create a new rule $X_u \rightarrow X_vX_w$ and put this new rule into both dictionaries. The dictionaries can be implemented as balanced search trees or hash tables, featuring $O(n \lg g)$ deterministic construction time or $O(n)$ expected construction time, respectively.

In the static setting (i.e., we do not work online), deterministic $O(n)$ time can be achieved with the enhanced suffix array [1], [36] supporting constant time longest common extension queries. We associate each node v of the Lyndon tree with the pair $(|T[i..j]|, rank(i))$, where $T[i..j]$ is the substring derived from the non-terminal representing v, and $rank(i)$ is the lexicographic rank of the suffix starting at position i. Then, sort all nodes according to their associated pairs with a linear-time integer sorting algorithm. By using longest common extension queries between adjacent nodes of equal length in the sorted order, we can determine in $O(1)$ time per node whether they represent the same string, and if so, assign the same variable (otherwise assign a new variable).

3.2 Lyndon array simulation

As a by-product, we can equip the Lyndon SLP of T with the indexing data structure of Bille et al. [8] to support character extraction and navigation in $O(\lg n)$ time. This allows us to compute the i-th entry of the Lyndon array [3] in $O(\lg n)$ time, where the i-th entry of the Lyndon array of T stores the length of the longest Lyndon word starting at $T[i]$. For that, given a text position i, we search for the highest Lyndon tree node having $T[i]$ as its leftmost leaf. Given the rightmost leaf of this node represents $T[j]$, the longest Lyndon word starting at $T[i]$ has the length $j - i + 1$. (Otherwise, there would be a higher node in the Lyndon tree representing a longer Lyndon word starting at $T[i]$.)

Lemma 3.2. There is a data structure of size $O(g)$ that can retrieve the longest Lyndon word starting at $T[i]$ in $O(\lg n)$ time.

3.3 Computational experiments

We empirically benchmark the grammar sizes obtained by the Lyndon SLP to highlight its potential as a grammar compressor. As benchmark datasets we used four highly repetitive texts consisting of the files `cere`, `einstein.de.txt`, `kernel`, and `world_leaders` from the Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl`). We used the natural order implied by the ASCII code for building the Lyndon SLPs. We compared the size of the resulting Lyndon grammars with the resulting grammars of Re-Pair, LCA, Recompression. We used existing implementations of Re-Pair (`https://users.dcc.uchile.cl/~gnavarro/software/`) and of LCA (`http://code.google.com/p/lcacomp/`). The outputs of LCA, Recompression and our method are SLPs, while those of Re-Pair are AGs (and not necessarily SLPs). For a fair comparison, we compared the resulting grammar sizes either in an SLP representation, or in a common AG representation.

SLP We keep the resulting grammar of the Lyndon SLP, LCA, and Recompression as it is, but transform the output of Re-Pair to an SLP. To this end, we observe that Re-Pair consists of (a) a list of non-terminals whose right hand sides are already of length two, and (b) a start symbol whose right hand side is a string of symbols of arbitrary size. Consequently, to transform this grammar to an SLP, it is left to focus on the start symbol: We replace greedily di-grams in the right hand side of the start symbol until it consists only of two symbols.

AG We process each grammar in the following way: First, we remove the production rules of the form $X_i \rightarrow a \in \Sigma$ by replacing all occurrences of X_i with a. Subsequently, we apply the reduction rule R-1 of Sect. 2.4.

We collected the obtained grammar sizes in **Table 2**. There, we observe that the Lyndon SLP is no match for Re-Pair, but competitive with LCA and Recompression. Although this evaluation puts Re-Pair in a good light, it seems hard to build an index data structure on this grammar that can be as efficient as the self-index data structure based on the Lyndon SLP, which we present in the next section.

4. Lyndon SLP based self-index

Given a Lyndon SLP of size g, we can build an indexing data structure on it to query all occurrences of a pattern P of

Table 2 Sizes of the resulting grammars benchmarked in Sect. 3.3.

collection		Re-Pair	LCA	Recompression	Lyndon SLP
`cere`	SLP	6,433,183	9,931,777	8,537,747	13,026,562
	AG	4,057,693	6,513,345	5,309,789	7,469,979
`einstein.de.txt`	SLP	125,343	251,411	202,749	205,348
	AG	84,493	168,193	127,790	123,963
`kernel`	SLP	2,254,840	4,065,522	3,587,382	4,201,895
	AG	1,373,244	2,507,291	2,135,779	2,400,211
`world_leaders`	SLP	601,757	1,243,757	1,023,739	911,222
	AG	398,234	809,163	636,700	552,497

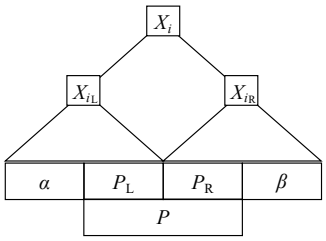


Fig. 3 A partition pair (P_L, P_R) of a pattern P with one of its associated tuples (X_i, α, β).

length $m \in [1..n]$ in T. We call this query *locate*(P). Our data structure is based on the approach of [12]. This approach separates the occurrences of a pattern into so-called primary occurrences and secondary occurrences. It first locates the primary occurrences and, with the help of these, it subsequently locates the secondary occurrences. To this end, it locates primary occurrences with a labeled binary relation data structure, and subsequently locates the secondary occurrences with the grammar tree. In our case, we find the primary occurrences with so-called partition pairs.

A *partition pair* (at position i) of a pattern $P[1..m]$ is a pair $(P[1..i], P[i+1..m])$ with $i \in [1..m]$ such that there exists a rule $X_i \to X_{i_L}X_{i_R}$ with $val(X_{i_L})$ and $val(X_{i_R})$ having $P[1..i]$ and $P[i+1..m]$ as a (not necessarily proper) suffix and as a prefix, respectively. Similar to the grammar proposed in Sect. 6.1 of [11], we can bound the number of partition pairs by $O(\lg m)$ by carefully selecting all possible partition pairs:

Given a partition pair (P_L, P_R) of P, let $X_i \to X_{i_L}X_{i_R}$ be a rule such that $val(X_{i_L})$ and $val(X_{i_R})$ have P_L and P_R as a suffix and as a prefix, respectively. Consequently, there exist two strings α and β such that $val(X_{i_L}) = \alpha P_L$ and $val(X_{i_R}) = P_R\beta$ (cf. **Fig. 3**). By the definition of the Lyndon tree of the text T, $(val(X_{i_L}), val(X_{i_R})) = (\alpha P_L, P_R\beta)$ is the standard factorization of $val(X_i) = \alpha P_L P_R \beta$. According to the standard factorization, $P_R\beta$ is the longest suffix of $val(X_i)$ that is a Lyndon word. For the proofs of Lemmas 4.7 and 4.8, we use this notation and call the tuple (X_i, α, β) a *tuple associated with* (P_L, P_R).

Let us take $P :=$ `bab` as an example. The only partition pair is (`b`, `ab`). Considering the Lyndon grammar of our example text given in Fig. 2, the tuples associated with (`b`, `ab`) are $(X_8, \mathtt{aa}, \varepsilon)$ and $(X_5, \mathtt{a}, \mathtt{b})$.

Note that $|\alpha| = 0$ if P is a Lyndon word. If P is a proper prefix of a Lyndon word[*2], then α may be empty. If P is a not a (not necessarily proper) prefix of a Lyndon word, then $|\alpha| > 0$ (since $\alpha P_L P_R \beta$ is a Lyndon word).

[*2] I.e., there is a string $S \in \Sigma^+$ such that PS is a Lyndon word.

4.1 Associated tuples with non-empty α

We want to reduce the number of possible partition pairs from m to $O(\lg m)$. A first idea is that only the beginning positions of the Lyndon factors of P contribute to potentially partition pairs. We prove this in Lemma 4.5, after defining the Lyndon factors:

The (composed) *Lyndon factorization* [10] of a string $P \in \Sigma^+$ is the factorization of P into a sequence $P_1^{\tau_1} \cdots P_p^{\tau_p}$ of lexicographically decreasing Lyndon words $P_1, \ldots, P_p$, where (a) each $P_x \in \Sigma^+$ is a Lyndon word, and (b) $P_x \succ P_{x+1}$ for each $x \in [1..p)$. P_x and $P_x^{\tau_x}$ are called *Lyndon factor* and *composed Lyndon factor*, respectively.

Lemma 4.1 (Algo. 2.1 of [15])**.** The Lyndon-factorization of a string can be computed in linear time.

We borrow from Sect. 2.2 of [23] the notation $lfs_P(x) := P_x^{\tau_x} \cdots P_p^{\tau_p}$ for the suffix of P starting with the x-th Lyndon factor. Given $\lambda_P \in [1..p]$ is the smallest integer such that $lfs_P(x+1)$ is a prefix of P_x for every $x \in [\lambda_P..p-1]$, $lfs_P(x)$ is called a *significant suffix* of P for every $x \in [\lambda_P..p]$. Consequently, $lfs_P(p) = P_p^{\tau_p}$ is a significant suffix.

In what follows, we show that P_R of a partition pair (P_L, P_R) has to start with a Lyndon factor (Lemma 4.5), and further has to start with a composed Lyndon factor (Lemma 4.7). Finally, we refine this result by restricting P_R to begin with a significant suffix (Lemma 4.8) whose number is bounded by the following lemma:

Lemma 4.2 (Lemma 12 of [23])**.** The number of significant suffixes of P is $O(\lg m)$.

In what follows, we study the occurrences of P in T under the circumstances that T is represented by its Lyndon tree induced by the standard factorization, while P is represented by its Lyndon factors.

Lemma 4.3 (Prop. 1.10 of [15])**.** The longest prefix of P that is a Lyndon word is the first Lyndon factor P_1 of P.

Lemma 4.4 (Lemma 5.4 of [3])**.** Given a production $X_j \to X_{j_L}X_{j_R} \in \mathcal{G}_{LYN}$, there is no Lyndon word that is a substring of $val(X_j) = val(X_{j_L})val(X_{j_R})$ beginning in $val(X_{j_L})$ and ending in $val(X_{j_R})$, except $val(X_j)$.

Lemma 4.5. Given (P_L, P_R) is a partition pair of a pattern P, P_R starts with a Lyndon factor of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.

Proof. Since $|\alpha| > 0$ holds, P is a proper substring of $val(X_i)$. Then P_R must start with a Lyndon factor of P according to Lemma 4.4. □

Lemma 4.6 (Prop. 1.3 of [15])**.** Given two Lyndon words α, β with $\alpha \prec \beta$, the concatenation $\alpha\beta$ is also a Lyndon word.

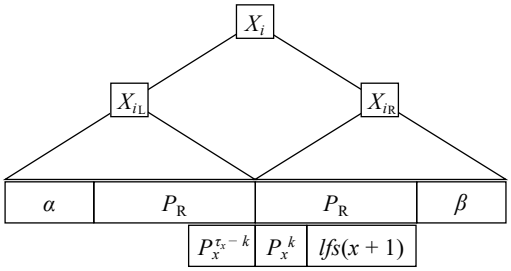


Fig. 4 Setting of the proof of Lemma 4.7.

Lemma 4.7. Given (P_L, P_R) is a partition pair of a pattern P, P_R starts with a composed Lyndon factor of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.

Proof. Let (X_i, α, β) be a tuple associated with (P_L, P_R). Assume for the contrary that P_R does not start with any composed Lyndon factors of P, namely, there exists $x \in [1..p]$ and $k \in [1..\tau_x - 1]$ such that P_L and P_R have $P_x^{\tau_x-k}$ and P_x^k as a suffix and prefix, respectively (cf. **Fig. 4**). By the assumption, $val(X_{i_R}) = P_x^k lfs_P(x+1)\beta$ is the longest Lyndon word that is a suffix of $val(X_i)$. Since $P_x \prec val(X_{i_R})$ and P_x is a Lyndon word, $P_x val(X_{i_R})$ is also a Lyndon word by Lemma 4.6. This contradicts that $val(X_{i_R})$ is the longest Lyndon word that is a suffix of $val(X_i)$. □

Lemma 4.7 helps us to concentrate on the *composed* Lyndon factors. Next, we show that only those composed Lyndon factors are interesting that start with a significant suffix:

Lemma 4.8. Given (P_L, P_R) is a partition pair of a pattern P, then P_R is a significant suffix of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.

Proof. Let (X_i, α, β) be a tuple associated with (P_L, P_R) and $|\alpha| > 0$. By Lemma 4.7, there exists $x \in [1..p]$ such that $P_R = P_x^{\tau_x} \cdots P_p^{\tau_p}$. Assume for the contrary that $x < \lambda_P$, i.e., P_R is not a significant suffix of P. By definition, $lfs_P(x) \succ lfs_P(x+1)$ holds. Since $lfs_P(x+1)$ is not a prefix of $lfs_P(x)$, $lfs_P(x)\beta \succ lfs_P(x+1)\beta$ also holds. This implies that $P_R = lfs_P(x)\beta$ is not a Lyndon word, a contradiction. □

This, together with Lemma 4.2, yields the following corollary.

Corollary 4.9. There are $O(\lg m)$ partition pairs of P associated with a tuple (X_i, α, β) with $|\alpha| > 1$.

Let us take $P :=$ `abacabadabacababa` as an elaborated example. Its composed Lyndon factorization is $P = P_1P_2P_3^2P_4$, where its Lyndon factors are $P_1 =$ `abacabad`, $P_2 =$ `abac`, $P_3 =$ `ab`, and $P_4 =$ `a` with $\lambda_P = 3$. Hence, $lfs_P(3)$ and $lfs_P(4)$ are significant suffixes. Its potential partition pairs are $(P_1P_2, P_3^2P_4)$, $(P_1P_2P_3^2, P_4)$. There is no Lyndon SLP such that another partitioning like $(P_1, P_2P_3^2P_4)$ or $(P_1P_2P_3, P_3P_4)$ would have an associated tuple according to Lemma 4.8 and Lemma 4.7, respectively.

4.2 Associated tuples with empty α

Given a partition pair (P_L, P_R) associated with a tuple $(X_i, \varepsilon, \beta)$, we consider two cases depending on $|P_L|$: In the case of $|P_L| = 1$, $(P[1], P[2..m])$ may be a partition pair of P. In the case of $|P_L| \geq 2$, suppose that $P' = P[2..m]$, $\alpha' = P[1]$ and (P'_L, P'_R) is a partition pair of P' with associated tuple (X_i, α', β). Then, $(P[1]P'_L, P'_R)$ is a partition pair of P with associated tuple $(X_i, \varepsilon, \beta)$. We can use Lemma 4.5, Lemma 4.7 and Lemma 4.8 to restrict P'_R starting with a significant suffix of $P[2..m]$ (cf. Cor. 4.9).

Corollary 4.10. There are $O(\lg m)$ partition pairs of P associated with a tuple $(X_i, \varepsilon, \beta)$.

Combining Cor. 4.9 with Cor. 4.10 yields the following theorem and the main result of this subsection:

Theorem 4.11. There are $O(\lg m)$ partition pairs of a pattern of length m.

4.3 Locating a pattern

In the following, we use the partition pairs to find all primary occurrences. We do this analogously as for the Γ-tree (Sect. 3.1. of [41]) or for special grammars (Sect. 6.1 of [11]).

Lemma 4.12 (Lemma 5.2 of [20])**.** Let $\mathcal{S}$ be a set of strings and assume that we can (a) extract a substring of length ℓ of a string in $\mathcal{S}$ in time $f_e(\ell)$ and (b) compute the Karp-Rabin fingerprint [26] of a substring of a string in $\mathcal{S}$ in time f_h. Then we can build a data structure of $O(|\mathcal{S}|)$ words solving the following problem in $O(m \lg \sigma/w + t(f_h + \lg m) + f_e(m))$ time: given a pattern $P[1..m]$ and $t > 0$ suffixes $Q_1, \ldots, Q_t$ of P, discover the ranges of strings in (the lexicographically-sorted) $\mathcal{S}$ prefixed by $Q_1, \ldots, Q_t$.

Lemma 4.13 (Thm. 1.1 of [8])**.** For an AG of size g representing a string of length n we can extract a substring of length ℓ in time $O(\ell + \lg n)$ after $O(g)$ preprocessing time and space.

Lemma 4.14 (Thm. 1 of [7])**.** Given a string of length n represented by an SLP of size g, we can construct a data structure supporting fingerprint queries in $O(g)$ space and $O(\lg n)$ deterministic query time. This data structure can be constructed in $O(n \lg n)$ randomized time (cf. Sect. 2.4 of [19]) by using Karp, Miller and Rosenberg's [25] renaming algorithm to make all fingerprints unique.

With Lemma 4.13 and Lemma 4.14 we have $f_e(\ell) = O(\ell + \lg n)$ and $f_h = O(\lg n)$ in Lemma 4.12, respectively, leading to:

Corollary 4.15. There is a data structure using $O(g)$ space such that, given a pattern $P[1..m]$ with $m \leq n$, it can find all variables whose derived strings have one of t selected suffix of P as a prefix in $O(m \lg \sigma/w + t(\lg n + \lg m) + \ell + \lg n)$ time.

Corollary 4.15 yields $O(m \lg n)$ time for $t = m$, i.e., when we need to split the pattern at each position. It yields $O(m \lg \sigma/w + \lg m \lg n)$ time for $t = \lg m$, i.e., the case for Sect. 6.1 of [11] and for the Lyndon SLP thanks to Lemma 4.8 (we assume that the pattern is not longer than the text).

We can retrieve the associated tuples of all primary occurrences by plugging the variables retrieved in Cor. 4.15 into a data structure for labeled binary relations [12].

For that, we generate two list $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of all variables $X_1, \cdots, X_g$ of the grammar sorted lexicographically by their derived strings and the reverses of their derived strings, respectively. Both lists allow us to answer a prefix (resp. suffix) query by returning a range of variables having the prefix (resp. suffix) in question. The query is performed by the data structure described in Lemma 4.12 (with $\mathcal{S}$ being either $\mathcal{L}$ or $\mathcal{L}^{\mathrm{REV}}$). Finally, we can plug the obtained ranges into the labeled binary relation data structure of Claude and Navarro [12]:

Lemma 4.16 (Thm. 3.1 of [12])**.** Given two list $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of variables sorted lexicographically by their expressions and its reversed strings, we can built a data structure of $O(g)$ words of space in $O(g \lg g)$ time for supporting the following query: Given a partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ and ranges in $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of those variables whose derived strings have $val(P_{\mathrm{R}})$ as a prefix and $val(P_{\mathrm{L}})$ as a suffix, this data structure can retrieve all associated tuples of $(P_{\mathrm{L}}, P_{\mathrm{R}})$ in $O((1 + \mathrm{occ}') \lg g)$ time, where occ′ denotes their number.

The time complexity of Cor. 4.15 and Lemma 4.12 is based on the assumption that we have (static) z-fast tries [6] built on the lists $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$,[*3] which we can build in $O(g)$ expected time and space (Sect. 6.6 (3) of [11]).

Since there are $O(\lg m)$ partition pairs according to Thm. 4.11, applying Lemma 4.16 over all $O(\lg m)$ partition pairs yields $O(\lg m \lg g + \mathrm{occ} \lg g)$ time, where occ denotes the number of all primary occurrences.

Corollary 4.17. We can find the primary occurrences of a pattern P in

$$\underbrace{O(m)}_{\textit{Lemma } 4.1} + \underbrace{O(m \lg \sigma/w + \lg m \lg n)}_{\textit{Cor. } 4.15} + \underbrace{O(\lg m \lg g + \mathrm{occ} \lg g)}_{\textit{Lemma } 4.16}$$

$= O(m + \lg m \lg n + \mathrm{occ} \lg g)$ time.

Finally, we use the derivation tree to find the remaining (secondary) occurrences of the pattern:

4.4 Search for secondary occurrences

We follow Claude and Navarro [13] improving the search of the secondary occurrences in [12] by applying reduction rule R-1 to enforce C-1 (see Sect. 2.4). The resulting admissible grammar $\mathcal{G}_{\mathrm{AG}}$ is no longer an SLP in general. Since we only remove variables with a single occurrence, the size of $\mathcal{G}_{\mathrm{AG}}$ is $O(g)$. Consequently, we can store both $\mathcal{G}_{\mathrm{AG}}$ *and* $\mathcal{G}_{\mathrm{SLP}}$ in $O(g)$ space.

Lemma 4.18 (Sect. 5.2 of [13])**.** Given the associated tuples of all partition pairs, we can find all occ occurrences of P in T with $\mathcal{G}_{\mathrm{AG}}$ in $O(\mathrm{occ} \lg g)$ time.

Remembering that we split the analysis of an associated tuple in the cases $|\alpha| = 0$ (Sect. 4.2) and $|\alpha| > 0$ (Cor. 4.17), we observe that the time complexity of the latter case dominates. Combining this time with Lemma 4.18 yields the time complexity for answering $locate(P)$ with the Lyndon SLP:

Theorem 4.19. Given the Lyndon SLP of T, there is a data structure using $O(g)$ words that can be constructed in $O(n \lg n)$ expected time, supporting $locate(P)$ in $O(m + \lg m \lg n + \mathrm{occ} \lg g)$ time for a pattern P of length m.

Note that the $O(n \lg n)$ expected construction time is due to the data structure described in Lemma 4.14.

5. Conclusion

We introduced a new class of SLPs, named the *Lyndon SLP*, and proposed a self-index structure of $O(g)$ words of space, which can be built from an input string T in $O(n \lg n)$ expected time, where n is the length of T and g is the size of the Lyndon SLP for T. By exploiting combinatorial properties on Lyndon SLPs, we showed that $locate(P)$ can be computed in $O(m + \lg m \lg n + \mathrm{occ} \lg g)$ time for a pattern P of length m, where occ is the number of occurrences of P. This is better than the $O(m^2 \lg \lg_{\hat{g}} n + (m + \mathrm{occ}) \lg \hat{g})$ query time of the SLP-index by Claude and Navarro [13] (cf. Table 1), which works for a *general* admissible grammar of size $\hat{g}$.

We have not implemented the proposed self-index structure, and comparing it with other self-index implementations such as the FM index [16], the LZ index [2], the ESP index [51], or the LZ-end index [30] will be a future work. Also, we want to speed up the query time to $O(m \lg \sigma/w + \lg m \lg n + \mathrm{occ} \lg g)$ by applying broadword techniques for determining the Lyndon factors of the pattern P (cf. Cor. 4.17), where σ is the alphabet size and w is the computer word length.

[*3] We use again the derived string or, respectively, the reverse of the derived string of each non-terminal in one of the lists as its respective keyword to insert into the trie.

References

[1] Abouelhoda, M. I., Kurtz, S. and Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algorithms*, Vol. 2, No. 1, pp. 53–86 (2004).

[2] Arroyuelo, D. and Navarro, G.: Space-efficient construction of Lempel-Ziv compressed text indexes, *Inf. Comput.*, Vol. 209, No. 7, pp. 1070–1102 (2011).

[3] Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M. and Tsuruta, K.: The "runs" theorem, *SIAM J. Comput.*, Vol. 46, No. 5, pp. 1501–1514 (online), DOI: 10.1137/15M1011032 (2017).

[4] Barcelo, H.: On the action of the symmetric group on the free Lie algebra and the partition lattice, *J. Comb. Theory, Ser. A*, Vol. 55, No. 1, pp. 93–129 (1990).

[5] Bassino, F., Clément, J. and Nicaud, C.: The standard factorization of Lyndon words: an average point of view, *Discret. Math.*, Vol. 290, No. 1, pp. 1–25 (2005).

[6] Belazzougui, D., Boldi, P., Pagh, R. and Vigna, S.: Fast prefix search in little space, with applications, *Proc. ESA*, LNCS, Vol. 6346, pp. 427–438 (2010).

[7] Bille, P., Gørtz, I. L., Cording, P. H., Sach, B., Vildhøj, H. W. and Vind, S.: Fingerprints in compressed strings, *J. Comput. Syst. Sci.*, Vol. 86, pp. 171–180 (2017).

[8] Bille, P., Landau, G. M., Raman, R., Sadakane, K., Satti, S. R. and Weimann, O.: Random access to grammar-compressed strings and trees, *SIAM J. Comput.*, Vol. 44, No. 3, pp. 513–539 (2015).

[9] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A. and Shelat, A.: The smallest grammar problem, *IEEE Trans. Information Theory*, Vol. 51, No. 7, pp. 2554–2576 (online), DOI: 10.1109/TIT.2005.850116 (2005).

[10] Chen, K. T., Fox, R. H. and Lyndon, R. C.: Free differential calculus, IV. The quotient groups of the lower central series, *Annals of Mathematics*, Vol. 68, No. 1, pp. 81–95 (1958).

[11] Christiansen, A. R., Ettienne, M. B., Kociumaka, T., Navarro, G. and Prezza, N.: Optimal-time dictionary-compressed indexes, *arxiv:1811.12779* (2018).

[12] Claude, F. and Navarro, G.: Self-indexed grammar-based compression, *Fundam. Inform.*, Vol. 111, No. 3, pp. 313–337 (2011).

[13] Claude, F. and Navarro, G.: Improved grammar-based compressed indexes, *Proc. SPIRE*, LNCS, Vol. 7608, pp. 180–192 (2012).

[14] Cormode, G. and Muthukrishnan, S.: The string edit distance matching problem with moves, *ACM Trans. Algorithms*, Vol. 3, No. 1, pp. 2:1–2:19 (online), DOI: 10.1145/1219944.1219947 (2007).

[15] Duval, J.: Factorizing words over an ordered alphabet, *J. Algorithms*, Vol. 4, No. 4, pp. 363–381 (1983).

[16] Ferragina, P., González, R., Navarro, G. and Venturini, R.: Compressed text indexes: From theory to practice, *ACM Journal of Experimental Algorithmics*, Vol. 13, pp. 1.12:1 – 1.12:31 (2008).

[17] Ferragina, P. and Manzini, G.: Opportunistic data structures with applications, *Proc. FOCS*, IEEE Computer Society, pp. 390–398 (online), DOI: 10.1109/SFCS.2000.892127 (2000).

[18] Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y. and Puglisi, S. J.: A faster grammar-based self-index, *Proc. LATA*, LNCS, Vol. 7183, pp. 240–251 (2012).

[19] Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y. and Puglisi, S. J.: LZ77-Based Self-indexing with Faster Pattern Matching, *Proc. LATIN*, LNCS, Vol. 8392, pp. 731–742 (2014).

[20] Gagie, T., Navarro, G. and Prezza, N.: Optimal-time text indexing in

BWT-runs bounded space, *Proc. SODA*, pp. 1459–1477 (2018).
[21] Grossi, R. and Vitter, J. S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract), *Proc. STOC*, pp. 397–406 (online), DOI: 10.1145/335305.335351 (2000).
[22] Hon, W., Lam, T. W., Sadakane, K. and Sung, W.: Constructing compressed suffix arrays with large alphabets, *Proc. ISAAC*, LNCS, Vol. 2906, pp. 240–249 (online), DOI: 10.1007/978-3-540-24587-2_26 (2003).
[23] I, T., Nakashima, Y., Inenaga, S., Bannai, H. and Takeda, M.: Faster Lyndon factorization algorithms for SLP and LZ78 compressed text, *Theor. Comput. Sci.*, Vol. 656, pp. 215–224 (2016).
[24] Jez, A.: Approximation of grammar-based compression via recompression, *Theor. Comput. Sci.*, Vol. 592, pp. 115–134 (2015).
[25] Karp, R. M., Miller, R. E. and Rosenberg, A. L.: Rapid Identification of Repeated Patterns in Strings, Trees and Arrays, *Proc. STOC*, pp. 125–136 (1972).
[26] Karp, R. M. and Rabin, M. O.: Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development*, Vol. 31, No. 2, pp. 249–260 (online), DOI: 10.1147/rd.312.0249 (1987).
[27] Kempa, D. and Prezza, N.: At the roots of dictionary compression: string attractors, *Proc. STOC*, pp. 827–840 (2018).
[28] Kieffer, J., Yang, E., Nelson, G. and Cosman, P.: Universal lossless compression via multilevel pattern matching, *IEEE Trans. Information Theory*, Vol. 46, No. 4, pp. 1227–1245 (2000).
[29] Kieffer, J. C. and Yang, E.: Grammar-based codes: A new class of universal lossless source codes, *IEEE Trans. Information Theory*, Vol. 46, No. 3, pp. 737–754 (online), DOI: 10.1109/18.841160 (2000).
[30] Kreft, S. and Navarro, G.: On compressing and indexing repetitive sequences, *Theor. Comput. Sci.*, Vol. 483, pp. 115–133 (2013).
[31] Lam, T. W., Sadakane, K., Sung, W. and Yiu, S.: A space and time efficient algorithm for constructing compressed suffix arrays, *Proc. COCOON*, LNCS, Vol. 2387, Springer, pp. 401–410 (2002).
[32] Larsson, N. J. and Moffat, A.: Offline dictionary-based compression, *Proc. DCC*, pp. 296–305 (1999).
[33] Lothaire, M.: *Combinatorics on Words*, Addison-Wesley (1983).
[34] Lyndon, R. C.: On Burnside's Problem, *Trans. AMS*, Vol. 77, No. 2, pp. 202–215 (1954).
[35] Mäkinen, V. and Navarro, G.: Compressed compact suffix arrays, *Proc. CPM*, LNCS, Vol. 3109, pp. 420–433 (online), DOI: 10.1007/978-3-540-27801-6_32 (2004).
[36] Manber, U. and Myers, E. W.: Suffix arrays: A new method for on-line string searches, *SIAM J. Comput.*, Vol. 22, No. 5, pp. 935–948 (1993).
[37] Maruyama, S., Sakamoto, H. and Takeda, M.: An online algorithm for lightweight grammar-based compression, *Algorithms*, Vol. 5, No. 2, pp. 2014–235 (2012).
[38] Maruyama, S., Tabei, Y., Sakamoto, H. and Sadakane, K.: Fully-online grammar compression, *Proc. SPIRE*, LNCS, Vol. 8214, pp. 218–229 (2013).
[39] Mehlhorn, K., Sundar, R. and Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time, *Algorithmica*, Vol. 17, No. 2, pp. 183–198 (online), DOI: 10.1007/BF02522825 (1997).
[40] Navarro, G. and Mäkinen, V.: Compressed full-text indexes, *ACM Comput. Surv.*, Vol. 39, No. 1, pp. 2:1–2:61 (online), DOI: 10.1145/1216370.1216372 (2007).
[41] Navarro, G. and Prezza, N.: Universal compressed text indexing, *Theor. Comput. Sci.*, Vol. 762, pp. 41–50 (2019).
[42] Nevill-Manning, C. G., Witten, I. H. and Maulsby, D. L.: Compression by induction of hierarchical grammars, *Proc. DCC*, pp. 244–253 (1994).
[43] Nishimoto, T., I, T., Inenaga, S., Bannai, H. and Takeda, M.: Dynamic index and LZ factorization in compressed space, *Discret. Appl. Math.*, Vol. 274 (online), DOI: https://doi.org/10.1016/j.dam.2019.01.014 (2019).
[44] Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression, *Theoretical Computer Science*, Vol. 302, No. 1–3, pp. 211–222 (2003).
[45] Sahinalp, S. C. and Vishkin, U.: Data compression using locally consistent parsing, Technical report, University of Maryland Department of Computer Science (1995).
[46] Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression, *J. Discrete Algorithms*, Vol. 3, No. 2–4, pp. 416–430 (2005).
[47] Sakamoto, H., Kida, T. and Shimozono, S.: A space-saving linear-time algorithm for grammar-based compression, *Proc. SPIRE*, LNCS, Vol. 3246, pp. 218–229 (2004).
[48] Sakamoto, H., Maruyama, S., Kida, T. and Shimozono, S.: A space-saving approximation algorithm for arammar-based compression, *IEICE Transactions*, Vol. 92-D, No. 2, pp. 158–165 (2009).
[49] Storer, J. A.: NP-completeness results concerning data compression, Technical Report 234, Dept. of Electrical Engineering and Computer Science, Princeton University (1977).
[50] Takabatake, Y., Nakashima, K., Kuboyama, T., Tabei, Y. and Sakamoto, H.: siEDM: An efficient string index and search algorithm for edit distance with moves, *Algorithms*, Vol. 9, No. 2, pp. 26:1–26:18 (2016).
[51] Takabatake, Y., Tabei, Y. and Sakamoto, H.: Improved ESP-index: A practical self-index for highly repetitive texts, *Proc. SEA*, LNCS, Vol. 8504, pp. 338–350 (online), DOI: 10.1007/978-3-319-07959-2_29 (2014).
[52] Welch, T. A.: A technique for high performance data compression, *IEEE Computer*, Vol. 17, pp. 8–19 (1984).
[53] Ziv, J. and Lempel, A.: Compression of individual sequences via variable-rate coding, *IEEE Trans. Information Theory*, Vol. 24, No. 5, pp. 530–536 (1978).
[54] Ziv, J. and Lempel, A.: A universal algorithm for sequential data compression, *IEEE Trans. Information Theory*, Vol. 23, No. 3, pp. 337–343 (1977).