Dynamic index, LZ factorization, and LCE queries in compressed space

Takaaki Nishimoto\(^1\), Tomohiro I\(^2\), Shunsuke Inenaga\(^1\), Hideo Bannai\(^1\), and Masayuki Takeda\(^1\)

\(^1\) Department of Informatics, Kyushu University, Japan
\{takaaki.nishimoto, inenaga, bannai, takeda\}@inf.kyushu-u.ac.jp
\(^2\) Department of Computer Science, TU Dortmund, Germany
tomohiro.i@cs.tu-dortmund.de

Abstract. In this paper, we present the following results: (1) We propose a new dynamic compressed index of \(O(w)\) space, that supports searching for a pattern \(P\) in the current text in \(O(|P| \log w \log |P| \log N \log^* M) + \text{occ} \log N)\) time and insertion/deletion of a substring of length \(y\) in \(O((y + \log N \log^* M) \log w \log N \log^* M)\) time, where \(N\) is the length of the current text, \(M\) is the maximum length of the dynamic text, \(z\) is the size of the Lempel-Ziv77 (LZ77) factorization of the current text, and \(w = O(z \log N \log^* M)\). (2) We propose a new space-efficient LZ77 factorization algorithm for a given text of length \(N\), which runs in \(O(N \log w' + z \log w' \log^* N \log^* N^2)\) time with \(O(w')\) working space, where \(w' = O(z \log N \log^* N)\). (3) We propose a data structure of \(O(w)\) space which supports longest common extension (LCE) queries on the text in \(O(\log N \log^* N)\) time. The LCE data structure can also maintain a grammar-compressed representation of a dynamic text efficiently. On top of the above contributions, we show several applications of our data structures which improve previous known results.

1 Introduction

1.1 Dynamic compressed index

In this paper, we consider the dynamic compressed text indexing problem of maintaining a compressed index for a text string that can be modified. Although there exits several dynamic non-compressed text indexes (see e.g. \([17,2]\) for recent work), there has been little work for the compressed variants. Hon et al. \([7]\) proposed the first dynamic compressed index of \(O(\frac{1}{\varepsilon}(NH_0 + N))\) bits of space which supports searching of \(P\) in \(O(|P| \log^2 N (\log^* N + \log |\Sigma|) + \text{occ} \log^{1+\varepsilon} N)\) time and insertion/deletion of a substring of length \(y\) in \(O((y + \sqrt{N}) \log^{2+\varepsilon} N)\) amortized time, where \(0 < \varepsilon \leq 1\) and \(H_0 \leq \log |\Sigma|\) denotes the zeroth order empirical entropy of the text of length \(N\) \([7]\). Salson et al. \([19]\) also proposed a dynamic compressed index, called dynamic FM-Index. Although their approach works well in practice, updates require \(O(N \log N)\) time in the worst case. To our knowledge, these are the only existing dynamic compressed indexes to date.

In this paper, we propose a new dynamic compressed index, as follows:
Theorem 1. Let $M$ be the maximum length of the dynamic text to index, $N$ the length of the current text $T$, and $z$ the number of factors in the Lempel-Ziv 77 factorization of $T$ without self-references. Then, there exist a dynamic index of $O(w)$ space which supports searching of a pattern $P$ in $O(|P| \log w + \log w \log |P| \log N \log^* M)^2 + \text{occ} \log N)$ time and insertion/deletion of a substring of length $y$ in amortized $O((y + \log N \log^* M) \log w \log N \log^* M)$ time, where $w = O(z \log N \log^* M)$.

Since $z \geq \log N$, $\log w = \max\{\log z, \log(\log^* M)\}$. Hence, our index is able to find pattern occurrences faster than the index of Hon et al. when the $|P|$ term is dominating in the pattern search times. Also, our index allows faster substring insertion/deletion on the text when the $\sqrt{N}$ term is dominating.

Related work. Our data structures are based on the data structure presented by Alstrup et al. [1], which was originally designed for pattern matching on a dynamic sequence of strings. The data structure of Alstrup et al. [1] requires at least $\Omega(N' \log N')$ bits of space, where $N'$ is the total length of the strings in the current sequence, and hence it is not compressed. Our dynamic compressed index improves the space bound, while using more pattern search time.

Technically speaking, our index has close relationship to the ESP-index [20]. There are two major differences between ours and theirs: The first difference is that our index is dynamic but the ESP-index is static. The second difference is that the pattern search time of the ESP-index is proportional to the number $\text{occ}_c$ of occurrences of the so-called “core” of a query pattern $P$, which corresponds to a maximal subtree of the ESP derivation tree of a query pattern $P$. If $\text{occ}$ is the number of occurrences of $P$ in the text, then it always holds that $\text{occ}_c \geq \text{occ}$, and $\text{occ}_c$ cannot be upper bounded by $\text{occ}$. In contrast, as can be seen in Theorem 1 the pattern search time of our index is proportional to the number $\text{occ}$ of occurrences of a query pattern $P$. This became possible due to our discovery of a new property of the signature encoding [1] (stated in Lemma 7). In relation to our problem, there exists the library management problem of maintaining a text collection (a set of text strings) allowing for insertion/deletion of texts (see [15] for recent work). While in our problem a single text is edited by insertion/deletion of substrings, in the library management problem a text can be inserted to or deleted from the collection. Hence, algorithms for the library management problem cannot be directly applied to our problem.

1.2 Applications and extensions

LZ77 factorization in small space. As an application to our dynamic compressed index, we present a new LZ77 factorization algorithm for a string $T$ of length $N$, running in $O(N \log w + z \log w \log^3 N \log^2 N) \log^* N)^2$ time and $O(w)$ working space. Goto et al. [6] showed how, given the grammar-like representation for string $T$ generated by the LCA algorithm [15], to compute the LZ77 factorization of $T$ in $O(z \log^2 m \log^3 N + m \log m \log^2 N)$ time and $O(m \log^2 m)$ space, where $m$ is the size of the given representation. Sakamoto et al. [15] claimed that $m = O(z \log^3 N \log^* N)$, however, it seems that in this bound they do not
consider the production rules to represent maximal runs of non-terminals in the derivation tree. The bound we were able to obtain with the best of our knowledge and understanding is $m = O(z \log^2 N \log^* N)$, and hence our algorithm seems to use less space than the algorithm of Goto et al. [6]. Recently, Gagie [5] presented an algorithm to compute an approximation of the LZ77 factorization in $O(N^{1+\epsilon})$ time and $O(z/\epsilon)$ space, where $0 < \epsilon \leq 1$. The output of his algorithm consists of $O(z/\epsilon)$ factors.

**Longest common extensions.** Furthermore, we consider the longest common extension (LCE) problems on: an uncompressed string $T$ of length $N$; a grammar-compressed string $T$ represented by an straight-line program (SLP) of size $n$, or an LZ77-compressed string $T$ with $z$ factors. Bille et al. [6] showed a Monte Carlo randomized data structure built on a given SLP of size $n$ which supports LCE queries in $O(\log N \log \ell)$ time, where $\ell$ is the output of the LCE query and $N$ is the length of the uncompressed text. Their data structure requires only $O(n)$ space, but requires $O(N)$ time to construct. We present a new LCE data structure using compressed space, namely $O(w)$ space, supporting LCE queries in $O(\log N \log^* N)$ time. We show how to construct this data structure in $O(N \log w)$ time given an uncompressed string of length $N$, $O(n \log \log n \log N \log^* N)$ time given an SLP of size $n$, and $O(z \log w \log N \log^* N)$ time given the LZ77 factorization of size $z$. We remark that all of our solutions are deterministic, and permits us faster LCE queries than the data structure of Bille et al. [5] when $\log^* N = o(\log \ell)$ which in many cases is true.

All proofs omitted due to lack of space can be found in the appendices.

## 2 Preliminaries

Let $\Sigma$ be an ordered alphabet and $\$ \ be the lexicographically largest character in $\Sigma$. An element of $\Sigma^*$ is called a string. For string $w = xyz$, $x$ is called a prefix, $y$ is called a substring, and $z$ is called a suffix of $w$, respectively. The length of string $w$ is denoted by $|w|$. The empty string $\varepsilon$ is a string of length 0, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the $i$-th character of $s$. For any $1 \leq i \leq j \leq |w|$, $w[i..j]$ denotes the substring of $w$ that begins at position $i$ and ends at position $j$. Let $w[i..] = w[i..|w|]$ and $w[..i] = w[1..i]$ for any $1 \leq i \leq |w|$. For any string $w$, let $w^R$ denote the reversed string of $w$, that is, $w^R = w[|w|] \cdots w[2|w|1]$. For any strings $w$ and $u$, let $LCP(w, u)$ (resp. $LCS(w, u)$) denote the length of the longest common prefix (resp. suffix) of $w$ and $u$. Given two string $s_1, s_2$ and two integers $i, j$, let $LCE(s_1, s_2, i, j)$ denote a query which returns $LCP(s_1[i..|s_1|], s_2[j..|s_2|])$.

For any strings $p$ and $s$, let $Occ(p, s)$ denote all occurrence positions of $p$ in $s$, namely, $Occ(p, s) = \{ i \mid p = s[i..i + |p| - 1], 1 \leq i \leq |s| - |p| + 1 \}$.

A straight-line program (SLP) $S$ of size $n$ is a set of productions $S = \{ X_i \to expr_i \}_{i=1}^n$, where each $X_i$ is a distinct variable and each $expr_i$ is either $expr_i = X_i X_r$ ($1 \leq \ell, r < i$), or $expr_i = a$ for some $a \in \Sigma$. Note that $X_n$ derives only a single string $w$ and, therefore, we view the SLP as a compressed representation of the string $w$ that is derived from the variable $X_n$. The length $N$ of the string
In this section, we recall the data structure of Alstrup et al. [1] on which our data structure was based. Their data structure was originally designed for pattern matching on a dynamic sequence of strings. Their dynamic index uses $\Omega(|T|)$ space, however, by removing the support of their pattern matching query, we get their data structure which represents a dynamic text $T$ and supports text edit operations in compressed space.
3.1 Signature encoding

A signature dictionary $D$ of size $w$ is a set $D = \{e_i \rightarrow \text{expr}_i\}_{i=1}^w$ of assignments such that each $e_i$ is a distinct positive integer called a signature and $\text{expr}_i$ is in one of the following forms:

$$\text{expr}_i = \begin{cases} a & (a \in \Sigma), \\ e_i e_r & (l, r < i, e_i, e_r \in E_D \text{ and } e_i \neq e_r), \\ e_j^k & (j < i, e_j \in E_D \text{ and } k > 1), \end{cases}$$

where $E_D$ denotes the set of signatures in $D$. Let $\text{Assgn}_D$ be a function such that $\text{Assgn}_D(e_i) = \text{expr}_i$ iff $e_i \rightarrow \text{expr}_i \in D$.

We define the function $\text{Sig}_D : \Sigma \cup E_D^+ \cup (E_D \times N) \rightarrow E_D$, as follows: If there exists an assignment $e \rightarrow x$ in $D$, then $\text{Sig}_D(x) = e$. If $x = e_1 \cdots e_k$ with $e_1, \ldots, e_k \in E_D$, $3 \leq k \leq 4$, and $e_i \neq e_{i+1}$ for $1 \leq i \leq k - 1$, then $\text{Sig}_D(x) = \text{Sig}_D(\text{Sig}_D(e_1 e_2) e_3 \cdots e_k)$. If $k > 4$, then $\text{Sig}_D(x)$ is not defined. Otherwise, $\text{Sig}_D(x) = x$. When clear from the context, we write $\text{Sig}_D$ and $\text{Assgn}_D$ as $\text{Sig}$ and $\text{Assgn}$, respectively.

We define the function $\text{val} : \Sigma \cup E_D^+ \cup (E_D \times N) \rightarrow \Sigma^+$, as follows: If the input $p$ is a character $a \in \Sigma$, $\text{val}(p) = a$. If the input $p$ is a sequence of signatures with $|p| \geq 2$, then $\text{val}(p) = \text{val}(p[1]) \cdots \text{val}(p[|p|])$. If the input $p$ is a single signature $e \in E_D$, $\text{val}(p) = \text{val}(\text{Assgn}(e))$. If $s = \text{val}(p)$ for $p \in \Sigma \cup E_D^+ \cup (E_D \times N)$, then we say that $p$ represents string $s$.

Alstrup et al. showed how to represent a sequence $\langle s_1 \$, \ldots, s_k \$ \rangle$ of strings using a signature dictionary $D$, where each string in $F$ terminates with a special end-marker $\$ \rangle$ which appears nowhere else in the string. In so doing, we restrict the signature dictionary $D$ so that each signature $e_i \in E_D$ satisfies one of the two following conditions: (1) there exists an assignment $e_j \rightarrow \text{expr}_j$ in $D$ such that $j > i$ and the right-hand side $\text{expr}_j$ contains $e_i$, or (2) there exists an assignment $e_i \rightarrow e_r e_r \in D$ such that $e_i \in E_D$ and $e_r \rightarrow \$ \rangle \in \Sigma$. We say that the signature dictionary $D$ represents the sequence $\langle s_1 \$, \ldots, s_k \$ \rangle$ of strings if for any $s$ in the sequence there exists a unique signature $e \in E_D$ such that $\text{val}(e) = s$. We then write $F_D = \langle s_1 \$, \ldots, s_k \$ \rangle$. For each string $s$ in $F_D$, the signature $id(s) \$ \rangle$ representing $s$ is determined by a process called signature encoding which constructs a hierarchy of sequences of signatures in $E_D^+$ inductively, as follows:

$$\text{Shrink}_i^h[j] = \begin{cases} \text{Sig}(s[j]) & \text{for } t = 0 \text{ and } 1 \leq j \leq |s|, \\ \text{Sig}(\text{Eblock}(\text{Pow}_i^{h-1}[j])) & \text{for } 0 < t \leq h \text{ and } 1 \leq j \leq |\text{Eblock}(\text{Pow}_i^{h-1})|, \end{cases}$$

$$\text{Pow}_i^h[j] = \text{Sig}(\text{Epow}(\text{Shrink}_i^h[j])) \text{ for } 0 \leq t \leq h \text{ and } 1 \leq j \leq |\text{Epow}(\text{Shrink}_i^h)|,$$

$$id(s) = \text{Sig}(\text{Eblock}(\text{Pow}_i^h \cdot \text{Sig}(s))).$$

where $h$ is the minimum integer satisfying $|\text{Pow}_i^h| = 1$. We remark that $\text{Shrink}_i^h$ and $\text{Pow}_i^h$ are sequences of signatures. Hence, $|\text{Shrink}_i^h| = |s|$ if $t = 0$, and $|\text{Shrink}_i^h| = |\text{Eblock}(\text{Pow}_i^{h-1})|$ otherwise. Also, $|\text{Pow}_i^h| = |\text{Eblock}(\text{Pow}_i^{h-1})|$. By the definition of the Eblock function and by Lemma 11 for any $1 \leq t \leq h$, $|\text{Shrink}_i^h| \leq |\text{Pow}_i^{h-1}|/2$ and $|\text{Pow}_i^h| \leq |\text{Pow}_i^{h-1}|/2$. Thus $h \leq \log |s|$.
Since we handle strings of length at most \( M \), the values of signatures in \( \mathcal{E}_D \) can be bounded by \( 2M - 1 \). We also remark that the inputs of the \( \text{Encblock} \) function above are sequences of signatures. Hence, \( \Delta_L \) of Lemma 2 is bounded by \( \log^* 2M + 6 = O(\log^* M) \).

By the definition of the signature encoding, for each string \( s \) in \( \mathcal{F}_D \), we can induce a tree for \( s \) from \( D \) and \( id(s) \), like the derivation tree of a context-free grammar which generates only \( s \). We call this tree the \textit{signature tree} of \( s \). See Fig. 4 which illustrates a signature tree. For \( e_i, e_j \in \mathcal{E}_D \), let \( v\text{Occ}(e_i, e_j) \) be the set of positions of the leftmost leaves of the subtrees rooted at the nodes labeled with \( e_i \) in the leaves of the signature tree of \( \text{val}(e_j) \). Namely,

\[
v\text{Occ}(e_i, e_j) = \begin{cases} \{k \mid k \in \text{vOcc}(e_i, e_l)\} & \text{if } e_j \rightarrow e_l e_r, \\ \cup\{\text{val}(e_l) + k \mid k \in \text{vOcc}(e_i, e_r)\} & \text{if } e_j = e^r, \\ \{1\} & \text{if } e_i = e_j, \\ \emptyset & \text{otherwise.} \end{cases}
\]

Let \( |v\text{Occ}(e)| = \sum_{j=1}^k |v\text{Occ}(e, id(s_j))| \), where \( \mathcal{F}_D = (s_1\$, \ldots, s_k\$) \).

We consider the following update operations on \( \mathcal{F}_D \), which actually invoke updates of \( D \).

- \( \text{CHAR}(c) \): Given a character \( c \in (\Sigma - \$) \), append \( c\$ \) to \( \mathcal{F}_D \).
- \( \text{CONCAT}(id(s_1\$, id(s_2\$, b)) \): Given two signatures \( id(s_1\$, id(s_2\$) \) for string \( s_1\$, s_2\$ \in \mathcal{F}_D \) and an integer \( b \in \{0, 1\} \), create a new string \( s = s_1 s_2\$ \) and append \( s \) to \( \mathcal{F}_D \). If \( b = 1 \) then remove \( s_1\$, s_2\$ \) from \( \mathcal{F}_D \), and if \( b = 0 \) then keep \( s_1\$, s_2\$ \) in \( \mathcal{F}_D \).
- \( \text{SPLIT}(id(s\$, k, b)) \): Given a signature \( id(s\$) \) for a string \( s\$ \in \mathcal{F}_D \) and two integers \( 1 \leq k < |s|, b \in \{0, 1\} \), create two new strings \( s_1 = s[1..k] \) and \( s_2 = s[k+1..|s|] \), and append \( s_1\$, s_2\$ \) to \( \mathcal{F}_D \). If \( b = 1 \) then remove \( s\$ \) from \( \mathcal{F}_D \), and if \( b = 0 \) then keep \( s\$ \) in \( \mathcal{F}_D \).
- \( \text{STRING}(s) \): Given a string \( s \in (\Sigma - \$)^+ \), append \( s\$ \) to \( \mathcal{F}_D \).
- \( \text{REMOVE}(id(s)) \): Given a signature \( id(s) \) for \( s \in \mathcal{F}_D \), remove \( s \) from \( \mathcal{F}_D \).

The operation \( \text{REMOVE} \) was not explicitly mentioned in [1], but were implicitly supported by the non-persistent version of their data structure. See also Example 1 in Appendix E for an example of the above operations.

During updates, a new assignment \( e \rightarrow \text{expr} \) is appended to \( D \) whenever it is needed, where \( e \) is the smallest positive integer that has not been used as a signature. Also, a signature is deleted as soon as it disappears from the signature trees of any strings in \( \mathcal{F}_D \). We remark that we do not consider reusing these deleted integers. Note also that a new string is not appended to \( \mathcal{F}_D \) if it already exists in \( \mathcal{F}_D \). In the next subsection, we describe how to implement \( D \).

**Lemma 2 (1).** Let \( D \) be a signature dictionary based on signature encoding, and let \( D' \) be the result of applying the \text{CONCAT} or \text{SPLIT} operation to \( D \). Then \(|\mathcal{E}_D \cup \mathcal{E}_{D'} - \mathcal{E}_D \cap \mathcal{E}_{D'}| = O(\log N \log^* M) \), where \( N \) is the length of the longest string in \( \mathcal{F}_{D'} \).
Lemma 3 ([16]). Let $T$ denote a string of length $N$ and let $\mathcal{D}$ denote a signature dictionary based on signature encoding for $\mathcal{F}_\mathcal{D} = \langle T \rangle$. Then $|\mathcal{E}_\mathcal{D}| = O(z \log N \log^* X)$, where $z$ is the number of LZ77 factors of $T$, $X = M$ if $T$ is dynamic, otherwise $X = N$.

Lemma 4 ([16]). Let $s$ be any string in $\mathcal{F}_\mathcal{D}$, and let $\text{id}(s) = e$. Let $p$ be any non-empty substring of $s$. Then, for any $1 \leq i \leq |\text{val}(e)|$ with $\text{val}(e)[i..i+|p|-1] = p$, there exists a common sequence $v = e_1, \ldots, e_d$ of signatures in the signature tree of $s$ such that: (1) $\text{val}(v) = \text{val}(e_1 \cdots e_d) = p$, (2) $i \in \text{vOcc}(e_1, e)$ and $i + |\text{val}(e_1 \cdots e_{d-1})| \in \text{vOcc}(e_d, e)$ for any $2 \leq h \leq d$, and (3) $|\text{Epow}(v)| = O(\log |p| \log^* M)$.

The sequence $v$ of signatures in Lemma 4 is called a common sequence of $p = \text{val}(e)[i..i+k-1]$ w.r.t. $\mathcal{D}$.

### 3.2 Data structure for $\mathcal{D}$

To process operations for a signature dictionary $\mathcal{D}$ with signature encoding efficiently, we introduce a data structure $\mathcal{H}$ for $\mathcal{D}$. $\mathcal{H}$ consists of two data structures. Let $w = |\mathcal{E}_\mathcal{D}|$. The first data structure is a balanced binary search tree for $\mathcal{D}$, which is used to compute $\text{Sig}(x)$ for a given $x \in \Sigma \cup \mathcal{E}_\mathcal{D}^+ \cup (\mathcal{E}_\mathcal{D} \times \mathcal{N})$, and/or to remove an assignment from $\mathcal{D}$ in $O(\log w)$ time. The second data structure is a DAG of total size $O(w)$ that is a compact representation of the set of signature trees of signatures in $\mathcal{E}_\mathcal{D}$. Each node represents a signature in $\mathcal{E}_\mathcal{D}$ and out-going edges represent the assignments in $\mathcal{D}$. For example, if there exists an assignment $e_i \rightarrow e_j e_r$, the edge from $e_r$ to $e_i$ is associated with it. For each node of signature $e$, we also associate $|\text{val}(e)|$. Once the node corresponding to a signature is identified, the DAG can be used to traverse any path in the signature tree in time linear in the length of the path. Each node will also hold information of incoming edges in order to compute $\text{vOcc}(e_i, e_j)$ and to dynamically maintain $\mathcal{D}$. See also Fig. 4 in Appendix 4 which illustrates an example of the above DAG.

If a data structure $\mathcal{H}$ to maintain $\mathcal{D}$ requires $O(a)$ time to compute $\text{Sig}(x)$ for any $x \in \Sigma \cup \mathcal{E}_\mathcal{D}^+ \cup (\mathcal{E}_\mathcal{D} \times \mathcal{N})$ or to update the data structure with any operation on $\mathcal{F}_\mathcal{D}$, and occupies $O(b)$ space, then we say that it is an $\mathcal{H}(a, b)$ signature dictionary. It follows from the arguments in the previous paragraph that there exists a deterministic $\mathcal{H}(\log w, w)$ signature dictionary.

Lemma 5 ([11]). Using $\mathcal{H}(\log w, w)$ for $\mathcal{F}_\mathcal{D}$, we can support

- $\text{CHAR}(c)$ in $O(\log w)$ time,
- $\text{CONCAT}(\text{id}(s_1), \text{id}(s_2), b)$ in $O(\log w \log |s_1 s_2| \log^* M)$ time,
- $\text{SPLIT}(\text{id}(s_1), i, b)$ in $O(\log w \log |s| \log^* M)$ time,
- $\text{REMOVE}(\text{id}(s))$ in $O(|s| \log w)$ time, and
- $\text{STRING}(P)$ in $O(|P| \log w)$ time and $O(|P|)$ working space,

where $s, s_1, s_2 \in \mathcal{F}_\mathcal{D}$, $P \in (\Sigma - \{\$\})^+$, and $1 < i < |s_1|$.

3 Alstrup et al. [11] used hashing to maintain $\mathcal{D}$ and obtained a randomized $\mathcal{H}(1, w)$ signature dictionary. However, since we are interested in the worst case time complexities, we use balanced binary search trees in place of hashing.
4 Compressed dynamic index

In this section, we present how to build a dynamic compressed index based on the signature encoding of Section 3.

First, we show our results on construction of an $\mathcal{H}(\log w, w)$ signature dictionary for $F_D$. It can be constructed from various types of inputs, such as (1) a plain (uncompressed) string $T$, (2) the LZ77 factorization of $T$, and (3) an SLP which represents $T$, as summarized by the following theorem.

**Theorem 2.** 1. Given a string $S$ of length $N$, we can construct $\mathcal{H}(\log w, w)$ for $F_D = \langle T$ $\rangle$ in $O(|T|)$ time and working space, or $O(|T| \log w)$ time and $O(w)$ working space.

2. Given an LZ77 factors without self reference of size $z$ representing $T$ of length $N$, we can construct $\mathcal{H}(\log w, w)$ for $F_D = \langle T$ $\rangle$ in $O(z \log w \log N \log^* M)$ time and $O(w)$ working space.

3. Given an SLP $S = \{X_i \to expr_i\}_{i=1}^n$ of size $n$ representing $T$ of length $N$, we can construct $\mathcal{H}(\log w, w')$ for $F_D = \langle val(X_1)\$, $\cdots,$ $val(X_n)\rangle$ in $O(n \log n \log N \log^* M)$ time and $O(w')$ working space, where $w' = O(n \log N \log^* M)$.

In the static case, the $M$ term of Theorem 2 can be replaced with the total length $N$ of static strings.

4.1 Static index

In this subsection, we show how to build a static index using the signature encoding, on which our dynamic index will be based. We describe an $\mathcal{H}(\log w, w)$ index for $F_D = \langle T$ $\rangle$ which computes $Occ(P, T)$ for a given pattern $P$. We will use operation STRING to temporarily add a query pattern $P$ to our data structure. The query pattern $P$ will be deleted by operation DELETE immediately after the search for $P$ has been finished. To describe the search algorithm, we define the following functions.

**Definition 2.** Let $E_{xy}$ denote the set of signatures $\{e \mid e \in E_D, e \not\sim a \in \Sigma\}$. For a signature $e \in E_{xy}$, let $e$.left and $e$.right denote $val(e_l)$ and $val(e_r)$ respectively if $e \rightarrow e_l e_r$. If $e \rightarrow q^*$, then $e$.left = $val(q)$ and $e$.right = $val(q^{-1})$.

**Definition 3.** For a string $P$ with $|P| > 1$, integers $i, k \in \mathbb{N}$, a signature $e \in E_{xy}$ and a set $P = \{1, \cdots, |P| - 1\}$ of integers, let

- $PrimOcc_D(e, P) = \{i \in P \mid P[.i]$ is a suffix of e.left, $P[i+1..]$ is a prefix of e.right$\}$,
- $PrimOcc_D(P, i) = \{(e, i) \mid e \in E_{xy}, i \in PrimOcc_D(e, P)\}$,
- $PrimOcc_D(P) = \{(e, j) \mid e \in E_{xy}, j \in PrimOcc_D(e, P)\}$,
- $PowOcc_D(e, i, k) = \{[val(e_i)] - i + 1\}$ if $e \rightarrow e_l e_r$,
- $\{[val(q)] - i + 1, \cdots, [val(q^*)] - i + 1\}$ if $e \rightarrow q^*$,

where $x$ is the maximum integer such that $|val(q^*)| - i + k \leq |val(e)|$.
We can compute $\text{Occ}_1 \leq |P| - 1$ such that $\text{val}(e)[|\text{val}(q)| - i + 1..|\text{val}(q)| - i + |P|] = P$, then $\text{val}(e)[x..x + |P| - 1] = P$ for $x \in \text{PowOcc}(e, i, |P|)$ because $\text{val}(e)$ is a run of $\text{val}(q)$. The following observation is clear from the above definitions.

**Observation 1** For any string $P$ with $|P| > 1$, $\text{Occ}(P, T) = \{j + k - 1 | (e, i) \in \text{PrimOcc}_D(P), j \in \text{PowOcc}_D(e, i, |P|), k \in \nu\text{Occ}(e, \text{id}(T))\}.$

When $|P| > 1$, we can compute $\text{PrimOcc}_D(P, i)$ efficiently by a 2D range reporting query as in the primary search algorithm for SLPs proposed by Claude and Navarro [4].

**Lemma 6.** Let $P$ be a pattern with $|P| > 1$. There exists a data structure for $D$ based on $H(\log w, w)$, which finds PrimOcc$_D(P, i)$ from a given id$(P)$ and an integer $i$ in $O(\log w \log |P| \log N (\log^* M)^2 + \text{occ}(\log w / \log \log w))$ time, occupying $O(w)$ space. We can update this data structure in $O(\log w \log N \log^* M)$ amortized time when a signature is inserted into or deleted from $E_D$.

### 4.2 Speeding up pattern matching

We could compute $\text{Occ}(P, T)$ by using the data structure of Lemma 6 for each $1 \leq i < |P|$ as in previous approaches [4]. Here we present two new ideas: (1) Efficient computation of 2D range reporting queries in Lemma 6 using LCE queries (Lemma 10) in compressed space. (2) Reducing the number of 2D range reporting queries from $O(|P|)$ to $O(\log |P| \log^* M)$ by the following lemma.

**Lemma 7.** Let $P$ denote a string and let $v = v_1, \ldots, v_k = \text{Epow}(u)$ where $u$ is the common sequence of $P$. Consider the set of integers $P' = \{|\text{val}(v_1)|, \ldots, |\text{val}(v_1, \ldots, v_{k-1})|\}$. If $|P| > 1$ and $|\text{Pow}_0^P| > 1$, then PrimOcc$_D(P) = \{(p, i) \in \text{PrimOcc}_D(P, i) \mid i \in P'\}$.

We can compute $P'$ by the following lemma.

**Lemma 8.** Given id$(P)$ of a string $P \in F_D$ and integer $i, k$, we can compute the common sequence $v$ of $P[i..i + k - 1]$ in $O((\log |P|) + \log k) \log^* M)$ time by $H(\log w, w)$ signature dictionary.

Therefore we can get the following lemma by Lemmas 3, 4, 6 and a short proof.

**Lemma 9.** For a text of length $N$, there exists a static index based on an $H(\log w, w)$ signature dictionary for $F_D = \langle T \rangle$ which finds all occ occurrences of a given pattern $P$ in $O((|P| + \log |P| \log N (\log^* M)^2) \log w + \text{occ} \log N)$ time, occupying $O(w)$ space. We can update this data structure in $O(\log w \log N \log^* M)$ amortized time when a signature is inserted into or deleted from $D$. 

9
4.3 Dynamic Index

We can dynamize our static index of the previous subsections as follows: Let \( \text{INSERT}(i, Y) \) and \( \text{DELETE}(i, k) \) denote an update operation of \( T \) which inserts a string \( Y \) of length \( k \) into \( T \) at position \( i \) and deletes a substring \( T[i..i+k] \) from \( T \), respectively. We can conduct \( \text{INSERT}(i, Y) \) on \( D \) in \( O(\log w \log N \log^* M + k \log w) \) time by a constant number of \( \text{STRING}, \text{REMOVE}, \text{CONCAT} \) and \( \text{SPLIT} \) operations. The number of created or removed signatures in \( E_D \) times can be improved to \( O(\log N \log^* M + k) \) by Lemma \( \ref{lem:dynamic-signature} \). Hence we can update the data structures of Lemma \( \ref{lem:dynamic-signature} \) and \( \ref{lem:dynamic-signature2} \) in \( O((\log N \log^* M + k) \log w \log N \log^* M) \) time. Therefore, we get Theorem \( \ref{thm:dynamic-index} \).

5 Applications

In this section, we present a number of applications of the data structures of Sections \( \ref{sec:dynamic} \) and \( \ref{sec:applications} \).

The first is an application of signature dictionaries to the longest common extension queries, which is summarized in the next lemma.

**Lemma 10.** Using an \( \mathcal{H}(\log w, w) \) signature dictionary, we can support queries \( \text{LCE}(s_1, s_2, i, j) \) and \( \text{LCE}(s_1^R, s_2^R, i, j) \) in \( O(\log |s_1 s_2| \log^* X) \) time, where \( s_1, s_2 \in \mathcal{F}_D \), and \( X = M \) (the maximum total length of strings in \( \mathcal{F}_D \)) in the dynamic case and \( X = N \) (the total length of strings in \( \mathcal{F}_D \)) in the static case.

Theorems \( \ref{thm:text-compression} \) and \( \ref{thm:compressed-string-processing} \) are applications to text compression.

**Theorem 3.** Given a string \( T \) of length \( N \), we can compute the LZ77 factorization of \( T \) in \( O(N \log w + z \log w \log^3 N (\log^* N)^2) \) time and \( O(w) \) working space, where \( z \) is the size of the LZ77 factorization of \( T \) and \( w = O(z \log N \log^* N) \).

**Theorem 4.** (1) Given an \( \mathcal{H}(\log w, w) \) signature dictionary for \( \mathcal{F}_D = \langle T \rangle \), we can compute an SLP \( S \) of size \( O(w \log |T|) \) representing \( T \) in \( O(|T|) \) time. (2) Let us conduct a single \( \text{INSERT} \) or \( \text{DELETE} \) operation on the string \( T \) represented by the SLP of (1). Let \( d \) be the length of the substring to be inserted or deleted, and let \( T' \) be the resulting string. During the above operation on the string, we can update, in \( O((d + \log M \log^* M) \log N) \) time, the SLP of (1) to an SLP \( S' \) of size \( O(w \log |T'|) \) which represents \( T' \), where \( M \) is the maximum length of the dynamic text, \( \mathcal{F}_D' = \langle T' \rangle \), and \( w' = |\mathcal{E}_D'| \).

Theorems \( \ref{thm:compressed-string-processing} \) and \( \ref{thm:compressed-string-processing2} \) are applications to compressed string processing (CSP), where the task is to process a given compressed representation of string(s) without explicit decompression.

**Theorem 5.** Given an SLP \( S \) of size \( n \) representing a string of length \( N \), we can construct, in \( O(n \log n \log N \log^* N) \) time, a data structure which occupies \( O(n \log N \log^* N) \) space and supports \( \text{LCP}(X_i, X_j) \) and \( \text{LCS}(X_i, X_j) \) queries for variables \( X_i, X_j \) in \( O(\log N) \) time. The \( \text{LCP}(X_i, X_j) \) and \( \text{LCS}(X_i, X_j) \) query times can be improved to \( O(1) \) using \( O(n \log n \log N \log^* N) \) preprocessing time.
**Theorem 6.** Given an SLP $S$ of size $n$ representing a string $T$ of length $N$, there is a data structure which occupies $O(z \log N \log^* N + n)$ space and supports queries $\text{LCE}(X_i, X_j, a, b)$ for variables $X_i, X_j$, $1 \leq a \leq |X_i|$ and $1 \leq b \leq |X_j|$ in $O(\log N \log^* N)$ time. The data structure can be constructed in $O(n \log \log n \log N \log^* N)$ preprocessing time and $O(n \log N \log^* N)$ working space, where $z \leq n$ is the size of the LZ77 factorization of $T$.

Let $h$ be the height of the derivation tree of a given SLP $S$. Note that $h \geq \log N$. Matsubara et al. [12] showed an $O(n h(n + \log N))$-time $O(n(n + \log N))$-space algorithm to compute an $O(n \log N)$-size representation of all palindromes in the string. Their algorithm uses a data structure which supports in $O(h^2)$ time, LCE queries of a special form $\text{LCE}(X_i, X_j, 1, p_j)$ [13]. This data structure takes $O(n^2)$ space and can be constructed in $O(n^2 h)$ time [11]. Using Theorem 6, we obtain a faster algorithm, as follows:

**Theorem 7.** Given an SLP of size $n$ representing a string of length $N$, we can compute an $O(n \log N)$-size representation of all palindromes in the string in $O(n \log^2 N \log^* N)$ time and $O(n \log N \log^* N)$ space.

A non-empty string $s$ is called a Lyndon word if $s$ is the lexicographically smallest suffix of $s$. The Lyndon factorization of a non-empty string $w$ is a sequence of pairs $([f_i], p_i)$ where each $f_i$ is a Lyndon word and $p_i$ is a positive integer such that $w = f_1^{p_1} \cdots f_m^{p_m}$ and $f_{i-1}$ is lexicographically smaller than $f_i$ for all $1 \leq i < m$. I et al. [3] proposed a Lyndon factorization algorithm running in $O(n h(n + \log n \log N))$ time and $O(n^2)$ space. Their algorithm use the LCE data structure on SLPs [14] which requires $O(n^2 h)$ preprocessing time, $O(n^2)$ working space, and $O(h \log N)$ time for LCE queries. We can obtain a faster algorithm using Theorem 6.

**Theorem 8.** Given an SLP of size $n$ representing a string of length $N$, we can compute the Lyndon factorization of the string in $O(n(n + \log n \log N \log^* N))$ time and $O(n(n + \log N \log^* N))$ space.

We can also solve the grammar compressed dictionary matching problem [10] with our data structures. We preprocess an input dictionary SLP (DSLP) $\langle G, m \rangle$ with $n$ productions that represent $m$ patterns. Given an uncompressed text $T$, the task is to output all occurrences of the patterns in $T$.

**Theorem 9.** Given a DSLP $\langle G, m \rangle$ of size $n$ that represents a dictionary $\Pi_{\langle G, m \rangle}$ for $m$ patterns of total length $N$, we can preprocess the DSLP in $O((n \log \log n + m \log m) \log N \log^* N)$ time and $O(n \log N \log^* N)$ space so that, given any text $T$ in a streaming fashion, we can detect all $occ$ occurrences of the patterns in $T$ in $O(|T| \log m \log N \log^* N + occ)$ time.

It was shown in [10] that we can construct in $O(n^4 \log n)$ time a data structure of size $O(n^2 \log N)$ which finds all occurrences of the patterns in $T$ in $O(|T| (h + m))$ time, where $h$ is the height of the derivation tree of DSLP $\langle G, m \rangle$. Note
that our data structure of Theorem 9 is always smaller, and runs faster when \( h = \omega(\log m \log N \log^* N) \).

**Acknowledgments.** We would like to thank Paweł Gawrychowski for drawing our attention to the work by Alstrup et al. [12] and for fruitful discussions.

**References**

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Dynamic pattern matching. Tech. rep., Department of Computer Science, University of Copenhagen (1998)
2. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Proc. SODA 2000. pp. 819–828 (2000)
3. Bille, P., Cording, P.H., Görtz, I.L., Sach, B., Vildhøj, H.W., Vind, S.: Fingerprints in compressed strings. In: Proc. WADS 2013. pp. 146–157 (2013)
4. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundamenta Informaticae 111(3), 313–337 (2011)
5. Gagie, T.: Approximating LZ77 in small space. CoRR abs/1503.02416 (2015)
6. Goto, K., Maruyama, S., Inenaga, S., Bannai, H., Sakamoto, H., Takeda, M.: Restructuring compressed texts without explicit decompression. CoRR abs/1107.2729 (2011)
7. Hon, W., Lam, T.W., Sadakane, K., Sung, W., Yiu, S.: Compressed index for dynamic text. In: DCC 2004. pp. 102–111 (2004)
8. I, T., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. In: Proc. SPIRE. pp. 174–185 (2013)
9. I, T., Matsubara, W., Shimohira, K., Inenaga, S., Bannai, H., Takeda, M., Narisawa, K., Shinohara, A.: Detecting regularities on grammar-compressed strings. Inf. Comput. 240, 74–89 (2015). [http://dx.doi.org/10.1016/j.ic.2014.09.009](http://dx.doi.org/10.1016/j.ic.2014.09.009)
10. I, T., Nishimoto, T., Inenaga, S., Bannai, H., Takeda, M.: Compressed automata for dictionary matching. Theor. Comput. Sci. 578, 30–41 (2015), [http://dx.doi.org/10.1016/j.tcs.2015.01.019](http://dx.doi.org/10.1016/j.tcs.2015.01.019)
11. Lifshits, Y.: Processing compressed texts: A tractability border. In: Proc. CPM 2007. LNCS, vol. 4580, pp. 228–240 (2007)
12. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theor. Comput. Sci. 410(8–10), 900–913 (2009)
13. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica 17(2), 183–198 (1997)
14. Miyazaki, M., Shinohara, A., Takeda, M.: An improved pattern matching algorithm for strings in terms of straight-line programs. In: Proc. CPM 1997. pp. 1–11 (1997)
15. Munro, J.I., Nekrich, Y., Vitter, J.S.: Dynamic data structures for document collections and graphs. CoRR abs/1503.05977 (2015)
16. Sahinalp, S.C., Vishkin, U.: Data compression using locally consistent parsing. TechnicM report, University of Maryland Department of Computer Science (1995)
17. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In: FOCS. pp. 320–328. IEEE Computer Society (1996)
18. Sakamoto, H., Maruyama, S., Kida, T., Shimozono, S.: A space-saving approximation algorithm for grammar-based compression. IEICE Transactions 92-D(2), 158–165 (2009)
19. Salson, M., Lecroq, T., Léonard, M., Mouchard, L.: Dynamic extended suffix arrays. J. Discrete Algorithms 8(2), 241–257 (2010)
20. Takabatake, Y., Tabei, Y., Sakamoto, H.: Improved esp-index: A practical self-index for highly repetitive texts. In: Proc. SEA 2014. pp. 338–350 (2014)
21. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–349 (1977)
A Appendix : Dynamic Index

A.1 Proof of Lemma 6

To prove Lemma 6, we use the following known results.

Definition 4 (Dynamic Two-Dimensional Orthogonal Range Reporting Problem). This problem is to maintain a data structure for a dynamic set $S$ of points in 2D which supports the following range reporting query: given a query rectangle $(x_1, x_2, y_1, y_2)$, return $\{(x, y) \in S \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$.

In addition, insertion and deletion of points on $S$ must be supported efficiently.

Lemma 11 ([24]). There exists the data structure for the two-dimensional orthogonal range reporting problem which supports range reporting queries in $O(\log n + \text{occ}(\log n/\log \log n))$ time, and insertion/deletion of a point in amortized $O(\log n)$ time, where $\text{occ}$ is the number of the elements to output and $n = |S|$. This structure uses $O(n)$ space.

Definition 5 (Order Maintenance Problem). This problem is to maintain a data structure for a dynamic list $V$ which supports the following operations and queries.

$\text{INSERT}(x, y)$: Insert an element $y$ immediately after its predecessor $x$ in $V$.

$\text{DELETE}(x)$: Delete an element $x$ from $V$.

$\text{ORDER}(x, y)$: Given two elements $x, y$ in $V$, return true if $x$ is before $y$ in $V$, and false otherwise.

Lemma 12 ([25]). There exists a data structure for the order maintenance problem which supports $\text{INSERT}, \text{DELETE},$ and $\text{ORDER}$ in $O(1)$ time. This structure uses $O(|V|)$ space.

For any string $w$, let $w^R$ denote the reversed string of $w$, that is, $w^R = w[|w|] \cdots w[2]w[1]$.

If a string $w$ is lexicographically smaller than another non-empty string $u$, then we write $w \prec u$ or $u \succ w$. We write $w \prec_R u$ if $w^R \prec u^R$, and $\prec_R$ is called the reversed lexicographical order of strings.

We are ready to show Lemma 6.

Proof. Now, we describe the data structure of Lemma 6. This data structure maintains a 5-tuple $(E_x, E_y, V_x, V_y, A)$, where: $E_x = (e_1, \cdots, e_d)$ is the permutation of $E_{xy}$ with $d = |E_{xy}|$, such that $e_i, \text{left} \preceq_R e_j, \text{left}$ for $1 \leq i < j \leq d$. We maintain $E_x$ by a balanced binary search tree. Similarly, $E_y$ is also the permutation of $E_{xy}$ such that $e_i, \text{right} \preceq_R e_j, \text{right}$ for $1 \leq i < j \leq d$. $V_x$ and $V_y$ are the order maintenance data structures of $E_x$ and $E_y$ of Lemma 12, respectively. We store each signature in $E_{xy}$ in a 2D plane based on $E_x$ and $E_y$, and let $S$ be the set of these points. $A$ is the data structure of Lemma 11 for the set $S$ of points. Note that the total space is $O(w)$. We remark that $E_x$ and $E_y$ will be used to compute the query rectangle from a given pattern, and $V_x$ and $V_y$ will be used to compare any two elements in constant time for $A$. 

14
Next we describe the search algorithm which finds $\text{PrimOcc}_{P}(P, i)$ from given an integer $i$ and signature $id(P)$. We compute the minimum and maximum integers $x, x'$ such that $P[i..x] \preceq_R e_x, \text{left} \preceq_R e_x, \text{left} \preceq_R \$P[i..i]$ on $E_x$, by a binary search combined with LCE queries of Lemma [10]. Similarly, we compute the minimum and maximum integers $y, y'$ such that $P[i+1..y] \preceq_R e_y, \text{right} \preceq_R e_y, \text{right} \preceq_R P[i+1..\$]$ on $E_y$. We use a range report query with a rectangle $(e_x, e_x', e_y, e_y')$ and let $G$ denote the answers. Then, $\text{PrimOcc}_{P}(P, i) = \{(e, i) \mid e \in G\}$. Therefore, the total running time is $O(\log w \log |P| \log N(\log^* M)^2 + \text{occ}(\log w/\log \log w))$ time.

Next we consider the update time of the data structure. The bottleneck is the update time of $E_x$ and $E_y$, which takes $O(\log w \log N \log^* M)$ time per signature using a binary search combined with LCE queries of Lemma [10].

Therefore Lemma [6] holds. \hfill \square

A.2 Proof of Lemma 4

Although Lemma [4] was established in [16], here we provide a proof of the lemma for completeness. Some of the lemmas to be established in this appendix will also be used to show other properties later.

In this subsection, we show that there exists a common sequence of $P$ for any string $P$. The idea is that we introduce a new signature encoding for any string $P$ by $X\text{Shrink}$ and $X\text{Pow}$ (Definition [7]). Although the signature encoding of $P$ by $\text{Shrink}$ and $\text{Pow}$ finally creates a single and unique signature, the new signature encoding finally creates a unique signature sequence $u$ which satisfies Lemma [4].

To show that $u$ satisfies the condition (2) of Lemma [4], we show that $X\text{Shrink}_i^u$ and $X\text{Pow}_i^u$ occur in $\text{Shrink}_i^u$, $\text{Shrink}_i^u$ respectively for any string $s$ such $P$ is a substring of $s$ (Lemma [14]).

Proof. Let $u$ be a character sequence, a block sequence, or a run sequence. Let $\text{Sig}^+(u) = \text{Sig}(u[1]), \ldots, \text{Sig}(u[|u|])$. Let $\Delta_L = \Delta_L$ and $\Delta_R = 1 + \Delta_R$. Let $g(p)$ denote a function defined by the function $f$ in Lemma [1] which, given $M$-colored sequence $p$, returns the bit sequence $d$. The following observation holds from Lemma [1].

**Observation 2.** For any $M$-colored sequence $a, b, c$ such that $a[[a]] \neq b[1]$ and $b[[b]] \neq c[1]$. (1) If $|b| > \Delta_L + \Delta_R$, then $g(b)[i] = g(ab)c[[a] + i]$ for any integer $\Delta_L < i \leq |b| - \Delta_R + 1$. (2) If $|b| > \Delta_L$, then $g(b)[i] = g(ab)c[[a] + i]$ for $\Delta_L < i \leq |b|$. (3) If $|b| > \Delta_R$, then $g(b)[i] = g(bc)[i]$ for $1 \leq i \leq |b| - \Delta_R + 1$.

For signature sequence $p$, let $CEncb(p)$ be the function which returns the blocks of $p[|L + 1..|p] - \hat{R}$ in $E\text{block}(p)$, where $\hat{L}$ is the minimum integer such that $g(p)[\hat{L} + 1] = 1$ and $\hat{L} \geq \Delta_L$, and $\hat{R}$ is the minimum integer such that $g(p)[|p] - \hat{R} + 1] = 1$ and $\hat{R} \geq \Delta_R$. Similarly, let $LEncb(p)$ and $REncb(p)$ denote the functions which return the blocks of $p[1..|p] - \hat{R}$ and $p[\hat{L} + 1..|p]$, respectively. Note that $\Delta_L \leq \hat{L} \leq \Delta_L + 3$ and $\Delta_R \leq \hat{R} \leq \Delta_R + 3$ because every block size is at most 4.
For any integer sequence $s$, let $\text{LEncb}(s)$, $\text{REncb}(s)$, and $\text{RShrink}(s)$ denote $\hat{L}, \hat{R}$ in $\text{CEncb}(s)$, respectively. (1) For any integer sequence $s$, let $\Delta_1^L$, $\Delta_1^R$ denote $\hat{L}, \hat{R}$ in $\text{CEncb}(s)$, respectively. (2) For any integer sequence $s$, let $\Delta_2^L$ denote $\hat{L}$ in $\text{RShrink}(s)$. Then $\Delta_2^L = \Delta_1^L$.

**Proof.** This is true by Observation 2.

**Definition 6.** For a string $P$, let

$$XShrink_t^P = \begin{cases} \text{Sig}^+(P) & \text{for } t = 0, \\ \text{Sig}^+(\text{CEncb}(XPow_{t-1}^P)) & \text{for } 0 < t \leq \hat{h}^P, \\ \text{Sig}^+(\text{CEncb}(XPow_{t-1}^P)) & \text{for } t > \hat{h}^P, \end{cases}$$

$$XPow_t^P = \begin{cases} \text{Sig}^+(\text{Epow}(XShrink_t^P[\omega_L^{(P,t)} + 1..]\text{Shrink}_t^P - \omega_R^{(P,t)})) & \text{for } 0 \leq t < \hat{h}^P, \\ \text{Sig}^+(\text{Epow}(XShrink_t^P[\omega_L^{(P,t)} + 1..]\text{Shrink}_t^P - \omega_R^{(P,t)})) & \text{for } t \geq \hat{h}^P, \end{cases}$$

where $\omega_L^{(P,t)}$, $\omega_R^{(P,t)}$ are the maximum integers such that $|\text{Epow}(XShrink_t^P[\omega_L^{(P,t)}])| = 1$ and $|\text{Epow}(XShrink_t^P[\omega_R^{(P,t)}])| = 1$ respectively, and $\hat{h}^P$ is a minimum integer such that $|\text{Epow}(XShrink_h^P[\omega_L^{(P,h)}])| \leq \Delta^L + \Delta^R + 3$. Let $\Delta_t^{(P,t)}$, $\Delta_t^{(P,t)}$ denote $\hat{L}, \hat{R}$ in $\text{CEncb}(XPow_t^P)$, respectively.

**Definition 7.** For a string $P$, let

$$LShrink_t^{(s,P)} = \begin{cases} s \text{LEncb}(LPow_{t-1}^{(s,P)} XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } t = 0 \\ s \text{LEncb}(LPow_{t-1}^{(s,P)} XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } 0 < t \leq \hat{h}^P \\ s \text{LEncb}(LPow_{t-1}^{(s,P)} XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } t > \hat{h}^P \end{cases}$$

$$RShrink_t^{(s,P)} = \begin{cases} s \text{REncb}(XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } t = 0 \\ s \text{REncb}(XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } 0 < t \leq \hat{h}^P \\ s \text{REncb}(XPow_{t-1}^{P}[,\Delta_t^{(P,t-1)} + \Delta_R]) & \text{for } t > \hat{h}^P \end{cases}$$

$$LShrink_t^{(s,P)} = \text{Sig}^+(LShrink_t^{(s,P)})$$

$$RShrink_t^{(s,P)} = \text{Sig}^+(RShrink_t^{(s,P)})$$

$$LPow_t^{(s,P)} = \text{Sig}^+(\text{Epow}(LShrink_t^{(s,P)} XShrink_t^{P}[\omega_L^{(P,t)}]))$$

$$RPow_t^{(s,P)} = \text{Sig}^+(\text{Epow}(XShrink_t^{P}[\text{Shrink}_t^{P} - \omega_R^{(P,t)} + 1..]RShrink_t^{(s,P)})))$$
Lemma 14. Let $s$ be a string and $i, k$ be integers such that $1 \leq i \leq i + k - 1 \leq |s|$ and $P = s[i..i + k - 1]$. Then, (1)

$$\text{Shrink}_{t}^{s} = L\text{Shrink}_{t}^{(s_{L}, P)}X\text{Shrink}_{t}^{P}R\text{Shrink}_{t}^{(s_{R}, P)} \text{ for } 0 \leq t \leq \hat{h}^{P}$$

$$\text{Pow}_{t}^{s} = L\text{Pow}_{t}^{(s_{L}, P)}X\text{Pow}_{t}^{P}R\text{Pow}_{t}^{(s_{R}, P)} \text{ for } 0 \leq t < \hat{h}^{P}$$

where $s_{L} = s[..i]$ and $s_{R} = s[i+k..]$. (2) If $i = 1$ and $k = |s|$, then $|\text{Shrink}_{t}^{(s_{L}, P)}| = O(\log^{*} M)$, $|\text{Shrink}_{t}^{(s_{R}, P)}| = O(\log^{*} M)$ for $0 \leq t \leq \hat{h}^{P}$.

Proof. Follows immediately from Lemma 13 and the definition of the signature encoding. See also Fig. 1. Note that for any integer $i$ such that $\text{Shrink}_{t}^{s}[i] = \text{Shrink}_{t}^{s}[i+1]$, $\text{Sig}^{+}(E\text{pow}(\text{Shrink}_{t}^{s})) \neq \text{Sig}^{+}(E\text{pow}(\text{Shrink}_{t}^{s}[i..i]E\text{pow}(\text{Shrink}_{t}^{s}[i+1..])))$. □

| L\text{Pow}_{t}^{(s_{L}, P)} | X\text{Pow}_{t}^{P} | R\text{Pow}_{t}^{(s_{R}, P)} |
|-----------------|---------|-----------------|
| L\text{Shrink}_{t}^{(s_{L}, P)} | X\text{Shrink}_{t}^{P} | R\text{Shrink}_{t}^{(s_{R}, P)} |

\[\hat{\Delta}_{L}^{(P, t-1)}, \hat{\Delta}_{R}^{(P, t-1)}\]

Fig. 1. The relation between $X\text{Shrink}_{t}^{P}$ and $X\text{Pow}_{t-1}^{P}$. $X\text{Pow}_{t-1}^{P}[..\hat{\Delta}_{L}^{(P, t-1)}]$ is encoded as a block of $L\text{Shrink}_{t}^{(s_{L}, P)}$. $X\text{Pow}_{t-1}^{P}[|X\text{Pow}_{t}^{P}|-\hat{\Delta}_{R}^{(P, t-1)}..]$ is encoded as a block of $R\text{Shrink}_{t}^{(s_{R}, P)}$.

Definition 8. For any string $P$, let

$$\text{Uniq}^{\ P} = a_{0}b_{0} \cdots a_{h-1}b_{h-1}X\text{Shrink}_{h}^{P}c_{h-1}d_{h-1} \cdots c_{a_{0}},$$

where

$$a_{t} = X\text{Shrink}_{t}^{P}[..\omega_{L}^{(P, t)}],$$

$$b_{t} = X\text{Pow}_{t}^{P}[..\hat{\Delta}_{L}^{(P, t)}],$$

$$c_{t} = X\text{Pow}_{t}^{P}[|X\text{Pow}_{t}^{P}|-\hat{\Delta}_{R}^{(P, t)}..],$$

$$d_{t} = X\text{Shrink}_{t}^{P}[|X\text{Shrink}_{t}^{P}|-\omega_{R}^{(P, t)} + 1..],$$

$$h = \hat{h}^{P}.$$ 

Note that $val(\text{Uniq}^{\ P}) = P$ and $|E\text{pow}(\text{Uniq}^{\ P})| = O(\log |P| \log^{*} M)$. We handle $\text{Uniq}^{\ P}$ as a common sequence of $P$ because this signature sequence satisfies all conditions of Lemma 4 by Lemma 14. See also Fig. 2 for an example of $\text{Uniq}^{\ P}$. □
A.3 Proof of Lemma 7

Proof. Consider the common sequence of $P$ of Lemma 4 and see also Fig. 2. Assume that there exists an integer $i$ and a signature $e \in \mathcal{E}_{xy}$ such that $(e,i) \in \text{PrimOcc}_P(P)$ and $i \not\in P'$. We consider two cases on $i$: Case (1) There exists an integer $c$ which satisfies $|\text{val}(u[..c-1])| < i < |\text{val}(u[..c])|$. Case (2) There exist an integer $d,r$ such that $i = |\text{val}(v_1, \ldots, v_{d-1}, q^r)|$, $1 \leq d \leq k$ and $1 \leq r < r_d$ where $q^r_j = v_j$ for $1 \leq j \leq k$.

(1) By the definitions of common sequence and $\text{PrimOcc}$, $p \in \text{vOcc}(u[c], e)$ and an interval $[p..p + |\text{val}(u[c]) - 1|]$ is consumed in $[1..|e.|\text{left}||i..|e.|\text{left} + 1..|\text{val}(e)|]$, where $p = |e.|\text{left} - (i - |\text{val}(u[..c - 1])|)$. However $p < |e.|\text{left} < p + |\text{val}(u[c])|$ by the assumption, we get a contradiction.

(2) Consider two sub-cases: (i) $e \rightarrow e_i e_r (e_i \neq e_r)$ (ii) $e \rightarrow q^m$. (i) $|\text{val}(e)| - |\text{val}(q_d)| + 1 \in \text{vOcc}(q_d, e_i)$ and $1 \in \text{vOcc}(q_d, e_r)$ hold. In other word, $v_d$ (or a maximal run which contains $v_d$) is not encoded to a single signature in the signature encoding of $T$. This contradicts the definition of signature encoding. (ii) Similarly, $|\text{val}(q)| - |\text{val}(q_d)| + 1 \in \text{vOcc}(q_d, q)$ and $1 \in \text{vOcc}(q_d, q)$ hold. If $q_d \neq q$, this is the same as Case (i). Otherwise, $q_d = q$. By the definition of $\text{PrimOcc}(e, P)$, $e.|\text{left} = \text{val}(q) = \text{val}(v_1..v_{d-1}q_d)$. This holds when $r = 1$ and $d = 1$, then $q = q_d \rightarrow a \in \Sigma$. Next, $|\text{val}(q_d^{a^{d-r}}, v_{d+1}, \ldots, v_k)|$ is a prefix of $e.|\text{right} = a^{m-1}$. Since $q_d \neq q_{d+1}$ and $k > d$ ($d = 1$ and $k > 1$), $|\text{val}(q_d^{a^{d-r}}, v_{d+1}, \ldots, v_k)|$ is not a prefix of $e.|\text{right}$.

Therefore, Lemma 7 holds. \hfill \Box

A.4 Proof of Lemma 8

Proof. By the definition $\text{Uniq}$ function, given a signature $id(T)$ and two integers $i, k$, we can compute a signature sequence $u = \text{Uniq}(T[i..i+k-1])$ in $O((\log k + \log |T|) \log^* M)$ time from the root of the signature tree of $T$. Note that we can store $u$ in $O(\log k \log^* M)$ space because $|\text{Epow}(\text{Uniq}(T[i..i+k-1]))| = O(\log k \log^* M)$. \hfill \Box

A.5 Proof of Lemma 9

Proof. Note that we can compute $id(P)$ from $P$ in $O(|P| \log w)$ time using $\mathcal{H}(\log w, w)$. There exist three cases: (1) $|P| = 1$, (2) $|P| > 1$ and $|\text{Pow}_P| > 1$, (3) $|P| > 1$ and $|\text{Pow}_P| = 1$. (1) $\text{Occ}(P, T) = \text{vOcc}(e, id(T))$ holds, where $e \rightarrow a = P$. Hence we can compute $\text{Occ}(P, T)$ in $O(\log \log N)$ time, because $\mathcal{F}_P = \{T\}$ and the height of the signature tree of $T$ is $O(\log N)$. (2) We can compute $\text{Occ}(P, T)$ in $O(|P| + \log |P| \log N (\log^* M)^2) \log w + \log \log N)$ time by Lemmas 6 and 8. (3) $P = a^r, a \in \Sigma$ and $r > 1$, so $\text{Shrink}_{P}^r = q^r$ and $q \rightarrow a$. Therefore

$$\text{Occ}(P, T) = \{j + k - 1 \mid e \in \mathcal{E}_P, j \in \text{PowOcc}_P(e, 1, P), k \in \text{vOcc}(e, id(T))\}.$$
Fig. 2. The abstract image of signature tree of $s$ and the common sequence of $P$. The endmarker $\$ is omitted for simplicity. The gray intervals show $XShrink_h^P$ and $XPow_h^P$. $h$ and $h^P$ are the minimum integers such that $|Pow_h^P| = 1$ and $|Epow(XShrink_h^P, P)| \leq \Delta_L + \Delta_R + 3$, respectively. $s = s_LPs_R$, where $|s_L| = i - 1$. Let $u = Uniq(P)$ denote the common sequence of $P$, then $Epow(u) = v_1, \ldots, v_{16}$. Note that val($v_1, \ldots, v_{15}$) occurs in the signature tree of $s$.

where $E_P = \{e \mid e \in E_P, e \rightarrow q^c, c \geq r\}$. We can compute $E_P$ in $O(|E_P| + \log N)$ time from given $id(P)$. To do so, for each $q \in E_P$, we consider the set $\{e \mid e \in E_P, e \rightarrow q^c, c > 0\}$ of signatures, and maintain a balanced binary search tree for this set, where the comparisons are based on the values of $c$. Therefore Lemma holds.

B Appendix: LZ77 Factorization

B.1 Proof of Theorem

For integers $j, k$ with $1 \leq j \leq j + k - 1 \leq N$, let $Fst(j, k)$ be the function which returns the minimum integer $i$ such that $i < j$ and $T[i..i+k-1] = T[j..j+k-1]$, if it exists. Then, we get the following observation.

Observation 3 Let $f_1 \cdots f_z$ be the LZ77-Factorization of a string $T$. Given $f_1 \cdots f_{i-1}$, we can compute $f_i$ with $O(\log |f_i|)$ calls of $Fst(j, k)$ (by doubling the value of $k$, followed by a binary search), where $j = |f_1 \cdots f_{i-1}| + 1$. 

19
Therefore we can get Theorem 3 from the following lemma.

**Lemma 15.** There exists a data structure for $\mathcal{H}(\log w, w)$ with $F_P = (T\$)$ which computes $Fst(j, k)$ in $O(\log w \log k \log N (\log^* N)^2)$ time, occupies $O(w)$ space, and requires $O(w \log w \log N \log^* N)$ time to construct.

**Proof.** We define $e.min = \min vOcc(e, id(T)) + |e.left|$ for signature $e \in E_{xy}$ and $FstOcc(P, i)$ for a string $P$ and an integer $i$ as follows:

$$
FstOcc(P, i) = \min\{ e.min \mid (e, i) \in PrimOcc_P(P, i) \}
$$

Then we represent $Fst(j, k)$ by $FstOcc(P, i)$ as follows.

$$
Fst(j, k) = \min\{ FstOcc(T[j..j+k-1], i) - i \mid i \in \{1, \cdots, k-1\} \}
= \min\{ FstOcc(T[j..j+k-1], i) - i \mid i \in \mathcal{P} \},
$$

where $\mathcal{P}$ is the set of integers of Lemma 16 with $|\mathcal{P}| = O(\log |T[j..j+k-1]| \log^* N)$. Now we describe a data structure which returns $FstOcc(P, x)$ from a given $P$ and $x$. Recall $E_{xy}$ represented on a plane and the range reporting described in Lemma 6. We define the weight of signature $e$ to be $e.min$. Then $Fst(j, k)$ equals the minimum weight of the points in the given rectangle of Lemma 6. To retrieve only such a point, we can use the following result:

**Lemma 16 ([22]).** For $n$ points where each point has a weight on a plane, there exists a data structure which supports the query which returns the minimum weight of points in a given rectangle in $O(\log^2 n)$ time, occupies $O(n)$ space, and requires $O(n \log n)$ time to construct.

We here need $E_x$, $E_y$ and $e.min$ for every signature $e \in E_{xy}$ to construct the data structure of Lemma 16. We can compute every $e.min$ in $O(w)$ time using the DAG of $\mathcal{H}$ and $E_x$, $E_y$ in $O(\log w \log w \log^* N)$ time by the LCE algorithm. Since we can compute $FstOcc(T[j..j+k-1], i)$ in $O(\log w \log N \log^* N) + O(\log^2 w) = O(\log w \log N \log^* N)$ time, by Lemma 5, we can compute $Fst(j, k)$ in $O(\log w \log k \log N (\log^* N)^2)$ time. Therefore Lemma 15 holds.

We remark that we can similarly compute the Lempel-Ziv77 Factorization with self-reference of a text (defined below) in the same time and same working space.

**Definition 9 (Lempel-Ziv77 Factorization with self-reference [21]).** The Lempel-Ziv77 (LZ77) factorization of a string $s$ with self-references is a sequence $f_1, \ldots, f_k$ of non-empty substrings of $s$ such that $s = f_1 \cdots f_k$, $f_1 = s[1]$, and for $1 < i \leq k$, if the character $s[[f_{i-1}+1]]$ does not occur in $s[[f_1..f_{i-1}]]$, then $f_i = s[[f_{i-1}+1]]$, otherwise $f_i$ is the longest prefix of $f_i \cdots f_k$ which occurs at some position $p$, where $1 \leq p \leq |f_1 \cdots f_{i-1}|$. 

20
C  Theorem

C.1 Proof of Theorem (1)

Proof. Firstly, we prove that an $\mathcal{H}(\log w, w)$ signature dictionary for $\mathcal{F}_D = \langle T$ space can be constructed in $O(|T|)$ time and space. To show this, we use the two following lemmas.

Lemma 17. Given $E_{\text{block}}(\text{Pow}_{i-1}^T)$, for $0 < t \leq h$, we can compute $\text{Shrink}_{i}^T$ in $O((b - a) + |\text{Pow}_{i-1}^T|)$ time and $O(b - a + |\text{Pow}_{i-1}^T|)$ space, where $b$ is the maximum integer in $\text{Pow}_{i-1}^T$ and $a$ is the minimum integer in $\text{Pow}_{i-1}^T$.

Proof. We assign signatures to the strings in $\mathcal{F}_D$ incrementally, in the order they appear in the sequence $\mathcal{F}_D$. Hence, for each element of $\text{Pow}_{i-1}[i]$ of $\text{Pow}_{i-1}^T$, $\text{Pow}_{i-1}^T[i] - a$ fits in an entry of a bucket of size $b - a$. Since $|E_{\text{block}}(\text{Pow}_{i-1}^T)| \leq 4$, we can sort all the blocks of $\text{Eblock}(\text{Pow}_{i-1}^T)$ by bucket sort in $O((b - a) + |\text{Pow}_{i-1}^T|)$ time and $O(b - a + |\text{Pow}_{i-1}^T|)$ space. Since $\text{Sig}$ is an injection and since we process the levels in increasing order, for any two different levels $0 \leq t' < t \leq h$, no elements of $\text{Shrink}_{t'}^T$ appear in $\text{Shrink}_{t-1}^T$, and hence no elements of $\text{Pow}_{i-1}^T$ appear in $\text{Pow}_{i-1}^T$. Thus, we can determine a new signature for each block in $E_{\text{block}}(\text{Pow}_{i-1}^T)$, without searching existing signatures in the lower levels. This completes the proof.

Lemma 18. Given $E_{\text{pow}}(\text{Shrink}_{i}^T)$, we can compute $\text{Pow}_{i}^T$ in $O(x + (b - a) + |E_{\text{pow}}(\text{Shrink}_{i}^T)|)$ time and $O(x + (b - a) + |E_{\text{pow}}(\text{Shrink}_{i}^T)|)$ space, where $x$ is the maximum length of runs in $E_{\text{pow}}(\text{Shrink}_{i}^T)$, $b$ is the maximum integer in $\text{Pow}_{i-1}^T$, and $a$ is the minimum integer in $\text{Pow}_{i-1}^T$.

Proof. We first sort all the elements of $E_{\text{pow}}(\text{Shrink}_{i}^T)$ by bucket sort in $O(b - a + |E_{\text{pow}}(\text{Shrink}_{i}^T)|)$ time and space, ignoring the powers of runs. Then, for each integer $r$ appearing in $\text{Shrink}_{i}^T$, we sort the runs of $r$’s by bucket sort with a bucket of size $x$. This takes a total of $O(x + |E_{\text{pow}}(\text{Shrink}_{i}^T)|)$ time and space for all integers appearing in $\text{Shrink}_{i}^T$. The rest is the same as the proof of Lemma

Since every node of the signature tree of $T$ has at least two children, the size of the signature tree is $O(|T|)$. Also, since the signatures are already sorted, we can construct a balanced search tree for signatures in $O(|T|)$ time. This balanced search tree is used to search for $\text{Sig}$ in $O(\log w)$ time. Thus, by Lemmas and we can compute $id(T)$ and hence an $\mathcal{H}(\log w, w)$ signature dictionary for $\mathcal{F}_D = \langle T$ time and space.

Secondly, we prove that an $\mathcal{H}(\log w, w)$ signature dictionary for $\mathcal{F}_D = \langle T$ can be constructed in $O(|T| \log w)$ time and $O(w)$ working space. To show this, we use the following lemma.

Lemma 19. Using an $\mathcal{H}(\log w, w)$ signature dictionary for $\mathcal{F}_D$, we can support $\text{STRING}(s)$ in $O(|s| \log w)$ time and $O(w)$ working space.
Proof. Recall that we can compute STRING(s) in $O(|s| \log w)$ time and $O(|s|)$ working space by Lemma\textsuperscript{5}. Let $B$ a positive integer, which will be specified later. We conduct the following sequence of operations: STRING($s_1$), STRING($s_2$), CONCAT(id($u_1$), id($s_2$), 1), STRING($s_3$), CONCAT(id($u_2$), id($s_3$), 1), \ldots, CONCAT(id($u_{i-1}$), id($s_i$), 1), where $s_i = s[(i-1)B + 1 \ldots iB]$ and $u_i = s_1 \ldots s_i$. Finally, we get id(T$)$ in $O(n \log w \log N \log^a M + |s| \log w)$ time and $O(w + B)$ space, where $N = \log |s|$. By choosing $B = \log N \log^a M$, our proof is finished.

Therefore we can construct $\mathcal{H}(\log w, w)$ in $O(|T| \log w)$ time and $O(w)$ working space. \hfill \Box

C.2 Proof of Theorem 2(2)

Proof. We consider $\mathcal{H}(\log w, w)$ for $\mathcal{F}_{D_i} = \langle f_1, \ldots, f_i, \$ \rangle$ for $1 \leq i < z$. We show that we can update $\mathcal{H}(\log w, w)$ for $\mathcal{F}_{D_1}$ to that for $\mathcal{F}_{D_{i+1}}$ in $O(\log w \log N \log^a M)$ time. If $|f_{i+1}| = 1$ then $f_{i+1}$ is a character. Hence we do the operations of CHAR($f_{i+1}$) and CONCAT(id($f_1..f_i\$), id($f_{i+1}\$), 1). If $|f_{i+1}| > 1$, there exists two integers $j, k$ which satisfies $f_{i+1} = (f_1..f_i)[j..j+k-1]$. Therefore we do the operation of CONCAT(id($f_1..f_i\$), id($f_{i+1}\$), 1). Note that we can compute id($f_{i+1}\$) by two SPLIT operations in $O(\log w \log N \log^a M)$ time by Lemma\textsuperscript{5}. Therefore Theorem 2(2) holds. \hfill \Box

C.3 Proof of Theorem 2(3)

Definition 10. Given an SLP $S$, for $X_i \in S$ we abbreviate Shrink$^{X_i}_t$ = Shrink$^{\text{val}(X_i)}_t$, Pow$^{X_i}_t$ = Pow$^t_{\text{val}(X_i)}$, XShrink$^{X_i}_t$ = XShrink$^{\text{val}(X_i)}_t$, and XPow$^{X_i}_t$ = XPow$^t_{\text{val}(X_i)}$. Moreover, we abbreviate $\omega^L_{X_i,t} = \omega^L_{\text{val}(X_i),t}$, $\omega^R_{X_i,t} = \omega^R_{\text{val}(X_i),t}$, $\Delta^L_{X_i,t} = \Delta^L_{\text{val}(X_i),t}$, $\Delta^R_{X_i,t} = \Delta^R_{\text{val}(X_i),t}$, LS$\text{Shrink}^{X_i}_t$ = $\text{LS}\text{Shrink}^{\text{val}(X_i)}_t$, R$\text{Shrink}^{X_i}_t$ = R$\text{Shrink}^{\text{val}(X_i)}_t$, LPow$^{X_i}_t$ = LPow$^t_{\text{val}(X_i)}$, RPow$^{X_i}_t$ = RPow$^t_{\text{val}(X_i)}$, and $\hat{h}^{X_i}_t = \hat{h}^{\text{val}(X_i)}_t$.

We begin with the following preliminary result:

Lemma 20. Given an SLP $S$ of size $n$ representing a string of length $N$, we can construct an $\mathcal{H}(\log w, 1, w)$ signature dictionary for $\mathcal{F}_D = \langle \text{val}(X_1)\$, \ldots, \text{val}(X_n)\rangle$ in $O(n \log w \log N \log^a M)$ time and $O(w)$ space, where $w = O(n \log N \log^a M)$.

Proof. We process each $X_i$ in increasing order of $i$. If $X_i \to c \in \Sigma$, then we conduct CHAR(c). If $X_i \to X_tX_r$, then we conduct CONCAT(id($X_t$), id($X_r$)). Therefore, Lemma 20 holds by Lemma 2 and 5. \hfill \Box

As above, we can construct $\mathcal{H}(\log w, w)$ from SLP $S$ in $O(n \log w \log N \log^a M)$ time. Lemma 20 is, however, slower than Theorem 2(3). We roughly describe our ideas for speed-up. The first idea is that we use the property of SLP.
Observation 4. For any production $X_i \rightarrow X_t X_r$ and an integer $t$, if $|XShrink_t^{X_i}| > 0$ and $|XShrink_r^{X_r}| > 0$, then there exists a signature sequence $u_t^{X_i}$ which satisfies:

$$Shrink_t^{X_i} = LShrink_t^{X_i} XShrink_t^{X_r} u_t^{X_i} XShrink_t^{X_i} RShrink_t^{X_r}$$

Observation 4 implies that (1) we can construct $Shrink_t^{X_i}$ by computing $u_t^{X_i}$ if we have already constructed $Shrink_t^{X_i}$ and $Shrink_t^{X_r}$, and (2) If there exist $u_t^{X_i}, \ldots, u_t^{X_n}$, we can construct $Shrink_t^{X_i}, \ldots, Shrink_t^{X_n}$ from $u_t^{X_i}, \ldots, u_t^{X_n}$. Hence, we compute $Shrink_0^{X_0}, \ldots, Shrink_0^{X_n}$, $Pow_0^{X_1}, \ldots, Pow_0^{X_n}$, $Shrink_1^{X_1}, \ldots, Shrink_1^{X_n}, \ldots$ in a bottom-up manner. Furthermore we determine assignments in $Shrink_t^{X_1}, \ldots, Shrink_t^{X_n}$ and $Pow_t^{X_1}, \ldots, Pow_t^{X_n}$ by bucket sort and an efficient sort algorithm as in the proof of Lemmas 17 and 18 respectively.

Definition 11. For $X_i \in \mathcal{S}$, we define some signature sequences as follows.

$$LXShrink_t^{X_i} = LShrink_t^{X_i} XShrink_t^{X_i};$$

$$XRShrink_t^{X_i} = XShrink_t^{X_i} RShrink_t^{X_i};$$

$$LXPow_t^{X_i} = \begin{cases} 
Sig^+(\text{Epow}(LXShrink_t^{X_i} | \text{LXShrink}_t^{X_i} - \omega_L^{(X_i)})]) & \text{for } 0 \leq t < \hat{h}^{X_i}, \\
\epsilon & \text{for } t \geq \hat{h}^{X_i},
\end{cases}$$

$$XRPow_t^{X_i} = \begin{cases} 
Sig^+(\text{Epow}(XRShrink_t^{X_i} | \omega_L^{(X_i,t)}) + 1..)) & \text{for } 0 \leq t < \hat{h}^{X_i}, \\
\epsilon & \text{for } t \geq \hat{h}^{X_i},
\end{cases}$$

$$APow_t^{X_i} = LXPow_t^{X_i} | \text{LXShrink}_t^{X_i} - (\Delta_L + \Delta_R^{(X_i,t)}) + 1..]$$

$$BPow_t^{X_i} = XRPow_t^{X_i} | (\Delta_L^{(X_i,t)} + \Delta_R)_t$$

$$AShrink_t^{X_i} = \begin{cases} 
LXShrink_t^{X_i} | \text{LXShrink}_t^{X_i} - \omega_R^{(X_i,t)} + 1..) & \text{for } 0 \leq t < \hat{h}^{X_i}, \\
LXShrink_t^{X_i} & \text{for } t \geq \hat{h}^{X_i},
\end{cases}$$

$$BShrink_t^{X_i} = \begin{cases} 
XRShrink_t^{X_i} | .\omega_L^{(X_i,t)} & \text{for } 0 \leq t < \hat{h}^{X_i}, \\
XRShrink_t^{X_i} & \text{for } t \geq \hat{h}^{X_i},
\end{cases}$$

Definition 12. We define $CShrink_t^{X_i}$, $CPow_t^{X_i}$ for any $X_i \in \mathcal{S}$ as follows. If $X_i \rightarrow a \in \Sigma$, then

$$CShrink_t^{X_i} = \text{Sig}(a) \text{ for } t = 0$$

$$CPow_t^{X_i} = \text{Sig}^+(\text{Epow}(CShrink_t^{X_i}))$$

If $X_i \rightarrow X_t X_r$, then

$$CShrink_t^{X_i} = \begin{cases} 
\epsilon & \text{for } t = 0, \\
CEncb(\text{APow}_{t-1}^{X_i} \text{CPow}_t^{X_i} \text{BPow}_t^{X_r}) & \text{for } 0 < t < h_x^{X_t}, h_x^{X_r}, \\
LENcb(\text{CPow}_{t-1}^{X_i} \text{BPow}_t^{X_r}) & \text{for } h_x^{X_t} \leq t < h_x^{X_r}, \\
RNencb(\text{APow}_{t-1}^{X_i} \text{CPow}_t^{X_r}) & \text{for } h_x^{X_r} \leq t < h_x^{X_t}, \\
Eblock(\text{CPow}_{t-1}^{X_i}) & \text{for } h_x^{X_t}, h_x^{X_r} \leq t \leq h_x^{X_i},
\end{cases}$$

$$CPow_t^{X_i} = \text{Sig}^+(\text{Epow}(AShrink_t^{X_i} CShrink_t^{X_i} BShrink_t^{X_r})) \text{ for } 0 \leq t < h_x^{X_i},$$
Observation 5 For any \( X_i \in S \) and \( 0 < t \leq \hat{h}^{X_i} \), then
\[
\text{LXShrink}_t^{X_i} = \text{Sig}^+(L\text{Encb}(\text{LPow}_{t-1}^{X_i}))
\]
\[
\text{XRShrink}_t^{X_i} = \text{Sig}^+(R\text{Encb}(\text{XR Pow}_{t-1}^{X_i}))
\]

Lemma 21. For any \( X_i \to X_tX_r \), let \( h \) is a minimal integer \( |\text{Pow}_{t}^{X_i}| = 1 \). Then for \( 0 \leq t \leq h \),
\[
\text{Shrink}_t^{X_i} = \text{LXShrink}_t^{X_i} \text{CShrink}_t^{X_i} \text{XRShrink}_t^{X_r}
\]
\[
\text{Pow}_t^{X_i} = \text{LXPow}_t^{X_i} \text{CPow}_t^{X_i} \text{XR Pow}_t^{X_r}
\]

Proof. These are true by the definitions. See also Fig. 3.

\[
\begin{array}{c|c|c}
\text{LXShrink}_t^{X_l} & \text{CShrink}_t^{X_l} & \text{XRShrink}_t^{X_r} \\
\hline
\text{LXPow}_{t-1}^{X_l} & \text{CPow}_{t-1}^{X_l} & \text{XR Pow}_{t-1}^{X_r} \\
\end{array}
\]

Fig. 3. The relation between \( \text{CShrink}_t^{X_l} \) and \( \text{CPow}_{t-1}^{X_l} \). \( \text{XR Pow}_{t-1}^{X_r} \) is encoded as a block of \( \text{CShrink}_t^{X_l} \). \( \text{LXPow}_{t-1}^{X_l} || \text{LXPow}_{t-1}^{X_l} - \Delta_r^{(X_r,t-1)} + 1 .. \) is encoded as a block of \( \text{CShrink}_t^{X_l} \).

Observation 6 For every variable \( X_i \) of SLP \( S \) and every integer \( t \geq 0 \), \( |\text{CPow}_t^{X_i}|, |\text{Pow}_t^{X_i}|, |\text{APow}_t^{X_i}|, |\text{BPow}_t^{X_i}|, |\text{LXPow}_t^{X_i}|, |\text{XR Pow}_t^{X_i}| \) are all bounded by \( O(\log^* M) \).

Definition 13. Let \( \text{LShrink}_t^{X_i} = \text{Sig}^+(\text{LShrink}^{X_i}) \) for any \( X_i \in S \). We define \( \text{RShrink}_t^{X_i} \), \( \text{LPow}_t^{X_i} \), \( \text{RPow}_t^{X_i} \), \( \text{CShrink}_t^{X_i} \) and \( \text{CPow}_t^{X_i} \) in similar ways.

Definition 14. Let \( S \) denote an SLP of size \( n \) and \( \Sigma \) denote an ordered alphabet \( \{a_1, \ldots, a_{|\Sigma|} \} \). Then we define \( \text{CPow}_S^t = (\text{CPow}_t^{X_1}, \ldots, \text{CPow}_t^{X_n}) \). Similarly, we define \( \text{CShrink}_S^t, \text{LShrink}_S^t, \text{RShrink}_S^t, \text{LPow}_S^t, \text{RPow}_S^t, \text{CPow}_S^t, \text{CShrink}_S^t, \text{LShrink}_S^t, \text{RShrink}_S^t, \text{LPow}_S^t, \text{RPow}_S^t \). Furthermore, we define \( \text{Shrink}_S^t = (\text{Sig}(a_1), \ldots, \text{Sig}(a_{|\Sigma|})) \).

We here describe two algorithms: One is Algorithm which builds \( (\text{CPow}_S^t, \text{LPow}_S^t, \text{RPow}_S^t) \) from \( (\text{CPow}_{t-1}^S, \text{LPow}_{t-1}^S, \text{RPow}_{t-1}^S) \), and the other is Algorithm which builds \( (\text{Pow}_t^{X_1}, \ldots, \text{Pow}_t^{X_n}) \) from \( (\text{CPow}_t^S, \text{LPow}_t^S, \text{RPow}_t^S) \). Since
we can construct \((CPow^S_0, LPow^S_0, RPow^S_0)\) from SLP \(S\) by Algorithm \(2\) we can construct \((Pow^X_1, \ldots, Pow^X_n)\) for every level \(t \geq 0\) using these algorithms. As soon as we find \(Pow^X_i\), such that \(|Pow^X_i| = 1\) for any \(X_i \in S\), then we determine \(id(X_i)\). This implies \(t = O(\log N)\). Finally, we can get an \(H_D\) signature dictionary for \(F_D = \langle val(X_1)\$, \ldots, val(X_n)\$\).

We remark that Algorithm \(1\) shows only a basic idea to compute \((Pow^X_1, \ldots, Pow^X_n)\), and hence requires \(O(N)\) time. In what follows, we will use the improved algorithm in Algorithm \(2\) See the proof of Lemma \(22\).

---

**Algorithm 1**: Basic algorithm to compute \(Pow^S_i\)

**Input**: \((CPow^S_i, LPow^S_i, RPow^S_i)\)

**Output**: \((Pow^X_1, \ldots, Pow^X_n)\)

1. foreach \(X_i \in S\) do
   2. if \(X_i \rightarrow X_j X_k\) then
      3. \(Pow^X_i \leftarrow LXPow^X_i CPow^X_i XRPow^X_i;\)
      4. \(LPow^X_i \leftarrow \| \cdot [Pow^X_i] - \|RPow^X_i|];\)
      5. \(XRPow^X_i \leftarrow \| \cdot LPow^X_i| + 1.];\)
   3. else if \(X_i \rightarrow a \in \Sigma\) then
      4. \(Pow^X_i \leftarrow \);
      5. \(LPow^X_i \leftarrow \);
      6. \(XRPow^X_i \leftarrow \);
3. return \((Pow^X_1, \ldots, Pow^X_n)\);

---

**Algorithm 2**: Algorithm to compute \((CPow^S_i, LPow^S_i, RPow^S_i)\)

**Input**: \((CPow^S_{i-1}, LPow^S_{i-1}, RPow^S_{i-1})\) or SLP \(S\)

**Output**: \((CPow^S_i, LPow^S_i, RPow^S_i)\)

1. if \(t = 0\) then
   2. Compute \(Shrink^S_i\) from SLP \(S\);
   3. Compute \((CPow^S_i, LPow^S_i, RPow^S_i)\) from \(Shrink^S_i\);
5. else
   6. Compute \((CShrink^S_i, LShrink^S_i, RShrink^S_i)\) from \((CPow^S_{i-1}, LPow^S_{i-1}, RPow^S_{i-1})\);
   7. Compute \((CShrink^S_i, LShrink^S_i, RShrink^S_i)\) from \((CShrink^S_i, LShrink^S_i, RShrink^S_i)\);
   8. Compute \((CPow^S_i, LPow^S_i, RPow^S_i)\) from \((CShrink^S_i, LShrink^S_i, RShrink^S_i)\);
9. return \((CPow^S_i, LPow^S_i, RPow^S_i)\);
Lemma 22. Algorithm can be implemented so that it runs in $O(n \log n \log^* M)$ time and $O(n \log^* M)$ working space.

Proof. Line 2 takes $O(n \log \log n \log^* M)$ time because $|\Sigma| \leq n$. At Line 3, we can compute $(\text{Shrink}^1, \ldots, \text{Shrink}^N)$ from SLP $S$ (see Algorithm 3). By modifying Algorithm 3 accordingly, we can conduct the procedures of Line 3 in $O(n \log^* M)$ time because the output size is $O(n \log^* M)$ by Observation 6. Similarly, we can conduct Lines 5 and 7 in $O(n \log^* M)$ time by modifying Algorithms 1 and 3 respectively. At Lines 6 and 8 we use the following lemmas.

Lemma 23. 1. Let $r_{\text{mag}}$ and $r_{\text{min}}$ be the maximum and minimum integers in $(\text{CShrink}^1, \text{LShrink}^1, \text{RShrink}^1)$, respectively. Then we can bind $r_{\text{max}} - r_{\text{min}} = O(n \log^* M)$.
2. Let $r_{\text{max}}$ and $r_{\text{min}}$ be the maximum and minimum integers in $(\text{CPow}^1, \text{LPow}^1, \text{RPow}^1)$, respectively. Then we can bind $r_{\text{max}} - r_{\text{min}} = O(n \log^* M)$.

Proof. Recall Lemmas 17 and 18. By Observation 6, Lemma 23 holds.

Lemma 24. 1. Given $(\text{CShrink}^1, \text{LShrink}^1, \text{RShrink}^1)$, we can determine $(\text{CShrink}^1, \text{LShrink}^1, \text{RShrink}^1)$ in $O(n \log^* M)$ time and working space.
2. Given $(\text{CPow}^1, \text{LPow}^1, \text{RPow}^1)$, we can determine $(\text{CPow}^1, \text{LPow}^1, \text{RPow}^1)$ in $O(n \log^* M)$ time and working space.

Proof. (1) This lemma immediately follows from Lemma 17 and 23. (2) We apply Lemma 17 but here we use, in place of bucket sort, the sorting algorithm of 26 on the RAM model which sorts $n$ integers over $[1..N]$ in $O(n \log \log n)$ time with $O(n)$ space.

Hence, Lemma 22 holds.

By Lemma 22, we can create a signature dictionary $D$ for $F_D = (\text{val}(X_1), \ldots, \text{val}(X_n))$ in $O(n \log \log n \log N \log^* M)$ time. Note that $|D| = O(n \log N \log^* M)$ by Observation 6. Since $D$ is sorted, we can construct $H(w, w)$ from $D$ in linear time as the proof Theorem 2. Therefore Theorem 2 (3) holds.

D Appendix: Applications

D.1 Proof of Lemma 10

Proof. Lemma 25. Let $s$ denote a string and $k$ denote an integer such that $1 \leq k \leq |s_1|$ and $P = s[.k]$. Then

$$X\text{Shrink}^t_s = X\text{Shrink}^t_P X\text{Shrink}^t_s |X\text{Shrink}^t_P| + 1\ldots \text{ for } 0 \leq t < \hat{t}^P$$

$$X\text{Pow}^t_s = X\text{Pow}^t_P X\text{Pow}^t_s |X\text{Pow}^t_P| + 1\ldots \text{ for } 0 \leq t < \hat{t}^P$$

Proof. The lemma holds by the definitions of $X\text{Shrink}$ and $X\text{Pow}$.

This means that $\text{val}(x) = s_1[i..i+k-1]$ and $|\text{Epow}(x)| \leq |\text{Epow}(\text{Uniq}(s_1[i..i+k-1]))|$, where $x$ is the signature sequence maintained by Algorithm 3. Hence the total running time is $O((\log |s_1| + \log |s_2|) \log^* M)$. Similarly, we can compute $\text{LCE}(s_1^P, s_2^R, i, j)$ in $O((\log |s_1| + \log |s_2|) \log^* M)$ time.
Algorithm 3: Algorithm to compute $\text{Shrink}_i^S$

**Input:** ($C\text{Shrink}_i^S$, $L\text{Shrink}_i^S$, $R\text{Shrink}_i^S$) or SLP $S$

**Output:** $\text{Shrink}_i^S$

1. foreach $X_i \in S$
   2. if $t = 0$
      3. if $X_i \rightarrow X_iX_i$ then
         4. $\text{Shrink}_i^{X_i} \leftarrow \text{Shrink}_i^{X_i} \times \text{Shrink}_i^{X_i}$;
      5. else if $X_i \rightarrow a \in \Sigma$ then
         6. $\text{Shrink}_i^{X_i} \leftarrow \text{Sig}(a)$;
      7. else
         8. if $X_i \rightarrow X_iX_i$ then
            9. $\text{Shrink}_i^{X_i} \leftarrow LX\text{Shrink}_i^{X_i} \times C\text{Shrink}_i^{X_i} \times XR\text{Shrink}_i^{X_i}$;
            10. $LX\text{Shrink}_i^{X_i} \leftarrow \text{Shrink}_i^{X_i} \times \lvert \text{Shrink}_i^{X_i} \rvert - \lvert R\text{Shrink}_i^{X_i} \rvert$;
            11. $XR\text{Shrink}_i^{X_i} \leftarrow \text{Shrink}_i^{X_i} \times \lvert L\text{Shrink}_i^{X_i} \rvert + 1$;
         12. else if $X_i \rightarrow a \in \Sigma$ then
            13. $\text{Shrink}_i^{X_i} \leftarrow C\text{Shrink}_i^{X_i}$;
            14. $LX\text{Shrink}_i^{X_i} \leftarrow \epsilon$;
            15. $XR\text{Shrink}_i^{X_i} \leftarrow \epsilon$;
      16. return $\text{Shrink}_i^S$;

D.2 Proof of Theorem 4

**Proof of Theorem 4 (1)**

Proof. For any signature $e \in \mathcal{E}_D$ such that $e \rightarrow e_1e_2e_3$, we can easily translate $e$ to a production of SLP because the assignment is a pair of signatures, like the right-hand side of the production rules of SLPs. For any signature $e \in \mathcal{E}_D$ such that $e \rightarrow q^k$, to translate $e$ to productions of SLP, we use the following lemma.

**Lemma 26.** Given a signature $e$ such that $e \rightarrow q^k$ and a variable $X_q$ such that $\text{val}(X_q) = \text{val}(q)$, we can create a variable $X_e$ such that $\text{val}(X_e) = \text{val}(e)$ by introducing $O(\log k)$ new productions in $O(\log k)$ time.

Proof. If $k$ is even and $k > 2$ then we compute $X_{k/2}$ such that $\text{val}(X_{k/2}) = \text{val}(q^{k/2})$ recursively and return a new production $X_k \rightarrow X_{k/2}X_{k/2}$. If $k$ is odd and $k > 2$ then we compute $X_{(k-1)/2}$ such that $\text{val}(X_{(k-1)/2}) = \text{val}(q^{(k-1)/2})$ recursively and return a new production $X_k \rightarrow X_{(k-1)/2}X_q$. If $k = 2$ then we return a new production $X_k \rightarrow X_qX_q$. Hence Lemma 26 holds.

Therefore we can compute $S$ in $O(w \log N)$ time.

**Proof of Theorem 4 (2)**
Algorithm 4: Computing \( k = LCE(s_1, s_2, i, j) \)

**Input**: Two signature sequences \( u = \text{Uniq}(s_1[i..]) \), \( v = \text{Uniq}(s_2[j..]) \)

**Output**: \( k \)

1. \( x \leftarrow \varepsilon; \)
2. while \( u \neq \varepsilon \) and \( v \neq \varepsilon \) do
3.   if \( u[1] = v[1] \) then
4.     \( p \leftarrow \text{LCP}(u, v); \)
5.     \( u \leftarrow u[p + 1..]; \)
6.     \( v \leftarrow v[p + 1..]; \)
7.     \( x \leftarrow xp; \)
8.   else
9.     if \( |\text{val}(u[1])| = 1 \) and \( |\text{val}(v[1])| = 1 \) then
10.    Break;
11.   else if \( |\text{val}(u[1])| \geq |\text{val}(v[1])| \) then
12.     \( u \leftarrow \text{Assgn}(u[1])u[2..]; \)
13.   else
14.     \( v \leftarrow \text{Assgn}(v[1])v[2..]; \)
15. return \( |\text{val}(x)|; \)

Proof. Note that the number of created or removed signatures in \( E_D \) is bounded \( O(\log N \log^* M + d) \) by Lemma 2. For each of the removed signatures, we remove the corresponding production from \( S \). For each of created signatures, we create the corresponding production and add it to \( S \) as in the proof of (1). Therefore Theorem 4 holds.

D.3 Proof of Theorem 5

We use the following known result.

**Lemma 27 ([1])**. Using an \( \mathcal{H}(\log w, w) \) signature dictionary for \( F_D \), we can support

- \( \text{LCP}(s_1, s_2) \) in \( O(\log |s_1s_2|) \) time,
- \( \text{LCS}(s_1, s_2) \) in \( O(\log |s_1s_2| \log^* M) \) time

where \( s_1, s_2 \in F_D \).

We are ready to prove Theorem 5.

Proof. The first result immediately follows from Lemma 27 and Theorem 2. To speed-up query times for \( \text{LCP} \) and \( \text{LCS} \), we sort variables in lexicographical order in \( O(n \log n \log N) \) time by \( \text{LCP} \) query and a standard comparison-based sorting. Constant-time \( \text{LCP} \) queries are then possible by using a constant-time \( \text{RMQ} \) data structure [23] on the sequence of the lcp values. \( \text{LCS} \) queries can be supported similarly.
D.4 Proof of Theorem 9

Proof. We can compute \( H(\log w, w') \) for \( \mathcal{F}_D = \langle \text{val}(X_1)$, \cdots, \text{val}(X_n) \rangle \) in \( O(n \log \log n \log^* N) \) time and \( O(w') \) working space using Theorem 2, where \( w' = O(n \log N \log^* N) \). Notice that each variable of the SLP appears at least once in the derivation tree \( T_n \) of the last variable \( X_n \) representing the string \( T \). Hence, if we store an occurrence of each variable \( X_i \) in \( T_n \) and \( |\text{val}(X_i)| \), we can reduce any LCE query on two variables to an LCE query on two positions of \( \text{val}(X_n) = T \). In so doing, for all \( 1 \leq i \leq n \) we first compute \(|\text{val}(X_i)|\) and then compute the leftmost occurrence \( t_i \) of \( X_i \) in \( T_n \), spending \( O(n) \) total time and space. Then, we remove all signatures from \( \mathcal{E}_D \) that do not appear in the signature encoding of the last variable \( X_n \), in \( O(w') = O(n \log N \log^* N) \) time. It follows from Lemma 8 that the number of remaining signatures is \( O(z \log N \log^* N) \), and hence the total space is \( O(z \log N \log^* N + n) \). By Lemma 10 each LCE query can be supported in \( O(\log N \log^* N) \) time. Since \( z \leq n \) [27], the total preprocessing time is \( O(n \log \log n \log N \log^* N) \) and working space is \( O(n \log N \log^* N) \).

\( \Box \)

D.5 Proof of Theorem 7

Proof. For a given SLP of size \( n \) representing a string of length \( N \), let \( P(n, N) \), \( S(n, N) \), and \( E(n, N) \) be the preprocessing time and space requirement for an LCE data structure on SLP variables, and each LCE query time, respectively.

Matsubara et al. [12] showed that we can compute an \( O(n \log N) \)-size representation of all palindromes in the string in \( O(P(n, N) + E(n, N) \cdot n \log N) \) time and \( O(n \log N + S(n, N)) \) space. Hence, using Theorem 6 we can find all palindromes in the string in \( O(n \log \log n \log N \log^* N + n \log \log N \log^* N) = O(n \log^2 N \log^* N) \) time and \( O(n \log N \log^* N) \) space.

\( \Box \)

D.6 Proof of Theorem 8

Proof. It is shown in [8] that we can compute the Lyndon factorization of the string in \( O(P(n, N) + E(n, N) \cdot n \log n) \) time using \( O(n^2 + S(n, N)) \) space. Hence, using Theorem 6 we can compute the Lyndon factorization of the string in \( O(n \log \log n \log N \log^* N + n \log \log N \log^* N) = O(n \log n \log N \log^* N) \) time. We remark that since \( m \leq n \) due to [8], the output size \( m \) is omitted in the total time complexity.

\( \Box \)

D.7 Proof of Theorem 9

Proof. In the preprocessing phase, we construct an \( \mathcal{H}(\log n, n \log N \log^* M) \) signature dictionary for the string sequence \( \mathcal{F}_D \) of DSLP \( \langle \mathcal{G}, m \rangle \) using Theorem 2 spending \( O(n \log \log n \log N \log^* M) \) time. Next we construct a compacted trie of size \( O(m) \) that represents the \( m \) patterns, which we denote by \( PTree \) (pattern tree). Formally, each non-root node of \( PTree \) represents either a pattern or the longest common prefix of some pair of patterns. \( PTree \) can be built by using
LCP of Theorem 5 in $O(m \log m \log N)$ time. We let each node have its string depth, and the pointer to its deepest ancestor node that represents a pattern if such exists. Further, we augment $PTree$ with a data structure for level ancestor queries so that we can locate any prefix of any pattern, designated by a pattern and length, in $PTree$ in $O(\log m)$ time by locating the string depth by binary search on the path from the root to the node representing the pattern. Supposing that we know the longest prefix of $T[i..|T|]$ that is also a prefix of one of the patterns, which we call the max-prefix for $i$, $PTree$ allows us to output $occ$ patterns occurring at position $i$ in $O(\log m + occ_i)$ time. Hence, the pattern matching problem reduces to computing the max-prefix for every position.

In the pattern matching phase, our algorithm processes $T$ in a streaming fashion, i.e., each character is processed in increasing order and discarded before taking the next character. Just before processing $T[j + 1]$, the algorithm maintains a pair of signature $p$ and integer $l$ such that $val(p)[1..l]$ is the longest suffix of $T[1..j]$ that is also a prefix of one of the patterns. When $T[j + 1]$ comes, we search for the smallest position $i \in \{j - l + 1, \ldots, j + 1\}$ such that there is a pattern whose prefix is $T[i..j + 1]$. For each $i \in \{j - l + 1, \ldots, j + 1\}$ in increasing order, we check if there exists a pattern whose prefix is $T[i..j + 1]$ by binary search on a sorted list of $m$ patterns. Since $T[i..j] = val(p)[i - j + 1..l]$, LCE with $p$ can be used for comparing a pattern prefix and $T[i..j + 1]$ (except for the last character $T[j + 1]$), and hence, the binary search is conducted in $O(\log m \log N \log^* M)$ time. For each $i$, if there is no pattern whose prefix is $T[i..j + 1]$, we actually have computed the max-prefix for $i$, and then we output the occurrences of patterns at $i$. The time complexity is dominated by the binary search, which takes place $O(|T|)$ times in total. Therefore, the algorithm runs in $O(|T| \log m \log N \log^* N + occ)$ time.

By the way, one might want to know occurrences of patterns as soon as they appear as Aho-Corasick automata do it by reporting the occurrences of the patterns by their ending positions. Our algorithm described above can be modified to support it without changing the time and space complexities. In the preprocessing phase, we additionally compute $RPTree$ (reversed pattern tree), which is analogue to $PTree$ but defined on the reversed strings of the patterns, i.e., $RPTree$ is the compacted trie of size $O(m)$ that represents the reversed strings of the $m$ patterns. Let $T[i..j]$ be the longest suffix of $T[1..j]$ that is also a prefix of one of the patterns. A suffix $T[i'..j]$ of $T[i..j]$ is called the max-suffix for $j$ if it is the longest suffix of $T[i..j]$ that is also a suffix of one of the patterns. Supposing that we know the max-suffix for $j$, $RPTree$ allows us to output $eocc_j$ patterns occurring with ending position $j$ in $O(\log m + eocc_j)$ time. Given a pair of signature $p$ and integer $l$ such that $T[i..j] = val(p)[1..l]$, the max-suffix for $j$ can be computed in $O(\log m \log N \log^* N)$ time by binary search on a list of $m$ patterns sorted by their “reversed” strings since each comparison can be done by “leftward” LCE with $p$. Except that we compute the max-suffix for every position and output the patterns ending at each position, everything else is the same as the previous algorithm, and hence, the time and space complexities are not changed.
Appendix: Examples

In this appendix, we supply some examples.

Example 1 (Sequence $\mathcal{O}$ of operations and sequence $\mathcal{F}_D$ of strings). Let us begin with an empty sequence $\mathcal{F}_D = \langle \cdot \rangle$ of strings. After conducting the sequence $\mathcal{O} = \langle \text{CHAR}(a), \text{CHAR}(b), \text{CONCAT}(s_1, s_2, 0), \text{CONCAT}(s_3, s_3, 0), \text{CONCAT}(s_4, s_4, 0), \text{SPLIT}(s_5, 3, 1) \rangle$ of operations, we get $\mathcal{F}_D = \langle s_1, \ldots, s_7 \rangle = \langle a\$,\$ b\$, \$ ab\$, \$ ab\$, aba\$ bab\$ \rangle$, where the marker \$ is attached to each string in $\mathcal{F}_D$ for convenience. Note that since the third argument of $\text{CONCAT}(s_4, s_4, 0)$ is 0, the string $s_4$ remains in $\mathcal{F}_D$. On the other hand, since the third argument of $\text{SPLIT}(s_5, 3, 1)$ is 1, the string $abababab$ has been deleted from $\mathcal{F}_D$.

Example 2 (Encblock$_d(p)$). Let $\log^* W = 2$, and then $\Delta_L = 8$, $\Delta_R = 4$. If $p = 1, 2, 3, 2, 5, 7, 6, 4, 3, 4, 3, 1, 2, 3, 4, 5$ and $d = 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0$, then Encblock$_d(p) = (1, 2, 3), (2, 5), (7, 6, 4), (3, 4, 3, 4), (1, 2), (3, 4, 5)$.

Example 3 (LEncb$_d(p)$, REncb$_d(p)$, and REncb$_d(p)$). On the example shown in Example 2, LEncb$_d(p) = (1, 2, 3), (2, 5), (7, 6, 4), (3, 4, 3, 4), CEncb$_d(p) = (3, 4, 3, 4)$, and REncb$_d(p) = (3, 4, 3, 4), (1, 2), (3, 4, 5)$.

Example 4 (Epow($s$)). For string $s = aabbbbab$, Epow($s$) = $a^2b^5a^1b^2$ and $|\text{Epow}(s)| = 4$ since $\text{Epow}(s)$ contains 4 maximal character runs.

Example 5 (Signature encoding). Let $\Sigma = \{A, B, C, \$\}$. With the signature dictionary $\mathcal{D} = \{1 \rightarrow C, 2 \rightarrow B, 3 \rightarrow A, 4 \rightarrow 1^4, 5 \rightarrow (1, 2), 6 \rightarrow (3, 1), 7 \rightarrow (2, 3), 8 \rightarrow (7, 2), 9 \rightarrow (3, 2), 10 \rightarrow (9, 3), 11 \rightarrow (4, 2), 12 \rightarrow (7, 1), 13 \rightarrow 7^6, 14 \rightarrow 12^2, 15 \rightarrow (5, 6), 16 \rightarrow (15, 8), 17 \rightarrow (10, 13), 18 \rightarrow (11, 9), 19 \rightarrow (15, 14), 20 \rightarrow (16, 17), 21 \rightarrow (20, 18), 22 \rightarrow (17, 19), 23 \rightarrow (21, 22), 24 \rightarrow 8, 25 \rightarrow (23, 24)\}$, we get $\mathcal{F}_D = \langle s\$\rangle$, where $s = \text{CABCABABABABABABABCCCCABABABABABABABABABABABABABABABABABABABABABABABCABCABC}$. Here,

$\text{Eblock}(\text{Pow}_0^s) = (1, 2), (3, 1), (2, 3, 2), (3, 2, 3), (2, 3)^6, (4, 2), (3, 2), (3, 2, 3), (2, 3)^6, (1, 2), (3, 1), (2, 3, 1)^2$,

$\text{Eblock}(\text{Pow}_1^s) = (5, 6, 8), (10, 13), (11, 9), (10, 13), (5, 6, 14)$,

$\text{Eblock}(\text{Pow}_2^s) = (16, 17, 18), (17, 19)$,

$\text{Eblock}(\text{Pow}_3^s) = (21, 22)$, and

$\text{id}(s\$) = 25$.

See also Fig. 4 which illustrates the signature tree induced from the above signature encoding of $s$. 31
Fig. 4. The signature tree of $s$ in Example 5, where the edges corresponding to $24 \to \$ \text{ and } 25 \to (23, 24)$ regarding the endmarker $\$$ are omitted for simplicity.
Fig. 5. The DAG for $\mathcal{D}$ of Example 33
References used in Appendix

22. Agarwal, P.K., Arge, L., Govindarajan, S., Yang, J., Yi, K.: Efficient external memory structures for range-aggregate queries. Comput. Geom. 46(3), 358–370 (2013), http://dx.doi.org/10.1016/j.comgeo.2012.10.003

23. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms 57(2), 75–94 (2005)

24. Blelloch, G.E.: Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In: Teng, S.H. (ed.) SODA. pp. 894–903. SIAM (2008)

25. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: Aho, A.V. (ed.) Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA. pp. 365–372. ACM (1987), http://doi.acm.org/10.1145/28395.28434

26. Han, Y.: Deterministic sorting in $O(n \log \log n)$ time and linear space. Proc. STOC 2002 pp. 602–608 (2002)

27. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1–3), 211–222 (2003)