Fully Dynamic Data Structure for LCE Queries in Compressed Space

Takaaki Nishimoto[1], Tomohiro I[2], Shunsuke Inenaga[3], Hideo Bannai[4], and Masayuki Takeda[5]

1 **Department of Informatics, Kyushu University, Japan**
`takaaki.nishimoto@inf.kyushu-u.ac.jp`
2 **Kyushu Institute of Technology, Japan**
`tomohiro@ai.kyutech.ac.jp`
3 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
5 **Department of Informatics, Kyushu University, Japan**
`takeda@inf.kyushu-u.ac.jp`

Abstract

A Longest Common Extension (LCE) query on a text T of length N asks for the length of the longest common prefix of suffixes starting at given two positions. We show that the signature encoding $\mathcal{G}$ of size $w = O(\min(z \log N \log^* M, N))$ [Mehlhorn et al., Algorithmica 17(2):183-198, 1997] of T, which can be seen as a compressed representation of T, has a capability to support LCE queries in $O(\log N + \log \ell \log^* M)$ time, where ℓ is the answer to the query, z is the size of the Lempel-Ziv77 (LZ77) factorization of T, and $M \geq 4N$ is an integer that can be handled in constant time under word RAM model. In compressed space, this is the fastest deterministic LCE data structure in many cases. Moreover, $\mathcal{G}$ can be enhanced to support efficient update operations: After processing $\mathcal{G}$ in $O(w f_{\mathcal{A}})$ time, we can insert/delete any (sub)string of length y into/from an arbitrary position of T in $O((y + \log N \log^* M) f_{\mathcal{A}})$ time, where $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$. This yields the first fully dynamic LCE data structure working in compressed space. We also present efficient construction algorithms from various types of inputs: We can construct $\mathcal{G}$ in $O(N f_{\mathcal{A}})$ time from uncompressed string T; in $O(n \log\log(n \log^* M) \log N \log^* M)$ time from grammar-compressed string T represented by a straight-line program of size n; and in $O(z f_{\mathcal{A}} \log N \log^* M)$ time from LZ77-compressed string T with z factors. On top of the above contributions, we show several applications of our data structures which improve previous best known results on grammar-compressed string processing.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Dynamic Texts, Longest Common Extension (LCE) Queries, Straight-line Program

Digital Object Identifier 10.4230/LIPIcs.MFCS.2016.72

1 Introduction

A *Longest Common Extension (LCE)* query on a text T of length N asks to compute the length of the longest common prefix of suffixes starting at given two positions. This fundamental query appears at the heart of many string processing problems (see text book [11] for example), and hence, efficient data structures to answer LCE queries gain a great attention.

41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016).
Editors: Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier; Article No. 72; pp. 72:1–72:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A classic solution is to use a data structure for lowest common ancestor queries [4] on the suffix tree of T. Although this achieves constant query time, the $\Theta(N)$ space needed for the data structure is too large to apply it to large scale data. Hence, recent work focuses on reducing space usage at the expense of query time. For example, time-space trade-offs of LCE data structure have been extensively studied [7, 24].

Another direction to reduce space is to utilize a compressed structure of T, which is advantageous when T is highly compressible. There are several LCE data structures working on grammar-compressed string T represented by a straight-line program (SLP) of size n. The best known deterministic LCE data structure is due to I et al. [13], which supports LCE queries in $O(h \log N)$ time, and occupies $O(n^2)$ space, where h is the height of the derivation tree of a given SLP. Their data structure can be built in $O(hn^2)$ time directly from the SLP. Bille et al. [5] showed a Monte Carlo randomized data structure which supports LCE queries in $O(\log N \log \ell)$ time, where ℓ is the output of the LCE query. Their data structure requires only $O(n)$ space, but requires $O(N)$ time to construct. Very recently, Bille et al. [6] showed a faster Monte Carlo randomized data structure of $O(n)$ space which supports LCE queries in $O(\log N + \log^2 \ell)$ time. The preprocessing time of this new data structure is not given in [6]. Note that, given the LZ77-compression of size z of T, we can convert it into an SLP of size $n = O(z \log \frac{N}{z})$ [22] and then apply the above results.

In this paper, we focus on the *signature encoding* $\mathcal{G}$ of T, which can be seen as a grammar compression of T, and show that $\mathcal{G}$ can support LCE queries efficiently. The signature encoding was proposed by Mehlhorn et al. for equality testing on a dynamic set of strings [19]. Alstrup et al. used signature encodings combined with their own data structure called anchors to present a pattern matching algorithm on a dynamic set of strings [2, 1]. In their paper, they also showed that signature encodings can support longest common prefix (LCP) and longest common suffix (LCS) queries on a dynamic set of strings. Their algorithm is randomized as it uses a hash table for maintaining the dictionary of $\mathcal{G}$. Very recently, Gawrychowski et al. improved the results by pursuing advantages of randomized approach other than the hash table [10]. It should be noted that the algorithms in [2, 1, 10] can support LCE queries by combining split operations and LCP queries although it is not explicitly mentioned. However, they did not focus on the fact that signature encodings can work in compressed space. In [9], LCE data structures on edit sensitive parsing, a variant of signature encoding, was used for sparse suffix sorting, but again, they did not focus on working in compressed space.

Our contributions are stated by the following theorems, where $M \geq 4N$ is an integer that can be handled in constant time under word RAM model. More specifically, $M = 4N$ if T is static, and $M/4$ is the upper bound of the length of T if we consider updating T dynamically. In dynamic case, N (resp. w) always denotes the current size of T (resp. $\mathcal{G}$). Also, $f_{\mathcal{A}}$ denotes the time for predecessor/successor queries on a set of w integers from an M-element universe, which is $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$ by the best known data structure [3].

▶ **Theorem 1** (LCE queries). *Let $\mathcal{G}$ denote the signature encoding of size $w = O(\min(z \log N \log^* M, N))$ for a string T of length N. Then $\mathcal{G}$ supports LCE queries on T in $O(\log N + \log \ell \log^* M)$ time, where ℓ is the answer to the query, and z is the size of the LZ77 factorization of T.*

▶ **Theorem 2** (Updates). *After processing $\mathcal{G}$ in $O(w f_{\mathcal{A}})$ time, we can insert/delete any (sub)string Y of length y into/from an arbitrary position of T in $O((y + \log N \log^* M) f_{\mathcal{A}})$ time. If Y is given as a substring of T, we can support insertion in $O(f_{\mathcal{A}} \log N \log^* M)$ time.*

▶ **Theorem 3** (Construction). *Let T be a string of length N, Z be LZ77 factorization without self reference of size z representing T, and $\mathcal{S}$ be an SLP of size n generating T. Then, we can construct the signature encoding $\mathcal{G}$ for T in (1a) in $O(N f_{\mathcal{A}})$ time and $O(w)$ working space from T, (1b) in $O(N)$ time and working space from T, (2) in $O(z f_{\mathcal{A}} \log N \log^* M)$ time and $O(w)$ working space from Z, (3a) in $O(n f_{\mathcal{A}} \log N \log^* M)$ time and $O(w)$ working space from $\mathcal{S}$, and (3b) in $O(n \log\log(n \log^* M) \log N \log^* M)$ time and $O(n \log^* M + w)$ working space from $\mathcal{S}$.*

The remarks on our contributions are listed in the following:

- We achieve an algorithm for the fastest deterministic LCE queries on SLPs, which even permits faster LCE queries than the randomized data structure of Bille et al. [6] when $\log^* M = o(\log \ell)$ which in many cases is true.
- We present the first fully dynamic LCE data structure working in compressed space.
- Different from the work in [2, 1, 10], we mainly focus on maintaining a single text T in compressed $O(w)$ space. For this reason we opt for supporting insertion/deletion as edit operations rather than split/concatenate on a dynamic set of strings. However, the difference is not much essential; our insert operations specified by a substring of an existing string can work as split/concatenate, and conversely, split/concatenate can simulate insert. Our contribution here is to clarify how to collect garbage being produced during edit operations, as directly indicated by a support of delete operations.
- The results (2) and (3a) of Theorem 3 immediately follow from the update operations considered in [2, 1], but others are nontrivial.
- Direct construction of $\mathcal{G}$ from SLPs is important for applications in compressed string processing, where the task is to process a given compressed representation of string(s) without explicit decompression. In particular, we use the result (3b) of Theorem 3 to show several applications which improve previous best known results. Note that the time complexity of the result (3b) can be written as $O(n \log\log n \log N \log^* M)$ when $\log^* M = O(n)$ which in many cases is true, and always true in static case because $\log^* M = O(\log^* N) = O(\log N) = O(n)$.

Proofs and examples omitted due to lack of space are in a full version of this paper [21].

2 Preliminaries

2.1 Strings

Let Σ be an ordered alphabet. An element of Σ^* is called a string. For string $w = xyz$, x, y and z are called a prefix, substring, and suffix of w, respectively. The length of string w is denoted by $|w|$. The empty string ε is a string of length 0. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the i-th character of w. 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 $\mathsf{LCP}(w, u)$ (resp. $\mathsf{LCS}(w, u)$) denote the length of the longest common prefix (resp. suffix) of w and u. Given two strings s_1, s_2 and two integers i, j, let $\mathsf{LCE}(s_1, s_2, i, j)$ denote a query which returns $\mathsf{LCP}(s_1[i..|s_1|], s_2[j..|s_2|])$. Our model of computation is the unit-cost word RAM with machine word size of $\Omega(\log_2 M)$ bits, and space complexities will be evaluated by the number of machine words. Bit-oriented evaluation of space complexities can be obtained with a $\log_2 M$ multiplicative factor.

▶ **Definition 4** (Lempel-Ziv77 factorization [25])**.** The Lempel-Ziv77 (LZ77) factorization of a string s without self-references is a sequence $f_1, \ldots, f_z$ of non-empty substrings of s such that $s = f_1 \cdots f_z$, $f_1 = s[1]$, and for $1 < i \leq z$, if the character $s[|f_1..f_{i-1}| + 1]$ does not occur in $s[|f_1..f_{i-1}|]$, then $f_i = s[|f_1..f_{i-1}| + 1]$, otherwise f_i is the longest prefix of $f_i \cdots f_z$ which occurs in $f_1 \cdots f_{i-1}$. The size of the LZ77 factorization $f_1, \ldots, f_z$ of string s is the number z of factors in the factorization.

2.2 Context free grammars as compressed representation of strings

Straight-line programs. A *straight-line program* (*SLP*) is a context free grammar in the Chomsky normal form that generates a single string. Formally, an SLP that generates T is a quadruple $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, such that Σ is an ordered alphabet of terminal characters; $\mathcal{V} = \{X_1, \ldots, X_n\}$ is a set of positive integers, called *variables*; $\mathcal{D} = \{X_i \to expr_i\}_{i=1}^{n}$ is a set of *deterministic productions* (or *assignments*) with each $expr_i$ being either of form $X_\ell X_r$ $(1 \leq \ell, r < i)$, or a single character $a \in \Sigma$; and $S := X_n \in \mathcal{V}$ is the start symbol which derives the string T. We also assume that the grammar neither contains *redundant* variables (i.e., there is at most one assignment whose righthand side is *expr*) nor *useless* variables (i.e., every variable appears at least once in the derivation tree of $\mathcal{G}$). The *size* of the SLP $\mathcal{G}$ is the number n of productions in $\mathcal{D}$. In the extreme cases the length N of the string T can be as large as 2^{n-1}, however, it is always the case that $n \geq \log_2 N$.

Let $val : \mathcal{V} \to \Sigma^+$ be the function which returns the string derived by an input variable. If $s = val(X)$ for $X \in \mathcal{V}$, then we say that the variable X *represents* string s. For any variable sequence $y \in \mathcal{V}^+$, let $val^+(y) = val(y[1]) \cdots val(y[|y|])$.

Run-length straight-line programs. We define *run-length SLPs* (*RLSLPs*), as an extension to SLPs, which allow *run-length encodings* in the righthand sides of productions, i.e., $\mathcal{D}$ might contain a production $X \to \hat{X}^k \in \mathcal{V} \times \mathcal{N}$. The *size* of the RLSLP is still the number of productions in $\mathcal{D}$ as each production can be encoded in constant space. Let $Assgn_{\mathcal{G}}$ be the function such that $Assgn_{\mathcal{G}}(X_i) = expr_i$ iff $X_i \to expr_i \in \mathcal{D}$. Also, let $Assgn_{\mathcal{G}}^{-1}$ denote the reverse function of $Assgn_{\mathcal{G}}$. When clear from the context, we write $Assgn_{\mathcal{G}}$ and $Assgn_{\mathcal{G}}^{-1}$ as $Assgn$ and $Assgn^{-1}$, respectively.

Representation of RLSLPs. For an RLSLP $\mathcal{G}$ of size w, we can consider a DAG of size w as a compact representation of the derivation trees of variables in $\mathcal{G}$. Each node represents a variable X in $\mathcal{V}$ and store $|val(X)|$ and out-going edges represent the assignments in $\mathcal{D}$: For an assignment $X_i \to X_\ell X_r \in \mathcal{D}$, there exist two out-going edges from X_i to its ordered children X_ℓ and X_r; and for $X \to \hat{X}^k \in \mathcal{D}$, there is a single edge from X to $\hat{X}$ with the multiplicative factor k.

3 Signature encoding

Here, we recall the *signature encoding* first proposed by Mehlhorn et al. [19]. Its core technique is *locally consistent parsing* defined as follows:

▶ **Lemma 5** (Locally consistent parsing [19, 1])**.** *Let W be a positive integer. There exists a function $f : [0..W]^{\log^* W + 11} \to \{0, 1\}$ such that, for any $p \in [1..W]^n$ with $n \geq 2$ and $p[i] \neq p[i+1]$ for any $1 \leq i < n$, the bit sequence d defined by $d[i] = f(\tilde{p}[i-\Delta_L], \ldots, \tilde{p}[i+\Delta_R])$ for $1 \leq i \leq n$, satisfies: $d[1] = 1$; $d[n] = 0$; $d[i] + d[i+1] \leq 1$ for $1 \leq i < n$; and $d[i] + d[i+1] + d[i+2] + d[i+3] \geq 1$ for any $1 \leq i < n-3$; where $\Delta_L = \log^* W + 6$, $\Delta_R = 4$,* *and* $\tilde{p}[j] = p[j]$ *for all* $1 \leq j \leq n$, $\tilde{p}[j] = 0$ *otherwise. Furthermore, we can compute* d *in* $O(n)$ *time using a precomputed table of size* $o(\log W)$, *which can be computed in* $o(\log W)$ *time.*

For the bit sequence d of Lemma 5, we define the function $Eblock_d(p)$ that decomposes an integer sequence p according to d: $Eblock_d(p)$ decomposes p into a sequence $q_1, \ldots, q_j$ of substrings called *blocks* of p, such that $p = q_1 \cdots q_j$ and q_i is in the decomposition iff $d[|q_1 \cdots q_{i-1}| + 1] = 1$ for any $1 \leq i \leq j$. Note that each block is of length from two to four by the property of d, i.e., $2 \leq |q_i| \leq 4$ for any $1 \leq i \leq j$. Let $|Eblock_d(p)| = j$ and let $Eblock_d(s)[i] = q_i$. We omit d and write $Eblock(p)$ when it is clear from the context, and we use implicitly the bit sequence created by Lemma 5 as d.

We complementarily use run-length encoding to get a sequence to which *Eblock* can be applied. Formally, for a string s, let $Epow(s)$ be the function which groups each maximal run of same characters a as a^k, where k is the length of the run. $Epow(s)$ can be computed in $O(|s|)$ time. Let $|Epow(s)|$ denote the number of maximal runs of same characters in s and let $Epow(s)[i]$ denote i-th maximal run in s.

The signature encoding is the RLSLP $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, where the assignments in $\mathcal{D}$ are determined by recursively applying *Eblock* and *Epow* to T until a single integer S is obtained. We call each variable of the signature encoding a *signature*, and use e (for example, $e_i \to e_\ell e_r \in \mathcal{D}$) instead of X to distinguish from general RLSLPs.

For a formal description, let $E := \Sigma \cup \mathcal{V}^2 \cup \mathcal{V}^3 \cup \mathcal{V}^4 \cup (\mathcal{V} \times \mathcal{N})$ and let $Sig : E \to \mathcal{V}$ be the function such that: $Sig(x) = e$ if $(e \to x) \in \mathcal{D}$; $Sig(x) = Sig(Sig(x[1..|x| - 1])x[|x|])$ if $x \in \mathcal{V}^3 \cup \mathcal{V}^4$; or otherwise undefined. Namely, the function *Sig* returns, if any, the lefthand side of the corresponding production of x by recursively applying the $Assgn^{-1}$ function from left to right. For any $p \in E^*$, let $Sig^+(p) = Sig(p[1]) \cdots Sig(p[|p|])$.

The signature encoding of string T is defined by the following *Shrink* and *Pow* functions: $Shrink^T_t = Sig^+(T)$ for $t = 0$, and $Shrink^T_t = Sig^+(Eblock(Pow^T_{t-1}))$ for $0 < t \leq h$; and $Pow^T_t = Sig^+(Epow(Shrink^T_t))$ for $0 \leq t \leq h$; where h is the minimum integer satisfying $|Pow^T_h| = 1$. Then, the start symbol of the signature encoding is $S = Pow^T_h$. We say that a node is in *level* t in the derivation tree of S if the node is produced by $Shrink^T_t$ or Pow^T_t. The height of the derivation tree of the signature encoding of T is $O(h) = O(\log |T|)$. For any $T \in \Sigma^+$, let $id(T) = Pow^T_h = S$, i.e., the integer S is the signature of T.

In this paper, we implement signature encodings by the DAG of RLSLP introduced in Section 2.

4 Compressed LCE data structure using signature encodings

In this section, we show Theorem 1.

Space requirement of the signature encoding. It is clear from the definition of the signature encoding $\mathcal{G}$ of T that the size of $\mathcal{G}$ is less than $4N \leq M$, and hence, all signatures are in $[1..M - 1]$. Moreover, the next lemma shows that $\mathcal{G}$ requires only *compressed space*:

▶ **Lemma 6** ([23]). *The size* w *of the signature encoding of* T *of length* N *is* $O(z \log N \log^* M)$, *where* z *is the number of factors in the LZ77 factorization without self-reference of* T.

Common sequences of signatures to all occurrences of same substrings. Here, we recall the most important property of the signature encoding, which ensures the existence of common signatures to all occurrences of same substrings by the following lemma.

▶ **Lemma 7** (common sequences [23]). *Let $\mathcal{G}$ be a signature encoding for a string T. Every substring P in T is represented by a signature sequence $Uniq(P)$ in $\mathcal{G}$ for a string P.*

$Uniq(P)$, which we call the *common sequence* of P, is defined by the following.

▶ **Definition 8.** For a string P, let

$$XShrink_t^P = \begin{cases} Sig^+(P) & \text{for } t = 0, \\ Sig^+(Eblock_d(XPow_{t-1}^P)[|L_t^P|..|XPow_{t-1}^P| - |R_t^P|]) & \text{for } 0 < t \leq h^P, \end{cases}$$

$$XPow_t^P = Sig^+(Epow(XShrink_t^P[|\hat{L}_t^P| + 1..|XShrink_t^P| - |\hat{R}_t^P|])|) \text{ for } 0 \leq t < h^P,$$

- L_t^P is the shortest prefix of $XPow_{t-1}^P$ of length at least Δ_L such that $d[|L_t^P| + 1] = 1$,
- R_t^P is the shortest suffix of $XPow_{t-1}^P$ of length at least $\Delta_R{+}1$ such that $d[|d|{-}|R_t^P|{+}1] = 1$,
- $\hat{L}_t^P$ is the longest prefix of $XShrink_t^P$ such that $|Epow(\hat{L}_t^P)| = 1$,
- $\hat{R}_t^P$ is the longest suffix of $XShrink_t^P$ such that $|Epow(\hat{R}_t^P)| = 1$, and
- h^P is the minimum integer such that $|Epow(XShrink_{h^P}^P)| \leq \Delta_L + \Delta_R + 9$.

Note that $\Delta_L \leq |L_t^P| \leq \Delta_L + 3$ and $\Delta_R + 1 \leq |R_t^P| \leq \Delta_R + 4$ hold by the definition. Hence $|XShrink_{t+1}^P| > 0$ holds if $|Epow(XShrink_t^P)| > \Delta_L + \Delta_R + 9$. Then,

$$Uniq(P) = \hat{L}_0^P L_0^P \cdots \hat{L}_{h^P-1}^P L_{h^P-1}^P XShrink_{h^P}^P R_{h^P-1}^P \hat{R}_{h^P-1}^P \cdots R_0^P \hat{R}_0^P.$$

We give an intuitive description of Lemma 7. Recall the locally consistent parsing of Lemma 5. Each i-th bit of bit sequence d of Lemma 5 for a given string s is determined by $s[i - \Delta_L..i + \Delta_R]$. Hence, for two positions i, j such that $P = s[i..i + k - 1] = s[j..j + k - 1]$ for some k, $d[i + \Delta_L..i + k - 1 - \Delta_R] = d[j + \Delta_L..j + k - 1 - \Delta_R]$ holds, namely, "internal" bit sequences of the same substring of s are equal. Since each level of the signature encoding uses the bit sequence, all occurrences of same substrings in a string share same internal signature sequences, and this goes up level by level. $XShrink_t^P$ and $XPow_t^P$ represent signature sequences obtained from only internal signature sequences of $XPow_{t-1}^T$ and $XShrink_t^T$, respectively. This means that $XShrink_t^P$ and $XPow_t^P$ are always created over P. From such common signatures we take as short signature sequence as possible for $Uniq(P)$: Since $val^+(Pow_{t-1}^P) = val^+(L_{t-1}^P XShrink_t^P R_{t-1}^P)$ and $val^+(Shrink_t^P) = val^+(\hat{L}_t^P XPow_t^P \hat{R}_t^P)$ hold, $|Epow(Uniq(P))| = O(\log|P| \log^* M)$ and $val^+(Uniq(P)) = P$ hold. Hence Lemma 7 holds [1].

The number of ancestors of nodes corresponding to $Uniq(P)$ is upper bounded by:

▶ **Lemma 9.** *Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be a signature encoding for a string T, P be a string, and let $\mathcal{T}$ be the derivation tree of a signature $e \in \mathcal{V}$. Consider an occurrence of P in s, and the induced subtree X of $\mathcal{T}$ whose root is the root of $\mathcal{T}$ and whose leaves are the parents of the nodes representing $Uniq(P)$, where $s = val(e)$. Then X contains $O(\log^* M)$ nodes for every level and $O(\log|s| + \log|P| \log^* M)$ nodes in total.*

LCE queries. In the next lemma, we show a more general result than Theorem 1, which states that the signature encoding supports (both forward and backward) LCE queries on a given arbitrary pair of signatures. Theorem 1 immediately follows from Lemma 10.

▶ **Lemma 10.** *Using a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ for a string T, we can support queries $\mathsf{LCE}(s_1, s_2, i, j)$ and $\mathsf{LCE}(s_1^R, s_2^R, i, j)$ in $O(\log|s_1| + \log|s_2| + \log \ell \log^* M)$ time for given two signatures $e_1, e_2 \in \mathcal{V}$ and two integers $1 \leq i \leq |s_1|$, $1 \leq j \leq |s_2|$, where $s_1 = val(e_1)$, $s_2 = val(e_2)$ and ℓ is the answer to the LCE query.*

[1] The common sequences are conceptually equivalent to the *cores* [17] which are defined for the *edit sensitive parsing* of a text, a kind of locally consistent parsing of the text.

Proof. We focus on $\mathsf{LCE}(s_1, s_2, i, j)$ as $\mathsf{LCE}(s_1^R, s_2^R, i, j)$ is supported similarly.

Let P denote the longest common prefix of $s_1[i..]$ and $s_2[j..]$. Our algorithm simultaneously traverses two derivation trees rooted at e_1 and e_2 and computes P by matching the common signatures greedily from left to right. Recall that s_1 and s_2 are substrings of T. Since the both substrings P occurring at position i in $val(e_1)$ and at position j in $val(e_2)$ are represented by $Uniq(P)$ in the signature encoding by Lemma 7, we can compute P by at least finding the common sequence of nodes which represents $Uniq(P)$, and hence, we only have to traverse ancestors of such nodes. By Lemma 9, the number of nodes we traverse, which dominates the time complexity, is upper bounded by $O(\log |s_1| + \log |s_2| + Epow(Uniq(P))) = O(\log |s_1| + \log |s_2| + \log \ell \log^* M)$. ◀

5 Updates

In this section, we show Theorem 2. Formally, we consider a *dynamic signature encoding* $\mathcal{G}$ of T, which allows for efficient updates of $\mathcal{G}$ in compressed space according to the following operations: $INSERT(Y, i)$ inserts a string Y into T at position i, i.e., $T \leftarrow T[..i-1]YT[i..]$; $INSERT'(j, y, i)$ inserts $T[j..j+y-1]$ into T at position i, i.e., $T \leftarrow T[..i-1]T[j..j+y-1]T[i..]$; and $DELETE(j, y)$ deletes a substring of length y starting at j, i.e., $T \leftarrow T[..j-1]T[j+y..]$.

During updates we recompute $Shrink_t^T$ and Pow_t^T for some part of new T (note that the most part is unchanged thanks to the virtue of signature encodings, Lemma 9). When we need a signature for $expr$, we look up the signature assigned to $expr$ (i.e., compute $Assign^{-1}(expr)$) and use it if such exists. If $Assign^{-1}(expr)$ is undefined we create a new signature, which is an integer that is currently not used as signatures (say $e_{new} = \min([1..M] \setminus \mathcal{V})$), and add $e_{new} \rightarrow expr$ to $\mathcal{D}$. Also, updates may produce a useless signature whose parents in the DAG are all removed. We remove such useless signatures from $\mathcal{G}$ during updates.

Note that the corresponding nodes and edges of the DAG can be added/removed in constant time per addition/removal of an assignment. In addition to the DAG, we need dynamic data structures to conduct the following operations efficiently: (A) computing $Assgn^{-1}(\cdot)$, (B) computing $\min([1..M] \setminus \mathcal{V})$, and (C) checking if a signature e is useless.

For (A), we use Beame and Fich's data structure [3] that can support predecessor/successor queries on a dynamic set of integers.[2] For example, we consider Beame and Fich's data structure maintaining a set of integers $\{e_\ell M^2 + e_r M + e \mid e \rightarrow e_\ell e_r \in \mathcal{D}\}$ in $O(w)$ space. Then we can implement $Assgn^{-1}(e_\ell e_r)$ by computing the successor q of $e_\ell M^2 + e_r M$, i.e., $e = q \bmod M$ if $\lfloor q/M \rfloor = e_\ell M + e_r$, and otherwise $Assgn^{-1}(e_\ell e_r)$ is undefined. Queries as well as update operations can be done in deterministic $O(f_\mathcal{A})$ time, where $f_\mathcal{A} = O\left(\min\left\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\right\}\right)$.

For (B), we again use Beame and Fich's data structure to maintain the set of maximal intervals such that every element in the intervals is signature. Formally, the intervals are maintained by a set of integers $\{e_i M + e_j \mid [e_i..e_j] \subseteq \mathcal{V}, e_i - 1 \notin \mathcal{V}, e_j + 1 \notin \mathcal{V}\}$ in $O(w)$ space. Then we can know the minimum integer currently not in $\mathcal{V}$ by computing the successor of 0.

For (C), we let every signature $e \in \mathcal{V}$ have a counter to count the number of parents of e in the DAG. Then we can know that a signature is useless if the counter is 0.

Lemma 11 shows that we can efficiently compute $Uniq(P)$ for a substring P of T.

[2] Alstrup et al. [1] used hashing for this purpose. However, since we are interested in the worst case time complexities, we use the data structure [3] in place of hashing.

▶ **Lemma 11.** *Using a signature encoding $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w, given a signature $e \in \mathcal{V}$ (and its corresponding node in the DAG) and two integers j and y, we can compute $Epow(Uniq(s[j..j+y-1]))$ in $O(\log |s| + \log y \log^* M)$ time, where $s = val(e)$.*

Proof of Theorem 2. It is easy to see that, given the static signature encoding of T, we can construct data structures (A)-(C) in $O(wf_{\mathcal{A}})$ time. After constructing these, we can add/remove an assignment in $O(f_{\mathcal{A}})$ time.

Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be the signature encoding before the update operation. We support $DELETE(j, y)$ as follows: (1) Compute the new start variable $S' = id(T[..j-1]T[j+y..])$ by recomputing the new signature encoding from $Uniq(T[..j-1])$ and $Uniq(T[j+y..])$. Although we need a part of d to recompute $Eblock_d(Pow_t^{T[..j-1]T[j+y..]})$ for every level t, the input size to compute the part of d is $O(\log^* M)$ by Lemma 5. Hence these can be done in $O(f_{\mathcal{A}} \log N \log^* M)$ time by Lemmas 11 and 9. (2) Remove all useless signatures Z from $\mathcal{G}$. Note that if a signature is useless, then all the signatures along the path from S to it are also useless. Hence, we can remove all useless signatures efficiently by depth-first search starting from S, which takes $O(f_{\mathcal{A}}|Z|)$ time, where $|Z| = O(y + \log N \log^* M)$ by Lemma 9.

Similarly, we can support $INSERT(Y, i)$ in $O(f_{\mathcal{A}}(y + \log N \log^* M))$ time by creating the new start variable S' from $Uniq(T[..i-1])$, $Uniq(Y)$ and $Uniq(T[i..])$. Note that we can naively compute $Uniq(Y)$ in $O(f_{\mathcal{A}}y)$ time. For $INSERT'(j, y, i)$, we can avoid $O(f_{\mathcal{A}}y)$ time by computing $Uniq(T[j..j+y-1])$ using Lemma 11. ◀

6 Construction

In this section, we give proofs of Theorem 3, but we omit proofs of the results (2) and (3a) as they are straightforward from the previous work [2, 1].

6.1 Theorem 3 (1a)

Proof of Theorem 3 (1a). Note that we can naively compute $id(T)$ for a given string T in $O(Nf_{\mathcal{A}})$ time and $O(N)$ working space. In order to reduce the working space, we consider factorizing T into blocks of size B and processing them incrementally: Starting with the empty signature encoding $\mathcal{G}$, we can compute $id(T)$ in $O(\frac{N}{B} f_{\mathcal{A}}(\log N \log^* M + B))$ time and $O(w + B)$ working space by using $INSERT(T[(i-1)B+1..iB], (i-1)B+1)$ for $i = 1, \ldots, \frac{N}{B}$ in increasing order. Hence our proof is finished by choosing $B = \log N \log^* M$. ◀

6.2 Theorem 3 (1b)

We compute signatures level by level, i.e., construct $Shrink_0^T, Pow_0^T, \ldots, Shrink_h^T, Pow_h^T$ incrementally. For each level, we create signatures by sorting signature blocks (or run-length encoded signatures) to which we give signatures, as shown by the next two lemmas.

▶ **Lemma 12.** *Given $Eblock(Pow_{t-1}^T)$ for $0 < t \leq h$, we can compute $Shrink_t^T$ in $O((b-a) + |Pow_{t-1}^T|)$ time and space, where b is the maximum integer in Pow_{t-1}^T and a is the minimum integer in Pow_{t-1}^T.*

Proof. Since we assign signatures to signature blocks and run-length signatures in the derivation tree of S in the order they appear in the signature encoding. $Pow_{t-1}^T[i] - a$ fits in an entry of a bucket of size $b - a$ for each element of $Pow_{t-1}^T[i]$ of Pow_{t-1}^T. Also, the length of each block is at most four. Hence we can sort all the blocks of $Eblock(Pow_{t-1}^T)$ by bucket sort in $O((b-a) + |Pow_{t-1}^T|)$ time and space. Since 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 Shrink^T_{t-1} appear in $\mathit{Shrink}^T_{t'-1}$, and hence no elements of Pow^T_{t-1} appear in $\mathit{Pow}^T_{t'-1}$. Thus, we can determine a new signature for each block in $\mathit{Eblock}(\mathit{Pow}^T_{t-1})$, *without* searching existing signatures in the lower levels. This completes the proof. ◀

▶ **Lemma 13.** *Given $\mathit{Epow}(\mathit{Shrink}^T_t)$, we can compute Pow^T_t in $O(x + (b - a) + |\mathit{Epow}(\mathit{Shrink}^T_t)|)$ time and space, where x is the maximum length of runs in $\mathit{Epow}(\mathit{Shrink}^T_t)$, b is the maximum integer in Pow^T_{t-1}, and a is the minimum integer in Pow^T_{t-1}.*

Proof. We first sort all the elements of $\mathit{Epow}(\mathit{Shrink}^T_t)$ by bucket sort in $O(b - a + |\mathit{Epow}(\mathit{Shrink}^T_t)|)$ time and space, ignoring the powers of runs. Then, for each integer r appearing in Shrink^T_t, we sort the runs of r's by bucket sort with a bucket of size x. This takes a total of $O(x + |\mathit{Epow}(\mathit{Shrink}^T_t)|)$ time and space for all integers appearing in Shrink^T_t. The rest is the same as the proof of Lemma 12. ◀

Proof of Theorem 3 (1b). Since the size of the derivation tree of $id(T)$ is $O(N)$, by Lemmas 5, 12, and 13, we can compute a DAG of $\mathcal{G}$ for T in $O(N)$ time and space. ◀

6.3 Theorem 3 (3b)

In this section, we sometimes abbreviate $val(X)$ as X for $X \in \mathcal{S}$. For example, Shrink^X_t and Pow^X_t represents $\mathit{Shrink}^{val(X)}_t$ and $\mathit{Pow}^{val(X)}_t$ respectively.

Our algorithm computes signatures level by level, i.e., constructs incrementally $\mathit{Shrink}^{X_n}_0$, $\mathit{Pow}^{X_n}_0, \ldots, \mathit{Shrink}^{X_n}_h, \mathit{Pow}^{X_n}_h$. Like the algorithm described in Section 6.2, we can create signatures by sorting blocks of signatures or run-length encoded signatures in the same level. The main difference is that we now utilize the structure of the SLP, which allows us to do the task efficiently in $O(n \log^* M + w)$ working space. In particular, although $|\mathit{Shrink}^{X_n}_t|, |\mathit{Pow}^{X_n}_t| = O(N)$ for $0 \le t \le h$, they can be represented in $O(n \log^* M)$ space.

In so doing, we introduce some additional notations relating to $\mathit{XShrink}^P_t$ and XPow^P_t in Definition 8. By Lemma 7, there exist $\hat{z}^{(P_1,P_2)}_t$ and $z^{(P_1,P_2)}_t$ for any string $P = P_1P_2$ such that the following equation holds: $\mathit{XShrink}^P_t = \hat{y}^{P_1}_t \hat{z}^{(P_1,P_2)}_t \hat{y}^{P_2}_t$ for $0 < t \le h^P$, and $\mathit{XPow}^P_t = y^{P_1}_t z^{(P_1,P_2)}_t y^{P_2}_t$ for $0 \le t < h^P$, where we define $\hat{y}^P_t$ and y^P_t for a string P as:

$$\hat{y}^P_t = \begin{cases} \mathit{XShrink}^P_t & \text{for } 0 < t \le h^P, \\ \varepsilon & \text{for } t > h^P, \end{cases} \qquad y^P_t = \begin{cases} \mathit{XPow}^P_t & \text{for } 0 \le t < h^P, \\ \varepsilon & \text{for } t \ge h^P. \end{cases}$$

For any variable $X_i \to X_\ell X_r$, we denote $\hat{z}^{X_i}_t = \hat{z}^{(val(X_\ell),val(X_r))}_t$ (for $0 < t \le h^{val(X_i)}$) and $z^{X_i}_t = z^{(val(X_\ell),val(X_r))}_t$ (for $0 \le t < h^{val(X_i)}$). Note that $|z^{X_i}_t|, |\hat{z}^{X_i}_t| = O(\log^* M)$ because $z^{X_i}_t$ is created on $\hat{R}^{X_\ell}_t \hat{z}^{X_i}_t \hat{L}^{X_r}_t$, similarly, $\hat{z}^{X_i}_t$ is created on $R^{X_\ell}_{t-1} z^{X_i}_{t-1} L^{X_r}_{t-1}$. We can use $\hat{z}^{X_1}_t, \ldots, \hat{z}^{X_n}_t$ (resp. $z^{X_1}_t, \ldots, z^{X_n}_t$) as a compressed representation of $\mathit{XShrink}^{X_n}_t$ (resp. $\mathit{XPow}^{X_n}_t$) based on the SLP: Intuitively, $\hat{z}^{X_n}_t$ (resp. $z^{X_n}_t$) covers the middle part of $\mathit{XShrink}^{X_n}_t$ (resp. $\mathit{XPow}^{X_n}_t$) and the remaining part is recovered by investigating the left/right child recursively (see also Fig. 1). Hence, with the DAG structure of the SLP, $\mathit{XShrink}^{X_n}_t$ and $\mathit{XPow}^{X_n}_t$ can be represented in $O(n \log^* M)$ space.

In addition, we define $\hat{A}^P_t$, $\hat{B}^P_t$, A^P_t and B^P_t as follows: For $0 < t \le h^P$, $\hat{A}^P_t$ (resp. $\hat{B}^P_t$) is a prefix (resp. suffix) of Shrink^P_t which consists of signatures of $A^P_{t-1} L^P_{t-1}$ (resp. $R^P_{t-1} B^P_{t-1}$); and for $0 \le t < h^P$, A^P_t (resp. B^P_t) is a prefix (resp. suffix) of Pow^P_t which consists of signatures of $\hat{A}^P_t \hat{L}^P_t$ (resp. $\hat{R}^P_t \hat{B}^P_t$). By the definition, $\mathit{Shrink}^P_t = \hat{A}^P_t \mathit{XShrink}^P_t \hat{B}^P_t$ for $0 \le t \le h^P$, and $\mathit{Pow}^P_t = A^P_t \mathit{XPow}^P_t B^P_t$ for $0 \le t < h^P$. See Fig. 2 for the illustration.

Since $\mathit{Shrink}^{X_n}_t = \hat{A}^{X_n}_t \mathit{XShrink}^{X_n}_t \hat{B}^{X_n}_t$ for $0 < t \le h^{X_n}$, we use $\hat{\Lambda}_t = (\hat{z}^{X_1}_t, \ldots, \hat{z}^{X_n}_t, \hat{A}^{X_n}_t, \hat{B}^{X_n}_t)$ as a compressed representation of $\mathit{Shrink}^{X_n}_t$ of size $O(n \log^* M)$. Similarly, for $0 \le$

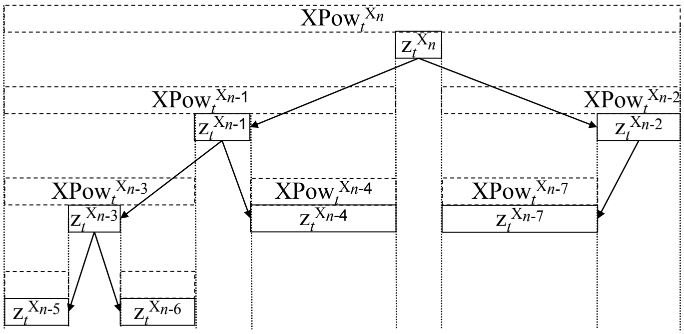

Figure 1 $XPow_t^{X_n}$ can be represented by $z_t^{X_1}, \dots, z_t^{X_n}$. In this example, $XPow_t^{X_n} = z_t^{X_{n-5}} z_t^{X_{n-3}} z_t^{X_{n-6}} z_t^{X_{n-1}} z_t^{X_{n-4}} z_t^{X_n} z_t^{X_{n-7}} z_t^{X_{n-2}}$.

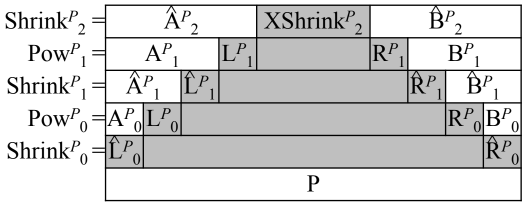

Figure 2 An abstract image of $Shrink_t^P$ and Pow_t^P for a string P. For $0 \leq t < h^P$, $A_t^P L_t^P$ (resp. $R_t^P B_t^P$) is encoded into $\hat{A}_{t+1}^P$ (resp. $\hat{B}_{t+1}^P$). Similarly, for $0 < t < h^P$, $\hat{A}_t^P \hat{L}_t^P$ (resp. $\hat{R}_t^P \hat{B}_t^P$) is encoded into A_t^P (resp. B_t^P).

$t < h^{X_n}$, we use $\Lambda_t = (z_t^{X_1}, \dots, z_t^{X_n}, A_t^{X_n}, B_t^{X_n})$ as a compressed representation of $Pow_t^{X_n}$ of size $O(n \log^* M)$.

Our algorithm computes incrementally $\Lambda_0, \hat{\Lambda}_1, \dots, \hat{\Lambda}_{h^{X_n}}$. Given $\hat{\Lambda}_{h^{X_n}}$, we can easily get $Pow_{h^{X_n}}^{X_n}$ of size $O(\log^* M)$ in $O(n \log^* M)$ time, and then $id(val(X_n))$ in $O(\log^* M)$ time from $Pow_{h^{X_n}}^{X_n}$. Hence, in the following three lemmas, we show how to compute $\Lambda_0, \hat{\Lambda}_1, \dots, \hat{\Lambda}_{h^{X_n}}$.

▶ **Lemma 14.** *Given an SLP of size n, we can compute Λ_0 in $O(n \log\log(n \log^* M) \log^* M)$ time and $O(n \log^* M)$ space.*

Proof. We first compute, for all variables X_i, $Epow(XShrink_0^{X_i})$ if $|Epow(XShrink_0^{X_i})| \leq \Delta_L + \Delta_R + 9$, otherwise $Epow(\hat{L}_0^{X_i})$ and $Epow(\hat{R}_0^{X_i})$. The information can be computed in $O(n \log^* M)$ time and space in a bottom-up manner, i.e., by processing variables in increasing order. For $X_i \to X_\ell X_r$, if both $|Epow(XShrink_0^{X_\ell})|$ and $|Epow(XShrink_0^{X_r})|$ are no greater than $\Delta_L + \Delta_R + 9$, we can compute $Epow(XShrink_0^{X_i})$ in $O(\log^* M)$ time by naively concatenating $Epow(XShrink_0^{X_\ell})$ and $Epow(XShrink_0^{X_r})$. Otherwise $|Epow(XShrink_0^{X_i})| > \Delta_L + \Delta_R + 9$ must hold, and $Epow(\hat{L}_0^{X_i})$ and $Epow(\hat{R}_0^{X_i})$ can be computed in $O(1)$ time from the information for X_ℓ and X_r.

The run-length encoded signatures represented by $z_0^{X_i}$ can be obtained by using the above information for X_ℓ and X_r in $O(\log^* M)$ time: $z_0^{X_i}$ is created over run-length encoded signatures $Epow(XShrink_0^{X_\ell})$ (or $Epow(\hat{R}_0^{X_\ell})$) followed by $Epow(XShrink_0^{X_r})$ (or $Epow(\hat{R}_0^{X_r})$). Also, by definition $A_0^{X_n}$ and $B_0^{X_n}$ represents $Epow(\hat{L}_0^{X_n})$ and $Epow(\hat{R}_0^{X_n})$, respectively.

Hence, we can compute in $O(n \log^* M)$ time $O(n \log^* M)$ run-length encoded signatures to which we give signatures. We determine signatures by sorting the run-length encoded signatures as Lemma 13. However, in contrast to Lemma 13, we do not use bucket sort for sorting the powers of runs because the maximum length of runs could be as large as N and we cannot afford $O(N)$ space for buckets. Instead, we use the sorting algorithm of Han [12]

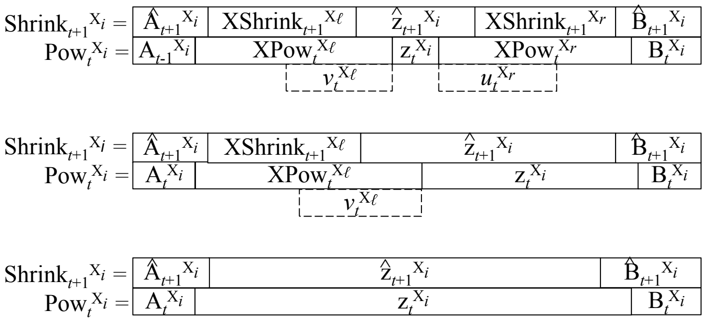

Figure 3 Abstract images of the needed signature sequence $v_t^{X_\ell} z_t^{X_i} u_t^{X_r}$ ($v_t^{X_\ell}$ and $u_t^{X_r}$ are not shown when they are empty) for computing $\hat{z}_{t+1}^{X_i}$ in three situations: Top for $0 \le t < h^{X_\ell}, h^{X_r}$; middle for $h^{X_r} \le t < h^{X_\ell}$; and bottom for $h^{X_\ell}, h^{X_r} \le t < h^{X_i}$.

which sorts x integers in $O(x \log\log x)$ time and $O(x)$ space. Hence, we can compute Λ_0 in $O(n \log\log(n \log^* M) \log^* M)$ time and $O(n \log^* M)$ space. ◀

▶ **Lemma 15.** *Given $\hat{\Lambda}_t$, we can compute Λ_t in $O(n \log\log(n \log^* M) \log^* M)$ time and $O(n \log^* M)$ space.*

Proof. The computation is similar to that of Lemma 14 except that we also use $\hat{\Lambda}_t$. ◀

▶ **Lemma 16.** *Given Λ_t, we can compute $\hat{\Lambda}_{t+1}$ in $O(n \log^* M)$ time and $O(n \log^* M)$ space.*

Proof. In order to compute $\hat{z}_{t+1}^{X_i}$ for a variable $X_i \to X_\ell X_r$, we need a signature sequence on which $\hat{z}_{t+1}^{X_i}$ is created, as well as its context, i.e., Δ_L signatures to the left and Δ_R to the right. To be precise, the needed signature sequence is $v_t^{X_\ell} z_t^{X_i} u_t^{X_r}$, where $u_t^{X_j}$ (resp. $v_t^{X_j}$) denotes a prefix (resp. suffix) of $y_t^{X_j}$ of length $\Delta_L + \Delta_R + 4$ for any variable X_j (see also Figure 3). Also, we need $A_t u_t^{X_n}$ and $v_t^{X_n} B_t$ to create $\hat{A}_{t+1}^{X_n}$ and $\hat{B}_{t+1}^{X_n}$, respectively.

Note that by Definition 8, $|z_t^X| > \Delta_L + \Delta_R + 9$ if $z_t^X \neq \varepsilon$. Then, we can compute $u_t^{X_i}$ for all variables X_i in $O(n \log^* M)$ time and space by processing variables in increasing order on the basis of the following fact: $u_t^{X_i} = u_t^{X_\ell}$ if $z_t^{X_\ell} \neq \varepsilon$, otherwise $u_t^{X_i}$ is the prefix of $z_t^{X_i}$ of length $\Delta_L + \Delta_R + 4$. Similarly $v_t^{X_i}$ for all variables X_i can be computed in $O(n \log^* M)$ time and space.

Using $u_t^{X_i}$ and $v_t^{X_i}$ for all variables X_i, we can obtain $O(n \log^* M)$ blocks of signatures to which we give signatures. We determine signatures by sorting the blocks by bucket sort as in Lemma 12 in $O(n \log^* M)$ time. Hence, we can get $\hat{\Lambda}_{t+1}$ in $O(n \log^* M)$ time and space. ◀

Proof of Theorem 3 (3b). Using Lemmas 14, 15 and 16, we can get $\hat{\Lambda}_{h^{X_n}}$ in $O(n \log\log (n \log^* M) \log N \log^* M)$ time by computing $\Lambda_0, \hat{\Lambda}_1, \ldots, \hat{\Lambda}_{h^{X_n}}$ incrementally. Note that during the computation we only have to keep Λ_t (or $\hat{\Lambda}_t$) for the current t and the assignments of $\mathcal{G}$. Hence the working space is $O(n \log^* M + w)$. By processing $\hat{\Lambda}_{h^{X_n}}$ in $O(n \log^* M)$ time, we can get the DAG of $\mathcal{G}$ of size $O(w)$. ◀

7 Applications

Theorem 17 is an application to text compression. Theorems 19-23 are applications to compressed string processing, where the task is to process a given compressed representation of string(s) without explicit decompression. We believe that only a few applications are listed here, considering the importance of LCE queries. As one example of unlisted applications, there is a paper [14] in which our LCE data structure was used to improve an algorithm of computing the Lyndon factorization of a string represented by a given SLP.

▶ **Theorem 17.** (1) *Given a dynamic signature encoding $\mathcal{G}$ for $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ of size w which generates T, we can compute an SLP $\mathcal{S}$ of size $O(w \log |T|)$ generating T in $O(w \log |T|)$ time.* (2) *Let us conduct a single INSERT or DELETE operation on the string T generated by the SLP of (1). Let y 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 $O((y + \log |T'| \log^* M)(f_{\mathcal{A}} + \log |T'|))$ time, the SLP of (1) to an SLP $\mathcal{S}'$ of size $O(w' \log |T'|)$ which generates T', where w' is the size of updated $\mathcal{G}$ which generates T'.*

We can get the next lemma using Theorem 3 (3b) and Theorem 2:

▶ **Lemma 18.** *Given an SLP of size n representing a string of length N, we can sort the variables of the SLP in lexicographical order in $O(n \log n \log N \log^* N)$ time and $O(n \log^* N + w)$ working space.*

Lemma 18 has an application to an SLP-based index of Claude and Navarro [8]. In the paper, they showed how to construct their index in $O(n \log n)$ time if the lexicographic order of variables of a given SLP is already computed. However, in order to sort variables they almost decompressed the string, and hence, needs $\Omega(N)$ time and $\Omega(N \log |\Sigma|)$ bits of working space. Now, Lemma 18 improves the sorting part yielding the next theorem.

▶ **Theorem 19.** *Given an SLP of size n representing a string of length N, we can construct the SLP-based index of [8] in $O(n \log n \log N \log^* N)$ time and $O(n \log^* N + w)$ working space.*

▶ **Theorem 20.** *Given an SLP $\mathcal{S}$ of size n generating a string T of length N, we can construct, in $O(n \log \log n \log N \log^* N)$ time, a data structure which occupies $O(n \log N \log^* N)$ space and supports $\mathsf{LCP}(val(X_i), val(X_j))$ and $\mathsf{LCS}(val(X_i), val(X_j))$ queries for variables X_i, X_j in $O(\log N)$ time. The $\mathsf{LCP}(val(X_i), val(X_j))$ and $\mathsf{LCS}(val(X_i), val(X_j))$ query times can be improved to $O(1)$ using $O(n \log n \log N \log^* N)$ preprocessing time.*

▶ **Theorem 21.** *Given an SLP $\mathcal{S}$ of size n generating a string T of length N, there is a data structure which occupies $O(w + n)$ space and supports queries $\mathsf{LCE}(val(X_i), val(X_j), a, b)$ for variables X_i, X_j, $1 \leq a \leq |X_i|$ and $1 \leq b \leq |X_j|$ in $O(\log N + \log \ell \log^* N)$ time, where $w = O(z \log N \log^* N)$. The data structure can be constructed in $O(n \log \log n \log N \log^* N)$ preprocessing time and $O(n \log^* N + w)$ working space, where $z \leq n$ is the size of the LZ77 factorization of T and ℓ is the answer of LCE query.*

Let h be the height of the derivation tree of a given SLP $\mathcal{S}$. Note that $h \geq \log N$. Matsubara et al. [18] showed an $O(nh(n + h \log N))$-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 $\mathsf{LCE}(val(X_i), val(X_j), 1, p_j)$ [20]. This data structure takes $O(n^2)$ space and can be constructed in $O(n^2 h)$ time [16]. Using Theorem 21, we obtain a faster algorithm, as follows:

▶ **Theorem 22.** *Given an SLP of size n generating 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 $O(n \log^* N + w)$ space.*

Our data structures also solve *the grammar compressed dictionary matching problem* [15].

▶ **Theorem 23.** *Given a DSLP $\langle \mathcal{S}, m \rangle$ of size n that represents a dictionary $\Pi_{\langle \mathcal{S}, m \rangle}$ 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 [15] 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 $\langle \mathcal{S}, m \rangle$. Note that our data structure of Theorem 23 is always smaller, and runs faster when $h = \omega(\log m \log N \log^* N)$.

References

1. Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Dynamic pattern matching. Technical report, Department of Computer Science, University of Copenhagen, 1998.
2. Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proc. SODA 2000*, pages 819–828, 2000.
3. Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002. `doi:10.1006/jcss.2002.1822`.
4. M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.
5. P. Bille, P. H. Cording, I. L. Gørtz, B. Sach, H. W. Vildhøj, and Søren Vind. Fingerprints in compressed strings. In *Proc. WADS 2013*, pages 146–157, 2013.
6. Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search, random access, and longest common extensions in grammar-compressed strings. *CoRR*, abs/1507.02853, 2015. URL: `http://arxiv.org/abs/1507.02853`.
7. Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 – July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2015. `doi:10.1007/978-3-319-19929-0_6`.
8. Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.
9. Johannes Fischer, Tomohiro I, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *LATIN 2016: Theoretical Informatics – 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, pages 483–496, 2016.
10. Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. *CoRR*, abs/1511.02612, 2015. URL: `http://arxiv.org/abs/1511.02612`.
11. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
12. Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Proc. STOC 2002*, pages 602–608, 2002.
13. Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015.
14. Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theoretical Computer Science*, 2016. in press. `doi:10.1016/j.tcs.2016.03.005`.

15 Tomohiro I, Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Compressed automata for dictionary matching. *Theor. Comput. Sci.*, 578:30–41, 2015. doi:10.1016/j.tcs.2015.01.019.

16 Yury Lifshits. Processing compressed texts: A tractability border. In *Proc. CPM 2007*, volume 4580 of *LNCS*, pages 228–240, 2007.

17 S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.

18 W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8–10):900–913, 2009.

19 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

20 M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. CPM 1997*, pages 1–11, 1997.

21 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. *CoRR*, abs/1605.01488, 2016. URL: http://arxiv.org/abs/1605.01488.

22 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

23 S. C Sahinalp and Uzi Vishkin. Data compression using locally consistent parsing. *TechnicM report, University of Maryland Department of Computer Science*, 1995.

24 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Proc. CPM 2016*, 2016. to appear.

25 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–349, 1977.