Dynamic Index and LZ Factorization 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] Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp

Abstract. In this paper, we propose a new *dynamic compressed index* of $O(w)$ space for a dynamic text T, where $w = O(\min(z \log N \log^* M, N))$ is the size of the signature encoding of T, z is the size of the Lempel-Ziv77 (LZ77) factorization of T, N is the length of T, and $M \geq 4N$ is an integer that can be handled in constant time under word RAM model. Our index supports searching for a pattern P in T in $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* M(\log N + \log |P| \log^* M) + occ \log N)$ 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 $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$. Also, we propose a new space-efficient LZ77 factorization algorithm for a given text of length N, which runs in $O(N f_{\mathcal{A}} + z \log w \log^3 N (\log^* N)^2)$ time with $O(w)$ working space.

1 Introduction

1.1 Dynamic compressed index

Given a text T, the string indexing problem is to construct a data structure, called an index, so that querying occurrences of a given pattern in T can be answered efficiently. As the size of data is growing rapidly in the last decade, many recent studies have focused on indexes working in compressed text space (see e.g. [11,12,7,6]). However most of them are static, i.e., they have to be reconstructed from scratch when the text is modified, which makes difficult to apply them to a dynamic text. Hence, 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 exists several dynamic *non-compressed* text indexes (see e.g. [24,3,9] for recent work), there has been little work for the compressed variants. Hon et al. [15] proposed the first dynamic compressed index of $O(\frac{1}{\epsilon}(NH_0 + N))$ bits of space which supports searching of P in $O(|P| \log^2 N(\log^\epsilon N + \log |\Sigma|) + occ \log^{1+\epsilon} N)$ time and insertion/deletion of a substring of length y in $O((y + \sqrt{N}) \log^{2+\epsilon} N)$ amortized time, where $0 < \epsilon \leq 1$ and $H_0 \leq \log |\Sigma|$ denotes the zeroth order empirical entropy of the text of length N [15]. Salson et al. [26] 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, $w = O(\min(z \log N \log^* M, N))$ the size of the signature encoding of T, and z the number of factors in the Lempel-Ziv 77 factorization of T without self-references. Then, there exists a dynamic index of $O(w)$ space*

Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Dynamic Index and LZ Factorization in Compressed Space, pp. 158–171.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

which supports searching of a pattern P in $O(|P|f_{\mathcal{A}} + \log w \log |P| \log^* M(\log N + \log |P| \log^* M) + occ \log N)$ time, where $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$, and insertion/deletion of a (sub)string Y into/from an arbitrary position of T in amortized $O((|Y| + \log N \log^* M) \log w \log N \log^* M)$ time. Moreover, if Y is given as a substring of T, we can support insertion in amortized $O(\log w(\log N \log^* M)^2)$ time.

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. To achieve the above result, technically speaking, we use the *signature encoding* $\mathcal{G}$ of T, which is based on the *locally consistent parsing* technique. The signature encoding was proposed by Mehlhorn et al. for equality testing on a dynamic set of strings [17]. Since then, the signature encoding and the related ideas have been used in many applications. In particular, Alstrup et al.'s proposed dynamic index (not compressed) which is based on the signature encoding of strings, while improving the update time of signature encodings [3] and the locally consistent parsing algorithm (details can be found in the technical report [2]).

Our data structure uses Alstrup et al.'s fast string concatenation/split algorithms (update algorithm) and linear-time computation of locally consistent parsing, but has little else in common than those. Especially, Alstrup et al.'s dynamic pattern matching algorithm [3,2] requires to maintain specific locations called *anchors* over the parse trees of the signature encodings, but our index does not use anchors. Our index has close relationship to the ESP-indices [27,28], but there are two significant differences between ours and ESP-indices: The first difference is that the ESP-index [27] is static and its online variant [28] allows only for appending new characters to the end of the text, while our index is fully dynamic allowing for insertion and deletion of arbitrary substrings at arbitrary positions. The second difference is that the pattern search time of the ESP-index is proportional to the number 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 occ is the number of occurrences of P in the text, then it always holds that $occ_c \geq occ$, and in general occ_c cannot be upper bounded by any function of occ. In contrast, as can be seen in Theorem 1, the pattern search time of our index is proportional to the number occ of occurrences of a query pattern P. This became possible due to our discovery of a new property of the signature encoding [2] (stated in Lemma 16).

As another application of signature encodings, Nishimoto et al. showed that signature encodings for a dynamic string T can support Longest Common Extension (LCE) queries on T efficiently in compressed space [20] (Lemma 10). They also showed signature encodings can be updated in compressed space (Lemma 12). Our algorithm uses properties of signature encodings shown in [20], more precisely, Lemmas 5-10 and 12, but Lemma 16 is a new property of signature encodings not described in [20].

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 [18] 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 Computing LZ77 factorization in compressed space.

As an application of our dynamic compressed index, we present a new LZ77 factorization algorithm working in compressed space.

The Lempel-Ziv77 (LZ77) factorization is defined as follows.

Definition 2 (Lempel-Ziv77 factorization [29]). *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.*

Although the primary use of LZ77 factorization is data compression, it has been shown that it is a powerful tool for many string processing problems [13,12]. Hence the importance of algorithms to compute LZ77 factorization is growing. Particularly, in order to apply algorithms to large scale data, reducing the working space is an important matter. In this paper, we focus on LZ77 factorization algorithms working in *compressed space.*

The following is our main result.

Theorem 3. *Given the signature encoding $\mathcal{G}$ of size w for a string T of length N, we can compute the LZ77 factorization of T in $O(z \log w \log^3 N (\log^* M)^2)$ time and $O(w)$ working space where z is the size of the LZ77 factorization of T.*

In [20], it was shown that the signature encoding $\mathcal{G}$ can be constructed efficiently from various types of inputs, in particular, in $O(N f_{\mathcal{A}})$ time and $O(w)$ working space from uncompressed string T. Therefore we can compute LZ77 factorization of a given T of length N in $O(N f_{\mathcal{A}} + z \log w \log^3 N (\log^* M)^2)$ time and $O(w)$ working space.

Related work. Goto et al. [14] showed how, given the grammar-like representation for string T generated by the LCA algorithm [25], to compute the LZ77 factorization of T in $O(z \log^2 m \log^3 N + m \log m \log^3 N)$ time and $O(m \log^2 m)$ space, where m is the size of the given representation. Sakamoto et al. [25] claimed that $m = O(z \log N \log^* N)$, 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. [14]. Recently, Fischer et al. [10] showed a Monte-Carlo randomized algorithms to compute an approximation of the LZ77 factorization with at most $2z$ factors in $O(N \log N)$ time, and another approximation with at most $(i+\epsilon)z$ factors in $O(N \log^2 N)$ time for any constant $\epsilon > 0$, using $O(z)$ space each.

Another line of research is LZ77 factorization working in compressed space in terms of Burrows-Wheeler transform (BWT) based methods. Policriti and Prezza recently proposed algorithms running in $NH_0 + o(N \log |\Sigma|) + O(|\Sigma| \log N)$ bits of space and $O(N \log N)$ time [21], or $O(R \log N)$ bits of space and $O(N \log R)$ time [22], where R is the number of runs in the BWT of the reversed string of T. Because their and our algorithms are established on different measures of compression, they cannot be easily compared. For example, our algorithm is more space efficient than the algorithm in [22] when $w = o(R)$, but it is not clear when it happens.

Examples and figures omitted due to lack of space are in a full version of this paper [19].

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 $\mathsf{LCP}(s_1[i..|s_1|], s_2[j..|s_2|])$. For any strings p and s, let $Occ(p, s)$ denote all occurrence positions of p in s, namely, $Occ(p, s) = \{i \mid p = s[i..i + |p| - 1], 1 \leq i \leq |s| - |p| + 1\}$. 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.

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*; $\mathcal{D} = \{X_i \to \mathit{expr}_i\}_{i=1}^{n}$ 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 $\mathit{val} : \mathcal{V} \to \Sigma^+$ be the function which returns the string derived by an input variable. If $s = \mathit{val}(X)$ for $X \in \mathcal{V}$, then we say that the variable X *represents* string s. For any variable sequence $y \in \mathcal{V}^+$, let $\mathit{val}^+(y) = \mathit{val}(y[1]) \cdots \mathit{val}(y[|y|])$. For any variable X_i with $X_i \to X_\ell X_r \in \mathcal{D}$, let $X_i.\mathrm{left} = \mathit{val}(X_\ell)$ and $X_i.\mathrm{right} = \mathit{val}(X_r)$, which are called the *left string* and the *right string* of X_i, respectively. For two variables $X_i, X_j \in \mathcal{V}$, we say that X_i occurs at position c in X_j if there is a node labeled with X_i in the derivation tree of X_j and the leftmost leaf of the subtree rooted at that node labeled with X_i is the c-th leaf in the derivation tree of X_j. We define the function $\mathit{vOcc}(X_i, X_j)$ which returns all positions of X_i in the derivation tree of X_j.

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 $\mathit{Assgn}_{\mathcal{G}}$ be the function such that $\mathit{Assgn}_{\mathcal{G}}(X_i) = \mathit{expr}_i$ iff $X_i \to \mathit{expr}_i \in \mathcal{D}$. Also, let $\mathit{Assgn}_{\mathcal{G}}^{-1}$ denote the reverse function of $\mathit{Assgn}_{\mathcal{G}}$. When clear from the context, we write $\mathit{Assgn}_{\mathcal{G}}$ and $\mathit{Assgn}_{\mathcal{G}}^{-1}$ as Assgn and Assgn^{-1}, respectively.

We define the left and right strings for any variable $X_i \to X_\ell X_r \in \mathcal{D}$ in a similar way to SLPs. Furthermore, for any $X \to \hat{X}^k \in \mathcal{D}$, let X.left $= val(\hat{X})$ and X.right $= val(\hat{X})^{k-1}$.

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 stores $|val(X)|$ 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. For $X \in \mathcal{V}$, let $parents(X)$ be the set of variables which have out-going edge to X in the DAG of $\mathcal{G}$. To compute $parents(X)$ for $X \in \mathcal{V}$ in linear time, we let X have a doubly-linked list of length $|parents(X)|$ to represent $parents(X)$: Each element is a pointer to a node for $X' \in parents(X)$ (the order of elements is arbitrary). Conversely, we let every parent X' of X have the pointer to the corresponding element in the list.

3 Signature encoding

Here, we recall the *signature encoding* first proposed by Mehlhorn et al. [17]. Its core technique is *locally consistent parsing* defined as follows:

Lemma 4 (Locally consistent parsing [17,2]). *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 4, 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 4 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 $Shrink_t^T = Sig^+(Eblock(Pow_{t-1}^T))$ 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 $|Pow_h^T| = 1$. Then, the start symbol of the signature encoding is $S = Pow_h^T$. 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 $id(T) = Pow_h^T = S$, i.e., the integer S is the signature of T. We let $N \leq M/4$. 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. Since all signatures are in $[1..M-1]$, we set $W = M$ in Lemma 4 used by the signature encoding. In this paper, we implement signature encodings by the DAG of RLSLP introduced in Section 2.

3.1 Commmon sequences

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 5 (common sequences [23,20]). *Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ 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, where $|Epow(Uniq(P))| = O(\log |P| \log^* M)$.*

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

Definition 6. *For a string P, let*

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

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

- *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 5. Recall that the locally consistent parsing of Lemma 4. Each i-th bit of bit sequence d of Lemma 4 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 which are 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 5 holds [1].

From the common sequences we can derive many useful properties of signature encodings like listed below (see the references for proofs).

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

Lemma 7 ([20]). *Let $\mathcal{G}$ 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.*

We can efficiently compute $Uniq(P)$ for a substring P of T.

Lemma 8 ([20]). *Using a signature encoding $\mathcal{G}$ 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 $s = val(e)$.*

The next lemma shows that $\mathcal{G}$ requires only *compressed space*:

Lemma 9 ([23,20]). *The size w of the signature encoding of T of length N is $O(\min(z\log N\log^* M, N))$, where z is the number of factors in the LZ77 factorization without self-reference of T.*

The next lemma shows that the signature encoding supports (both forward and backward) LCE queries on a given arbitrary pair of signatures.

Lemma 10 ([20]). *Using a signature encoding $\mathcal{G}$ 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 $1 \le i \le |s_1|$, $1 \le j \le |s_2|$, where $s_1 = val(e_1)$, $s_2 = val(e_2)$ and ℓ is the answer to the LCE query.*

3.2 Dynamic signature encoding

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 7).

[1] The common sequences are conceptually equivalent to the *cores* [16] which are defined for the *edit sensitive parsing* of a text, a kind of locally consistent parsing of the text.

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 e_{new}, which is an integer that is currently not used as signatures, 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.

We can upper bound the number of signatures added to or removed from $\mathcal{G}$ after a single update operation by the following lemma. [2]

Lemma 11. *After* $\mathit{INSERT}(Y, i)$ *or* $\mathit{DELETE}(j, y)$ *operation,* $O(y + \log N \log^* M)$ *signatures are added to or removed from* $\mathcal{G}$*, where* $|Y| = y$*. After* $\mathit{INSERT}'(j, y, i)$ *operation,* $O(\log N \log^* M)$ *signatures are added to or removed from* $\mathcal{G}$*.*

Proof. Consider $\mathit{INSERT}'(j, y, i)$ operation. Let $T' = T[..i-1]T[j..j+y-1]T[i..]$ be the new text. Note that by Lemma 5 the signature encoding of T' is created over $\mathit{Uniq}(T[..i-1])\mathit{Uniq}(T[j..j+y-1])\mathit{Uniq}(T[i..])$, and hence, $O(\log N \log^* M)$ signatures can be added by Lemma 7. Also, $O(\log N \log^* M)$ signatures, which were created over $\mathit{Uniq}(T[..i-1])\mathit{Uniq}(T[i..])$, may be removed.

For $\mathit{INSERT}(Y, i)$ operation, we additionally think about the possibility that $O(y)$ signatures are added to create $\mathit{Uniq}(Y)$. Similarly, for $\mathit{DELETE}(j, y)$ operation, $O(y)$ signatures, which are used in and under $\mathit{Uniq}(T[j..j+y-1])$, can be removed. ☐

In [20], it was shown how to augment the DAG representation of $\mathcal{G}$ to add/remove an assignment to/from $\mathcal{G}$ in $O(f_{\mathcal{A}})$ time, where $f_{\mathcal{A}} = O\left(\min\left\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\right\}\right)$ is the time complexity of Beame and Fich's data structure [4] to support predecessor/successor queries on a set of w integers from an M-element universe.[3] Note that there is a small difference in our DAG representation from the one in [20]; our DAG has a doubly-linked list representing the parents of a node. We can check if a signature is useless or not by checking if the list is empty or not, and the lists can be maintained in constant time after adding/removing an assignment. Hence, the next lemma still holds for our DAG representation.

Lemma 12 (Dynamic signature encoding [20]). *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. Moreover, if* Y *is given as a substring of* T*, we can support insertion in* $O(f_{\mathcal{A}} \log N \log^* M)$ *time.*

4 Dynamic Compressed Index

In this section, we present our dynamic compressed index based on signature encoding. As already mentioned in the introduction, our strategy for pattern matching is different from that of Alstrup et al. [2]. It is rather similar to the one taken in the static index for SLPs of Claude and Navarro [6]. Besides applying their idea to RLSLPs, we show how to speed up pattern matching by utilizing the properties of signature encodings.

Index for SLPs. Here we review how the index in [6] for SLP $\mathcal{S}$ generating a string T computes $Occ(P, T)$ for a given string P. The key observation is that, any occurrence

[2] The property is used in [20], but there is no corresponding lemma to state it clearly.

[3] The data structure is, for example, used to compute $Assgn^{-1}(\cdot)$. Alstrup et al. [2] used hashing for this purpose. However, since we are interested in the worst case time complexities, we use the data structure [4] in place of hashing.

of P in T can be uniquely associated with the lowest node that covers the occurrence of P in the derivation tree. As the derivation tree is binary, if $|P| > 1$, then the node is labeled with some variable $X \in \mathcal{V}$ such that P_1 is a suffix of X.left and P_2 is a prefix of X.right, where $P = P_1P_2$ with $1 \leq |P_1| < |P|$. Here we call the pair $(X, |X.\mathrm{left}| - |P_1| + 1)$ a *primary occurrence* of P, and let $pOcc_{\mathcal{S}}(P, j)$ denote the set of such primary occurrences with $|P_1| = j$. The set of all primary occurrences is denoted by $pOcc_{\mathcal{S}}(P) = \bigcup_{1 \leq j < |P|} pOcc_{\mathcal{S}}(P, j)$. Then, we can compute $Occ(P, T)$ by first computing primary occurrences and enumerating the occurrences of X in the derivation tree.

The set $Occ(P, T)$ of occurrences of P in T is represented by $pOcc_{\mathcal{S}}(P)$ as follows: $Occ(P, T) = \{j + k - 1 \mid (X, j) \in pOcc_{\mathcal{S}}(P), k \in vOcc(X, S)\}$ if $|P| > 1$; $Occ(P, T) = vOcc(X, S)((X \to P) \in \mathcal{D})$ if $|P| = 1$.

Hence the task is to compute $pOcc_{\mathcal{S}}(P)$ and $vOcc(X, S)$ efficiently. Note that $vOcc(X, S)$ can be computed in $O(|vOcc(X, S)|h)$ time by traversing the DAG in a reversed direction from X to the source, where h is the height of the derivation tree of S. Hence, in what follows, we explain how to compute $pOcc_{\mathcal{S}}(P)$ for a string P with $|P| > 1$. We consider the following problem:

Problem 13 (Two-Dimensional Orthogonal Range Reporting Problem). Let $\mathcal{X}$ and $\mathcal{Y}$ denote subsets of two ordered sets, and let $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{Y}$ be a set of points on the two-dimensional plane, where $|\mathcal{X}|, |\mathcal{Y}| \in O(|\mathcal{R}|)$. A data structure for this problem supports a query $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$; given a rectangle (x_1, x_2, y_1, y_2) with $x_1, x_2 \in \mathcal{X}$ and $y_1, y_2 \in \mathcal{Y}$, returns $\{(x, y) \in \mathcal{R} \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$.

Data structures for Problem 13 are widely studied in computational geometry. There is even a dynamic variant, which we finally use for our dynamic index. Until then, we just use any data structure that occupies $O(|\mathcal{R}|)$ space and supports queries in $O(\hat{q}_{|\mathcal{R}|} + q_{|\mathcal{R}|} qocc)$ time with $\hat{q}_{|\mathcal{R}|} = O(\log |\mathcal{R}|)$, where $qocc$ is the number of points to report.

Now, given an SLP $\mathcal{S}$, we consider a two-dimensional plane defined by $\mathcal{X} = \{X.\mathrm{left}^R \mid X \in \mathcal{V}\}$ and $\mathcal{Y} = \{X.\mathrm{right} \mid X \in \mathcal{V}\}$, where elements in $\mathcal{X}$ and $\mathcal{Y}$ are sorted by lexicographic order. Then consider a set of points $\mathcal{R} = \{(X.\mathrm{left}^R, X.\mathrm{right}) \mid X \in \mathcal{V}\}$. For a string P and an integer $1 \leq j < |P|$, let $y_1^{(P,j)}$ (resp. $y_2^{(P,j)}$) denote the lexicographically smallest (resp. largest) element in $\mathcal{Y}$ that has $P[j+1..]$ as a prefix. If there is no such element, it just returns NIL and we can immediately know that $pOcc_{\mathcal{S}}(P, j) = \emptyset$. We define $x_1^{(P,j)}$ and $x_2^{(P,j)}$ in a similar way over $\mathcal{X}$. Then, $pOcc_{\mathcal{S}}(P, j)$ can be computed by a query $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$.

Using this idea, we can get the next result:

Lemma 14. *For an SLP $\mathcal{S}$ of size n, there exists a data structure of size $O(n)$ that computes, given a string P, $pOcc_{\mathcal{S}}(P)$ in $O(|P|(h + |P|) \log n + q_n |pOcc_{\mathcal{S}}(P)|)$ time.*

Proof. For every $1 \leq j < |P|$, we compute $pOcc_{\mathcal{S}}(P, j)$ by $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$. We can compute $y_1^{(P,j)}$ and $y_2^{(P,j)}$ in $O((h + |P|) \log n)$ time by binary search on $\mathcal{Y}$, where each comparison takes $O(h + |P|)$ time for expanding the first $O(|P|)$ characters of variables subjected to comparison. In a similar way, $x_1^{(P,j)}$ and $x_2^{(P,j)}$ can be computed in $O((h+|P|) \log n)$ time. Thus, the total time complexity is $O(|P|((h+|P|) \log n + \hat{q}_n) + q_n |pOcc_{\mathcal{S}}(P)|) = O(|P|(h + |P|) \log n + q_n |pOcc_{\mathcal{S}}(P)|)$. □

Index for RLSLPs. We extend the idea for the SLP index described above to RLSLPs. The difference from SLPs is that we have to deal with occurrences of P

that are covered by a node labeled with $X \to \hat{X}^k$ but not covered by any single child of the node in the derivation tree. In such a case, there must exist $P = P_1P_2$ with $1 \leq |P_1| < |P|$ such that P_1 is a suffix of X.left $= val^+(\hat{X})$ and P_2 is a prefix of X.right $= val^+(\hat{X}^{k-1})$. Let $j = |val(\hat{X})| - |P_1| + 1$ be a position in $val^+(\hat{X}^d)$ where P occurs, then P also occurs at $j + c|val(\hat{X})|$ in $val^+(\hat{X}^k)$ for every positive integer c with $j + c|val(\hat{X})| + |P| - 1 \leq |val^+(\hat{X}^k)|$. Using this observation, the index for SLPs can be modified for RLSLPs to achieve the same bounds as in Lemma 14.

Index for signature encodings. Since signature encodings are RLSLPs, we can compute $Occ(P, T)$ by querying $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$ for "every" $1 \leq j < |P|$. However, the properties of signature encodings allow us to speed up pattern matching as summarized in the following two ideas: (1) We can efficiently compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ using LCE queries in compressed space (Lemma 15). (2) We can reduce the number of $report_{\mathcal{R}}$ queries from $O(|P|)$ to $O(\log |P| \log^* M)$ by using the property of the common sequence of P (Lemma 16).

Lemma 15. *Assume that we have the signature encoding $\mathcal{G}$ of size w for a string T of length N, $\mathcal{X}$ and $\mathcal{Y}$ of $\mathcal{G}$. Given a signature $id(P) \in \mathcal{V}$ for a string P and an integer j, we can compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ in $O(\log w(\log N + \log |P| \log^* M))$ time.*

Proof. By Lemma 10 we can compute $x_1^{(P,j)}$ and $x_2^{(P,j)}$ on $\mathcal{X}$ by binary search in $O(\log w(\log N + \log |P| \log^* M))$ time. Similarly, we can compute $y_1^{(P,j)}$ and $y_2^{(P,j)}$ in the same time. □

Lemma 16. *Let P be a string with $|P| > 1$. If $|Pow_0^P| = 1$, then $pOcc_{\mathcal{G}}(P) = pOcc_{\mathcal{G}}(P, 1)$. If $|Pow_0^P| > 1$, then $pOcc_{\mathcal{G}}(P) = \bigcup_{j \in \mathcal{P}} pOcc_{\mathcal{G}}(P, j)$, where $\mathcal{P} = \{|val^+(u[1..i])| \mid 1 \leq i < |u|, u[i] \neq u[i+1]\}$ with $u = Uniq(P)$.*

Proof. If $|Pow_0^P| = 1$, then $P = a^{|P|}$ for some character $a \in \Sigma$. In this case, P must be contained in a node labeled with a signature $e \to \hat{e}^d$ such that $\hat{e} \to a$ and $d \geq |P|$. Hence, all primary occurrences of P can be found by $pOcc_{\mathcal{G}}(P, 1)$.

If $|Pow_0^P| > 1$, we consider the common sequence u of P. Recall that substring P occurring at j in $val(e)$ is represented by u for any $(e, j) \in pOcc(P)$ by Lemma 5 Hence at least $pOcc_{\mathcal{G}}(P) = \bigcup_{i \in \mathcal{P}'} pOcc_{\mathcal{G}}(P, i)$ holds, where $\mathcal{P}' = \{|val^+(u[1])|, \ldots, |val^+(u[..|u| - 1])|\}$. Moreover, we show that $pOcc_{\mathcal{G}}(P, i) = \emptyset$ for any $i \in \mathcal{P}'$ with $u[i] = u[i+1]$. Note that $u[i]$ and $u[i+1]$ are encoded into the same signature in the derivation tree of e, and that the parent of two nodes corresponding to $u[i]$ and $u[i+1]$ has a signature e' in the form $e' \to u[i]^d$. Now assume for the sake of contradiction that $e = e'$. By the definition of the primary occurrences, $i = 1$ must hold, and hence, $Shrink_0^P[1] = u[1] \in \Sigma$. This means that $P = u[1]^{|P|}$, which contradicts $|Pow_0^P| > 1$. Therefore the statement holds. □

Using Lemmas 5, 15 and 16, we get a static index for signature encodings:

Lemma 17. *For a signature encoding $\mathcal{G}$ of size w which generates a text T of length N, there exists a data structure of size $O(w)$ that computes, given a string P, $pOcc_{\mathcal{G}}(P)$ in $O(|P|f_{\mathcal{A}} + \log w \log |P| \log^* M(\log N + \log |P| \log^* M) + q_w|pOcc_{\mathcal{S}}(P)|)$ time.*

Proof. We focus on the case $|Pow_0^P| > 1$ as the other case is easier to be solved. We first compute the common sequence of P in $O(|P|f_{\mathcal{A}})$ time. Taking $\mathcal{P}$ in Lemma 16, we recall that $|\mathcal{P}| = O(\log|P|\log^* M)$ by Lemma 5. Then, in light of Lemma 16, $pOcc_{\mathcal{G}}(P)$ can be obtained by $|\mathcal{P}| = O(\log|P|\log^* M)$ range reporting queries. For each query, we spend $O(\log w(\log N+\log|P|\log^* M))$ time to compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ by Lemma 15. Hence, the total time complexity is

$$\begin{aligned} &O(|P|f_{\mathcal{A}} + \log|P|\log^* M(\log w(\log N + \log|P|\log^* M) + \hat{q}_w) + q_w|pOcc_{\mathcal{S}}(P)|) \\ &\quad = O(|P|f_{\mathcal{A}} + \log w\log|P|\log^* M(\log N + \log|P|\log^* M) + q_w|pOcc_{\mathcal{S}}(P)|). \end{aligned}$$

□

In order to dynamize our index of Lemma 17, we consider a data structure for "dynamic" two-dimensional orthogonal range reporting that can support the following update operations:

– $insert_{\mathcal{R}}(p, x_{pred}, y_{pred})$: given a point $p = (x, y)$, $x_{pred} = \max\{x' \in \mathcal{X} \mid x' \le x\}$ and $y_{pred} = \max\{y' \in \mathcal{Y} \mid y' \le y\}$, insert p to $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.
– $delete_{\mathcal{R}}(p)$: given a point $p = (x, y) \in \mathcal{R}$, delete p from $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.

We use the following data structure for the dynamic two-dimensional orthogonal range reporting.

Lemma 18 ([5]). *There exists a data structure that supports $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$ in $O(\log|\mathcal{R}|+occ(\log|\mathcal{R}|/\log\log|\mathcal{R}|))$ time, and $insert_{\mathcal{R}}(p, i, j)$, $delete_{\mathcal{R}}(p)$ in amortized $O(\log|\mathcal{R}|)$ time, where occ is the number of the elements to output. This structure uses $O(|\mathcal{R}|)$ space.* [4]

Proof (Proof of Theorem 1). Our index consists of a dynamic signature encoding $\mathcal{G}$ and a dynamic range reporting data structure of Lemma 18 whose $\mathcal{R}$ is maintained as they are defined in the static version. We maintain $\mathcal{X}$ and $\mathcal{Y}$ in two ways; self-balancing binary search trees for binary search, and Dietz and Sleator's data structures for order maintenance. Then, primary occurrences of P can be computed as described in Lemma 17. Adding the $O(occ\log N)$ term for computing all pattern occurrences from primary occurrences, we get the time complexity for pattern matching in the statement.

Concerning the update of our index, we described how to update $\mathcal{G}$ after $INSERT$, $INSERT'$ and $DELETE$ in Lemma 12. What remains is to show how to update the dynamic range reporting data structure when a signature is added to or deleted from $\mathcal{V}$. When a signature e is deleted from $\mathcal{V}$, we first locate $e.\mathrm{left}^R$ on $\mathcal{X}$ and $e.\mathrm{right}$ on $\mathcal{Y}$, and then execute $delete_{\mathcal{R}}(e.\mathrm{left}^R, e.\mathrm{right})$. When a signature e is added to $\mathcal{V}$, we first locate $x_{pred} = \max\{x' \in \mathcal{X} \mid x' \le e.\mathrm{left}^R\}$ on $\mathcal{X}$ and $y_{pred} = \max\{y' \in \mathcal{Y} \mid y' \le e.\mathrm{right}\}$ on $\mathcal{Y}$, and then execute $insert_{\mathcal{R}}((e.\mathrm{left}^R, e.\mathrm{right}), x_{pred}, y_{pred})$. The locating can be done by binary search on $\mathcal{X}$ and $\mathcal{Y}$ in $O(\log w\log N\log^* M)$ time as Lemma 15.

Since the number of signatures added to or removed from $\mathcal{G}$ during a single update operation is upper bounded by Lemma 11, we can get the desired time bounds of Theorem 1. □

[4] The original problem considers a real plane in the paper [5], however, his solution only need to compare any two elements in $\mathcal{R}$ in constant time. Hence his solution can apply to our range reporting problem by maintains $\mathcal{X}$ and $\mathcal{Y}$ using the data structure of order maintenance problem proposed by Dietz and Sleator [8], which enables us to compare any two elements in a list L and insert/delete an element to/from L in constant time.

5 LZ77 factorization in compressed space

In this section, we show Theorem 3. Note that since each f_i can be represented by the pair $(x_i, |f_i|)$, we compute incrementally $(x_i, |f_i|)$ in our algorithm, where x_i is an occurrence position of f_i in $f_1 \cdots f_{i-1}$.

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. Our algorithm is based on the following fact:

Fact 1 *Let $f_1, \ldots, f_z$ be the LZ77-factorization of a string T. Given $f_1, \ldots, f_{i-1}$, 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 $j = |f_1 \cdots f_{i-1}| + 1$.*

We explain how to support queries $Fst(j,k)$ using the signature encoding. We define $e.\text{min} = \min vOcc(e,S) + |e.\text{left}|$ for a signature $e \in \mathcal{V}$ with $e \to e_\ell e_r$ or $e \to \hat{e}^k$. We also define $FstOcc(P,i)$ for a string P and an integer i as follows:

$$FstOcc(P,i) = \min\{e.\text{min} \mid (e,i) \in pOcc_{\mathcal{G}}(P,i)\}$$

Then $Fst(j,k)$ can be represented by $FstOcc(P,i)$ as follows:

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

where $\mathcal{P}$ is the set of integers in Lemma 16 with $P = T[j..j+k-1]$.

Recall that in Section 4 we considered the two-dimensional orthogonal range reporting problem to enumerate $pOcc_{\mathcal{G}}(P,i)$. Note that $FstOcc(P,i)$ can be obtained by taking $(e,i) \in pOcc_{\mathcal{G}}(P,i)$ with $e.\text{min}$ minimum. In order to compute $FstOcc(P,i)$ efficiently instead of enumerating all elements in $pOcc_{\mathcal{G}}(P,i)$, we give every point corresponding to e the weight $e.\text{min}$ and use the next data structure to compute a point with the minimum weight in a given rectangle.

Lemma 19 ([1]). *Consider n weighted points on a two-dimensional plane. There exists a data structure which supports the query to return a point with the minimum weight in a given rectangle in $O(\log^2 n)$ time, occupies $O(n)$ space, and requires $O(n \log n)$ time to construct.*

Using Lemma 19, we get the following lemma.

Lemma 20. *Given a signature encoding $\mathcal{G}$ of size w which generates T, we can construct a data structure of $O(w)$ space in $O(w \log w \log N \log^* M)$ time to support queries $Fst(j,k)$ in $O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time.*

Proof. For construction, we first compute $e.\text{min}$ in $O(w)$ time using the DAG of $\mathcal{G}$. Next, we prepare the plane defined by the two ordered sets $\mathcal{X}$ and $\mathcal{Y}$ in Section 4. This can be done in $O(w \log w \log N \log^* M)$ time by sorting elements in $\mathcal{X}$ (and $\mathcal{Y}$) by LCE algorithm (Lemma 10) and a standard comparison-based sorting. Finally we build the data structure of Lemma 19 in $O(w \log w)$ time.

To support a query $Fst(j,k)$, we first compute $Epow(Uniq(P))$ with $P = T[j..j+k-1]$ in $O(\log N + \log k \log^* M)$ time by Lemma 8, and then get $\mathcal{P}$ in Lemma 16. Since $|\mathcal{P}| = O(\log k \log^* M)$ by Lemma 5, $Fst(j,k) = \min\{FstOcc(P,i) - i \mid i \in \mathcal{P}\}$ can be computed by answering $FstOcc$ $O(\log k \log^* M)$ times. For each computation of $FstOcc(P,i)$, we spend $O(\log w(\log N + \log k \log^* M))$ time to compute

$x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ by Lemma 15, and $O(\log^2 w)$ time to compute a point with the minimum weight in the rectangle $(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$. Hence it takes $O(\log k \log^* M(\log w(\log N + \log k \log^* M) + \log^2 w)) = O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time in total. $\square$

We are ready to prove Theorem 3 holds.

Proof (Proof of Theorem 3). We compute the z factors of the LZ77-factorization of T incrementally by using Fact 1 and Lemma 20 in $O(z \log w \log^3 N (\log^* M)^2)$ time. Therefore the statement holds. $\square$

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 21 (Lempel-Ziv77 factorization with self-reference [29]). *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 $s[|f_1..f_{i-1}|+1]$ 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 $f_i \cdots f_k$ which occurs at some position p, where $1 \leq p \leq |f_1 \cdots f_{i-1}|$.*

Acknowledgments. We would like to thank Paweł Gawrychowski for drawing our attention to the work by Alstrup et al. [2,3] and for fruitful discussions.

References

1. P. K. Agarwal, L. Arge, S. Govindarajan, J. Yang, and K. Yi: *Efficient external memory structures for range-aggregate queries.* Comput. Geom., 46(3) 2013, pp. 358–370.
2. S. Alstrup, G. S. Brodal, and T. Rauhe: *Dynamic pattern matching*, tech. rep., Department of Computer Science, University of Copenhagen, 1998.
3. S. Alstrup, G. S. Brodal, and T. Rauhe: *Pattern matching in dynamic texts*, in Proc. SODA 2000, 2000, pp. 819–828.
4. P. Beame and F. E. Fich: *Optimal bounds for the predecessor problem and related problems.* J. Comput. Syst. Sci., 65(1) 2002, pp. 38–72.
5. G. E. Blelloch: *Space-efficient dynamic orthogonal point location, segment intersection, and range reporting*, in SODA, S.-H. Teng, ed., SIAM, 2008, pp. 894–903.
6. F. Claude and G. Navarro: *Self-indexed grammar-based compression.* Fundamenta Informaticae, 111(3) 2011, pp. 313–337.
7. F. Claude and G. Navarro: *Improved grammar-based compressed indexes*, in SPIRE'12, 2012, pp. 180–192.
8. P. F. Dietz and D. D. Sleator: *Two algorithms for maintaining order in a list*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, A. V. Aho, ed., ACM, 1987, pp. 365–372.
9. A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S. Woo: *Position heaps: A simple and dynamic text indexing data structure.* J. Discrete Algorithms, 9(1) 2011, pp. 100–121.
10. J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka: *Approximating LZ77 via small-space multiple-pattern matching*, in ESA 2015, 2015, pp. 533–544.
11. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *A faster grammar-based self-index*, in LATA'12, 2012, pp. 240–251.
12. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *LZ77-based self-indexing with faster pattern matching*, in Proc. LATIN 2014, 2014, pp. 731–742.
13. T. Gagie, P. Gawrychowski, and S. J. Puglisi: *Approximate pattern matching in lz77-compressed texts.* J. Discrete Algorithms, 32 2015, pp. 64–68.

14. K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda: *Restructuring compressed texts without explicit decompression.* CoRR, abs/1107.2729 2011.
15. W. Hon, T. W. Lam, K. Sadakane, W. Sung, and S. Yiu: *Compressed index for dynamic text*, in DCC 2004, 2004, pp. 102–111.
16. S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto: *ESP-index: A compressed index based on edit-sensitive parsing.* J. Discrete Algorithms, 18 2013, pp. 100–112.
17. K. Mehlhorn, R. Sundar, and C. Uhrig: *Maintaining dynamic sequences under equality tests in polylogarithmic time.* Algorithmica, 17(2) 1997, pp. 183–198.
18. J. I. Munro, Y. Nekrich, and J. S. Vitter: *Dynamic data structures for document collections and graphs.* CoRR, abs/1503.05977 2015.
19. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Dynamic index and LZ factorization in compressed space.* CoRR, abs/1605.09558 2016.
20. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Fully dynamic data structure for LCE queries in compressed space.* CoRR, abs/1605.01488 2016.
21. A. Policriti and N. Prezza: *Fast online lempel-ziv factorization in compressed space*, in String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings, C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, eds., vol. 9309 of Lecture Notes in Computer Science, Springer, 2015, pp. 13–20.
22. A. Policriti and N. Prezza: *Computing LZ77 in run-compressed space*, in 2016 Data Compression Conference (DCC 2016), 2016, pp. 23–32, to appear.
23. S. C. Sahinalp and U. Vishkin: *Data compression using locally consistent parsing.* TechnicM report, University of Maryland Department of Computer Science, 1995.
24. S. C. Sahinalp and U. Vishkin: *Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract)*, in FOCS, IEEE Computer Society, 1996, pp. 320–328.
25. H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono: *A space-saving approximation algorithm for grammar-based compression.* IEICE Transactions, 92-D(2) 2009, pp. 158–165.
26. M. Salson, T. Lecroq, M. Léonard, and L. Mouchard: *Dynamic extended suffix arrays.* J. Discrete Algorithms, 8(2) 2010, pp. 241–257.
27. Y. Takabatake, Y. Tabei, and H. Sakamoto: *Improved esp-index: A practical self-index for highly repetitive texts*, in Proc. SEA 2014, 2014, pp. 338–350.
28. Y. Takabatake, Y. Tabei, and H. Sakamoto: *Online self-indexed grammar compression*, in SPIRE 2015, 2015, pp. 258–269.
29. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.