A compressed dynamic self-index for highly repetitive text collections

Takaaki Nishimoto[1]   Yoshimasa Takabatake[2] and Yasuo Tabei[1]

[1] RIKEN Center for Advanced Intelligence Project, Chuo-ku, Tokyo, Japan

{takaaki.nishimoto, yasuo.tabei}@riken.jp

[2] Kyushu Institute of Technology, Fukuoka, Japan

takabatake@ai.kyutech.ac.jp

Abstract

We present a novel compressed dynamic self-index for highly repetitive text collections. Signature encoding, an existing self-index of this type, has a large disadvantage of slow pattern search for short patterns. We obtain faster pattern search by leveraging the idea behind a truncated suffix tree (TST) to develop the first compressed dynamic self-index, called the *TST-index*, that supports not only fast pattern search but also dynamic update operations for highly repetitive texts. Experiments with a benchmark dataset show that the pattern search performance of the TST-index is significantly improved.

1   Introduction

A *highly repetitive text collection* is a set of texts such that any two texts can be converted into each other with a few modifications. Such collections have become common in various fields of research and industry. Examples include genome sequences for the same species, version-controlled documents, and source code repositories. For human genome sequences, it is said that the difference between individual human genomes is around 0.1%, and there is a huge collection of human genomes, such as the 1000 Genome Project [1]. As another example, Wikipedia belongs to the category of version-controlled documents. There is clearly a strong, growing need for powerful methods to process huge collections of repetitive texts.

A *self-index* is a data representation for a text with support for random access and pattern search operations. Quite a few self-indexes for highly repetitive texts have been proposed thus far. Early methods include the SLP-index [5] and LZ-end [9]. The block tree index (BT-index) [14] is a recently proposed self-index that reaches compression close to that of Lempel-Ziv. The run-length FM-index (RLFM-index) [7] is a recent breakthrough built on the notion of a run-length-encoded Burrows-Wheeler transform (BWT). The RLFM-index can be constructed in a space-efficient, online manner while supporting fast operations. Although existing self-indexes for highly repetitive texts are constructible in a compressed space and also support fast operations, there are no proposed methods supporting dynamic updating of self-indexes for editing a set of texts with a highly repetitive structure. Thus, an important open challenge is to develop a self-index for highly repetitive texts that supports not only fast operations for random access and pattern search but also dynamic updating of the index.

The *ESP-index* [18] and *signature encoding* [15] are two self-indexes using the notion of a *locally consistent parsing (LCPR)* [11] for highly repetitive texts. While they have the advantages of being constructible in an online manner and supporting dynamic updates of data structures, they have a large disadvantage of slow pattern search for short patterns. From a given string, these methods build a parse tree that guarantees upper bounds for parsing discrepancies between different appearances of the same substring. For pattern search, the ESP-index performs top-down search of the parse tree to find candidate pattern appearances, and then it checks whether the pattern does occur around each candidate. In contrast, signature encoding uses a 2D range of reporting queries for pattern search. Traversing the search space, especially for short patterns, takes a long time for both methods, which limits their applicability in practice.

In this paper, we present a novel, compressed dynamic index for highly repetitive text collections, called a *truncated suffix tree- (TST-) based index (TST-index)*. The TST-index improves on existing

self-indexes on an LCPR to support faster pattern search by leveraging the idea behind a *q-truncated suffix tree (TST)* [13]. A *q*-TST is built from an input text, and a self-index is created for the transformed text by using the *q*-TST, which greatly improves the search performance. In addition, the TST-index supports dynamic updates, which is a useful property when input texts can be edited, as noted above. Experiments with a benchmark dataset of highly repetitive texts show that pattern search with the TST-index is much faster than that with signature encoding. In addition, the TST-index has pattern search performance competitive with that of the RLFM-index, yet the size of the TST-index can be smaller than that of the RLFM-index.

2 Preliminaries

Let Σ be an ordered alphabet and $\sigma = |\Sigma|$. Let string T be an an element in Σ^*, with its length denoted by $|T|$. In general, a string $P \in \Sigma^*$ is called a *pattern*. Let $N = |T|$ and $m = |P|$ throughout this paper. For string $T = xyz$, x, y, and z are called a prefix, substring, and suffix of T, respectively. For two strings x and y, let $x \cdot y = xy$.

The empty string ε is a string of length 0. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \le i \le |T|$, $T[i]$ denotes the i-th character of T. For any $1 \le i \le j \le |T|$, $T[i..j]$ denotes the substring of T that begins at position i and ends at position j. Let $T[i..] = T[i..|T|]$ and $T[..i] = T[1..i]$ for any $1 \le i \le |T|$. For strings T and K and indexes i and k, we define insertion and deletion operations as follows: $insert(T, i, K) = T[..i-1] \cdot K \cdot T[i..]$ and $delete(T, i, k) = T[..i-1] \cdot T[i+k-1..]$. For any strings P and T, let $Occ(P, T)$ denote all occurrence positions of P in T; that is, $Occ(P, T) = \{i \mid P = T[i..i + |P| - 1], 1 \le i \le |T| - |P| + 1\}$. Let $occ = |Occ(P, T)|$. Similarly, $cOcc(c, T) = Occ(c, T)$ for $c \in \Sigma$.

For a string P and integer q, we say that P is a *q-gram* if $|P| = q$. We then say that P is a *q-short pattern* if $|P| \le q$, or a *q-long pattern* if $|P| > q$. Similarly, we say that a suffix of length at most q of P is a *q-short suffix* of P. Let Σ_T^q be the set of all substrings of length q and all suffixes of length at most q in T, i.e., $\Sigma_T^q = \{T[i.. \min\{i + q - 1, |T|\}] \mid i \in 1 \le i \le |T|\}$. For example, if $T = bababababbabab\$$ and $q = 4$, then $\Sigma_T^q = \{\$, ab\$, abab, abba, b\$, bab\$, baba, babb, bbab\}$. For a string $T \in \Sigma^+$, we say that T is a $|\Sigma|$-*colored sequence* if $T[i] \ne T[i+1]$ holds for any $1 \le i < |T|$.

For a string T and an integer $1 < i \le |T|$, the longest prefix f of $T[i..]$ such that f occurs in $T[..i]$ is called the *longest previous factor without self-reference* at position i of T. Furthermore, $f_1, \ldots, f_d$ is called a *factorization* of T if $f_1 \cdots f_d = T$ holds, where factor f_i for each $i \in \{1, 2, ..., d\}$ is a substring of T and d is the size of the factorization. Then, the run-length encoding $RLE(T)$ is a factorization of string T such that each factor is a maximal run of the same character in T. Such a run is denoted as a^k for length k and character $a \in \Sigma$. For example, for $T = aabbbbbabb$, $RLE(T) = a^2 b^5 a^1 b^2$. $RLE(T)$ is a $|RLE(T)|$-colored sequence when we treat each factor as a character.

Next, the *Lempel-Ziv77 (LZ77) factorization without self-reference* [21] for a string T is a factorization $\mathsf{LZ}(T) = f_1, \ldots, f_z$ satisfying the following conditions: (1) $f_1 \cdots f_z = T$; (2) $f_1 = T[1]$; (3) f_i is the longest previous factor without self-reference at position $\ell = |f_1 \cdots f_{i-1}| + 1$ of T, if it exists for $1 < i \le z$; and (4) otherwise, $f_i = T[\ell]$. Hence, z is the number of factors in LZ77, i.e., $z = |\mathsf{LZ}(T)|$. For example, if $T = ababcababcabababcd$, then $\mathsf{LZ}(T) = a, b, ab, ab, c, ababc, ababc, d$.

A self-index is a data structure built on T and supporting the following operations:

- **Count:** Given a pattern P, count the number of appearances of P in T.
- **Locate:** Given a pattern P, return all positions at which P appears in T.
- **Extract:** Given a range $[i..i + \ell - 1]$, return $T[i..i + \ell - 1]$.

Such a self-index is considered static. In contrast, a dynamic self-index supports not only the above three operations but also an update operation on the self-index for insertion and deletion of a (sub)string of length k on string T of maximal length $M \ge N$. A compressed dynamic self-index is a compressed representation of a dynamic self-index. We assume that $M = N$ for a static setting and $M \ge N$ for a dynamic setting.

Our model of computation is a unit-cost word RAM with a machine word size of $W = \Omega(\log_2 M)$ bits. We evaluate the space complexity here in terms of the number of machine words. A bitwise evaluation of space complexity can be obtained with a $\log_2 M$ multiplicative factor.

Finally, let $\log^{(1)} n = \log_2 n$, $\log^{(i+1)} n = \log \log^{(i)} n$, and $\log^* n = \min\{i \mid \log^i n \le 1, i \ge 1\}$ for a real positive number n.

3 Literature Review

Self-indexes for highly repetitive text collections constitute an active research area, and many methods have been proposed thus far. In this section, we review the representative methods summarized in the upper part of Table 1. See [7] for a complete survey.

Self-indexes for highly repetitive text collections are broadly classified into three categories. The first category is a grammar-based self-index. Claude and Navarro [5] presented the *SLP-index*, which is built on a *straight-line program (SLP)*, a context-free grammar (CFG) in Chomsky normal form for deriving a single text. For the size n of a CFG built from text T, the SLP-index takes $O(n)$ space in memory while supporting locate queries in $O(\frac{m^2}{\epsilon} \log(\frac{\log N}{\log n}) + (m + occ) \log n)$ time, where $\epsilon \in (0, 1]$ is a parameter [5].

Two other grammar-based self-indexes, the ESP-index [18] and signature encoding [15], use the notion of LCPR. Each takes $w = O(z \log N \log^* M)$ space in memory while supporting locate queries in $O(mf_{\mathcal{A}} + occ \log N + \log w \log m \log^* M(\log N + \log m \log^* M))$ time, where $f_{\mathcal{A}} = f(w, M)$ and $f(a, b) = O(\min\{\frac{\log \log b \log \log a}{\log \log \log b}, \sqrt{\frac{\log a}{\log \log a}}\})$. Signature encoding also supports dynamic updates in $O((k + \log N \log^* M) \log w \log N \log^* M)$ time for highly repetitive texts. Although the ESP-index and signature encoding have an advantage in that they can be built in an online manner, and although signature encoding also supports dynamic updates, these self-indexes have a large disadvantage in that locate queries are slow for short patterns.

The second category includes self-indexes based on LZ77 factorization. Recently, Bille et al. [4] presented a self-index with a time-space trade-off on LZ77. Their self-index takes $O(\hat{z} \log(N/\hat{z}))$ space while supporting locate queries in $O(m(1 + \frac{\log^{\epsilon'} \hat{z}}{\log(N/\hat{z})}) + occ(\log \log N + \log^{\epsilon'} \hat{z}))$ time, where $\hat{z}$ is the number of factors in LZ77 with self-reference on T and $\epsilon' \in (0, 1)$ is an arbitrary constant. Navarro [14] presented a self-index based on a block tree (BT), called the *BT-index*. The BT-index uses $O(z \log(n/z))$ space and locates a pattern in $O(m^2 \log(N/z) + m \log^{\epsilon''} z + occ(\log \log N + \log^{\epsilon''} z))$ time for any constant $\epsilon'' > 0$.

The third category includes the RLFM-index built on the notion of a run-length-encoded BWT. The RLFM-index was originally proposed in [10] but has recently been extended to support locate queries [7]. It uses $O(r)$ space for the number r of BWT runs and takes $O(m \log \log_W(\sigma + N/r) + occ \log \log_W(N/r))$ time for locate queries. Although the RLFM-index supports fast locate queries in a compressed space, it has a serious issue with its inability to dynamically update indexes.

Despite the importance of compressed dynamic self-indexes for highly repetitive text collections, no previous method has both supported fast queries and dynamic updates of indexes while achieving a high compression ratio for highly repetitive texts. We thus present a compressed dynamic self-index, called the TST-index, that meets both demands and is applicable to highly repetitive text collections in practice. The TST-index has the following theoretical property, which will be proven over the remaining sections.

Theorem 1. *For a string T and an integer q, the TST-index takes space $w' = O(z(q^2 + \log N \log^* M))$ while supporting the following four operations: (i) count queries in $O(m(\log \log \sigma)^2)$ time for a pattern of length $m \leq q$; (ii) locate queries in $O(m(\log \log \sigma)^2 + occ \log N)$ time for a pattern of length $m \leq q$; (iii) extract queries in $O(\ell + \log N)$ time; and (iv) update operations in $O(f_{\mathcal{B}}(k + q + \log N \log^* M) + (k + q)q(\log \log \sigma)^2)$ time, where $f_{\mathcal{B}} = f(w', M)$.*

4 Fast queries with truncated suffix trees

In this section we present a novel text transformation, called a *q-TST transformation*, to improve the search performance of a self-index. We first introduce the TST in the next subsection and then present the *q*-TST transformation in the following subsection.

4.1 Tries, compact tries and truncated suffix trees

A *trie* $\mathcal{X}$ for a set of strings F is a rooted tree whose nodes represent all prefixes of the strings in F (see the left side of Figure 1). Let U be the set of nodes in $\mathcal{X}$, and let U_L be the set of leaves in $\mathcal{X}$. We then define the following five operations for $\mathcal{X}$, $u, v \in U$, $c \in \Sigma$, and $P \in \Sigma^*$:

- $path_{\mathcal{X}}(u)$: Returns the string P starting at the root and ending at node u.
- $locus_{\mathcal{X}}(P)$: Returns the node u such that $path_{\mathcal{X}}(u) = P$.

Table 1: Summary of self-indexes for highly repetitive text collections. Here, $occ_c \geq occ$ is the number of candidate occurrences of a given pattern as obtained by the ESP-index [18].

Method	Space	Locate Time	Update time
RLFM-index [7]	$O(r)$	$O(m \log\log_W(\sigma + N/r) + occ \log\log_W(N/r))$	Unsupported
Bille et al. [4]	$O(\hat{z}\log(N/\hat{z}))$	$O(m(1 + \frac{\log^{\epsilon'}\hat{z}}{\log(N/\hat{z})}) + occ(\log\log N + \log^{\epsilon'}\hat{z}))$	Unsupported
BT-index [14]	$O(z\log(N/z))$	$O(m^2\log(N/z) + m\log^{\epsilon''}z$ $+ occ(\log\log N + \log^{\epsilon''}z))$	Unsupported
SLP-index [5]	$O(n)$	$O(\frac{m^2}{\epsilon}\log(\frac{\log N}{\log n}) + (m + occ)\log n)$	Unsupported
Signature encoding [15]	$O(z\log N\log^* M)$	$O(mf_{\mathcal{A}} + occ\log N + \log z$ $\log m\log^* M(\log N + \log m\log^* M)$	$O((k + \log N\log^* M)\log z$ $\log N\log^* M)$ on average
TST-index-s (this study)	$O(z(q + \log N\log^* N))$	$O(m + occ)$ $(m \leq q)$ $O(m + occ_c\log m\log^* N\log N)(m > q)$	Unsupported
TST-index-d (this study)	$O(z(q^2 + \log N\log^* M))$	$O(m(\log\log\sigma)^2 + occ\log N)$ $(m \leq q)$	$O(f_{\mathcal{B}}(k + q + \log N\log^* M)$ $+ (k+q)q(\log\log\sigma)^2)$

- $leave_{\mathcal{X}}(P)$: Returns the set of all leaves whose prefixes contain P.
- $child_{\mathcal{X}}(u, c)$: Returns the node v such that $path_{\mathcal{X}}(u) \cdot c = path_{\mathcal{X}}(v)$ holds if it exists.
- $slink_{\mathcal{X}}(u) = v$: Returns the node v such that $path_{\mathcal{X}}(v) = path_{\mathcal{X}}(u)[2..]$ holds if it exists.

All nodes in $\mathcal{X}$ are categorized into two types. If node v is an internal node and has only one child, then v is called an *implicit node*; otherwise, v is called an *explicit node*. Let U_{imp} and U_{exp} be the respective sets of implicit and explicit nodes in $\mathcal{X}$.

For $u \in U_{imp}$, the function $expl_{\mathcal{X}}(u)$ returns the lowest explicit node $v \in U_{exp}$ such that u is an ancestor of v. The computation times for $expl_{\mathcal{X}}(u)$, $locus_{\mathcal{X}}(P)$, $path_{\mathcal{X}}(u) = P$, and $leave_{\mathcal{X}}(P)$ are constant, $O(|P|g)$, $O(|P|)$, and $O(|P|g + |leave_{\mathcal{X}}(P)|)$, respectively, where g is the computation time for $child_{\mathcal{X}}(u, c)$. Here, $g = O(1)$ when we use *perfect hashing* [6] in $O(|U_{exp}|)$ space. Moreover, $slink_{\mathcal{X}}(u) = v$ can be computed in constant time if node u stores a pointer to v in constant space. Hence, the data structure requires $O(|U_{exp}|)$ space.

A *compact trie* is a space-efficient representation of a trie $\mathcal{X}$ such that all chains of implicit nodes in $\mathcal{X}$ are collapsed into single edges (see the right side of Figure 1). We use two representations for node labels in a compact trie. The first representation uses a string $Y \in \Sigma^*$ such that each edge label in $\mathcal{X}$ is represented as a pair of start and end positions in Y, resulting in a space-efficient representation of the edge labels in $\mathcal{X}$. This representation is called a *compact trie with a reference string* and takes $O(|U_{exp}| + |Y|)$ space. The second approach represents each node label explicitly and is called a *compact trie without a reference string*. This representation takes $O(|U|)$ space. The compact trie with a reference string is more space efficient and is used in the static case, while the compact trie without a reference string is used in the dynamic case.

We can insert or delete a string K into or from a compact trie without a reference string in $O(|K|\hat{g})$ time [12], where $\hat{g}$ is the computation time for updating the data structure for $child_{\mathcal{X}}(u, c)$ when a child is inserted into or removed from u (assume $g \leq \hat{g}$). Here, $g, \hat{g} = f(u', \sigma) = O((\log\log\sigma)^2)$ when we use the predecessor/successor approach of Beame and Fich [3] in $O(|U_{exp}|)$ space, where u' is the number of children of u.

Example 2. *The trie on the left side of Figure 1 is built on* $F = \{\$, ab\$, abab, abba, b\$, bab\$, baba, babb, bbab\}$. *In this trie,* $U = \{1, \ldots 10, A, \ldots, I\}$, $U_{exp} = \{1, 3, 6, 8, A, \ldots, I\}$, $U_{imp} = 2, 4, 5, 7, 9, 10$, $path_{\mathcal{X}}(3) = $ ab, $locus_{\mathcal{X}}($baba$) = G$, $leave_{\mathcal{X}}($b$) = \{E, F, G, H, I\}$, $child_{\mathcal{X}}(5, a) = D$, $slink_{\mathcal{X}}(4) = 7$, *and* $expl(4) = C$.

The q-TST $\mathcal{X}$ [13] of a string T is a trie for Σ_T^q. We assume that T ends with a special character \$ not included in Σ. Since $\Sigma_T^q = F$ for Example 2, the right side of Figure 1 shows a q-TST built on $T = bababab bbabab\$$. Trie $\mathcal{X}$ is a 4-TST built on $T = bababab babab\$$, because $\Sigma_T^q = F = \{\$, ab\$, abab, abba, b\$, bab\$, baba, babb, bbab\}$.

An important fact is that $slink_{\mathcal{X}}(u)$ always exists for every leaf node u in q-TST. We thus explicitly store $slink_{\mathcal{X}}(u)$ for every leaf u.

Vitale et al. showed that the reference string Y of a q-TST can be represented as a string of length $O(|\Sigma_T^q|)$, and that a q-TST for a string T can be constructed in $O(|T|\hat{g})$ time in an online manner while using $O(|\Sigma_T^q|)$ working space [20].
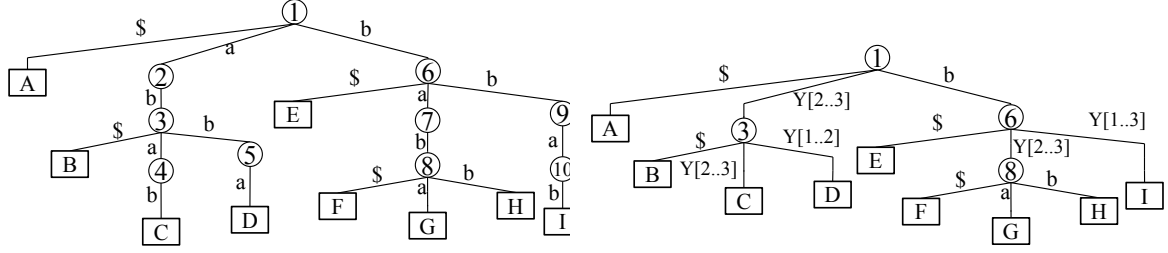
Figure 1: Trie for $F = \{\$, ab\$, abab, abba, b\$, bab\$, baba, babb, bbab\}$ (left) and its corresponding compact trie with reference string $Y = bab\$$ (right).

4.2 q-TST transformation

We now present a string transformation using a q-TST, which we call a *q-TST transformation*. A string T is transformed into a new string T_q by a q-TST transformation built on T. A self-index with improved search performance can then be built on T_q.

A q-TST transformation using a q-TST $\mathcal{X}$ on T is a transformation of T into a new string T_q by replacing every q-gram P in T by $locus_{\mathcal{X}}(P)$ (i.e., by its node id). Formally, $T_q = C(T, 1) \cdots C(T, |T|)$, where $C(T, i) = locus_{\mathcal{X}}(T[i.. \min\{i + q - 1, |T|\}])$. Similarly, a pattern P is transformed into P_q by a q-TST transformation using the same q-TST $\mathcal{X}$, i.e., $P_q = C(P, 1) \cdots C(P, |P| - q + 1)$. Given these definitions, the following lemma holds.

Lemma 3. *For two strings T, P and q-TST of T, the following equation holds.*

$$Occ(P, T) = \begin{cases} \bigcup_{c \in leave_{\mathcal{X}}(P)} cOcc(c, T_q) & \text{for } |P| \leq q \\ Occ(P_q, T_q) & \text{for } |P| > q \end{cases} \tag{1}$$

For $|P| > q$, $Occ(P, T) = \phi$ when P_q cannot be computed, i.e, when P contains a new q-gram not occurring in T.

Proof. Recall that a q-gram beginning at position i in T is transformed to the corresponding q-TST leaf beginning at position i in T_q. This means that a substring P of length at most q and beginning at position i in T is transformed to a q-TST leaf containing P as a prefix beginning at position i in T_q. Let L be the set of all leaves containing P as a prefix. Then all occurrence positions of all leaves of L in T_q are given by $Occ(P, T)$. Since $L = leave_{\mathcal{X}}(P)$, Equation 1 holds for q-short patterns.

Similarly for a q-long pattern P, a substring P beginning at position i in T is transformed to P_q beginning at position i in T_q. $Occ(P, T) = \phi$ clearly holds when P_q cannot be computed. Therefore, Equation 1 also holds for q-long patterns. □

Example 4. *Let $T = bababababbabab\$$ and $q = 4$, and let the trie in Figure 1 be a q-TST of T. Then $T_q = GCGCHDIGCFBEA$. Let $P = ab$ and $P' = babab$. Then $Occ(P, T) = cOcc(B, T_q) \cup cOcc(C, T_q) \cup cOcc(D, T_q)$ holds, because $leave_{\mathcal{X}}(P) = \{B, C, D\}$, $cOcc(B, T_q) = \{11\}$, $cOcc(C, T_q) = \{2, 4, 9\}$, and $cOcc(D, T_q) = \{6\}$. Similarly, $Occ(P', T) = Occ(P'_q, T_q) = Occ(GC, T_q) = \{1, 3, 8\}$, because $|P'| > q$.*

We can compute $leave_{\mathcal{X}}(P)$ in $O(|P|g + |leave_{\mathcal{X}}(P)|)$ time. P_q also can be computed in $O(|P|g)$ time by using $slink_{\mathcal{X}}$ and $child_{\mathcal{X}}$. This is because for $u = C(P, i + 1)$, $v = C(P, i)$, and $q \leq i < m$, $u = child_{\mathcal{X}}(slink_{\mathcal{X}}(v), P[i + q])$ holds if u and v exist.

Lemma 3 tells us that we can improve search queries on a general self-index for q-short patterns on a q-TST in $O(|P|g + c_1 \times |leave_{\mathcal{X}}(P)|)$ time if the computation time for pattern search on the general self-index is greater than $O(|P|g + c_1 \times |leave_{\mathcal{X}}(P)|)$, where c_1 is the computation time for $cOcc$ with the general self-index. If the length of pattern P is at most q, i.e., $|P| \leq q$, then we perform search queries on the q-TST. Otherwise, we perform search queries of P_q on a self-index for T_q.

We obtain the following theorem by using Lemma 3.

Theorem 5. *Let $\mathcal{I}(T)$ be an index supporting $cOcc(P, T)$ in $O(c_1)$ time and $Occ(P, T)$ in $O(c_2)$ time. Then there exists an index of $O(|\Sigma_T^q| + |\mathcal{I}(T_q)|)$ space that supports $Occ(P, T)$ in $O(m + c_1 \times |leave_{\mathcal{X}}(P)|)$ time for a q-short pattern P, and that supports $Occ(P, T)$ in $O(m + c_2)$ time for a q-long pattern P, where $|\mathcal{I}(T)|$ is the size of $\mathcal{I}(T)$.*

We apply Theorem 5 to signature encoding and present a novel self-index named TST-index in the next section.

5 TST-index

We obtain the TST-index by combining Theorem 5 with signature encoding. First, we introduce LCPR and signature encoding, and then we develop the TST-index.

5.1 Locally consistent parsing (LCPR)

LCPR [11] is a factorization of a string T by using a bit string $\tau(T)$ computed for T. Let p_i be the position of the i-th 1 in $\tau(T)$. Then $\tau(T)$ and p_i satisfy the conditions given in Lemma 6.

Lemma 6 ([11]). *For a c-colored sequence T of length at least 2, there exists a function $\tau(T)$ that returns a bit sequence of length $|T|$ satisfying the following properties: (1) $2 \leq p_{i+1} - p_i \leq 4$ and $p_1 = 1$ hold for $1 \leq i \leq d$, where $d = |cOcc(1, \tau(T))|$ and $p_{d+1} = |T| + 1$. (2) If $T[i - \Delta_L..i + \Delta_R] = T'[j - \Delta_L..j + \Delta_R]$ holds for integers i and j, then $\tau(T)[i] = \tau(T')[j]$ holds, where $\Delta_L = \log^* c + 6, \Delta_R = 4$, and T and T' are c-colored sequences.*

An LCPR for a c-colored string T using p_i for $1 \leq i \leq d + 1$ is defined as $LC_c(T) = T[p_1..p_2 - 1], \ldots, T[p_d..p_{d+1} - 1]$.

Example 7. *Let $\Delta_L = 2, \Delta_R = 1$, and $T = \mathrm{abcabcdabcab}$, and assume that $\tau(T) = 100100010010$. Then $LC_{|\Sigma|}(T) = \mathrm{abc}, \mathrm{abcd}, \mathrm{abc}, \mathrm{ab}$, $p_1 = 1$, $p_2 = 4$, $p_3 = 8$, $p_4 = 11$, $p_5 = 13$, and $2 \leq p_{i+1} - p_i \leq 4$ holds for any $1 \leq i \leq 4$. Since $T[3 - \Delta_L..3 + \Delta_R] = T[10 - \Delta_L..10 + \Delta_R] = \mathrm{abca}$ holds, $\tau(T)[3] = \tau(T)[10] = 0$. Similarly, since $T[4 - \Delta_L..4 + \Delta_R] = T[11 - \Delta_L..11 + \Delta_R] = \mathrm{bcab}$ holds, $\tau(T)[3] = \tau(T)[10] = 1$.*

5.2 Signature encoding

A signature encoding [11] of a string T is a context-free grammar $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ generating the single text T and built using $RLE(T)$ and LC_{4M}. Here, $\mathcal{V} = \{e_1, \ldots, e_w\}$ is a set of positive integers called *variables*. $\mathcal{D} = \{e_i \to f_i\}_{i=1}^w$ is a set of *deterministic production rules* (a.k.a. *assignments*), with each f_i being either of a sequence of variables in V or a single character in Σ. S is the start symbol deriving string T.

A signature encoding corresponds to a balanced derivation tree of height $O(\log N)$ built on the given string T, where each internal node (respectively, leaf node) has a variable in V (respectively, a character in Σ), as illustrated in Figure 2. Since this derivation tree is balanced, the node labels at each level can be considered as a sequence from the leftmost node to the rightmost node at a level. We define $Assgn^+(f_1, f_2, \ldots, f_d)$ as a function returning a variable sequence $e_1, e_2, \ldots, e_d$, where f_i is a single character in Σ or a sequence of variables in V, and e_i is a single character in Σ. In addition, $(e_j \to f_j) \in \mathcal{D}$ holds for $1 \leq j \leq d$. Then, SE_t^T is a sequence of node labels at the t-th level of the derivation tree built from string T and defined using $Assgn^+(f_1, f_2, \ldots, f_d)$ as follows.

$$SE_t^T = \begin{cases} Assgn^+(T[1], \ldots, T[|T|]) & \text{for } t = 0, \\ Assgn^+(LC_{4M}(SE_{t-1}^T)) & \text{for } t = 2, 4, \ldots, \\ Assgn^+(RLE(SE_{t-1}^T)) & \text{for } t = 1, 3, \ldots. \end{cases}$$

Here, $S = SE_h^T[1]$ holds for the minimum positive integer h satisfying $|SE_h^T| = 1$. Hence, $h + 1$ is the height of the derivation tree of S. Let $w = |\mathcal{V}|$ be the size of $\mathcal{G}$, whose bound is given by the following lemma.

Lemma 8 ([17]). *The size w of G is bounded by $O(\min(z \log N \log^* M, N))$.*

$\mathcal{G}$ satisfies the following properties by the definition. (1) $h = O(\log N)$ holds because $|SE_t^T| \leq \frac{1}{2}|SE_{t-1}^T|$ for a positive even integer t. (2) Every variable is at most $4M$ because SE_t^T must be a $4M$-colored sequence to apply an LCPR to SE_t^T. (3) $\mathcal{G}$ can be stored in $O(w)$ space, because every variable e in $\mathcal{D}$ is in one of three cases: (i) $e \to c \in \Sigma$; (ii) $e \to \hat{e}^k$, where $\hat{e} \in \mathcal{V}$ and $1 \leq k \leq N$; or (iii) e derives a variable sequence of length $2 \leq d \leq 4$.

Example 9. *Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be a context-free grammar, where $\Sigma = \{A, B\}$, $\mathcal{V} = \{1, \ldots, 20\}$, $\mathcal{D} = \{1 \to A, 2 \to B, 3 \to 1^1, 4 \to 2^1, 5 \to 2^2, 6 \to 1^2, 7 \to (3, 4), 8 \to (3, 5, 3), 9 \to (4, 3), 10 \to (4, 3, 4, 6), 11 \to 7^3, 12 \to 8^1, 13 \to 9^1, 14 \to 10^1, 15 \to 9^3, 16 \to (11, 12, 13), 17 \to (14, 15), 18 \to 16^1, 19 \to 17^1, 20 \to (18, 19)\}$, and $S = 20$. Assume that $LC_{4N}(SE_1^T) = (3, 4)^3, (3, 5, 3), (4, 3), (4, 3, 4, 6), (4, 3)^3$, $LC_{4N}(SE_3^T) = (11, 12, 13), (14, 15)$, and $LC_{4N}(SE_5^T) = (18, 19)$ hold. Then $\mathcal{G}$ is the signature encoding of $T = \mathrm{ABABABABBAABABABAABABABA}$.*
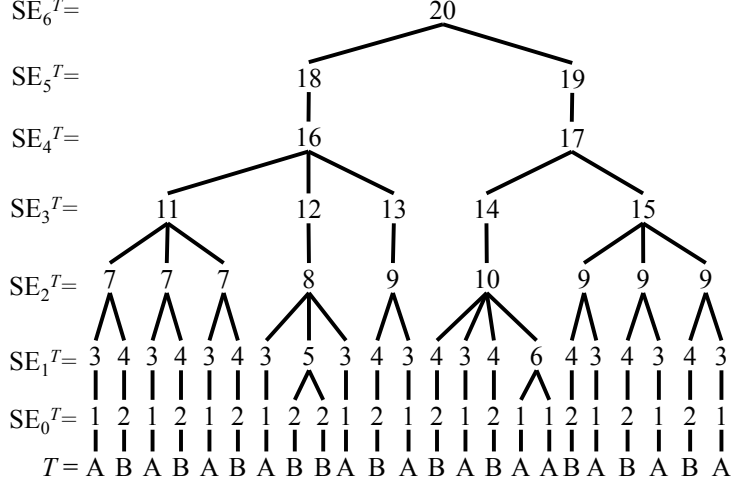
Figure 2: The derivation tree of $\mathcal{G}$ of Example 9.

5.3 Static TST-indexes

We can extend signature encoding as a self-index for pattern search. A key idea for pattern search is that there exists a common variable sequence of length $O(\log |P| \log^* M)$ representing every substring P in the parse tree of $\mathcal{G}$. This sequence is called the *core* of P and is computable from P, as described elsewhere [18, 15] in detail.

Since we can perform top-down searches of the parse tree for the core of P instead of P itself, we obtain the following lemma. [1]

Lemma 10 ([18]). *For a signature encoding, there exists an index of $O(w)$ space supporting count and locate queries in $O(m + occ_c \log m \log^* M \log N)$ time, $cOcc(c, T)$ in $O(occ)$ time, and extract queries in $O(\ell + \log N)$ time, where $occ_c \geq occ$ is the number of candidate occurrences of P by this index.*

Proof. In the original paper of Takabatake et al. [18], their index performs count and locate queries in $O(\log \log w(m \log^* M + occ_c \log m \log^* M \log N))$ time and $cOcc(c, T)$ in $O(occ \log N)$ time. We can remove the $\log \log w$ term, however, by using perfect hashing [6]. Since the $m \log^* M$ term is the computation time for $\tau(P)$, it can also be reduced to m by using the data structure proposed by Alstrup et al. [2]. Both of these data structures use $O(w)$ space.

Next, we show that we can compute $cOcc(c, T)$ in $O(occ)$ time by using $O(w)$ space. The directed acyclic graph (DAG) of $\mathcal{D}$ represents the derivation tree $\mathcal{T}$ of $\mathcal{G}$, and each node in the DAG represents a distinct variable in $\mathcal{V}$. For an edge p in the DAG, let the *relevant offset* between the parent u and child v be the length of the string generated by $e_1 \cdots e_{i-1}$, where u represents $e \to e_1 \cdots e_k \in \mathcal{D}$ and p represents the i-th edge in u. Then the occurrence position of a node in $\mathcal{T}$ is the sum of relevant offsets in the path starting at the root and ending at the node. Hence, we can compute $cOcc(c, T)$ in $O(occ \log N)$ time by storing all relevant offsets, because the height of $\mathcal{T}$ is $O(\log N)$. Furthermore, the offsets can be stored in $O(w)$ space, because we can represent the relevant offsets between e and $\hat{e}$ for $e \to \hat{e}^k \in \mathcal{D}$ by using $O(1)$ space.

Next, for a node v representing $e \in \mathcal{V}$, we store the lowest ancestor u of v such that the in-degree of u is at least 2 or u is the root in the DAG, i.e, the node representing S. We also store the sum of relevant offsets on the path starting at u and ending at v. By summing up these values, we can reduce $O(occ \log N)$ to $O(occ)$ time, and the data structures use $O(w)$ space. $\square$

We call the signature encoding built on T_q a *q-signature encoding*. Our static self-index consists of the q-TST $\mathcal{X}$ of T and the self-index of Lemma 10 for $\mathcal{G}$ representing T_q. In turn, we construct T_q from $\mathcal{X}$, $\mathcal{G}$ from T_q, and the self-index of Lemma 10 from $\mathcal{G}$. The search algorithm is based on Theorem 5, meaning that, for locate queries, we compute $locus_{\mathcal{X}}(P)$ by using $\mathcal{X}$ and output all occurrences of $locus_{\mathcal{X}}(P)$ on T_q by using locate queries on T_q if $|P| \leq q$. Otherwise, we compute P_q by using $\mathcal{X}$ and output all occurrences of P_q on T_q by using locate queries on T_q. Similarly, we can perform count queries. Hence, we obtain the following performance for our self-index from Theorem 5 and Lemma 10.

---

[1]Takabatake et al. [18] construct the grammar representing an input text by using a technique called *alphabet reduction*, which is essentially equal to LCPR. Although the definition of their grammar is slightly different from ours, we can use their search algorithm with our signature encoding.

Theorem 11. *For a string T and an integer q, there exists an index using $O(|\Sigma_T^q| + w')$ space and supporting (i) locate queries in $O(m + occ)$ time for q-short patterns, (ii) locate and count queries in $O(m + occ_c \log m \log^* M \log N)$ time for q-long patterns, and (iii) extract queries in $O(\ell + \log N)$ time, where w' is the size of the q-signature encoding of T.*

We give the following upper bound for w'.

Theorem 12. *For a q-signature encoding of T, $w' = O(z(q + \log N \log^* M))$ holds.*

Proof. See Section 6. $\qquad\square$

We also give an upper bound $|\Sigma_T^q| = O(zq)$ [19]. Thus, the size of our index is $O(z(q + \log N \log^* M))$. The results of an empirical evaluation of the TST-index are given in Section 7.

5.4 Dynamic TST-indexes

In this section, we consider maintaining our self-index for q-short patterns with a dynamic text T. Recall that our index consists of a q-TST $\mathcal{X}$ of T and an index of T_q. In the dynamic setting, the index is still based on Theorem 5, but we use the following data structure for T_q instead of that given by Lemma 10.

Lemma 13 (Dynamic signature encoding [16]). *With the addition of data structures taking $O(w)$ space, $\mathcal{G}$ can support update operations in $O(f_\mathcal{A}(k + \log N \log^* M))$ time. The modified data structure also supports computing $cOcc(c, T)$ in $O(occ \log N)$ time for $c \in \Sigma$ and extract queries in $O(\ell + \log N)$ time.*

Hence, our dynamic index supports locate queries in $O(mg + occ \log N)$ time for q-short patterns. For count queries, we append $|Occ(path_\mathcal{X}(v), T)|$ to $v \in U_{exp}$ as extra information. We also append $slink_\mathcal{X}(v)$ to $v \in U_L$.

The remaining problem is how to update these data structures when T is changed. The main problem is how T_q changes when T is changed. If we can understand that process, then we can update the data structures through each update operation.

Let $C(T, i, j) = C(T, i) \cdots C(T, j)$ for a string T and two integers i and j, i.e., $C(T, i, j) = T_q[i..j]$. For strings $x, y, z \in \Sigma^*$, the following equations hold, where $s = x[|x| - q + 1]z[1..q]$ and $s' = x[|x| - q + 1] \cdot y \cdot z[1..q]$:

$$
\begin{aligned}
C(xz, 1, |xz|) &= C(x, 1, |x| - q + 1)\,C(s, 1, |s| - q + 1)\,C(z, 1, |z|) \\
C(xyz, 1, |xyz|) &= C(x, 1, |x| - q + 1)\,C(s', 1, |s'| - q + 1)\,C(z, 1, |z|)
\end{aligned}
\tag{2}
$$

We can explain the changes in T_q due to update operations by using Equation 2. First, $T_q = C(xz, 1, |xz|)$ changes to $C(xyz, 1, |xyz|)$ when we update T by $insert(T, i, K)$, where $x = T[1..i - 1]$, $y = K$, and $z = T[i..|T|]$. Similarly, $T_q = C(xyz, 1, |xyz|)$ changes to $C(xz, 1, |xz|)$ when we update T by $delete(T, i, k)$, where $x = T[1..i - 1]$, $y = T[i..i + k - 1]$, and $z = T[i + k..|T|]$. A substring of length at most $|y| + 2q$ only changes in the q-TST leaf sequence of xz when we insert y between x and z. This means that we can update T_q with two insertions and deletions when T is updated by an insertion or deletion.

Next, we consider the update algorithm for our self-index. For $insert(T, i, K)$, we update $\mathcal{X}$ and $\mathcal{G}$ by the following four steps:

(i) Insert all q-grams in s' into $\mathcal{X}$.

(ii) Compute $C(s', 1, |s'| - q + 1)$ by using $\mathcal{X}$.

(iii) Insert $C(s', 1, |s'| - q + 1)$ into T_q, and remove $C(s, 1, |s| - q + 1)$ from T_q.

(iv) Remove old (unused) q-grams from $\mathcal{X}$.

Similarly, we update the q-TST and $\mathcal{G}$ for $delete(T, i, k)$. Step (iii) can run in $O(f_\mathcal{B}(k + q + \log N \log^* M))$ time by Lemma 13. Note that we can detect old q-grams by checking V, because $\mathcal{G}$ removes old variables during updating.

Next, we consider how to update the q-TST $\mathcal{X}$ of T. If $\mathcal{X}$ does not have extra information, then it can be updated in $O((k + q)q\hat{g})$ time. Hence, we need to show that $\mathcal{X}$ can maintain extra information without increasing the update time. When a new q-gram P is created in T, we insert P into $\mathcal{X}$ and increment the value representing $|Occ(path_\mathcal{X}(v), T)|$ for each explicit node v on the path from the root

to $u = locus_{\mathcal{X}}(P)$. This runs in $O(|P|\hat{g})$ time. If u is a node created by this insertion, then we need to compute $slink_{\mathcal{X}}(u)$. We can also do this in $O(|P|\hat{g})$ time by computing $locus_{\mathcal{X}}(P[2..])$. Note that $locus_{\mathcal{X}}(P[2..])$ always exists after computing Step 1 in the update algorithm. Similarly, we can update the extra information in the same time when a q-gram is removed from T. Hence Steps (i), (ii), and (iv) can run in $O((k+q)q\hat{g})$ time.

The only remaining point of discussion is maintenance of leaf IDs in the q-TST. When a new leaf v is created in $\mathcal{X}$, we need to assign an unused integer to v. In the dynamic setting, we use the address representing leaf v as this integer, so every character in T_q uses W bits.

Since $|U| = O(q|\Sigma_T^q|) = O(zq^2)$, we obtain Theorem 1 from Lemma 13 and Theorem 12. Note that our data structure can support extract queries in $O(\ell + \log N)$ time by using Lemma 13.

6 Tight upper bound for signature encodings of T_q

In this section, we obtain the proof of Theorem 12 by using the proof of Lemma 8. Note that we can simply show that $w' = O(zq \log N \log^* M)$ by using the following lemma and Lemma 8, but this order is larger than that in Theorem 12.

Lemma 14. $z' = O(zq)$ holds for a string T and integer q, where $z' = \mathsf{LZ}(T_q)$.

Proof. Let $LPF(i)$ and $LPF_q(i)$ be the longest previous factors without self-reference at position i in T and T_q, respectively. Then $|LPF_q(i)| \geq \max\{|LPF(i)| - (q-1), 0\}$ holds for $1 \leq i \leq N$ by a q-TST transformation. This means that every LZ77 factor of $\mathsf{LZ77}(T)$ is divided into at most q LZ77 factors on T_q. Hence, $z' = O(zq)$. $\square$

6.1 The proof of Lemma 8

We begin by explaining our approach for the proof of Lemma 8. Since $|V|$ is equal to X, the number of distinct variables in the derivation tree of $\mathcal{G}$, we try to bound X through the following idea: (1) Since each variable is determined by a local context (substring) of T obtained by signature encoding, the variables for common local contexts are the same. (2) Let $\mathsf{LZ}(T) = f_1, \ldots f_z$. Since each factor is a longest previous factor in T, a variable whose local context is contained in an LZ77 factor also occurs at another position. Hence, the upper bound of X is Y, the number of variables whose local context is on two or more LZ77 factors of T. (3) Finally, since there exist $O(\log N \log^* M)$ such variables in a factor, we can obtain Lemma 8.

We can now work through our approach more formally. Every variable in $\mathcal{G}$ is determined by a strictly local context of T. To be precise, every variable for $t = 0$ is determined only by the represented character. For a positive even integer t, every variable is determined by the SE_t^T used by the bit sequence representing the form of the variable. Recall Example 7. The factor representing $abcd$ does not change as long as $g(T)[4..8]$ does not change. We can compute $g(T)[4..8]$ from $T[4-\Delta_L..8+\Delta_R]$, so the factor depends locally on $T[4-\Delta_L..8+\Delta_R]$. For a positive odd integer t, every variable is determined by a run representing e and the left and right characters at either end of the run. Recall the example of $RLE(T)$. The factor b^5 changes into b^6 if the left character is b, but the factor always remains b^5 as long as the characters at both ends do not change. Hence, the factor depends locally on $T[2..8]$. In short, a variable on SE_t^T depends on an interval on SE_{t-1}^T. By recursively applying this rule, a variable ultimately depends on an interval on T. For a variable on $SE_t^T[i]$, let $dep_t^T(i)$ be the corresponding interval on T on which it depends. Formally, let $SE_t^T[i]$ represent $SE_{t-1}^T[L..R]$ when t is a positive even integer, and $SE_{t-1}^T[L'..R']$ when t is a positive odd integer. Then, we define $dep_t^T(i)$ as follows.

$$
dep_t^T(i) = \begin{cases} \{i\} & \text{for } t = 0, \\ \bigcup_{x=L-\Delta_L}^{R+1+\Delta_R} dep_{t-1}^T(x) & \text{for } t = 2, 4, \ldots, \\ \bigcup_{x=L'-1}^{R'+1} dep_{t-1}^T(x) & \text{for } t = 1, 3, \ldots, \end{cases}
$$

Example 15. *Recall Example 9. Let $\Delta_L = 2$ and $\Delta_R = 1$. Then $dep_0^T[7] = \{7\}$, $dep_1^T[8] = \{7, \ldots, 10\}$, and $dep_2^T[5] = dep_1^T[10 - \Delta_L] \cup \cdots \cup dep_1^T[12 + \Delta_R] = \{7, \ldots, 15\}$.*

The local context of $SE_t^T[i]$ is a substring on $dep_t^T(i)$ in T. Note that each $dep_t^T(i)$ represents an interval on $[1, N]$. The following observation holds with respect to signature encodings and *dep*.

Observation 16. *For two integers $i < j$ and an integer t, let $dep_t^T(i) = [\ell, r]$ and $dep_t^T(j) = [\ell', r']$. Then $\ell < \ell'$ and $r < r'$ hold.*

Observation 17. *Let $[\ell, r]$ and $[\ell', r']$ be two intervals on $[1, N]$ such that $T[\ell..r] = T[\ell'..r']$ holds. If there exists two integers i and t such that $dep_t^T(i) = [\ell, r]$, then there exists an integer j such that $dep_t^T(j) = [\ell', r']$.*

Hence, variables having a common local context are the same.

Next, let $K(x, t) = \{i \mid 1 \leq i \leq |SE_t^T|, x \in dep_t^T(i)\}$, constituting the set of variables on SE_t^T that depend on position x in T. We obtain the following inequality from Sentence (2) and the local context property:

$$w = |V| \leq \sum_{i=1}^{z} \sum_{t=0}^{h} |K(x_i, t)| \tag{3}$$

holds, where $x_i = |f_1 \cdots f_i|$. We can then bound $|K(x, t)|$ by using the following lemma.

Lemma 18 ([2]). *(1) For a positive even integer t, if $|K(x, t-1)| = d$ holds, then $|K(x, t)| \leq (d + \Delta_L + \Delta_R)/2$ also holds. (2) For a positive odd integer t, if $|K(x, t-1)| = d$ holds, then $|K(x, t)| \leq d$ also holds.*

Proof. By Observation 16, we can represent $K(x, t-1)$ by an interval $E = [p, p + d - 1]$. Let X be the set of positions SE_t^T whose variable depends on a position in E on SE_t^T. Then $X = K(x, t)$ holds by dep. Let $SE_t^T[i]$ represent $SE_{t-1}^T[L_i..R_i]$. Then $|X| \leq (d + \Delta_L + \Delta_R)/2$ holds, because $SE_t^T[i]$ depends on $SE_{t-1}^T[L_i - \Delta_L..R_i + \Delta_R + 1]$ and every factor $LC_{4M}(SE_{t-1}^T)$ has a length of at least 2. (2) Similarly, for a positive odd integer t, if $|K(x, t-1)| = d$ holds, then $|K(x, t)| \leq d$ also holds. □

Hence, $|K(x, t)| = O(\log^* M)$ holds for any integers x and t by Lemma 18 and $|K(x, 1)| \leq 2$. Therefore, Lemma 8 holds by $h = O(\log N)$, Lemma 18, and Inequality 3, where $h + 1$ is the height of the derivation tree of $\mathcal{G}$.

6.2 The proof of Theorem 12

We can now easily prove Theorem 12 by using the same approach for the proof of Lemma 8. The point is that we can regard a character at i in T_q as depending on $T[i..i + q - 1]$. The means that we can regard a variable on $SE_t^{T_q}$ as depending on a local context in T, not T_q. Formally, we modify $dep_t^{T_q}(i)$ for a q-signature encoding as follows.

$$dep_t^{T_q}(i) = \begin{cases} [i, i + q - 1] & \text{for } t = 0, \\ \bigcup_{x=L}^{R} dep_{t-1}^{T_q}(x) & \text{for } t = 2, 4, \ldots, \\ \bigcup_{x=L'-1}^{R'+1} dep_{t-1}^{T_q}(x) & \text{for } t = 1, 3, \ldots, \end{cases}$$

Similarly, $K(x, t)$ is redefined by the modified version of $dep_t^{T_q}(i)$. Note that Inequality 3 and Lemma 18 still hold. Hence, $\sum_{t=0}^{h'} |K(i, t)| = O(q + h' \log^* M)$ immediately holds by $|K(x, 0)| \leq q$, where $h' + 1$ is the height of the derivation tree of the signature encoding representing T_q. Therefore, Theorem 12 holds by $h' = O(\log N)$, where z is the number of LZ77 factors of T, not T_q.

7 Experiments

In this section, we demonstrate the effectiveness of the TST-index in a static setting with a benchmark dataset of highly repetitive texts.

For a benchmark dataset we used nine highly repetitive texts consisting of the files DNA, english.200MB, einstein.en.txt, einstein.de.txt, Escherichia_Coli, cere, influenza, para, and world_leaders from the Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl`). We sampled 1000 substrings of each length $m = \{4, 8, 16, \ldots, 2048\}$ from each benchmark text and used those substrings as queries. We used the memory consumption and search time for count and locate queries as evaluation measures. We

Table 2: Index size in megabytes for each text. The TST-index is denoted as q-TST for each value of parameter q in $\{4, 8, 16, 32\}$.

	DNA	english 200MB	einstein en.txt	einstein de.txt	Escherichia Coli	cere	influenza	para	world leaders
Text	385	200	445	88	107	439	147	409	44
ESP	438	248	2	1	42	47	23	60	5
RLFM	2,429	760	3	1	146	124	31	164	6
4-TST	501	388	8	3	48	54	24	71	13
8-TST	826	1,987	27	10	87	94	37	124	47
16-TST	23,927	10,615	48	17	1,794	1,417	277	1,960	88
32-TST	32,287	13,199	69	24	2,292	1,876	762	2,835	155

performed all the experiments on one core of a quad-core Intel(R) Xeon(R) E5-2680 v2 (2.80 GHz) CPU with 256 GB of memory.

We compared our TST-index with the ESP-index [18] and the RLFM-index [7]. The ESP-index provides a baseline for evaluating the effectiveness of the TST-index, while the RLFM-index is a state-of-the-art self-index for highly repetitive text collections. We used the C++ language to implement the TST-index [2] as a combination of a q-TST and the ESP-index, the self-index on an LCPR given in Lemma 10. We varied the parameter q by testing q-gram lengths from $\{4,8,16,32\}$. We used existing implementations of the ESP-index (https://github.com/tkbtkysms/esp-index-I) and the RLFM-index (https://github.com/nicolaprezza/r-index).

7.1 Results

Figures 3 and 4 show the measured search times for count and locate queries for each method. In addition, Table 2 lists the index sizes obtained with each method. The results show that the size of the TST-index increased exponentially with the size of q. This is because the variety of q-grams also increased exponentially. From a practical standpoint, the size of the TST-index is small when q is at most 8.

The TST-index was much faster than the ESP-index, especially when searching for short patterns of length at most 64, which demonstrates the effectiveness of the TST-index as a combination of a q-TST and the ESP-index. The TST-index was much more efficient for count queries than for locate queries. For the file DNA, in fact, the TST-index was 10,000 times faster than the ESP-index for count queries with patterns of length 8 but only 2 times faster than the ESP-index for locate queries with patterns of the same length. The improvement in locate queries for short patterns with the TST-index was small because the computation time of $cOcc$ for the ESP-index was slow. The performance could be improved by modifying the ESP-index implementation. Although the size of the TST-index was at most three times larger than that of the ESP-index, it remained small for highly repetitive texts in practice, showing the practicality of the TST-index.

For short patterns, pattern search with the TST-index was competitive with respect to that with the RLFM-index. The size of the TST-index was smaller than that of the RLFM-index, especially for highly repetitive texts and small q. In addition to the smaller index size, the TST-index has a large advantage in that it supports dynamic updates, unlike the RLFM-index.

8 Conclusions

We have presented a novel self-index, the TST-index, that supports fast pattern searches and dynamic update operations for highly repetitive text collections. Experimental results demonstrated that the search performance of the TST-index was significantly improved in comparison with other self-indexes on an LCPR, while the index size remained small. In addition, the search performance was competitive with that of the RLFM-index.

Note that a q-TST can be combined with any index. Kärkkäinen and Sutinen [8] proposed an index for q- short patterns. By combining their index with a q-TST, we obtain the following result.

---

[2] https://github.com/TNishimoto/TSTESP

Theorem 19 ([8] and [20])**.** *For a string T and integer q, there exists an index using $O(|\Sigma_T^q| + \mathsf{LZ}(T_q))$ space while supporting locate queries in $O(m + occ)$ time for q-short patterns.*

The size of their index is $O(zq)$, because $|\Sigma_T^q|, |\mathsf{LZ}(T_q)| = O(zq)$. Note that $|\Sigma_T^q|, |\mathsf{LZ}(T_q)| = O(zq)$ still holds even if we replace z with $\hat{z}$, where $\hat{z} \leq z$ is the number of factors in LZ77 with self-reference on T. Although their index is smaller than our static TST-index, it does not support extract queries.
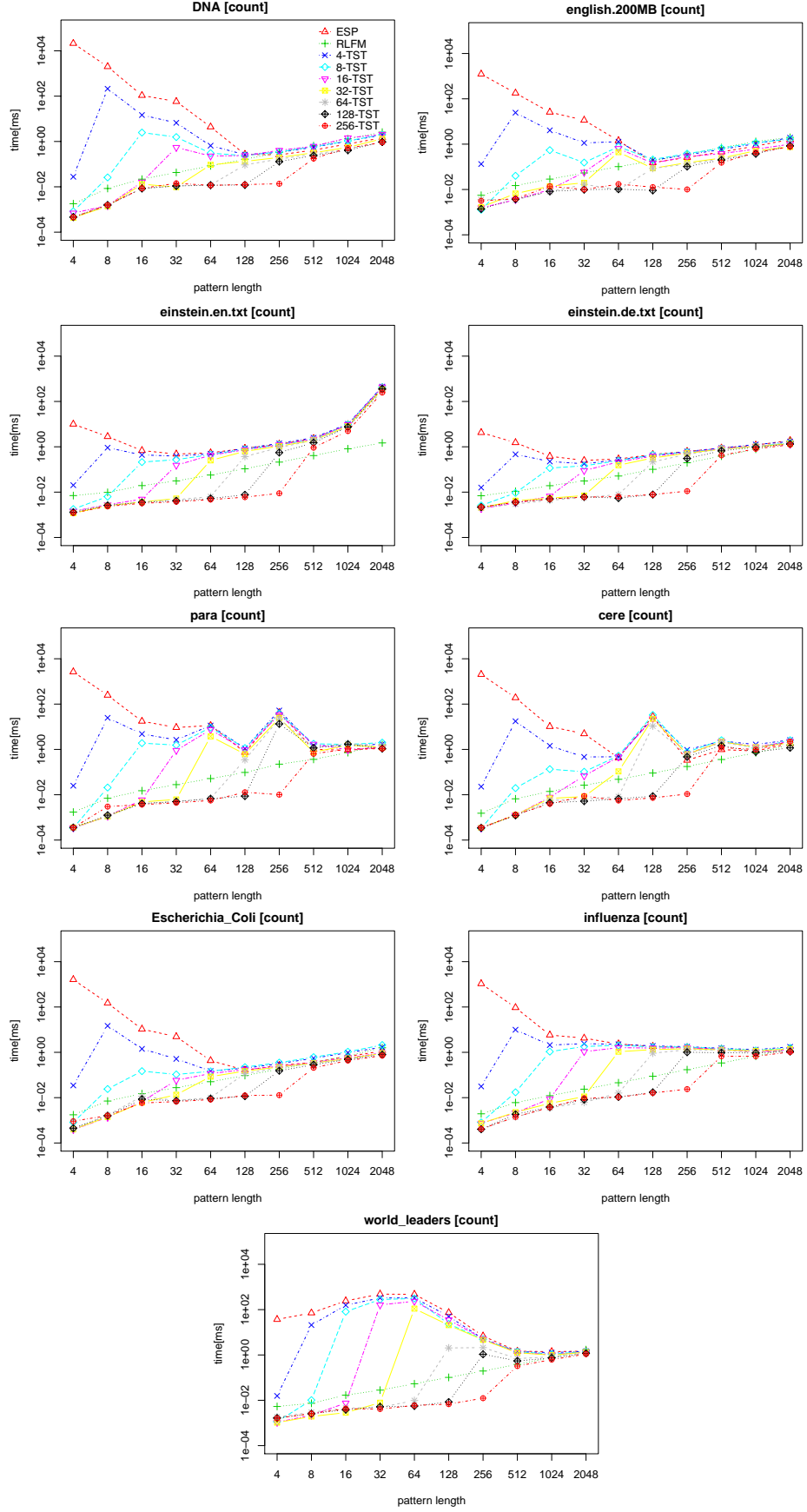
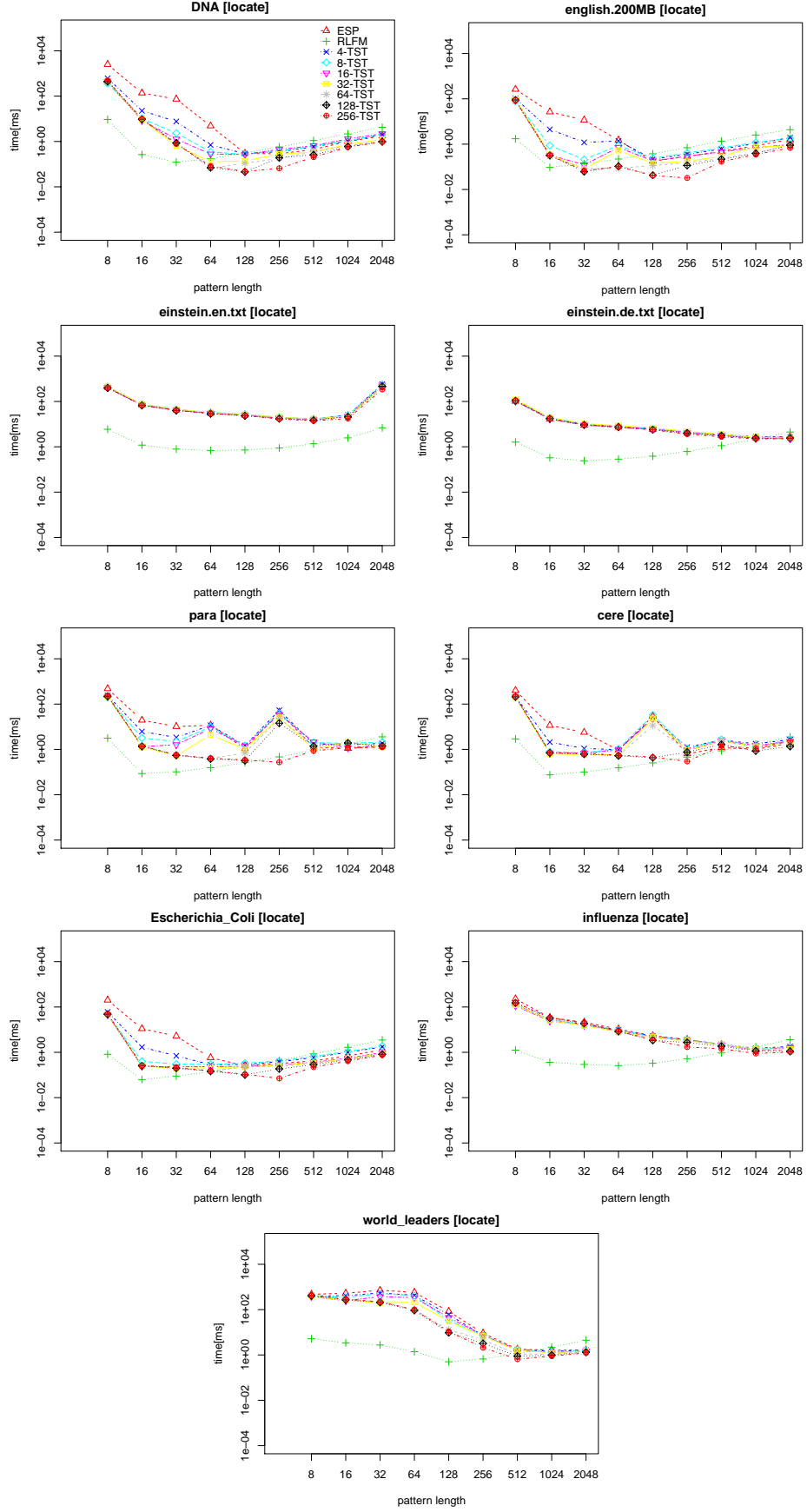Figure 3: Times for count queries on each benchmark text.

Figure 4: Times for locate queries on each benchmark text.

References

[1] 1000 Genomes Project Consortium: A map of human genome variation from population-scale sequencing. Nature 467, 1061–1073 (2010)

[2] Alstrup, S., Brodal, G.S., Rauhe, T.: Dynamic pattern matching. Technical report, Department of Computer Science, University of Copenhagen (1998)

[3] Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. Journal of Computer and System Sciences 65(1), 38–72 (2002)

[4] Bille, P., Ettienne, M.B., Gørtz, I.L., Vildhøj, H.W.: Time-space trade-offs for lempel-ziv compressed indexing. In: Proceedings of 28th Annual Symposium on Combinatorial Pattern Matching. LIPIcs, vol. 78, pp. 16:1–16:17 (2017)

[5] Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Proceedings of the 19th Symposium on String Processing and Information Retrieval. pp. 180–192 (2012)

[6] Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. Journal of the ACM 31, 538–544 (1984)

[7] Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. CoRR abs/1705.10382 (2017)

[8] Kärkkäinen, J., Sutinen, E.: Lempel-ziv index for q-grams. Algorithmica 21, 137–154 (1998)

[9] Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science 483, 115–133 (2013)

[10] Mäkinen, V., Navarro, G.: Storage and retrieval of individual genomes. Nordic Journal of Computing 12, 40–66 (2005)

[11] Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica 17, 183–198 (1997)

[12] Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM 15, 514–534 (1968)

[13] Na, J.C., Apostolico, A., Iliopoulos, C.S., Park, K.: Truncated suffix trees and their application to data compression. Theoretical Computer Science 304, 87–101 (2003)

[14] Navarro, G.: A self-index on block trees. In: Proceedings of the 24th Symposium on String Processing and Information Retrieval. pp. 278–289 (2017)

[15] Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Dynamic index and LZ factorization in compressed space. In: Proceedings of the 20th Annual Symposium on Prague Stringology Conference. pp. 158–170 (2016)

[16] Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Fully dynamic data structure for LCE queries in compressed space. In: Proceedings of 41st International Symposium on Mathematical Foundations of Computer Science. pp. 72:1–72:15 (2016)

[17] Sahinalp, S.C., Vishkin, U.: Data compression using locally consistent parsing. Technical report, University of Maryland Department of Computer Science (1995)

[18] Takabatake, Y., Tabei, Y., Sakamoto, H.: Improved ESP-index: A practical self-index for highly repetitive texts. In: Proceedings of the 13th International Symposium on Experimental Algorithms. pp. 338–350 (2014)

[19] Tanimura, Y., Nishimoto, T., Bannai, H., Inenaga, S., Takeda, M.: Small-space encoding LCE data structure with constant-time queries. CoRR abs/1702.07458 (2017)

[20] Vitale, L., Martín, A., Seroussi, G.: Space-efficient representation of truncated suffix trees, with applications to markov order estimation. Theoretical Computer Science 595, 34–45 (2015)

[21] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 337–343 (1977)