Small-Space LCE Data Structure with Constant-Time Queries

Yuka Tanimura[1], **Takaaki Nishimoto**[2], **Hideo Bannai**[3], **Shunsuke Inenaga**[4], **and Masayuki Takeda**[5]

1 **Department of Informatics, Kyushu University, Japan**

2 **RIKEN Center for Advanced Intelligence Project, Chuo-ku, Tokyo, Japan**
takaaki.nishimoto@riken.jp

3 **Department of Informatics, Kyushu University, Japan**
bannai@inf.kyushu-u.ac.jp

4 **Department of Informatics, Kyushu University, Japan**
inenaga@inf.kyushu-u.ac.jp

5 **Department of Informatics, Kyushu University, Japan**
takeda@inf.kyushu-u.ac.jp

Abstract

The *longest common extension* (*LCE*) problem is to preprocess a given string w of length n so that the length of the longest common prefix between suffixes of w that start at any two given positions is answered quickly. In this paper, we present a data structure of $O(z\tau^2 + \frac{n}{\tau})$ words of space which answers LCE queries in $O(1)$ time and can be built in $O(n \log \sigma)$ time, where $1 \leq \tau \leq \sqrt{n}$ is a parameter, z is the size of the Lempel-Ziv 77 factorization of w and σ is the alphabet size. The proposed LCE data structure does not access the input string w when answering queries, and thus w can be deleted after preprocessing. On top of this main result, we obtain further results using (variants of) our LCE data structure, which include the following:

- For highly repetitive strings where the $z\tau^2$ term is dominated by $\frac{n}{\tau}$, we obtain a *constant-time and sub-linear space* LCE query data structure.
- Even when the input string is not well compressible via Lempel-Ziv 77 factorization, we still can obtain a *constant-time and sub-linear space* LCE data structure for suitable τ and for $\sigma \leq 2^{o(\log n)}$.
- The time-space trade-off lower bounds for the LCE problem by Bille et al. [J. Discrete Algorithms, 25:42-50, 2014] and by Kosolobov [CoRR, abs/1611.02891, 2016] do not apply in some cases with our LCE data structure.

1998 ACM Subject Classification G.2.1 Combinatorial Algorithms

Keywords and phrases longest common extension, truncated suffix trees, t-covers

Digital Object Identifier 10.4230/LIPIcs.MFCS.2017.10

1 Introduction

1.1 The LCE problem

The *longest common extension* (*LCE*) problem is to preprocess a given string w of length n so that the length of the longest common prefix of suffixes of w starting at two query positions is answered quickly. The LCE problem often appears as a sub-problem of many different string processing problems, e.g., approximate pattern matching [35, 17], string comparison [34],

42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017).
Editors: Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin; Article No. 10; pp. 10:1–10:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and finding string regularities such as maximal repetitions (a.k.a. runs) [30, 1], distinct squares [22, 2], gapped repeats [11, 31, 18, 14], palindromes and gapped palindromes [21, 29, 39], and 2D palindromes [20].

A well known solution to the LCE problem is achieved by the suffix tree [47] augmented with a constant-time linear-space longest common ancestor (LCA) data structure [23, 3], or equivalently the inverse suffix array (ISA) and longest common prefix (LCP) array augmented with a constant-time range minimum query (RMQ) data structure [37, 3]. Either combination uses $O(n)$ words of space, answer LCE queries in $O(1)$ time, and can be constructed using $O(n)$ words of working space, in $O(n)$ time for integer alphabets or in $O(n \log \sigma)$ time for general ordered alphabets of size σ. The $O(n)$ space requirements, however, can be prohibitive for massive text, and hence the main focus of recent research has been on more space-efficient solutions with trade-offs for query time.

1.2 Space-efficient LCE data structures

In this paper, we will call data structures that use $o(n)$ words, or equivalently $o(n \log n)$ bits as *sub-linear space* data structures. Bille et al. [9] proposed the first *sub-linear* space LCE query data structure which occupies $O(\frac{n}{\tau})$ words of space, answers LCE queries in $O(\tau^2)$ time, and can be built in $O(\frac{n^2}{\tau})$ time using $O(\frac{n}{\tau})$ words of working space, for parameter range $1 \leq \tau \leq \sqrt{n}$. Bille et al. [8] developed an improved sub-linear space data structure which occupies $O(\frac{n}{\tau})$ words of space, answers LCE queries in $O(\tau)$ time, and can be built in $O(n^{\frac{3}{2}})$ *expected* time using $O(\frac{n}{\tau})$ words of working space, or in $O(n^{2+\epsilon})$ time using $O(\frac{n}{\tau})$ words of working space for parameter $1 \leq \tau \leq n$, where $0 < \epsilon < 1$. Tanimura et al. [45] proposed an LCE data structure of $O(\frac{n}{\tau})$ words of space, which can be built in faster $O(n\tau)$ time using $O(\frac{n}{\tau})$ words of working space, but takes slower $O(\tau \log \min\{\tau, \frac{n}{\tau}\})$ time for LCE queries, for parameter $1 \leq \tau \leq n$. All of these sub-linear space LCE data structures need to access to the input string when answering queries. Therefore, these data structures require extra $n\lceil \log \sigma \rceil$ *bits* of space for storing the input string. An in-place LCE data structure based on fingerprints was recently proposed [42].

There also exist *compressed* LCE data structures which store a compressed form of the input string represented as a straight-line program (a.k.a. grammar-based text compression) [41, 25, 6, 7, 24]. For compressible strings, the space usage of these data structures can be sub-linear.

1.3 Our LCE data structure

This paper proposes the *first* $O(1)$-*time* LCE data structure which takes *sub-linear space* in several reasonable cases, namely, when the string is compressible, and/or, when the alphabet size is suitably small. Our data structure has both flavours of sub-linear space and compressed LCE data structures. Namely, for parameter $1 \leq \tau \leq \sqrt{n}$, we present an LCE data structure which takes $O(z\tau^2 + \frac{n}{\tau})$ words of space, answers LCE queries in $O(1)$ time, and can be built in $O(n \log \sigma)$ time for general ordered alphabets of size σ using $O(z\tau^2 + \frac{n}{\tau})$ words of working space, where z is the size of the Lempel-Ziv 77 factorization [48] of the input string. It is known that z is a lower bound on the size of *any* grammar-based compression of the string [44], and can be very small for highly repetitive strings. In such cases where the $z\tau^2$ term is dominated by $\frac{n}{\tau}$, our LCE data structure uses sub-linear space. An interesting feature is that we do *not* actually compress the input string, i.e., do not compute the Lempel-Ziv 77 factorization, but we construct a data structure whose size is bounded by $O(z\tau^2 + \frac{n}{\tau})$.

Table 1 Deterministic LCE query data structures. n is the length of the input string, σ is the alphabet size, z is the size of the Lempel-Ziv 77 factorization of w, l is the length of the LCE, ω is the machine word size, $\epsilon > 0$ is an arbitrarily small constant, and τ is a trade-off parameter ($\dagger : 1 \leq \tau \leq n$, $\diamond : 1 \leq \tau \leq \sqrt{n}$). ISA+ consists of the inverse suffix array of w, the LCP array and the RMQ data structure. $\star$ is valid for $\omega = \Theta(\log n)$ and $\sigma \leq 2^{o(\log n)}$.

Data structure		Preprocessing		Ref
Space (bits)	Query Time	Working space	Construction time	
$O(\omega)$	$O(n)$	$O(\omega)$	-	naïve
$O(n\omega)$	$O(1)$	$O(n\omega)$	$O(n)$	ISA+
$n\lceil\log\sigma\rceil + O(\frac{n\omega}{\tau})$	$O(\tau^2)$	$O(\frac{n\omega}{\tau})$	$O(n^2/\tau)$	$\diamond$ [9]
$n\lceil\log\sigma\rceil + O(\frac{n\omega}{\tau})$	$O(\tau)$	$O(\frac{n\omega}{\tau})$	$O(n^{3/2})$ exp.	$\dagger$ [8](1)
$n\lceil\log\sigma\rceil + O(\frac{n\omega}{\tau})$	$O(\tau)$	$O(\frac{n\omega}{\tau})$	$O(n^{2+\epsilon})$	$\dagger$ [8](2)
$n\lceil\log\sigma\rceil + O(\frac{n\omega}{\tau})$	$O(\tau\log\min\{\tau, \frac{n}{\tau}\})$	$O(\frac{n\omega}{\tau})$	$O(n\tau)$	$\dagger$ [45]
$n\lceil\log\sigma\rceil + O(\omega\log n)$	$O(\log l)$	$O(\omega\log n)$	$O(n\log n)$ exp.	[42]
$O(z\omega\log n\log^* n)$	$O(\log n\log^* n)$	$O(z\omega\log n\log^* n)$	$O(n\log\sigma)$	[41],[25]
$O(z\omega\log\frac{n}{z})$	$O(\log n)$	$O(n\omega)$	$O(n)$	[24]
$O(z\omega\log\frac{n}{z})$	$O(\log n)$	$O(n\log\sigma + z\omega\log\frac{n}{z})$	$O(n\log\log\sigma + z\log^2\frac{n}{z})$	[24]+[32]
$O((z\tau^2 + \frac{n}{\tau})\omega)$	$O(1)$	$O((z\tau^2 + \frac{n}{\tau})\omega)$	$O(n\log\sigma)$	$\diamond$ ours
$O(z^{1/3}n^{2/3}\omega)$	$O(1)$	$O(z^{1/3}n^{2/3}\omega)$	$O(n\log\sigma\log n)$	ours
$o(n\log n)$	$O(1)$	$o(n\log n)$	$o(n\log^2 n)$	$\star$ ours
$O(\sqrt{nz}\omega)$	$O(\sqrt{\frac{n}{z}})$	$O(\sqrt{nz}\omega)$	$O(n\log\sigma\log n)$	ours

Even when the input string is not well compressible via Lempel-Ziv 77, for suitably small alphabets, we can build a sub-linear space LCE data structure with $O(1)$ query time using appropriate values of τ. By choosing $\tau = (\frac{n}{z})^{\frac{1}{3}}$, our LCE query data structure takes $O(z^{\frac{1}{3}}n^{\frac{2}{3}})$ words of space, which translates to $O(n/(\log_\sigma n)^{\frac{1}{3}})$ using the well-known fact that $z = O(n/\log_\sigma n)$ (e.g. [26]). This means that our data structure can be stored in $O(n\log n/(\log_\sigma n)^{\frac{1}{3}}) = O(n(\log n)^{\frac{2}{3}}(\log\sigma)^{\frac{1}{3}})$ *bits* of space. This implies that for alphabets of size $\sigma \leq 2^{o(\log n)}$ (note that these contain polylogarithmic alphabets), our data structure takes only $o(n\log n)$ bits of space, yet answers LCE queries in $O(1)$ time. Also, our LCE data structure does not access the input string when answering queries, and hence the input string does not have to be kept. To our knowledge, this is the first sub-linear space LCE data structure for strings which are not well compressible with Lempel-Ziv 77.

The key to our efficient LCE query data structure is a hybrid use of the *truncated suffix trees* [38] and block-wise LCE queries based on t-covers [43, 9]. The q-truncated suffix tree of a string w is the compact trie (a.k.a. Patricia tree) which represents all substrings of w of length at most q. We observe that, for any $1 \leq q \leq n$, the q-truncated suffix tree can be stored in $O(zq)$ words of space, including a string to which the edges label pointers refer. We also show that the block-wise LCE query data structure based on t-covers can be efficiently built by the t-truncated suffix tree, leading to our result. Several variants of our data structure are considered, as summarized in Table 1.

The rest of this paper is organized as follows. Section 2 gives some definitions and introduces tools which will be used as building-blocks of our LCE data structure. In Section 3 we propose our new LCE data structure and analyze its time/space complexities. In Section 4 we review some lower bounds on the LCE problem and show that using our LCE data structure, these lower bounds do not apply in some cases. We conclude in Section 5.

2 Preliminaries

2.1 Notations

Let Σ be an ordered alphabet of size σ. Each element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is the string of length zero, namely $|\varepsilon| = 0$. If $w = xyz$ for some strings w, x, y, z, then x, y, and z are respectively called a *prefix*, *substring*, and *suffix* of w. For any $1 \leq i \leq |w|$, let $w[i]$ denote the ith character of w. For any $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j, namely, $w[i..j] = w[i] \cdots w[j]$. A string of length q is called a *q-gram*. For any $1 \leq q \leq |w|$, let $\mathit{Substr}_q(w)$ denote the set of all q-grams occurring in w and the $q-1$ suffixes of w of length shorter than q, namely, $\mathit{Substr}_q(w) = \{w[i..\min\{i+q-1, |w|\}] \mid 1 \leq i \leq |w|\}$.

For any string w, let $\mathsf{LCE}^w(i,j)$ denote the length of the longest common prefix of $w[i..|w|]$ and $w[j..|w|]$. We will write $\mathsf{LCE}(i,j)$ when w is clear from the context. Since $\mathsf{LCE}^w(i,i) = |w| - i$, we will only consider the case when $i \neq j$. For any integers $i \leq j$, let $[i..j]$ denote the set of integers from i to j (including i and j).

The *Lempel-Ziv 77 factorization with self-references* [48] of a string w is a sequence $\mathsf{LZ}(w) = f_1, \ldots, f_z$ of z non-empty substrings of w such that $w = f_1 \cdots f_z$ and for $1 \leq i \leq z$,

- $f_i = w[|f_1 \cdots f_{i-1}|+1] \in \Sigma$ if $s[|f_1 \cdots f_{i-1}|+1]$ is a character not occurring in $f_0 \cdots f_{i-1}$,
- f_i is the longest prefix of $f_i \cdots f_z$ such that f_i is a substring of w beginning at a position in range $[1..|f_1 \cdots f_{i-1}|]$,

where $f_0 = \varepsilon$. The *size* of $\mathsf{LZ}(w)$ is the number z of factors $f_1, \ldots, f_z$, and is denoted as $|\mathsf{LZ}(w)| = z$. For instance, for string $w =$ `abababcabababcababababcd` of length 22, $\mathsf{LZ}(w) =$ `a`, `b`, `abab`, `c`, `abababcabababc`, `d` and $|\mathsf{LZ}(w)| = 6$.

Our model of computation is a standard word RAM with machine word size $\omega \geq \log n$. The space requirements will be evaluated by the number of words unless otherwise stated.

2.2 Tools

We will use the following tools as building blocks of our LCE data structure.

t-covers. For any positive integer t, a set $D \subseteq [0..t-1]$ is called a *t-difference-cover* if $[0..t-1] = \{(x-y) \bmod t \mid x, y \in D\}$, namely, every element in $[0..t-1]$ can be expressed by a difference between two elements in D modulo t. For any positive integer n, a set $S \subseteq [1..n]$ is called a *t-cover* of $[1..n]$ if $S = \{i \in [1..n] \mid (i \bmod t) \in D\}$ with some t-difference-cover D, and there is a constant-time computable function $h(i,j)$ that for any $1 \leq i, j \leq n-t$, $0 \leq h(i,j) \leq t$ and $i + h(i,j), j + h(i,j) \in S$.

▶ **Lemma 1** ([36]). *For any integer t, there exists a t-difference-cover $D(t)$ of size $O(\sqrt{t})$ which can be computed in $O(\sqrt{t})$ time.*

▶ **Lemma 2** ([12]). *For any integer t $(\leq n)$, there exists a t-cover of size $O(\frac{n}{\sqrt{t}})$ which can be computed in $O(\frac{n}{\sqrt{t}})$ time.*

In what follows, we will denote by $S(t)$ an arbitrary t-cover of $[1..n]$ which satisfies the conditions of Lemma 2. See Figure 1 for an example of a t-cover $S(t)$.

Truncated suffix trees. For convenience, we assume that any string w ends with a special end-marker \$ that appears nowhere else in w. Let $n = |w|$. For any $1 \leq q \leq n$, the *q-truncated suffix tree* of w, denoted q-$\mathsf{TST}(w)$, is a Patricia tree which represents $\mathit{Substr}_q(w)$. Namely, q-$\mathsf{TST}(w)$ is an edge-labeled rooted tree such that: (1) Each edge is labeled with a non-empty

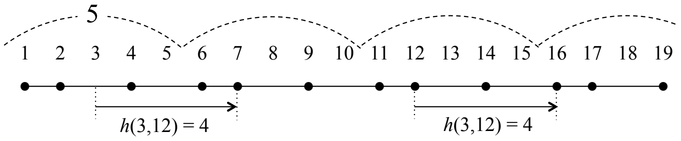

Figure 1 Let $t = 5$ and $D = \{1, 2, 4\}$. This figure shows an example of a 5-cover $S(5) = \{1, 2, 4, 6, 7, 9, 11, 12, 14, 16, 17, 19\}$. The black dots represent the elements in $S(5)$. For instance, we have $h(3, 12) = 4$, namely, $3 + 4, 12 + 4 \in S(5)$.

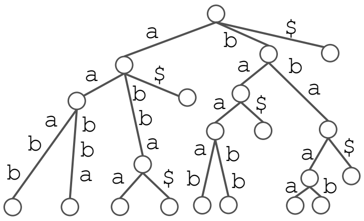

Figure 2 5-$\mathsf{TST}(w)$ with string $w =$ `baabbaabbaaabbaabba$`.

substring of w; (2) Each internal node v has at least two children, and the labels of the out edges of v begin with distinct characters; (3) For any leaf u, there is at least one position $1 \leq i \leq n$ such that $w[i..\min\{i + q - 1, n\}]$ is the string obtained by concatenating the edge labels from the root to u; (4) For any position $1 \leq i \leq n$ in w, there is a unique leaf u such that $w[i..\min\{i + q - 1, n\}]$ is the string obtained by concatenating the edge labels from the root to u.

Informally speaking, q-$\mathsf{TST}(w)$ can be obtained by trimming the full suffix tree of w so that any path from the root represents a substring of length at most q. Clearly, the number of leaves in q-$\mathsf{TST}(w)$ is equal to $|\mathit{Substr}_q(w)|$. We assume that the leaves of q-$\mathsf{TST}(w)$ are sorted in lexicographical order. Figure 2 shows an example of a q-$\mathsf{TST}(w)$. For any node u of q-$\mathsf{TST}(w)$, $\mathit{str}(u)$ denotes the string spelled out by the path from the root to u.

In the case of the full suffix tree (n-$\mathsf{TST}(w)$) of string w of length n, each edge label x is represented by a pair (i, j) of positions in w such that $x = w[i..j]$. We call w as the *reference string* for the full suffix tree, and this way the full suffix tree can be stored in $O(n)$ space. For q-$\mathsf{TST}(w)$, Vitale et al. [46] showed how to represent q-$\mathsf{TST}(w)$ in $O(|\mathit{Substr}_q(w)|)$ space, including the reference string, and how to construct them efficiently, both in time and space.

▶ **Lemma 3** ([46]). *Let w be any string of length n over an ordered alphabet of size σ. For any $1 \leq q \leq n$, let $y = |\mathit{Substr}_q(w)|$. Then, there exists a reference string w' of length $O(y)$ for q-$\mathsf{TST}(w)$. Moreover, q-$\mathsf{TST}(w)$ with the leaves sorted in lexicographical order, and a reference string w' can be constructed in $O(n \log \sigma)$ time with $O(y)$ working space.*

We also show the following lemma.

▶ **Lemma 4.** *q-$\mathsf{TST}(w)$ can be represented in $O(zq)$ space, where $z = |\mathsf{LZ}(w)|$.*

Proof. By Lemma 3, it suffices to show that $|\mathit{Substr}_q(w)| = O(zq)$. For each q-gram $p \in \mathit{Substr}_q(w)$, let $\mathit{locc}_w(p)$ be the beginning position of the leftmost occurrence of p in w. If $q = 1$, then clearly $|\mathit{Substr}_1(w)| \leq z$ and hence the lemma holds. If $q \geq 2$ then the interval $[\mathit{locc}_w(p)..\mathit{locc}_w(p) + q - 1]$ must cross the boundary of two adjacent factors of $\mathsf{LZ}(w)$, since otherwise the interval is completely contained in a single factor of $\mathsf{LZ}(w)$ but this contradicts that $[\mathit{locc}_w(p)..\mathit{locc}_w(p) + q - 1]$ is the leftmost occurrence of p in w. Clearly, the maximum number of q-grams that can cross a boundary of $\mathsf{LZ}(w)$ is $q-1$. Hence, the total number of distinct q-grams in w is $O(zq)$. Also, $\mathit{Substr}_q(w)$ contains q substrings $w[n-q+1], \ldots, w[n]$ of w which are shorter than q. Overall, we obtain $|\mathit{Substr}_q(w)| = O(zq)$. ◀

The next theorem follows from Lemmas 3 and 4 and an obvious fact that $|\mathit{Substr}_q(w)| \leq n$.

▶ **Theorem 5.** *Given a string w of length n over an ordered alphabet of size σ and integer $1 \leq q \leq n$, we can construct an $O(\min\{zq, n\})$-space representation of q-$\mathsf{TST}(w)$ in $O(n \log \sigma)$ time with $O(\min\{zq, n\})$ working space.*

In what follows, we will only consider interesting cases where $zq < n$ for a given $1 \leq q \leq n$, and will simply use $O(zq)$ to denote the size of q-$\mathsf{TST}(w)$.

3 Our LCE data structure

3.1 Overview of our algorithm

The general framework of our space-efficient LCE algorithm follows the approach of Gawrychowski et al.'s LCE algorithm for strings over a general ordered alphabet [19]. Namely, we compute $\mathsf{LCE}(i, j)$ using the two following types of queries:

$$\begin{aligned} \mathsf{ShortLCE}_t(i,j) &= \min(\mathsf{LCE}(i,j), t), \\ \mathsf{LongLCE}_t(i,j) &= \begin{cases} \lfloor \mathrm{LCE}(i,j)/t \rfloor & \text{if } i, j \in S(t), \\ \perp & \text{otherwise.} \end{cases} \end{aligned}$$

$\mathsf{LCE}(i, j)$ is computed in the following manner. Let $\delta = h(i, j)$. Recall that $\delta \leq t$ can be computed in constant time and that $i + \delta, j + \delta \in S(t)$. First, we compare up to the first δ characters of $w[i..|w|]$ and $w[j..|w|]$ using $\mathsf{ShortLCE}_t(i, j)$. If $l_1 = \mathsf{ShortLCE}_t(i, j)$ is shorter than t, then $\mathsf{LCE}(i, j) = l_1$. If $l_1 = t$, then $\mathsf{LCE}(i, j)$ is at least t long. To check if it further extends, we compute $l_2 = \mathsf{LongLCE}_t(i + \delta, j + \delta)$, and $l_3 = \mathsf{ShortLCE}_t(i + \delta + l_2, j + \delta + l_2)$. Finally, we get $\mathsf{LCE}(i, j) = \delta + t \cdot l_2 + l_3$. See also Figure 3.

The main difference between Gawrychowski et al.'s method and ours is in how to compute $\mathsf{ShortLCE}_t(i, j)$. While they use a Union-Find structure that takes $O(n)$ working space (for $O(n)$ queries) as a main tool, we use an augmented $2t$-$\mathsf{TST}(w)$ for $\mathit{Substr}_{2t}(w)$ which occupies $O(zt + \frac{n}{\sqrt{t}})$ total space, answers $\mathsf{ShortLCE}_t(i, j)$ queries in $O(1)$ time, and can be constructed in $O(n \log \sigma)$ time with $O(zt)$ working space. How to answer $\mathsf{LongLCE}_t(i, j)$ queries is equivalent to Gawrychowski et al.'s, namely, we sample the positions from $S(t)$ so that LCE queries for these sampled positions can be answered in $O(1)$ time. We show how to build the data structure for $\mathsf{LongLCE}_t(i, j)$ queries by using t-$\mathsf{TST}(w)$ for $\mathit{Substr}_t(w)$ in $O(n \log \sigma)$ time with $O(zt)$ working space.

3.2 $\mathsf{ShortLCE}_t$ queries

For $\mathsf{ShortLCE}_t(i, j)$ queries, we use $2t$-$\mathsf{TST}(w)$ which represents the set $\mathit{Substr}_{2t}(w)$ of all substrings of w of length at most $2t$. For any position $1 \leq i \leq n$, let p_i denote the substring of w that begins at position i and is of length at most $2t$, namely, $p_i = w[i..\min\{i+2t-1, n\}]$. Notice that $\mathit{Substr}_{2t}(w) = \bigcup_{i=1}^{n}\{p_i\}$. For any position $1 \leq i \leq n$ in w, let $\ell(i) = u$ iff u is the leaf of $2t$-$\mathsf{TST}(w)$ such that $\mathit{str}(u) = p_i$. Basically, we will compute $\mathsf{ShortLCE}_t(i, j)$ by efficiently finding the LCA of the corresponding leaves $\ell(i)$ and $\ell(j)$ on $2t$-$\mathsf{TST}(w)$. The reason that we use $2t$-$\mathsf{TST}(w)$ rather than t-$\mathsf{TST}(w)$ will become clear later.

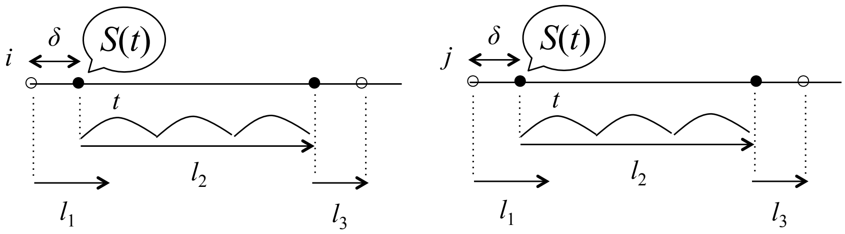

Figure 3 Illustration of an overview of our $\mathsf{LCE}(i, j)$ algorithm. We are given two positions i and j in string w. First, we compute $l_1 = \mathsf{ShortLCE}_t(i, j)$. If $l_1 < t$, then $\mathsf{LCE}(i, j) = l_1$. Otherwise, we compute $\mathsf{LongLCE}_t(i + \delta, j + \delta)$ where $i + \delta, j + \delta \in S(t)$ and $0 \leq \delta \leq t$. We finally compute $l_3 = \mathsf{ShortLCE}(i + \delta + l_2, j + \delta + l_2)$ where $l_2 = t\mathsf{LongLCE}_t(i + \delta, j + \delta)$. Then $\mathsf{LCE}(i, j) = \delta + l_2 + l_3$.

Now the key is how to access $\ell(i)$ for a given position i in w. As our goal is to build a sub-linear space data structure for $\mathsf{ShortLCE}_t$ queries, we cannot afford to store a pointer to $\ell(i)$ from every position $1 \leq i \leq n$. Thus, we store such a pointer only from every t-th positions in w. We call these positions as *sampled positions*. Formally, for every sampled position $j \in Q_{t,n} = \{1 + kt \mid 0 \leq k \leq \lceil \frac{n}{t} \rceil - 1\}$ we explicitly store a pointer from j to its corresponding leaf $\ell(j)$ on $2t$-$\mathsf{TST}(w)$. Also, for each position $1 \leq i \leq n$ in w, let $\alpha(i) = \max\{j \in Q_{t,n} \mid j \leq i\}$. Namely, $\alpha(i)$ is the closest sampled position in $Q_{t,n}$ to the left of i (or it is i itself if $i \in Q_{t,n}$).

Given a position $1 \leq i \leq n$, $\alpha(i)$ can be computed in $O(1)$ time with a simple arithmetic operation. Hence, we can access the leaf $\ell(\alpha(i))$ for the closest sampled position $\alpha(i)$ in $O(1)$ time. The next task is to locate $\ell(i)$. To describe our constant-time algorithm, let us consider a conceptual graph $G = (V, E)$ such that $V = \mathit{Substr}_{2t}(w)$ and $E = \{(u, c, v) \mid u[1..2t - 1] = v[2..2t], c = v[1]\}$, where (u, c, v) represents a directed edge labeled c from u to v. This graph G is equivalent to the edge-reversed de Bruijn graph of order $2t$, with extra nodes for the $2t - 1$ suffixes of w which are shorter than $2t$. It is clear that there is a one-to-one correspondence between the leaves of $2t$-$\mathsf{TST}(w)$ and the nodes of the graph G. Thus, we will identify each leaf of $2t$-$\mathsf{TST}(w)$ with the nodes of graph G.

▶ **Lemma 6.** *Given $2t$-$\mathsf{TST}(w)$ for a string w of length n and for any $1 \leq 2t \leq n$, we can construct the graph G in $O(n)$ time using $O(zt)$ working space.*

Proof. The de Bruijn graph of order q for a string of length n can be constructed in $O(n)$ time using space linear in the size of the output de Bruijn graph, provided that q-$\mathsf{TST}(w)$ is already constructed [13]. By setting $q = 2t$, adding extra $2t - 1$ nodes for the suffixes that are shorter than $2t$, and reversing all the edges, we obtain our graph $G = (V, E)$.

The number of nodes in V is clearly equal to $|\mathit{Substr}_{2t}(w)|$. Also, since each edge in E corresponds to a distinct substring in $\mathit{Substr}_{2t+1}(w)$, the number of edges in E is equal to $|\mathit{Substr}_{2t+1}(w)|$. By a similar argument to the proof of Lemma 4, we obtain $|V| = |\mathit{Substr}_{2t}(w)| = O(zt)$ and $|E| = |\mathit{Substr}_{2t+1}(w)| = O(zt)$. ◀

Let $d = i - \alpha(i)$. A key observation here is that there is a path of length d from node p_i to node $p_{\alpha(i)}$ in this graph G. Since G is a graph, however, it is not easy to quickly move from $p_{\alpha(i)}$ to p_i. To overcome this difficulty, we consider a spanning tree of G of which the root is $p_n = w[n] = \$$. Let T denote any spanning tree of G. See Figure 4 for examples of the graph G and its spanning tree T. Although some edges are lost in spanning tree T, it is enough for our purpose. Namely, the following lemma holds.

▶ **Lemma 7.** *Any spanning tree T of G satisfies the following properties: (1) There is a non-branching path of length $2t$ from the root p_n to the node p_{n-2t}. (2) For any $1 \leq i \leq n-2t-1$ and $0 \leq d < t$, let g be the d-th ancestor of $p_{\alpha(i)}$. Then, $g[1..t] = p_i[1..t]$.*

Proof. The first property is immediate from the fact that the last character $w[n] = \$$ occurs nowhere else in w and the root represents $p_n = \$$.

Since $d < t$ and $|p_{\alpha(i)}| = 2t$, we have $p_{\alpha(i)}[d..d+t-1] = p_i[1..t]$. By the first property and $\alpha(i) \leq i \leq n-2t-1$, the depth of node $p_{\alpha(i)}$ is at least $2t$. Also, by following the in-coming edge of each node in the reversed direction, we delete the first character of the corresponding string. Hence, $p_i[1..t]$ is a prefix of the d-th ancestor g of $p_{\alpha(i)}$. ◀

We are ready to show the main result of this section.

▶ **Theorem 8.** *For any string w of length n and integer $1 \leq t \leq n$, a data structure of size $O(zt + \frac{n}{t})$ can be constructed in $O(n \log \sigma)$ time using $O(zt)$ working space such that subsequent $\mathsf{ShortLCE}_t(i,j)$ queries for any $1 \leq i,j \leq n$ can be answered in $O(1)$ time, where $z = |\mathsf{LZ}(w)|$.*

Proof. We use a spanning tree T enhanced with a *level ancestor* data structure [5, 4] which can be constructed in time and space linear in the size of the input tree T.

Given two positions i,j in w, we answer $\mathsf{ShortLCE}_t(i,j)$ query as follows:

1. Compute the closest sampled positions $\alpha(i)$ and $\alpha(j)$ by simple arithmetics.
2. Access the nodes $p_{\alpha(i)}$ and $p_{\alpha(j)}$ in the spanning tree T using pointers from the sampled positions $\alpha(i)$ and $\alpha(j)$, respectively.
3. Let $d = i - \alpha(i)$ and $d' = j - \alpha(j)$. Access the d-th ancestor u of $p_{\alpha(i)}$ and the d'-th ancestor of $p_{\alpha(j)}$ using level ancestor queries on T.
4. Compute the LCA x of the two leaves u and v on $2t$-$\mathsf{TST}(w)$, and return $\min\{|str(x)|, t\}$.

The correctness follows from Lemma 7. Since each step of the above algorithm takes $O(1)$ time, we can answer $\mathsf{ShortLCE}_t(i,j)$ in $O(1)$ time. By Lemma 4, the size of $2t$-$\mathsf{TST}(w)$ with an LCA data structure is $O(zt)$, and also the size of the spanning tree T with a level ancestor data structure is $O(|Substr_{2t}(w)|) = O(zt)$. In addition, we store pointers from the $\Theta(\frac{n}{t})$ sampled positions to their corresponding nodes in T. Overall, the total space requirement of our data structures is $O(zt + \frac{n}{t})$. We can build these data structures in a total of $O(n \log \sigma)$ time using $O(zt)$ working space by Theorem 5 and Lemma 6. ◀

3.3 $\mathsf{LongLCE}_t$ queries

At a high level, our $\mathsf{LongLCE}_t(i,j)$ query algorithm is an adaptation of the t-cover based algorithm by Puglisi and Turpin [43], which was later re-discovered by Bille et al. [9]. Gawrychowski et al. [19] showed that an $O(\frac{n}{\sqrt{t}})$-space data structure, which answers $\mathsf{LongLCE}_t(i,j)$ query in $O(1)$ time, can be constructed in $O(n \log t)$ time with $t = \Omega(\log^2 n)$ for a string of length n over a general ordered alphabet. In this section, we show the same data structure as Gawrychowski et al. can be constructed in $O(n \log \sigma)$ time with $O(zt + \frac{n}{t})$ working space for a general ordered alphabet of size σ and any $1 \leq t \leq n$.

Consider a t-cover $S(t)$ of $[1..n]$ for some t-difference-cover D. For each position $i \in S(t)$ such that $i + t - 1 \leq n$, the substring $b_i = w[i..i+t-1]$ is said to be a *t-block*. The goal here is to answer the block-wise LCE value $\mathsf{LongLCE}_t(i,j)$ for two given positions in the t-cover $S(t)$. Since we query $\mathsf{LongLCE}_t(i,j)$ only for positions $i,j \in S(t)$ and the answer to $\mathsf{LongLCE}_t(i,j)$ is a multiple of t, we can regard each t-block as a single character. Thus,

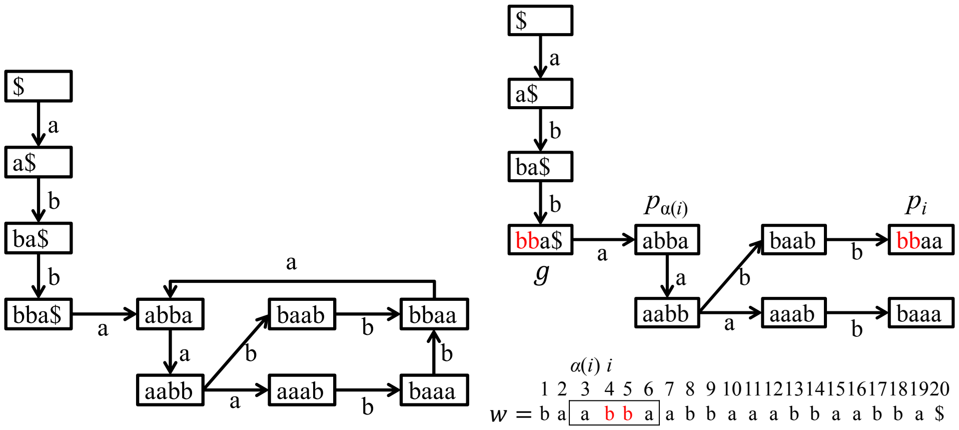

Figure 4 The left graph G is the edge-reversed de Bruijn graph of order $2t$, with extra nodes for the $2t-1$ suffixes of w which are shorter than $2t$, where $t=2$ and w is the same string as in Figure 2. An edge from u to v labeled character c represents $c \cdot u[1..2t-1] = v$. The right tree is a spanning tree of the left graph. Let $i=4$ and $\alpha(i)=3$. Then $p_i = bbaa$, $p_{\alpha(i)} = abba$. Let g be the d-th ancestor of $p_{\alpha(i)}$ in the right tree, where $0 \le d < t$. Then $g[1..t] = p_i[1..t]$ holds by Lemma 7.

we sort all t-blocks in lexicographical order, and encode each t-block by its lexicographical rank. Since each t-block is of length t, we can sort the t-blocks in $O(\frac{n}{\sqrt{t}} \log \frac{n}{\sqrt{t}})$ time with $O(zt + \frac{n}{\sqrt{t}})$ working space by using any suitable comparison-based sorting algorithm and our $O(1)$-time $\mathsf{ShortLCE}_t$ query data structure of Section 3.2. The next lemma shows that we can actually compute the lexicographical ranks of all t-blocks more efficiently.

▶ **Lemma 9.** *Let w be an input string of length n and $1 \le t \le n$ be an integer. Given the data structure for $\mathsf{ShortLCE}_t$ queries of Theorem 8 for w, we can sort all t-blocks of w in lexicographic order in $O(zt + \frac{n}{\sqrt{t}})$ time using $O(zt + \frac{n}{t})$ working space, where $z = |\mathsf{LZ}(w)|$.*

Proof. We insert new (non-branching) nodes to $2t$-$\mathsf{TST}(w)$ such that every t-gram in w is represented by an explicit node. This increases the size of the tree by a constant factor. We also associate each node u such that $|str(u)| = t$ with the lexicographical rank of the t-gram $str(u)$ among all t-grams in w. Then, we associate each leaf ℓ of the tree such that $|str(\ell)| \ge t$ with its ancestor v which represents a t-gram. All these can be preformed in $O(zt)$ total time by standard depth-first traversals on the tree.

Then, for each t-block $b_i = w[i..i+t-1]$, we can access a leaf ℓ of $2t$-$\mathsf{TST}(w)$ such that $str(\ell)[1..t] = w[i..i+t-1]$ in $O(1)$ time using the algorithm of Theorem 8, and we return the rank of the ancestor v of ℓ that represents $b_i = w[i..i+t-1]$. Since there are $O(\frac{n}{\sqrt{t}})$ t-blocks in w, it takes a total of $O(zt + \frac{n}{\sqrt{t}})$ time. The working space is $O(zt + \frac{n}{t})$ by Theorem 8. ◀

There is an alternative algorithm to sort the t-blocks, as follows:

▶ **Lemma 10.** *For any string w of length n over an alphabet of size σ, any integer $1 \le t \le n$, we can sort all t-blocks in lexicographic order in $O(n \log \sigma)$ time using $O(zt)$ working space, where $z = |\mathsf{LZ}(w)|$.*

Proof. We use t-$\mathsf{TST}(w)$ and the reversed de Bruijn graph of order t. We associate each leaf of the tree representing a t-gram with its lexicographical rank among all leaves in the tree.

Let r be the graph node which represents $w[n] = \$$. We simply traverse the graph while scanning the input string w from right to left. For each $1 \le i \le n$, this gives us the graph node representing $b_i = w[i..i+t-1]$ and hence the corresponding leaf of t-$\mathsf{TST}(w)$.

t-$\mathsf{TST}(w)$ and the reversed de Bruijn graph can be constructed in $O(n \log \sigma)$ time with $O(zt)$ working space. The ranks of the leaves in t-$\mathsf{TST}(w)$ can be easily computed in $O(zt)$ time by a standard tree traversal. Traversing the reversed de Bruijn graph takes $O(n \log \sigma)$ time. Hence the lemma holds. ◀

For each $i \in S(t)$, let r_i be the rank of the t-block $b_i = w[i..i+t-1]$ computed by any of the algorithms above. Clearly $r_i \in [1..n]$. For simplicity, assume $\sqrt{t}$ is an integer. For each position $i \in D$ (where D is the underlying t-difference cover), let $\#_i = r_i r_{i+t} \cdots r_{i+m_i t}$, where $m_i = \frac{n-i+1}{\sqrt{t}} - 1$. We create a string $\mathit{code}(w) = \#_1\$_1 \cdots \#_k\$_k$ of length $|S(t)| = O(\frac{n}{\sqrt{t}})$. Since each $\#_i$ is a string over the integer alphabet $[1..S(t)] \subset [1..n]$ and $|D| = O(\sqrt{t})$, we can regard $\mathit{code}(w)$ as a string over an integer alphabet of size $O(n)$. Then, we build the suffix array, the inverse suffix array, the LCP array [37] of $\mathit{code}(w)$ and an range minimum query (RMQ) data structure [3] for the LCP array. For any position $i \in S(t)$ on the original string w, we can compute its corresponding position i' on $\mathit{code}(w)$ as $i' = |\#_1\$_1\#_2\$_2 \cdots \#_{x-1}\$_{x-1}| + \frac{i-x}{t} + 1$ where $x = i \bmod t$. Now, $\mathsf{LongLCE}_t(i,j)$ query for two positions $i, j \in S(t)$ on the original string w reduces to an LCE query for the corresponding positions on $\mathit{code}(w)$, which can be answered in $O(1)$ time using an RMQ on the LCP array. All these arrays and the RMQ data structure can be built in $O(\frac{n}{\sqrt{t}})$ time [27, 28, 3].

▶ **Theorem 11.** *For any string of length n and integer $1 \leq t \leq n$, a data structure of size $O(\frac{n}{\sqrt{t}})$ can be constructed in $O(n \log \sigma)$ time using $O(zt + \frac{n}{t})$ working space such that subsequent $\mathsf{LongLCE}_t(i,j)$ queries for any $1 \leq i, j \leq n$ can be answered in $O(1)$ time, where $z = |\mathsf{LZ}(w)|$.*

Proof. We need $O(\frac{n}{\sqrt{t}})$ working space for the encoded string $\mathit{code}(w)$ and its suffix array plus LCP array enhanced with an RMQ data structure. Then the theorem follows from Theorem 8, and Lemma 9 or Lemma 10. ◀

3.4 Main result and variants

In what follows, let w be an input string of length n and $z = |\mathsf{LZ}(w)|$. By Theorem 8 and Theorem 11 shown in the previous subsections, we obtain the main theorem of this paper:

▶ **Theorem 12.** *For any integer $1 \leq t \leq n$, an LCE data structure of size $O(zt + \frac{n}{\sqrt{t}})$ can be constructed in $O(n \log \sigma)$ time with $O(zt + \frac{n}{\sqrt{t}})$ working space such that subsequent $\mathsf{LCE}(i,j)$ query for any $1 \leq i, j \leq n$ can be answered in $O(1)$ time.*

We can also obtain the following variants of our LCE data structure.

▶ **Corollary 13.** *For any integer $1 \leq t \leq n$, an LCE data structure of size $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ can be constructed in $O(n \log \sigma \log n)$ time with $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ working space such that subsequent $\mathsf{LCE}(i,j)$ query for any $1 \leq i, j \leq n$ can be answered in $O(1)$ time.*

Proof. The LCE data structure of Theorem 12 for $t = (\frac{n}{z})^{\frac{2}{3}} < n$ takes $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ space. Since we do not compute z, we are not able to compute the exact value of $(\frac{n}{z})^{\frac{2}{3}}$. However, we can find the value of t for which the difference between the actual size of t-$\mathsf{TST}(w)$ and $\lceil \frac{n}{\sqrt{t}} \rceil$ is smallest, by doubling-then-binary searches for t. Since the size of the resulting LCE data structure can be by a constant factor larger than the smallest variant of our LCE data structure, it is clearly bounded by $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$. The above method takes $O(n \log \sigma \log n)$ total time and uses $O(z^{\frac{1}{3}} n^{\frac{2}{3}})$ total working space. ◀

▶ **Corollary 14.** *For alphabets of size $\sigma \leq 2^{o(\log n)}$, an LCE data structure of size $o(n \log n)$* bits *can be constructed in $o(n \log^2 n)$ time with $o(n \log n)$* bits *of working space such that subsequent* $\mathsf{LCE}(i, j)$ *query for any $1 \leq i, j \leq n$ can be answered in $O(1)$ time.*

Proof. By plugging the well-known fact that $z = O(n/\log_\sigma n)$ into the result of Corollary 13, we get $O(n/(\log_\sigma n)^{\frac{1}{3}})$ for the space bound. Thus our data structure can be stored in $\mathcal{S}(n) = O(n(\log n)^{\frac{2}{3}}(\log \sigma)^{\frac{1}{3}})$ *bits* of space in the transdichotomous word RAM [16] with machine word size $\omega = \Theta(\log n)$. Hence, for alphabets of size $\sigma \leq 2^{o(\log n)}$, we obtain an LCE data structure with the claimed bounds. ◀

We can also obtain a new time-space trade-off LCE data structure. Observe that using the data structure of Theorem 8 for $1 \leq d \leq n$, we can answer $\mathsf{ShortLCE}_t$ queries for any $1 \leq t \leq n$ in $O(\max\{1, \frac{t}{d}\})$ time. Hence the following theorem holds.

▶ **Theorem 15.** *For any integers $1 \leq t' \leq t \leq n$, a data structure of size $O(zt' + \frac{n}{\sqrt{t}} + \frac{n}{t'})$ can be constructed in $O(n \log \sigma)$ time with $O(zt + \frac{n}{\sqrt{t}} + \frac{n}{t'})$ working space such that subsequent* $\mathsf{LCE}(i, j)$ *query for any $1 \leq i, j \leq n$ can be answered in $O(\frac{t}{t'})$ time.*

Theorem 15 implies the following: (1) By setting $t' = t$, we obtain Theorem 12. Moreover, by choosing also $t \leftarrow (\frac{n}{z})^{2/3}$, we obtain a data structure of size $O(z^{1/3}n^{2/3})$ answering LCE queries in constant time, which coincides with Corollary 13. This is the smallest data structure among the fastest data structures with two parameters t and t'. (2) By setting $t' = \sqrt{t}$ and for $t = n/z$, we get a data structure of size $O(\sqrt{nz})$ answering LCE queries in $O(\sqrt{\frac{n}{z}})$ time. This is the fastest data structure among the smallest data structures with two parameters t and t'. Note that when we do not know z, this data structure of at most $O(\sqrt{nz})$ space can be constructed in $O(n \log \sigma \log n)$ preprocessing time and $O(\sqrt{nz})$ working space as in Corollary 13. Although the parameters cannot be arbitrarily chosen, the space-query time product obtained here is optimal with fastest construction to date.

Moreover, we can reduce the zt term in the working space of Theorem 15 to zt' by increasing the preprocessing time. The bottleneck of the working space is in sorting t-blocks, i.e., Lemma 9 or Lemma 10. Since any two t-blocks can be compared in $O(\frac{t}{t'})$ time using $O(\frac{t}{t'})$ $\mathsf{ShortLCE}_{t'}$ queries, we can get the following theorem using any suitable comparison-based sorting algorithm instead of Lemma 9 or Lemma 10.

▶ **Theorem 16.** *We can construct the data structure of Theorem 15 in $O(\frac{n}{t'} \log \frac{n}{t} + n \log \sigma)$ time and $O(zt' + \frac{n}{\sqrt{t}} + \frac{n}{t'})$ working space.*

4 Lower bounds vs upper bounds for the LCE problem

Let $\mathcal{T}(n)$ and $\mathcal{S}(n)$ respectively denote the query time and data structure size (in *bits*) of an arbitrary LCE data structure for an input string of length n.

Brodal et al. [10] showed that in the non-uniform cell probe model, any RMQ data structure for a string of length n which uses $\frac{n}{t}$ bits of additional space for any $1 \leq t \leq n$ must take $\Omega(t)$ query time (i.e., $\Omega(t)$ character accesses or cell probes). Their proof assumes that each character in the string is stored in a separate cell, and counted the minimum number of character accesses required to answer an RMQ. Although their proof uses a binary string of length n where each character takes only a single bit, the above assumption is valid in a commonly accepted case that the underlying alphabet size is 2^ω, where ω denotes the size of each cell (i.e. machine word). Then, Bille et al. [9] showed that RMQ queries on any binary string of length n can be reduced to LCE queries on the same binary string, with

$\Theta(\log n)$ additional bits of space. This implies that, again assuming that each character is stored in a separate cell, any LCE data structure for a binary string of length n which uses $\mathcal{S}(n) = \frac{n}{t} + \Theta(\log n)$ additional bits of space must take $\mathcal{T}(n) = \Omega(t)$ query time, for parameter $1 \leq t \leq \frac{n}{\log n}$. Recently, Kosolobov [33] showed another result on time-space product trade-off lower bound in the non-uniform cell probe model, which can be formalized as follows:

▶ **Theorem 17** ([33]). *In the non-uniform cell probe model where each character is stored in a separate cell, for any $\mathcal{S}(n)$, there exists $\sigma = 2^{\Omega(\mathcal{S}(n)/n)}$ such that for any LCE data structure for a string over the alphabet $\Sigma = \{1, \ldots, \sigma\}$, which takes $\mathcal{S}(n)$ bits of space and answers LCE queries in $\mathcal{T}(n)$ time (i.e., with $\mathcal{T}(n)$ character accesses or cell probes), $\mathcal{T}(n)\mathcal{S}(n) = \Omega(n \log n)$ holds.*

The lower bound by Kosolobov is optimal for the considered range of the alphabet size $\sigma = 2^{\Omega(\mathcal{S}(n)/n)}$, since the data structure of Bille et al. [8] achieves $\mathcal{T}(n)\mathcal{S}(n) = O(n \log n)$.

Interestingly, our LCE data structure proposed in Section 3 reveals that there are some cases where the above lower bounds do not apply. For highly compressible strings where zt is dominated by $\frac{n}{\sqrt{t}}$, our LCE data structure of Theorem 12 takes $O(\frac{n \log n}{t})$ *bits* of space for $1 \leq t \leq n$ with machine word of size $\omega = \Theta(\log n)$. Hence, for parameter $1 \leq t' \leq \frac{\sqrt{n}}{\log n}$ we get $\mathcal{S}(n) = O(\frac{n}{t'})$. Since our data structure of Theorem 12 always achieves $\mathcal{T}(n) = O(1)$ for any parameter setting, we observe that Bille et al.'s lower bound do not apply for highly repetitive strings. Notice also that our LCE data structure of Corollary 14 achieves $\mathcal{T}(n)\mathcal{S}(n) = o(n \log n)$ for alphabet size $\sigma \leq 2^{o(\log n)}$. This shows that the alphabet size $\sigma = 2^{\Omega(\mathcal{S}(n)/n)}$ is important for Kosolobov's lower bound to hold.

Kosolobov [33] did suggest a possibility to overcome his lower bound when σ is small, and the input string can be *packed*, where $\log_\sigma n$ characters can occupy a memory cell, allowing the algorithm to read $\log_\sigma n$ characters with one memory access. We show below that this is also possible. An input string of length n can be considered as a bit string of length $n \log \sigma$. Let $t = \log n$, and first consider the $\mathsf{ShortLCE}_{\log n}$ queries on the bit string. When the original string is available in a packed representation, the longest common prefix of two substrings strings of length $\log n$ bits can be computed in constant time using no extra space using bit operations, namely, by taking the bitwise exclusive or (XOR) and computing the position of the most significant set bit (msb), or without msb, by multiple lookups on a table of total size $o(n)$ bits. Next, consider the $\mathsf{LongLCE}_{\log n}$ queries on the bit string. By simply using the same data structure as described in Section 3.3 for the bit string of length $n \log \sigma$, we can answer $\mathsf{LongLCE}_{\log n}$ queries in constant time using a data structure of size $O(\frac{n \log \sigma}{\sqrt{\log n}} \log(n \log \sigma)) = O(n\sqrt{\log n} \log \sigma)$ bits. Using the two queries, we can answer an LCE query for arbitrary positions i, j of the original string in constant time with $\lfloor (\mathsf{LCE}(i \cdot \log \sigma, j \cdot \log \sigma)) / \log \sigma \rfloor$. Since the size of the data structure is $\mathcal{S}(n) = O(n\sqrt{\log n} \log \sigma)$ bits, we obtain $\mathcal{T}(n)\mathcal{S}(n) = o(n \log n)$ for $\sigma \leq 2^{o(\sqrt{\log n})}$. Our LCE data structure based on truncated suffix trees is superior for larger σ, and also when the input string is highly repetitive and compressible since it does not require the original string.

5 Conclusions and open questions

In this paper, we presented an LCE data structure which uses $O(zt + \frac{n}{\sqrt{t}})$ words of space and answers in LCE queries in $O(1)$ time, for parameter $1 \leq t \leq \sqrt{n}$. This data structure can be constructed in $O(n \log \sigma)$ time with $O(zt + \frac{n}{\sqrt{t}})$ working space. Using the fact that

$z = O(n/\log_\sigma n)$ and suitably choosing t, our method achieves the first $O(1)$-time sub-linear space LCE data structure for alphabets of size $\sigma \leq 2^{o(\log n)}$.

An interesting open question is whether we can improve the total space requirement to $O(zt+\frac{n}{t})$. The bottleneck is the $\mathsf{LongLCE}_t$ data structure that uses $O(zt+\frac{n}{\sqrt{t}})$ space. Another open question is whether we can compute the size z of the Lempel-Ziv 77 factorization in $O(n\log\sigma)$ time with sub-linear working space. This is motivated for computing the value of t which optimizes our space bound $O(zt+\frac{n}{t})$. A little has been done in this line of research: Nishimoto et al. [40] showed how to compute the Lempel-Ziv 77 factorization in $O(n\mathrm{polylog}(n))$ time with $O(z\log n\log^* n)$ working space. Fischer et al. [15] showed an algorithm which computes an approximation of the Lempel-Ziv 77 factorization of size $(1+\epsilon)z$ in $O(\frac{1}{\epsilon}n\log n)$ time with $O(z)$ working space, for any $0 < \epsilon \leq 1$.

Another direction of further research is to give a tighter upper bound for the size of the t-truncated suffix trees than zt. We observed that there exists a string of length n for which zt is greater by a factor of $\sqrt{n}$ than the actual size of the t-truncated suffix tree for some t.

Acknowledgments. We thank Dmitry Kosolobov for explaining his work [33] to us.

References

1 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In *Proc. SODA 2015*, pages 562–571, 2015.

2 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. *CoRR*, abs/1610.03421, 2016.

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. Latin 2000*, pages 88–94, 2000.

4 Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

5 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

6 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. In *Proc. FSTTCS 2016*, pages 36:1–36:16, 2016.

7 Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *J. Comput. Syst. Sci.*, 86:171–180, 2017.

8 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Proc. CPM 2015*, pages 65–76, 2015.

9 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014.

10 Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.

11 Gerth Stølting Brodal, Rune B. Lyngsø, Christian N. S. Pedersen, and Jens Stoye. Finding maximal pairs with bounded gap. In *Proc. CPM 1999*, pages 134–149, 1999.

12 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. CPM 2003*, pages 55–69, 2003.

13 Bastien Cazaux, Thierry Lecroq, and Eric Rivals. Construction of a de Bruijn graph for assembly from a truncated suffix tree. In *LATA 2015*, pages 109–120, 2015.

14 Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing α-gapped repeats. In *Proc. LATA 2016*, pages 245–255, 2016.

15 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. *CoRR*, abs/1504.06647, 2015.

16 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

17 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17:52–54, 1986.

18 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Efficiently finding all maximal α-gapped repeats. In *Proc. STACS 2016*, pages 39:1–39:14, 2016.

19 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In *Proc. CPM 2016*, pages 5:1–5:13, 2016.

20 Sara Geizhals and Dina Sokol. Finding maximal 2-dimensional palindromes. In *Proc. CPM 2016*, pages 19:1–19:12, 2016.

21 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

22 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

23 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

24 Tomohiro I. Longest common extensions with recompression. In *Proc. CPM 2017*, 2017. To appear.

25 Shunsuke Inenaga. A faster longest common extension algorithm on compressed strings and its applications. In *Proc. PSC 2015*, pages 1–4, 2015.

26 Juha Kärkkäinen. Repetition-based text indexes. *Ph.D. thesis, University of Helsinki, Department of Computer Science*, 1999.

27 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

28 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM 2001*, pages 181–192, 2001.

29 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.

30 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS 1999*, pages 596–604, 1999.

31 Roman M. Kolpakov and Gregory Kucherov. Finding repeats with fixed gap. In *Proc. SPIRE 2000*, pages 162–168, 2000.

32 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC 2016*, pages 3–12, 2016.

33 Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *CoRR*, abs/1611.02891, 2016.

34 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.

35 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986.

36 Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

37 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

38 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 1-3(304):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.

39 Shintaro Narisada, Diptarama, Kazuyuki Narisawa, Shunsuke Inenaga, and Ayumi Shinohara. Computing longest single-arm-gapped palindromes in a string. In *Proc. SOFSEM 2017*, pages 375–386, 2017.

40 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In *Proc. PSC 2016*, pages 158–170, 2016.

41 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *MFCS 2016*, pages 72:1–72:15, 2016.

42 Nicola Prezza. In-place longest common extensions. *CoRR*, abs/1608.05100, 2016.

43 Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. ISAAC 2008*, pages 124–135, 2008.

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

45 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Proc. CPM 2016*, pages 1:1–1:10, 2016.

46 Luciana Vitale, Alvaro Martín, and Gadiel Seroussi. Space-efficient representation of truncated suffix trees, with applications to Markov order estimation. *Theor. Comput. Sci.*, 595:34–45, 2015.

47 P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

48 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977.