Compressed Suffix Trees: Efficient Computation and Storage of LCP-Values

SIMON GOG, The University of Melbourne
ENNO OHLEBUSCH, Ulm University

The suffix tree is a very important data structure in string processing, but typical implementations suffer from huge space consumption. In large-scale applications, compressed suffix trees (CSTs) are therefore used instead. A CST consists of three (compressed) components: the suffix array, the longest common prefix (LCP)-array and data structures for simulating navigational operations on the suffix tree. The LCP-array stores the lengths of the LCPs of lexicographically adjacent suffixes, and it can be computed in linear time. In this article, we present a new LCP-array construction algorithm that is fast and very space efficient. In practice, our algorithm outperforms alternative algorithms. Moreover, we introduce a new compressed representation of LCP-arrays.

Categories and Subject Descriptors: E.1 [**Data Structures**]; E.2 [**Data Storage Representations**]; E.4 [**Coding and Information Theory**]: Data Compaction and Compression; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures; sorting and searching*; H.3.1 [**Context Analysis and Indexing**]: *Indexing methods*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Algorithm engineering, LCP-array, text indexing, text compression, Burrows-Wheeler transform, wavelet tree, balanced parentheses

ACM Reference Format:
Gog, S. and Ohlebusch, E. 2011. Compressed suffix trees: Efficient computation and storage of LCP-values. ACM J. Exp. Algor. 18, 2, Article 2.1 (May 2013), 31 pages.
DOI: http://dx.doi.org/10.1145/2444016.2461327

1. INTRODUCTION

A suffix tree for a text (string) T of length n is a compact trie storing all the suffixes of T (see Section 2 for details). It is an extremely important data structure with applications in string matching, bioinformatics, and document retrieval, to mention only a few examples (see Gusfield [1997] for an extensive list of applications). The drawback of suffix trees is their huge space consumption of about 20 times the text size, even in carefully engineered implementations. To reduce their size, several authors provided compressed suffix trees (CSTs) or fully-compressed suffix trees (FCST) (for a survey, see, e.g., Sadakane [2007], Russo et al. [2008], Ohlebusch and Gog [2009], Fischer et al. [2009], Ohlebusch et al. [2010] and Navarro and Mäkinen [2007]). A CST of T can be divided into three components: (i) the suffix array SA, specifying the lexicographic order of T's suffixes; (ii) the longest common prefix (LCP)-array storing the lengths of

Authors' addresses: S. Gog, Computing and Information Systems Department, The University of Melbourne, VIC 3010, Parkville, Australia. E. Ohlebusch, Institute for Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1084-6654/2013/05-ART2.1 $15.00
DOI: http://dx.doi.org/10.1145/2444016.2461327

the longest common prefixes of lexicographically adjacent suffixes; and (iii) additional data structures for simulating navigational operations on the suffix tree.

Particular emphasis has been put on efficient construction algorithms for all three components of CSTs. (Here, "efficiency" encompasses both construction *time* and *space*.) This is especially true for the first component. In the last decade, much effort has gone into the development of efficient suffix array construction algorithms (SACAs) (for a survey, see Puglisi et al. [2007]). Although linear time direct SACAs have been known since 2003, experiments showed [Puglisi et al. 2007] that these were outperformed in practice by SACAs having a worst-case time complexity of $O(n^2 \log n)$. To date, however, one of the fastest SACA is a linear time algorithm [Nong et al. 2009] (see Section 3.1.2 for details).

As discussed in Section 3.2.1, today's best LCP-array construction algorithms (LACAs) [Kärkkäinen et al. 2009; Kasai et al. 2001] are linear time algorithms, but they suffer from poor memory locality. In Section 4, we present a space-efficient (using $2n$ bytes only) and fast LACA. Based on the observation that one cache miss takes approximately the time of 20 character comparisons, we try to trade character comparisons for cache misses. The algorithm uses the text T, the suffix array, and the Burrows-Wheeler Transform ($\mathsf{T}^{\mathsf{BWT}}$). Since most CSAs are based on $\mathsf{T}^{\mathsf{BWT}}$ anyway, we basically get it for free. Section 4.5 shows the significance of the algorithm. More precisely, experimental results show that our algorithm outperforms state-of-the-art algorithms [Kärkkäinen et al. 2009; Kasai et al. 2001] on non–highly repetitive data. For large texts, it is always faster than the previously best algorithms, while at the same time using less than half of the space. The superiority of our new LACA varies with the text size (the larger the better), the alphabet size (the smaller the better), the number of "large" values in the LCP-array (the less the better), and the runs in the $\mathsf{T}^{\mathsf{BWT}}$ (the more the better). The algorithm works particularly well on two types of data that are of importance in practice: long DNA sequences (small alphabet size) and large collections of XML documents (long runs in the BWT).

As discussed in Section 3.2.2, there are two main types of compressed LCP-representations: the representation in suffix array order [Kurtz 1999; Brisaboa et al. 2009] and the representation in text order [Sadakane 2002; Fischer et al. 2009]. In Section 5, we will present a third alternative: the *tree compressed* representation. The idea is to use the topology information of the CST to compress the LCP information.

This article is organized as follows: After introducing notations and basic concepts in Section 2, we revisit the design of CSTs in Section 3. We present the different implementation choices for each component and its construction. In Section 4, we review and evaluate existing LCP-construction algorithms and present our new solution. Finally, we present and study our new compressed LCP-representation in Section 5.

2. PRELIMINARIES

Let Σ be an ordered alphabet of constant size σ whose smallest element is the so-called sentinel character \$. In the following, T is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $0 \le i \le n-1$, $\mathsf{T}[i]$ denotes the *character at position* i in T. For $i \le j$, $\mathsf{T}[i..j]$ denotes the *substring* of T starting with the character at position i and ending with the character at position j. Furthermore, $\mathsf{T}_i = \mathsf{T}[i..n-1]$ denotes suffix i of T.

The suffix tree of T has the following properties.

—It is a rooted directed ordered tree with exactly n leaves numbered 0 to $n-1$.
—Each inner node has at least two children. We call such a node branching.
—Each edge is labeled with a nonempty substring of T.
—No two edges out of a node can have edge-labels beginning with the same character.

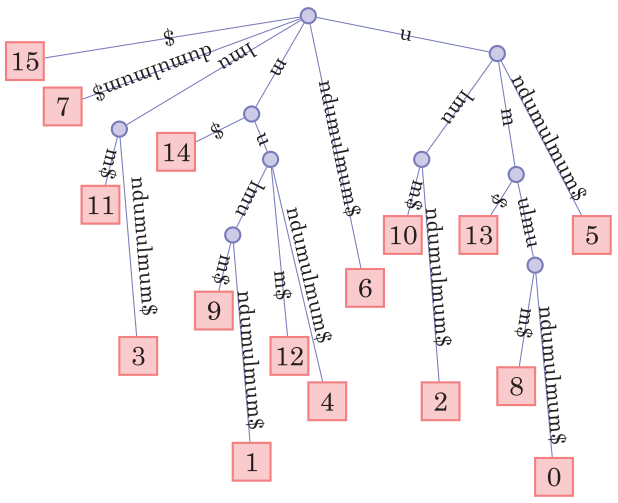


Fig. 1. The suffix tree of string T=`umulmundumulmum$`.

i	SA	LCP	$\mathsf{T}^{\mathsf{BWT}}$	T
0	15	-1	m	$
1	7	0	n	dumulmum$
2	11	0	u	lmum$
3	3	3	u	lmundumulmum$
4	14	0	u	m$
5	9	1	u	mulmum$
6	1	5	u	mulmundumulmum$
7	12	2	l	mum$
8	4	2	l	mundumulmum$
9	6	0	u	ndumulmum$
10	10	0	m	ulmum$
11	2	4	m	ulmundumulmum$
12	13	1	m	um$
13	8	2	d	umulmum$
14	0	6	$	umulmundumulmum$
15	5	1	m	undumulmum$
16		-1		

Fig. 2. The suffix array of the text T=`umulmundumulmum`, the corresponding LCP-array, and the Burrows-Wheeler transformed string $\mathsf{T}^{\mathsf{BWT}}$.

—The concatenation of the edge-labels on the path from the root to a leaf i equals suffix $T[i..n-1]$.

Figure 1 depicts the suffix tree of the string T=`umulmundumulmum$`.

The suffix array of T is an array SA$[0..n-1]$ of length n that contains the lexicographic ordering of all suffixes of T, that is, SA[0] contains the starting position of the lexicographically smallest suffix of T, SA[1] contains the starting position of the lexicographically second smallest suffix, and so on. The suffix array is the sequence of all leaf node labels of the suffix tree of T (from left to right). For example, the suffix array corresponding to the suffix tree in Figure 1 is depicted in Figure 2. Clearly, a suffix array occupies $n \log n$ bits. Its inverse permutation ISA is called the *inverse suffix array.*

Information about common substrings of T is stored in the LCP-array. It stores for two lexicographically adjacent suffixes in SA the length of their longest common prefix. If we use $lcp(u, v)$ to denote the length of the longest common prefix between two strings u and v, then the LCP-array is an array of integers in the range 0 to n such that LCP$[0] = -1$, LCP$[n] = -1$, and LCP$[i] = lcp(\mathsf{T}_{\mathsf{SA}[i-1]}, \mathsf{T}_{\mathsf{SA}[i]})$ for $1 \le i \le n-1$. In

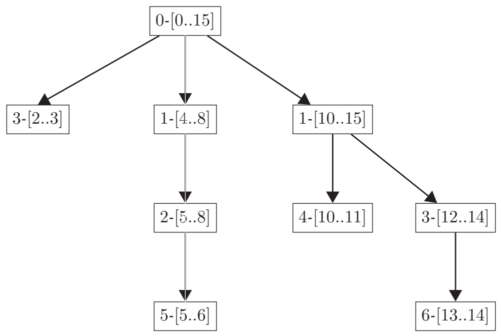


Fig. 3. The LCP-interval tree of the LCP-array from Figure 2.

Figure 2, we can see the LCP-array for our running example. Note that the entry at position 0 is defined to be -1 only for technical reasons (i.e., to avoid case distinctions in the pseudocode). Later, in the real implementation, we will define $\mathsf{LCP}[0] = 0$. In uncompressed form, the LCP-array takes $n \log n$ bits like the uncompressed suffix array. For $i < j$, a range minimum query $\mathsf{RMQ}(i, j)$ on the interval $[i..j]$ in the LCP-array returns an index k such that $\mathsf{LCP}[k] = \min\{\mathsf{LCP}[l] \mid i \leq l \leq j\}$. It is not difficult to show that $lcp(\mathsf{T}_{\mathsf{SA}[i]}, \mathsf{T}_{\mathsf{SA}[j]}) = \mathsf{LCP}[\mathsf{RMQ}(i+1, j)]$.

According to Abouelhoda et al. [2004], an interval $[i..j]$, where $1 \leq i < j \leq n$, in an LCP-array is called an *LCP-interval of LCP-value* ℓ (denoted by ℓ-$[i..j]$) if

(1) $\mathsf{LCP}[i] < \ell$,
(2) $\mathsf{LCP}[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$,
(3) $\mathsf{LCP}[k] = \ell$ for at least one k with $i + 1 \leq k \leq j$,
(4) $\mathsf{LCP}[j+1] < \ell$.

Every index k, $i + 1 \leq k \leq j$, with $\mathsf{LCP}[k] = \ell$ is called ℓ-index. Note that each LCP-interval has at most $|\Sigma| - 1$ many ℓ-indices.

An LCP-interval m-$[p..q]$ is said to be *embedded* in an LCP-interval ℓ-$[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq p < q \leq j$) and $m > \ell$. The interval $[i..j]$ is then called the interval *enclosing* $[p..q]$. If $[i..j]$ encloses $[p..q]$ and there is no interval embedded in $[i..j]$ that also encloses $[p..q]$, then $[p..q]$ is called a *child interval* of $[i..j]$. This parent–child relationship constitutes a tree, which is called the *LCP-interval tree*. It is not difficult to see that the LCP-interval tree is a representation of the suffix tree. Each node in the suffix tree corresponds to an interval in the LCP-interval tree. Figure 3 shows the lcp-interval tree corresponding to the suffix tree from Figure 1.

Kärkkäinen et al. [2009] called the following special permutation of the LCP-array the *permuted longest common prefix* (PLCP) array:

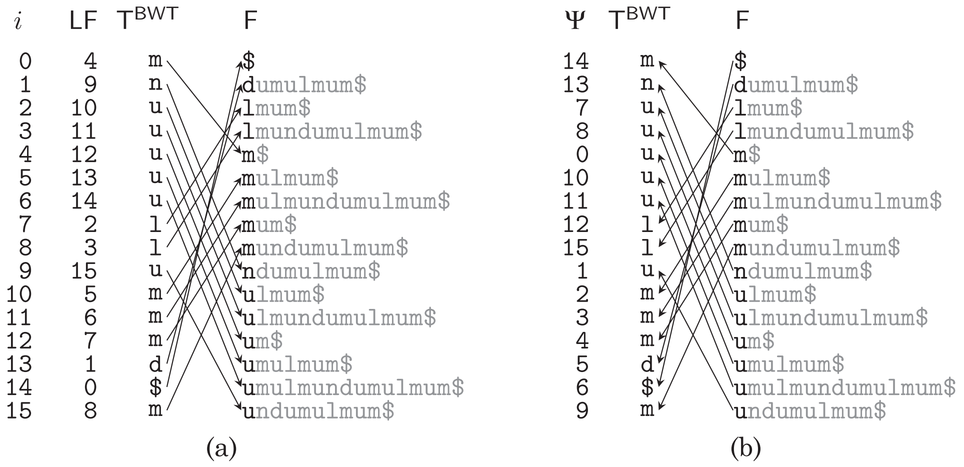
$$\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i] \text{ or equivalently } \mathsf{LCP}[\mathsf{ISA}[j]] = \mathsf{PLCP}[j]. \tag{1}$$

While the values in LCP appear in the order of the suffix array—called SA-*order*—the values in PLCP appear in text order.

The text $\mathsf{T}^{\mathsf{BWT}}$ obtained from T by the *Burrows-Wheeler transform* can be defined as follows:

$$\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}[\mathsf{SA}[i] - 1 \bmod n]. \tag{2}$$

Figure 4 shows an example. In uncompressed form, $\mathsf{T}^{\mathsf{BWT}}$ takes $n \log \sigma$ bits of memory, which is exactly the size of the input string. The Burrows-Wheeler Transform is popular in lossless data compression because it is computationally easy to compress.



i	LF	$\mathsf{T}^{\mathsf{BWT}}$	F
0	4	`m`	`$`
1	9	`n`	`dumulmum$`
2	10	`u`	`lmum$`
3	11	`u`	`lmundumulmum$`
4	12	`u`	`m$`
5	13	`u`	`mulmum$`
6	14	`u`	`mulmundumulmum$`
7	2	`l`	`mum$`
8	3	`l`	`mundumulmum$`
9	15	`u`	`ndumulmum$`
10	5	`m`	`ulmum$`
11	6	`m`	`ulmundumulmum$`
12	7	`m`	`um$`
13	1	`d`	`umulmum$`
14	0	`$`	`umulmundumulmum$`
15	8	`m`	`undumulmum$`

(a)

Ψ	$\mathsf{T}^{\mathsf{BWT}}$	F
14	`m`	`$`
13	`n`	`dumulmum$`
7	`u`	`lmum$`
8	`u`	`lmundumulmum$`
0	`u`	`m$`
10	`u`	`mulmum$`
11	`u`	`mulmundumulmum$`
12	`l`	`mum$`
15	`l`	`mundumulmum$`
1	`u`	`ndumulmum$`
2	`m`	`ulmum$`
3	`m`	`ulmundumulmum$`
4	`m`	`um$`
5	`d`	`umulmum$`
6	`$`	`umulmundumulmum$`
9	`m`	`undumulmum$`

(b)

Fig. 4. The arrows depict the (a) LF-function and (b) the Ψ-function and how it can be expressed by $\mathsf{T}^{\mathsf{BWT}}$ and F for the running example T=`umulmundumulmum$`.

The LF-*mapping* or LF-*function* is defined by

$$\mathsf{LF}[i] = \mathsf{ISA}[(\mathsf{SA}[i] - 1) \bmod n]. \tag{3}$$

That is, for suffix $j = \mathsf{SA}[i]$ the value of $\mathsf{LF}[j]$ is the index of suffix $j - 1$ in the suffix array (see Figure 4). Its long name *last-to-first column mapping* stems from the fact that it maps the last column $L = \mathsf{T}^{\mathsf{BWT}}$ to the first column F, where F contains the first character of the suffixes in the suffix array, that is, $F[i] = \mathsf{T}[\mathsf{SA}[i]]$.

The Ψ-*function* goes the other way around:

$$\Psi[i] = \mathsf{ISA}[(\mathsf{SA}[i] + 1) \bmod n]. \tag{4}$$

So for suffix $j = \mathsf{SA}[i]$, the value of $\Psi[j]$ is the index of suffix $j + 1$ in the suffix array (see Figure 4). Thus, $\mathsf{LF}[i]$ or $\Psi[i]$ can be calculated in constant time in the presence of SA and ISA. Also note that Ψ and LF are inverse permutations of each other.

Given a balanced parentheses sequence of length n, the following operations can be supported in constant time with $o(n)$ bits of extra space (see Gog and Fischer [2010], Sadakane and Navarro [2010], and references therein): $rank_{(}(i)$ returns the number of opening parentheses up to position i, $select_{(}(i)$ returns the position of the i-th opening parenthesis, $find\_close(i)$ returns the position of the closing parenthesis matching the opening parenthesis at position i, $enclose(i)$ for an opening parenthesis at position i returns the position j of the opening parenthesis such that $(j, find\_close(j))$ encloses $(i, find\_close(i))$ most tightly, and $double\_enclose(i, j)$ for two opening parentheses at positions i and j so that $find\_close(i) < j$ returns the position of the opening parenthesis of the parenthesis pair that most tightly encloses $(i, find\_close(i))$ and $(j, find\_close(j))$.

Of course, a balanced parentheses sequence can be viewed as a bit string, where, for example, 1 stands for an opening parenthesis and 0 denotes a closing parenthesis. Jacobson [1989] and Clark [1996] showed that rank and select queries on arbitrary bit strings can be answered in constant time using only sublinear space.

3. COMPRESSED SUFFIX TREES

A compressed suffix tree (CST) on T consists of three components:

(1) a compressed suffix array (CSA),
(2) a compressed LCP-array,

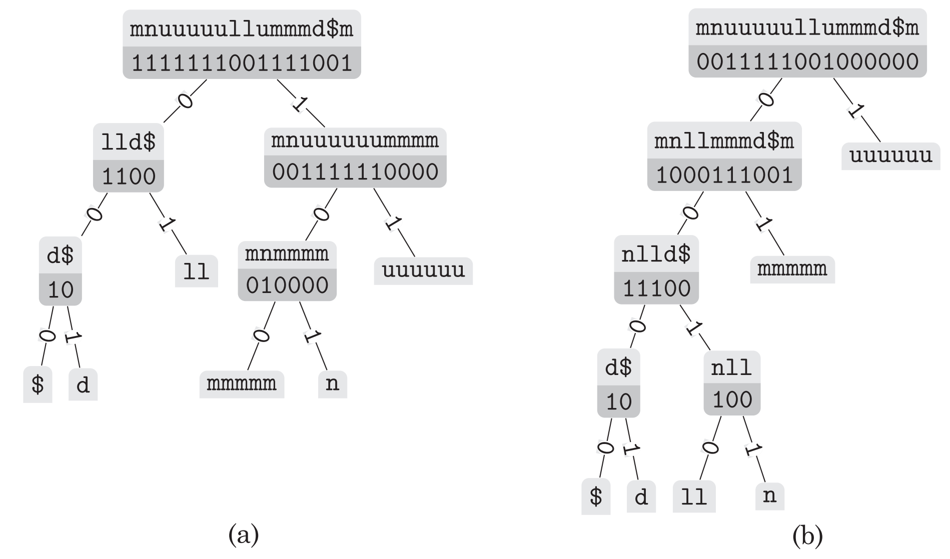


Fig. 5. For $\mathsf{T}^{\mathsf{BWT}}$=mnuuuuullummmd$m, we show (a) the balanced wavelet tree of depth σ and (b) the wavelet tree in Huffman shape.

(3) a succinct data structure for simulating navigational operations on the suffix tree.

The goal is to construct and store each of these three components efficiently.

3.1. Compressed Suffix Arrays

3.1.1. Compression. There are dozens of papers on CSAs, offering different trade-offs between occupied space and access time t_{SA}. The reader is referred to Navarro and Mäkinen [2007] for a survey paper on the subject. Here, we briefly review the wavelet tree devised by Grossi et al. [2003] because (i) it provides the basis for one of the best CSAs, and (ii) we will use it later for storing LCP-values efficiently. First, we assign to each character $c \in \Sigma$ a unique codeword $code(c) \in \{0,1\}^*$. For example, we can assign to each character a fixed-length binary codeword in $\{0,1\}^{\lceil \log \sigma \rceil}$ or use a prefix code like the Huffman code [Huffman 1952]. The wavelet tree is now formed as follows (see Figure 5(a) for an example): The root node v is on level 0 and contains the text S of length n. We store a bit vector b_v of length n, which contains at position i the most significant bit $code(S[i])[0]$ of character $S[i]$. Now the tree is built recursively: If a node v contains a text S_v that contains at least two different characters, then we create two child nodes v_ℓ and v_r. All characters that are marked with a 0 go to the left node v_ℓ, and all other characters go to the right node. Thereby, the order of the characters is preserved. Note that we store in the implementation only the bit vectors, the rank and select data structures for the bit vectors, and two maps *c_to_leaf*and *leaf_to_c*, which map each character c to its leaf node and vice versa. With a fixed-length code the depth of the tree is bounded by $\lceil \log \sigma \rceil$. As each level, except the last one, contains n bits, we get a space complexity of $n \log \sigma + o(n)$ bits for a constant-size alphabet.

With a fixed-length code, the wavelet tree of $\mathsf{T}^{\mathsf{BWT}}$ supports the operations $rank_{\mathsf{T}^{\mathsf{BWT}}}(i,c)$ and $select_{\mathsf{T}^{\mathsf{BWT}}}(i,c)$ in $\mathcal{O}(\log \sigma)$ time, where $rank_{\mathsf{T}^{\mathsf{BWT}}}(i,c)$ returns the number of occurrences of character c in the prefix $\mathsf{T}^{\mathsf{BWT}}[0..i-1]$ and $select_{\mathsf{T}^{\mathsf{BWT}}}(i,c)$ returns the position of the i-th occurrence of character c in $\mathsf{T}^{\mathsf{BWT}}$. Using these operations, the time t_{LF} (t_Ψ) to compute $\mathsf{LF}(i)$ ($\Psi(i)$) is in $\mathcal{O}(\log \sigma)$. Additionally, the character at index

i in $\mathsf{T}^{\mathsf{BWT}}$ can be retrieved in $\mathcal{O}(\log \sigma)$ time from the wavelet tree, without storing $\mathsf{T}^{\mathsf{BWT}}$ itself. Moreover, the wavelet tree of $\mathsf{T}^{\mathsf{BWT}}$ supports backward search [Ferragina and Manzini 2000]. In practice, a Huffman-shaped wavelet tree (see Figure 5) of $\mathsf{T}^{\mathsf{BWT}}$ performs better, but it cannot guarantee that the operations run in $\mathcal{O}(\log \sigma)$ time in the worst case.

Using a sampling parameter s_{SA}, a practical CSA is then obtained by storing every s_{SA}-th entry of the suffix array SA as sample. An entry $\mathsf{SA}[i]$ can be retrieved from the sampled suffix array as follows: If $i = k \cdot s_{\mathsf{SA}}$, then $\mathsf{SA}[i]$ is sampled and can be returned in constant time. Otherwise, we apply the LF function d times until we reach a sampled value $x = \mathsf{SA}[k \cdot s_{\mathsf{SA}}]$ and infer $\mathsf{SA}[i] = x + d \bmod n$. Alternatively, the Ψ function can be used instead of LF. If needed, the ISA can be sampled similarly.

3.1.2. Construction. The implementations of the first linear time suffix array construction algorithms (SACAs) [Kärkkäinen and Sanders 2003; Ko and Aluru 2003] take about $10n$ bytes, that is, in the worst case 20 times the size of the resulting CST. After that, much work was done to improve the practical running time and space for SACAs. Puglisi et al. [2007] published a survey article that reviews 20 different SACAs that were proposed until 2007. At that time, the fastest SACA in practice used $5n$ to $6n$ bytes and had a theoretical worst-case time complexity of $\mathcal{O}(n^2 \log n)$. Algorithms that use only $5n$ bytes are called *lightweight* algorithms. This term was coined by Manzini and Ferragina [2004] when they presented their first lightweight SACA (basically only the text and SA are in memory). Today, the SACA implementation of Yuta Mori called `libdivsufsort`[1] is the fastest. His lightweight algorithm has a worst-case time complexity of $\mathcal{O}(n \log n)$. Moreover, there is also a fast implementation of a linear time SACA. The algorithm was devised by Nong et al. [2009], and the implementation was improved by Yuta Mori. It is also lightweight and takes about twice the time of the `libdivsufsort` implementation. An alternative way to construct the CSA is to first compute $\mathsf{T}^{\mathsf{BWT}}$ directly from T. Okanohara and Sadakane [2009] recently presented an algorithm for this task that needs about $2n$ bytes of space. Experiments in Beller et al. [2011] showed that the runtime of their implementation is about twice the runtime of the fastest SACA for small inputs (200 MB). Then, one can construct the wavelet tree of $\mathsf{T}^{\mathsf{BWT}}$ and use LF (or Ψ) to generate the SA and ISA samples in $\mathcal{O}(n \cdot t_{\mathsf{LF}})$ (or $\mathcal{O}(n \cdot t_{\Psi})$) time.

For the construction of the suffix array SA, we use `libdivsufsort`. Afterward, we store SA to disk, because we only access SA in sequential order and, therefore, SA can be streamed from disk. Next, we construct the CSA by computing $\mathsf{T}^{\mathsf{BWT}}$ from SA using Equation (2_) (note that we only need T in memory if we write $\mathsf{T}^{\mathsf{BWT}}[i]$ directly to disk in each step) and constructing its wavelet tree. The latter can be done with constant space by streaming, or in space two times the text size in a straightforward implementation. A CSA is then constructed by streaming SA another time and taking every s_{SA}-th entry as SA sample.

3.2. Compressed LCP-Arrays

In most CST implementations, the compressed LCP-array is obtained from the uncompressed LCP-array. For this reason, we first review LCP-array construction algorithms (LACAs).

3.2.1. Construction. Kasai et al. [2001] proposed the first linear time LACA that does not use the suffix tree. Their algorithm uses T, SA, ISA, and of course the LCP-array (in total, $13n$ bytes for a text of length $< 2^{32}$ and an ASCII alphabet). The main advantage

[1]Version 2.0.1 is available at http://code.google.com/p/libdivsufsort.

of their algorithm is that it is simple and uses at most $2n$ character comparisons. But its poor locality behavior results in many cache misses, which is a severe disadvantage on current computer architectures. Mäkinen [2003] and Manzini [2004] reduced the space occupancy of that algorithm to $9n$ bytes, and Manzini [2004] showed that his solution has a slowdown of about 5% to 10% compared to the $13n$-bytes solution. He also proposed an even more space-efficient (but slower) algorithm that overwrites the suffix array. Recently, Kärkkäinen et al. [2009] developed the so-called Φ-algorithm, which computes PLCP, that is, the LCP-array in text order. The Φ-algorithm takes only $5n$ bytes and is much faster than Kasai et al.'s algorithm because it has a better locality of references. However, in virtually all applications, LCP-values are required to be in suffix array order, so that in a final step, the PLCP-array must be converted into the LCP-array. Although this final step suffers (again) from a poor locality behavior, the overall algorithm is still faster than Kasai et al.'s. They also proposed two sparse versions of the Φ-algorithm, which—for an integer $q \geq 1$—have a running time of $\mathcal{O}(nq)$ and use only $5n + \frac{4}{q}n$ and $n + \frac{4}{q}n$ bytes of space. In a different approach, Puglisi and Turpin [2008] tried to avoid cache misses by using the difference cover method of Kärkkäinen and Sanders [2003]. The worst-case time complexity of their algorithm is $\mathcal{O}(nv)$ and the space requirement is $n + \mathcal{O}(n/\sqrt{v} + v)$ bytes, where v is the size of the difference cover. Experiments showed that the best runtime is achieved for $v = 64$, but the algorithm is still slower than Kasai et al.'s. This is because it uses a succinct data structure for constant time range minimum queries, which takes considerable time in practice. Very recently, Beller et al. [2011] showed that the LCP-array can be directly computed from the wavelet tree of $\mathsf{T}^{\mathsf{BWT}}$, but that algorithm is slower than the Φ-algorithm. To sum up, currently, the fastest lightweight LACA is that of Kärkkäinen et al. [2009]. In Section 4, we propose a new lightweight linear time algorithm that is specially designed for the use in CST construction because it reuses temporary results of the CST construction process.

3.2.2. Compression. Here, we categorize existing concepts from a practical perspective and refer to Ohlebusch et al. [2010] for a list of theoretical results. There are two main types of compressed LCP-representations.

(1) The *representation in suffix array order*. The length of the longest common prefix of the suffixes $\mathsf{SA}[i-1]$ and $\mathsf{SA}[i]$ is stored at position i. Examples of this representation are the the solution of Kurtz [1999], the direct accessible code solution of Brisaboa et al. [2009], and the usage of a wavelet tree of the LCP-array.
(2) The *representation in text order*. The length of the longest common prefix of the suffixes $\mathsf{SA}[i-1]$ and $\mathsf{SA}[i]$ is stored at position $\mathsf{SA}[i]$. This representation is called *PLCP-representation.* To recover the LCP-entry at position i, we have to access the suffix array at position i, which is very slow on a CST. The advantage of this ordering is that it uses only a fraction of the space of the representation in suffix array order. An example for the PLCP-representation is the solution proposed by Sadakane [2002]. This solution takes only $2n + o(n)$ bits and has to perform only one *select* operation on top of the suffix array access. Fischer et al. [2009] further compress the size for this kind of representation to at most $n \cdot H_k \cdot (2 \log \frac{1}{H_k} + \mathcal{O}(1))$ bits, where H_k denotes the empirical entropy of order k of the input text. However, experiments of Cánovas and Navarro [2010] show that the space consumption for different kinds of texts is not smaller than the implementation of the $2n + o(n)$ bits solution of Sadakane, but the query time is surprisingly more or less equal. Fischer [2010b] proposed another data structure which is basically the select data structure of Sadakane but needs the original text to work fast in practice.

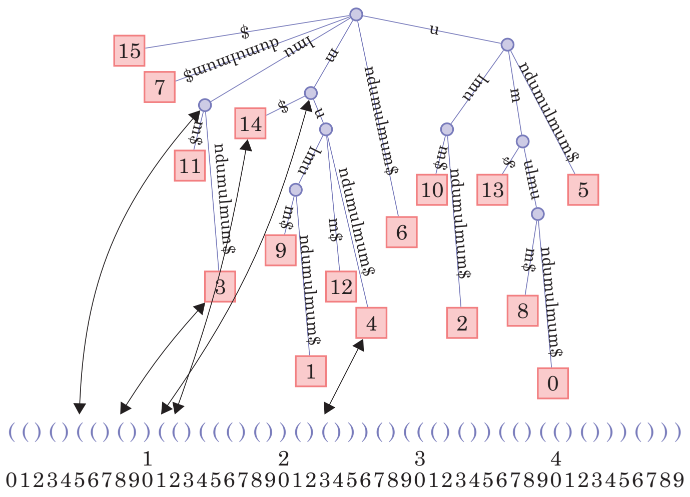


Fig. 6. Top: Suffix tree of the running example. Bottom: BPS$_{dfs}$ of the suffix tree and the positions of the parentheses. The arrows are explained in the text.

In Section 5, we will present a third alternative: the *tree compressed* representation. The idea is to use the topology information of the CST to compress the LCP information. In combination with further compression techniques like the use of wavelet trees and the idea of sampling LCP-values (see Sirén [2010]), we obtain a very attractive solution.

3.3. Succinct Tree Navigation

There are two main lines of research to simulate navigational operations on the suffix tree with the help of a succinct data structure:

(1) storing the tree topology *explicitly* by a sequence of balanced parentheses,
(2) deriving the tree topology *implicitly* from intervals in the LCP-array.

A list of existing concepts can be found in [Ohlebusch et al. 2010]. Here, we will briefly review one particular representative of each of the two research lines because they will be used later. Sadakane [2007] proposed to use the balanced parentheses sequence BPS$_{dfs}$ to represent the tree topology explicitly. The BPS$_{dfs}$ can be constructed by a depth-first search traversal of the (uncompressed) suffix tree as follows: At each node v (starting at the root), we write an opening parenthesis, recursively process the child nodes of v, and write a closing parenthesis afterward (see Figure 6). While the time complexity for the construction is optimal, the space complexity is not, because in the worst case, we need at least $n \log n$ bits for the stack of the depth-first search traversal. More importantly, one should avoid the construction of the (uncompressed) suffix tree. Hon and Sadakane [2002] showed how to construct BPS$_{dfs}$ solely from CSA, a compressed LCP-representation, and a succinct stack of size $\mathcal{O}(n)$ bits in time $\mathcal{O}(n \log^{\epsilon} n)$. Välimäki et al. [2009] describe a similar algorithm that takes only $\mathcal{O}(n)$ bits in total but has a time complexity of $\mathcal{O}(n \log n)$. Note that in both solutions, the final output has to be in memory, and this adds $4n$ bits to the space complexities. To sum up, the space-efficient construction of BPS$_{dfs}$ is a complex task that needs only $\mathcal{O}(n)$ bits, but the hidden constant is not negligible. The BPS$_{dfs}$ needs up to $4n + o(n)$ bits for the sequence of parentheses because the suffix tree has n leaves and up to $n - 1$ additional internal nodes (the $o(n)$-term comes from the extra data structures for navigation).

A node in the tree is represented by the position of the corresponding opening parenthesis in BPS$_{dfs}$. Note that a node is a leaf if and only if its corresponding opening parenthesis is immediately followed by a closing parenthesis. For example, in Figure 6 the leaf labeled with 14 corresponds to the opening parenthesis at position 12. While the construction of the BPS_{dfs} is rather complex, the simulation of navigational operations on the suffix tree is quite easy. For instance, the parent operation in the tree corresponds to the *enclose* operation in BPS_{dfs}. Figure 6 shows an example. The leaf node v labeled with 3 corresponds to position 8 in BPS_{dfs}, and $enclose(8) = 5$ corresponds to the position of the opening parenthesis of the parent of v. A second tree operation that also boils down to an operation on the BPS_{dfs} is to determine the lowest common ancestor (*lca*) of two nodes. This operation corresponds to the *double_enclose* operation. Figure 6 exemplifies that the lowest common ancestor of the leaf nodes labeled with 14 and 4 are represented by positions 12 and 23, and $double\_enclose(12, 23) = 11$ corresponds to the position of the lowest common ancestor of the two nodes. The reader is referred to Sadakane [2007] for a detailed description of how the BPS_{dfs} can be used to simulate other navigational operations in the suffix tree.

Ohlebusch et al. [2010] proposed to use the balanced parentheses sequence BPS_{sct} of the Super-Cartesian tree of the LCP-array and an additional bit vector rc to simulate navigational operations. The BPS_{sct} of an LCP-array of size n consists of exactly $2n$ parentheses and the bit vector rc of n bits, so it occupies $3n + o(n)$ bits in total. Every element is represented by an opening parenthesis and a corresponding closing parenthesis. Algorithm 1 shows pseudocode for the construction. We first initialize BPS_{sct} in Line 01 to a sequence of $2n$ closing parentheses. On Line 03, we set the current position in BPS_{sct} to 0 and initialize a stack s on Line 04. Next, we scan the LCP-array from left to right. For every element $\mathsf{LCP}[i]$, we check if there exists previous elements on the stack s that are greater than $\mathsf{LCP}[i]$. If this is the case, we pop them from the stack and increment the current position in BPS_{sct}. As we have initialized BPS_{sct} with closing parentheses, this means that we write a closing parentheses at position *ipos*. After the while loop, the stack contains only positions of elements that are smaller than or equal to $\mathsf{LCP}[i]$. We write an opening parenthesis for element $\mathsf{LCP}[i]$ at position *ipos* and push the position of $\mathsf{LCP}[i]$ onto the stack. Note that the produced parentheses sequence is balanced because we only write closing parentheses for elements that were pushed onto the stack; therefore, we have written an opening parenthesis before we write the matching closing one. Also, all elements that remain on the stack after the n-th iteration have corresponding closing parentheses, since we have initialized BPS_{sct} with closing parentheses. The whole construction algorithm has a linear running time because in total we cannot enter the while loop more than n times. The working set of the algorithm only depends on the size of the stack, as all other data structures are

ALGORITHM 1: Construction of BPS_{sct} of the LCP-array.

```
01 $\mathsf{BPS}_{sct}[0..2n+1] \leftarrow [0, 0, \ldots, 0]$
02 $\mathsf{rc}[0..n] \leftarrow [0, 0, \ldots, 0, 1]$
03 $ipos \leftarrow 0$
04 $s \leftarrow stack()$
05 for $i \leftarrow 0$ to $n - 1$ do
06    while $s.size() > 0$ and $\mathsf{LCP}[s.top()] > \mathsf{LCP}[i]$ do
07       $jpos \leftarrow s.pop()$
08       if $\mathsf{LCP}[s.top()] \neq \mathsf{LCP}[jpos]$ then
09          $\mathsf{rc}[ipos - i] \leftarrow 1$
10       $ipos \leftarrow ipos + 1$
11    $\mathsf{BPS}_{sct}[ipos] \leftarrow 1$
12    $s.push(i)$
13    $ipos \leftarrow ipos + 1$
```

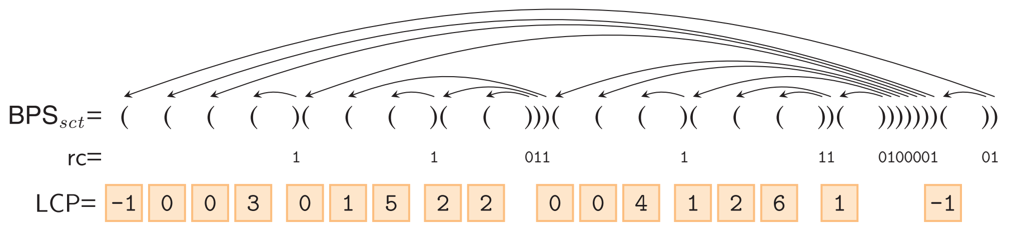

BPS$_{sct}$= (((( )((( )(( )))((( )((( ))( ))))))( ))
rc= 1 1 011 1 11 0100001 01
LCP= -1 0 0 3 0 1 5 2 2 0 0 4 1 2 6 1 -1

Fig. 7. The LCP-array of our running example and the corresponding balanced parentheses sequence BPS_{sct}. Each (conceptual) arrow points from a closing parenthesis to its corresponding opening one. Note that we have placed the i-th opening parentheses directly above its corresponding LCP-entry $\mathsf{LCP}[i]$.

accessed sequentially. Since the stack values are stored in increasing order, we can realize the stack with only $n + o(n)$ bits by using the proposal of Fischer [2010a].

As mentioned in Section 2, an inner node in the suffix tree can be represented as an LCP-interval ℓ-$[i..j]$. For example, the root of the suffix tree of Figure 6 is represented by the LCP-interval 0-[0..15]. In the BPS_{sct}, the closing parentheses of the ℓ-indices of an LCP-interval ℓ-$[i..j]$ form a contiguous sequence. For example, the 0-indices of the LCP-interval 0-[0..15] are 1, 2, 4, 9, and 10 because the LCP-values at these indices is 0. The closing parentheses of these 0-indices are the five parentheses left to the last opening parenthesis in the BPS_{sct} of Figure 7. The bit vector rc marks only the rightmost of these closing parentheses. With the help of the bit vector rc, all navigational operations based on the BPS_{sct} take only constant time, but it is beyond the scope of this article to explain them. The reader is referred to Ohlebusch et al. [2010] for details of the implementation.

One of the virtues of the BPS_{sct} is that we can use it to construct BPS_{dfs} fast and space-efficiently. That is, we can use a depth-first order iterator to walk through the tree and calculate BPS_{dfs} in sequential order. Practical experiments in Gog [2011] have shown that this approach is faster and more space-efficient than the solutions of Hon and Sadakane [2002] and Välimäki et al. [2009].

4. AN LACA FOR CST CONSTRUCTION

Before presenting our new LACA, we analyze the LACAs developed by Kasai et al. [2001] and Kärkkäinen et al. [2009] to identify their weaknesses and improve them.

The following inequality was proven in Kasai et al. [2001]:

$$\mathsf{LCP}[i] \geq \mathsf{LCP}[\mathsf{ISA}[(\mathsf{SA}[i]-1) \bmod n]] - 1. \tag{5}$$

We get an analogous inequality for PLCP by first setting $i = \mathsf{ISA}[j]$ in Inequality (5), so $\mathsf{LCP}[\mathsf{ISA}[j]] \geq \mathsf{LCP}[\mathsf{ISA}[(j-1) \bmod n]] - 1$. This in conjunction with Equation (1) yields

$$\mathsf{PLCP}[j] \geq \mathsf{PLCP}[(j-1) \bmod n] - 1. \tag{6}$$

We shall see that the practical runtimes of the algorithms depend on the number of reducible LCP-values. A value $\mathsf{LCP}[i]$ is called *reducible* if and only if $\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}^{\mathsf{BWT}}[i-1]$; otherwise, it is *irreducible*. Because $\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}^{\mathsf{BWT}}[i-1]$ is equivalent to $\mathsf{LF}[i] = \mathsf{LF}[i-1] + 1$, it follows that a reducible value $\mathsf{LCP}[i]$ can be computed by the equation (see Manzini [2004, Lemma 1])

$$\mathsf{LCP}[i] = \mathsf{LCP}[\mathsf{LF}[i]] - 1 \text{ iff } \mathsf{LCP}[\mathsf{ISA}[(\mathsf{SA}[i]-1) \bmod n]] - 1. \tag{7}$$

A value $\mathsf{PLCP}[j]$ is called *reducible* if $\mathsf{T}[j-1] = \mathsf{T}[\Phi[j]-1]$, where $\Phi[j] = \mathsf{SA}[(\mathsf{ISA}[j]-1) \bmod n]$; otherwise, it is *irreducible*. Setting $i = \mathsf{ISA}[j]$ or equivalently $j = \mathsf{SA}[i]$, we obtain $\mathsf{T}[j-1] = \mathsf{T}[(\mathsf{SA}[i]-1) \bmod n] = \mathsf{T}^{\mathsf{BWT}}[i]$ and $\mathsf{T}[\Phi[j]-1] = \mathsf{T}[\mathsf{SA}[(i-1) \bmod n]-1] = \mathsf{T}^{\mathsf{BWT}}[i-1]$. Thus, $\mathsf{PLCP}[j]$ is reducible if and only if $\mathsf{LCP}[\mathsf{ISA}[j]]$ is reducible.

It follows as a consequence that a reducible value $\mathsf{PLCP}[j]$ can be computed by the equation (see Kärkkäinen et al. [2009, Lemma 4])

$$\mathsf{PLCP}[j] = \mathsf{PLCP}[(j-1) \bmod n] - 1. \tag{8}$$

4.1. Linear Time LACAs Revisited

The key idea in the LACA devised by Kasai et al. [2001] is to use Inequality (5) to calculate all values in linear time. They first calculate ISA from SA (see Line 01 in Algorithm (2)). This is the first part of the algorithm, which is expensive because the write accesses (to ISA) are not in sequential order. Then, in the main loop (Lines 03–10), they sequentially calculate for each suffix i the lexicographically preceding suffix $j = \mathsf{SA}[\mathsf{ISA}[i]-1]$ (i.e. $j = \Phi[i]$) and store it in their original implementation to $\mathsf{LCP}[\mathsf{ISA}[i]]$ (Line 09). That is, there is a random read access to SA and a random write access to LCP in each iteration. Furthermore, the first read access to T at position $j = \Phi[i]$ is only cheap if $\mathsf{T}[j]$ is cached. An important observation, which will help us to explain the practical runtime of the algorithm on different kind of texts, is that if $\mathsf{LCP}[\mathsf{ISA}[i]] = \mathsf{PLCP}[i]$ is reducible, then $j = \Phi[i] = \Phi[i-1]+1$, and hence $\mathsf{T}[j]$ is cached.

ALGORITHM 2: Kasai et al.'s LACA. With streaming, we get a $5n$-byte semiexternal version and without streaming, we get the $9n$-byte solution.

```
01 for $i \leftarrow 0$ to $n-1$ do $\mathsf{ISA}[\mathsf{SA}[i]] \leftarrow i$ // stream SA from disk
   // store ISA to disk
02 $\ell \leftarrow 0$
03 for $i \leftarrow 0$ to $n-1$ do
04    $\mathsf{ISA}_i \leftarrow \mathsf{ISA}[i]$ // stream ISA from disk
05    if $\mathsf{ISA}_i > 0$ do
06       $j \leftarrow \mathsf{SA}[\mathsf{ISA}_i - 1]$
07       while $\mathsf{T}[i+\ell] = \mathsf{T}[j+\ell]$ do
08          $\ell \leftarrow \ell + 1$
09       $\mathsf{SA}[\mathsf{ISA}_i - 1] \leftarrow \ell$ // overwrite $\mathsf{SA}[\mathsf{ISA}_i - 1]$ with $\mathsf{LCP}[\mathsf{ISA}_i]$
10    $\ell \leftarrow max(\ell - 1, 0)$
11 $\mathsf{SA}[n-1] \leftarrow -1$
12 // store overwritten SA as result to disk and shift indices by 1
```

Kärkkäinen et al. [2009] improved the practical runtime of Kasai et al.'s algorithm, essentially by removing the random write access to LCP. Algorithm 3 shows pseudo-code of their Φ-algorithm. It first computes in Line 01 all values of Φ and stores these in an array, which is called *Φ-array*. Note that this is as expensive as calculating ISA in Algorithm 2. In the main loop (Lines 03–07), the read accesses to T are exactly the same as in Algorithm 2. However, this time the resulting LCP-values are stored sequentially (in text order) to PLCP. That is, we have sequential write accesses to PLCP, in contrast to Algorithm 2, where we have random write accesses to LCP. Finally, in Line 09, LCP is constructed sequentially by n random read accesses on PLCP. Since Algorithm 2 has also n additional random read accesses to SA in Line 06, we can conclude that the difference of the practical runtimes of Algorithms 2 and 3 is due to the removal of the random write access to LCP in Algorithm 3.

Kärkkäinen et al. [2009] also showed how to reduce the space consumption and get a lightweight algorithm. The key idea is to first calculate LCP-values for a smaller set of suffixes. To be precise, they compute LCP-values for all suffixes x with $x \equiv 0 \mod q$ for a $q \geq 1$. In a second step, each remaining LCP-value is calculated with at most q

ALGORITHM 3: Kärkkäinen et al.'s Φ-algorithm. With streaming and buffered writing to disk, we get the $5n$-byte semiexternal version; without streaming and using the same array for PLCP and Φ, we get the fast $9n$-byte version.

```
01 for $i \leftarrow 0$ to $n-1$ do $\Phi[\mathsf{SA}[i]] \leftarrow \mathsf{SA}[i-1]$ // stream SA from disk
02 $\ell \leftarrow 0$
03 for $i \leftarrow 0$ to $n-1$ do
04    $\ell \leftarrow max(\ell-1, 0)$
05    while $\mathsf{T}[i+\ell] = \mathsf{T}[\Phi[i]+\ell]$ do
06       $\ell \leftarrow \ell+1$
07    $\mathsf{PLCP}[i] \leftarrow \ell$
08 for $i \leftarrow 0$ to $n-1$ do
09    $\mathsf{LCP}[i] \leftarrow \mathsf{PLCP}[\mathsf{SA}[i]]$ // stream SA from disk, write LCP buffered to disk
```

character comparisons, so the overall time complexity is $\mathcal{O}(nq)$. We present pseudo-code for the approach in Algorithm 4. In Lines 01–03, a sparse Φ-array Φ_q of length n/q is initialized. Then, all LCP-values of suffixes x with $x \equiv 0 \mod q$ are calculated in Lines 04–09. If Inequality 6 is applied q times, then we get

$$\mathsf{PLCP}[i] \geq \mathsf{PLCP}[i-q] - q, \tag{9}$$

so in each of the n/q iterations Algorithm 4 makes at most $q-1$ more character comparisons than in the corresponding iteration in Algorithm 3. In total, the time for initializing PLCP_q is $\mathcal{O}(n)$, and only $n + 4n/q$ bytes of space are required when we stream SA from disk.

ALGORITHM 4: Kärkkäinen et al.'s sparse LACA. It requires $5n + \frac{4}{q}n$ bytes when the same array is used for SA and LCP, that is, SA is overwritten by LCP. The semiexternal version, which streams SA and writes LCP buffered to disk, requires only $n + \frac{4}{q}n$ bytes.

```
01 for $i \leftarrow 0$ to $n-1$ do
02    if $\mathsf{SA}[i] \bmod q = 0$ then // stream SA from disk
03       $\Phi_q[\mathsf{SA}[i]/q] \leftarrow \mathsf{SA}[i-1]$

04 $\ell \leftarrow 0$
05 for $i \leftarrow 0$ to $\lfloor (n-1)/q \rfloor$ do
06    while $\mathsf{T}[iq+\ell] = \mathsf{T}[\Phi_q[i]+\ell]$ do
07       $\ell \leftarrow \ell+1$
08    $\mathsf{PLCP}_q[i] \leftarrow \ell$        // overwrite $\Phi_q$ with $\mathsf{PLCP}_q$
09    $\ell \leftarrow max(\ell-q, 0)$

10 $(x, y) \leftarrow (0, 0)$
11 for $i \leftarrow 0$ to $n-1$ do
12    $(x, y) \leftarrow (y, \mathsf{SA}[i])$ // stream SA from disk
13    $\ell \leftarrow \max(0,\ \mathsf{PLCP}_q[y/q] - (y - \lfloor y/q \rfloor \cdot q))$
14    while $\mathsf{T}[y+\ell] = \mathsf{T}[x+\ell]$ do
15       $\ell \leftarrow \ell+1$
16    $\mathsf{LCP}[i] \leftarrow \ell$
```

Finally, in Lines 10–16, LCP is calculated in sequential SA-order as follows: For each pair (x, y) of suffixes $y = \mathsf{SA}[i]$ and $x = \mathsf{SA}[i-1]$ the rightmost sampled suffix $y' \leq y$ is calculated by $y' = q\lfloor y/q \rfloor$. The LCP-value of y' is stored in $\mathsf{PLCP}_q[y/q]$ and it is at most $y - y'$ characters smaller than the LCP-value for suffix y. So, Algorithm 4 makes in each iteration at most $q-1$ more character comparisons than in the corresponding

i	SA	LCP	LF	T^{BWT}	T
0	15	−1	4	m	$
1	7	0	9	n	dumulmum$
2	11	0	10	u	lmum$
3	3	3	11	u	lmundumulmum$
4	14	0	12	u	m$
5	9	1	13	u	mulmum$
6	1	5	14	u	mulmundumulmum$
7	12	2	2	l	mum$
8	4	2	3	l	mundumulmum$
9	6	0	15	u	ndumulmum$
10	10	0	5	m	ulmum$
11	2	4	6	m	ulmundumulmum$
12	13	1	7	m	um$
13	8	2	1	d	umulmum$
14	0	6	0	$	umulmundumulmum$
15	5	1	8	m	undumulmum$

i/j	1	2	3	4	5	6	7	8	9	10
0	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1
1		0	0	0	0	0	0	0	0	0
2			0	0	0	0	0	0	0	0
3				3	3	3	3	3	3	3
4					0	0	0	0	0	0
5						1	1	1	1	1
6							5	5	5	5
7								2	2	2
8									2	2
9		0	0	0	0	0	0	0	0	0
10			0	0	0	0	0	0	0	0
11				4	4	4	4	4	4	4
12					1	1	1	1	1	1
13						2	2	2	2	2
14							6	6	6	6
15										1

Fig. 8. Left: LCP-array, LF-mapping, and T^{BWT} of our running example. Right: The LCP-array after the j-th iteration of Algorithm 5. A value that is calculated in step j is marked gray in column j. Values which are not yet computed are omitted.

iteration in Algorithm 3, and we get a time complexity of $\mathcal{O}(qn)$ for the last step and the space remains at $n + 4n/q$ bytes when LCP is streamed to disk. It is worth mentioning that all of the additional character comparisons are in sequential order and, therefore, are very fast in practice. Hence, the practical runtime does not depend linearly on q.

4.2. Ideas for an LACA in the CST Construction

In the previous section, we saw that all LACAs have to make expensive random accesses to arrays. Our first observation is that in the context of CST construction, some of the expensive random accesses have already been made before we construct the LCP-array. One example is the calculation of T^{BWT} with Equation (2) from SA and T, where we make n random accesses to T. We can store T^{BWT} to disk and reuse this information in sequential order in our new algorithm. Note that we also get sequential access to LF for free, when T^{BWT} is present. We will also see that we can deduce some LCP-values with LF and range minimum queries (RMQs), which can be answered very fast in our algorithm in practice. We recall that for two indices i and j with $i \leq j$, a range minimum query $\mathsf{RMQ}(i, j)$ on the LCP-array returns the smallest index k such that $\mathsf{LCP}[k] = \min\{\mathsf{LCP}[q] \mid i \leq q \leq j\}$. Finally, we observe that the naïve approach of calculating LCP-values by comparing suffixes $\mathsf{SA}[i]$ and $\mathsf{SA}[i-1]$ for each i is fast when $\mathsf{LCP}[i]$ is small. Therefore, we have two phases in the algorithm. In phase 1, we calculate all small LCP-values $\leq m$, and in phase 2, we insert the remaining big values.

4.3. Phase 1

The pseudocode of phase 1 of our LACA can be found in Algorithm 5. The main loop (Lines 03–15) calculates the LCP-array in sequential SA-order, but the naïve computation of $\mathsf{LCP}[i]$ by comparing suffixes $\mathsf{SA}[i]$ with $\mathsf{SA}[i-1]$ in Line 10 does not have to be done for every suffix. To exemplify this, we take a look at Figure 8, which illustrates the application of the algorithm on our running example. In iteration $j = 4$, we calculate $\mathsf{LCP}[3] = 3$ by comparing suffix $\mathsf{SA}[i-1] = 11$ with suffix $\mathsf{SA}[i] = 3$. But because the characters before suffixes 11 and 3 are equal, that is, $T^{BWT}[i-1] = T^{BWT}[i]$, we can deduce that $\mathsf{LCP}[\mathsf{LF}[3]] = \mathsf{LCP}[3] + 1 = 4$ by Equation (7), and insert that value in the same iteration at index $\mathsf{LF}[3] = 11$ into LCP. Note that this write access is not a random access, since LF points to at most σ regions in LCP. We will now formally show that we can insert a second LCP-entry in each iteration i in which $\mathsf{LF}[i] > i$.

To do this, we first define a function *prev* by

$$prev(i) = \max\{j \mid 0 \le j < i \text{ and } \mathsf{T}^{\mathsf{BWT}}[j] = \mathsf{T}^{\mathsf{BWT}}[i]\},$$

where $prev(i) = -1$ if the maximum is taken over an empty set. Intuitively, if we start at index $i-1$ and scan the $\mathsf{T}^{\mathsf{BWT}}$ upward, then $prev(i)$ is the first smaller index at which the same character $\mathsf{T}^{\mathsf{BWT}}[i]$ occurs.

We claim that the following equation holds

$$\mathsf{LCP}[\mathsf{LF}[i]] = \begin{cases} 0, & \text{if } prev(i) = -1 \\ 1 + \mathsf{LCP}[\mathsf{RMQ}(prev(i)+1, i)], & \text{otherwise.} \end{cases} \tag{10}$$

Proof. We prove the claim by cases. If $prev(i) = -1$, then $\mathsf{T}_{\mathsf{LF}[i]} = \mathsf{T}[\mathsf{LF}[i]..n-1]$ is the lexicographically smallest suffix among all suffixes having $\mathsf{T}^{\mathsf{BWT}}[i]$ as first character. Hence $\mathsf{LCP}[\mathsf{LF}[i]] = 0$. Otherwise, $\mathsf{LF}[prev(i)] = \mathsf{LF}[i] - 1$. In this case, it follows that

$$\begin{aligned} \mathsf{LCP}[\mathsf{LF}[i]] &= lcp(\mathsf{T}_{\mathsf{SA}[\mathsf{LF}[i]-1]}, \mathsf{T}_{\mathsf{SA}[\mathsf{LF}[i]]}) = lcp(\mathsf{T}_{\mathsf{SA}[\mathsf{LF}[prev(i)]]}, \mathsf{T}_{\mathsf{SA}[\mathsf{LF}[i]]}) \\ &= 1 + lcp(\mathsf{T}_{\mathsf{SA}[prev(i)]}, \mathsf{T}_{\mathsf{SA}[i]}) = 1 + \mathsf{LCP}[\mathsf{RMQ}(prev(i)+1, i)]. \quad \square \end{aligned}$$

We will now show that Algorithm 5 correctly computes the LCP-array.

ALGORITHM 5: Phase 1 of the construction of the LCP-array.

```
01 last_occ[0..$\sigma - 1$] $\leftarrow [-1, -1, \dots, -1]$
02 LCP[0] $\leftarrow -1$; LCP[LF[0]] $\leftarrow 0$
03 for $i \leftarrow 1$ to $n - 1$ do
04   if LCP[$i$] $= \perp$ then        // LCP[$i$] is undefined
05     $\ell \leftarrow 0$
06     if LF[$i$] $< i$ then
07       $\ell \leftarrow \max\{\mathsf{LCP}[\mathsf{LF}[i]] - 1, 0\}$
08       if $\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}^{\mathsf{BWT}}[i-1]$ then
09         continue at Line 12
10     while T[SA[$i$] $+ \ell$] $=$ T[SA[$i - 1$] $+ \ell$] do
11       $\ell \leftarrow \ell + 1$
12     LCP[$i$] $\leftarrow \ell$
13   if LF[$i$] $> i$ then
14     LCP[LF[$i$]] $\leftarrow$ LCP[RMQ(last_occ[$\mathsf{T}^{\mathsf{BWT}}[i]$] $+ 1, i$)] $+ 1$
15   last_occ[$\mathsf{T}^{\mathsf{BWT}}[i]$] $\leftarrow i$
```

Proof. Under the assumption that all entries in the LCP-array in the first $i - 1$ iterations of the for-loop have been computed correctly, we consider the i-th iteration and prove the following.

(1) If $\mathsf{LCP}[i] = \perp$, then the entry $\mathsf{LCP}[i]$ will be computed correctly.
(2) If $\mathsf{LF}[i] > i$, then the entry $\mathsf{LCP}[\mathsf{LF}[i]]$ will be computed correctly.

(1) If the if-condition in Line 06 is not true, then $\mathsf{T}_{\mathsf{SA}[i-1]}$ and $\mathsf{T}_{\mathsf{SA}[i]}$ are compared character by character (Lines 10–11) and $\mathsf{LCP}[i]$ is assigned the correct value in Line 12. Otherwise, if the if-condition in Line 06 is true, then ℓ is set to $\max\{\mathsf{LCP}[\mathsf{LF}[i]]-1, 0\}$. We claim that $\ell \le \mathsf{LCP}[i]$. This is certainly true if $\ell = 0$, so suppose that the claim is not true and that $\ell = \mathsf{LCP}[\mathsf{LF}[i]] - 1 > 0$. According to (the proof of) Equation (10), $\mathsf{LCP}[\mathsf{LF}[i]] - 1 = lcp(\mathsf{T}_{\mathsf{SA}[prev(i)]}, \mathsf{T}_{\mathsf{SA}[i]})$. Obviously, $lcp(\mathsf{T}_{\mathsf{SA}[prev(i)]}, \mathsf{T}_{\mathsf{SA}[i]}) \le lcp(\mathsf{T}_{\mathsf{SA}[i-1]}, \mathsf{T}_{\mathsf{SA}[i]}) = \mathsf{LCP}[i]$, so our claim follows.

Now, if $\mathsf{T}^{\mathsf{BWT}}[i] \ne \mathsf{T}^{\mathsf{BWT}}[i-1]$, then $\mathsf{T}_{\mathsf{SA}[i-1]}$ and $\mathsf{T}_{\mathsf{SA}[i]}$ are compared character by character (Lines 10–11), but the first ℓ characters are skipped because they are identical. Again, $\mathsf{LCP}[i]$ is assigned the correct value in Line 12. Finally, if $\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}^{\mathsf{BWT}}[i-1]$, then $prev(i) = i - 1$. This, in conjunction with Equation (10), yields $\mathsf{LCP}[\mathsf{LF}[i]] - 1 = lcp(\mathsf{T}_{\mathsf{SA}[prev(i)]}, \mathsf{T}_{\mathsf{SA}[i]}) = lcp(\mathsf{T}_{\mathsf{SA}[i-1]}, \mathsf{T}_{\mathsf{SA}[i]}) = \mathsf{LCP}[i]$. Thus, $\ell = \mathsf{LCP}[\mathsf{LF}[i]] - 1$ is already the correct value of $\mathsf{LCP}[i]$, so Lines 10 and 11 can be skipped and the assignment in Line 12 is correct.

(2) In the linear scan of the LCP-array, we always have $\mathsf{last\_occ}[\mathsf{T}^{\mathsf{BWT}}[i]] = prev(i)$. Therefore, it is a direct consequence of Equation (10) that the assignment in Line 14 is correct. □

We still have to explain how the index $j = \mathsf{RMQ}(\mathsf{last\_occ}[\mathsf{T}^{\mathsf{BWT}}[i]] + 1, i))$ and the LCP-value $\mathsf{LCP}[j]$ in Line 14 can be computed efficiently. To this end, we use a technique also described in Dhaliwal et al. [2012], which uses a stack K of size $\mathcal{O}(\sigma)$. Each element on the stack is a pair consisting of an index and an LCP-value. We first push $(0, -1)$ onto the initially empty stack K. It is an invariant of the for-loop that the stack elements are strictly increasing in both components (from bottom to top). In the i-th iteration of the for-loop, before Line 13, we update the stack K by removing all elements whose LCP-value is greater than or equal to $\mathsf{LCP}[i]$. Then, we push the pair $(i, \mathsf{LCP}[i])$ onto K. Clearly, this maintains the invariant. Let $x = \mathsf{last\_occ}[\mathsf{T}^{\mathsf{BWT}}[i]] + 1$. The answer to $\mathsf{RMQ}(x, i)$ is the pair (j, ℓ), where j is the minimum of all indices that are greater than or equal to x. This pair can be found by an inspection of the stack. Moreover, the LCP-value $\mathsf{LCP}[\mathsf{RMQ}(x, i)] + 1$ we are looking for is $\ell + 1$. To meet the $\mathcal{O}(\sigma)$ space condition of the stack, after each σ-th update, we check whether the size s of K is greater than σ. If so, we can remove $s - \sigma$ elements from K because there are at most σ possible queries. With this strategy, the stack size never exceeds 2σ and the amortized time for the updates is $\mathcal{O}(1)$. Furthermore, an inspection of the stack takes $\mathcal{O}(\sigma)$ time. In practice, this works particularly well when there is a run in the $\mathsf{T}^{\mathsf{BWT}}$ because then the element we are searching for is on top of the stack.

Note that Algorithm 5 has a quadratic runtime in the worst case; consider, for example, the string $\mathsf{T} = ababab...ab\$$.

At first glance, Algorithm 5 does not have any advantage over Kasai et al.'s algorithm because it holds T, SA, LF, $\mathsf{T}^{\mathsf{BWT}}$, and LCP in main memory. A closer look, however, reveals that the arrays SA, LF, and $\mathsf{T}^{\mathsf{BWT}}$ are accessed sequentially in the for-loop. So they can be streamed from disk. We cannot avoid the random access to T, but we can avoid the random access to LCP, as we show next.

Most problematic are the "jumps" upward (Line 06 when $\mathsf{LF}[i] < i$) and downward (Line 13 when $\mathsf{LF}[i] > i$). The key idea is to buffer LCP-values in queues (FIFO data structures) and to retrieve them when needed.

First, we show, how we can replace the condition $\mathsf{LCP}[i] = \perp$ in Line 04, which checks if $\mathsf{LCP}[i]$ was already calculated. We have to replace it because, in the new scenario, we are not allowed to write LCP-values in a previous iteration $j < i$ to a position $i = \mathsf{LF}[j]$ (see Line 14). We will now show how we can decide if $\mathsf{LCP}[i]$ was already calculated by checking if $\Psi[i] = j < i$. Recall that $\Psi[i]$ corresponds to the position of the k-th occurrence of character $\mathsf{F}[i]$ in $\mathsf{T}^{\mathsf{BWT}}$, where $\mathsf{F}[i]$ is the starting character of the current suffix $\mathsf{SA}[i]$ and k is the index in the $\mathsf{F}[i]$-interval. For example, in Figure 8, for $i = 4$, we get $\mathsf{F}[4] = \mathtt{m}$, $k = 1$ and $\Psi[4] = 0 < 4$, while for the next index $i = 5$, we get $k = 2$ and $\Psi[5] = 10 > 5$. To decide if $\Psi[i] < i$, we first determine $c = \mathsf{F}[i]$ and k and then calculate $r = rank(i, c)$, the number of occurrences of cs in $\mathsf{T}^{\mathsf{BWT}}[0..i-1]$. If $r \geq k$, then the k-th occurrence of c is smaller than i and, therefore, $\Psi[i] < i$ and otherwise $\Psi[i] > i$. Note that we can use σ different counters to keep track of $rank(i, c)$ for all $c \in \Sigma$.

Second, we have to replace the write to location $\mathsf{LF}[i]$ in Line 14. Note that $\mathsf{LF}[i]$ lies in between $\mathsf{C}[\mathsf{T}^{\mathsf{BWT}}[i]]$ and $\mathsf{C}[\mathsf{T}^{\mathsf{BWT}}[i]] + rank(n, \mathsf{T}^{\mathsf{BWT}}[i])$, the interval of all suffixes that start with character $\mathsf{T}^{\mathsf{BWT}}[i]$ and that there are at most σ different such intervals. We exploit this fact in the following way: For each character $c \in \Sigma$, we use a queue Q_c. During the for-loop, we add (enqueue) the values $\mathsf{LCP}[\mathsf{C}[c]], \mathsf{LCP}[\mathsf{C}[c]+1], \ldots, \mathsf{LCP}[\mathsf{C}[c]+rank(n,c)]$ in exactly this order to Q_c. In iteration i, an operation $enqueue(Q_c, x)$ is done for $c = \mathsf{T}^{\mathsf{BWT}}[i]$ and $x = \mathsf{LCP}[\mathsf{RMQ}(\mathsf{last\_occ}[\mathsf{T}^{\mathsf{BWT}}[i]]+1, i)] + 1$ in Line 14 provided that $\mathsf{LF}[i] > i$, and in Line 12 for $c = F[i]$ and $x = \ell$. Also in iteration i, an operation $dequeue(Q_c)$ is done for $c = \mathsf{T}^{\mathsf{BWT}}[i]$ in Line 07 provided that $\mathsf{LF}[i] < i$. This dequeue operation returns the value $\mathsf{LCP}[\mathsf{LF}[i]]$, which is needed in Line 07. Moreover, if $i < \mathsf{C}[\mathsf{F}[i]] + rank(i, \mathsf{T}^{\mathsf{BWT}}[i])$, then we know that $\mathsf{LCP}[i]$ has been computed previously but is still in one of the queues. Thus, an operation $dequeue(Q_c)$ is done for $c = F[i]$ immediately before Line 13, and it returns the value $\mathsf{LCP}[i]$.

The space used by the algorithm now only depends on the size of the queues. We use constant size buffers for the queues and read/write the elements to/from disk if the buffers are full/empty (this even allows to answer an RMQ by binary search in $\mathcal{O}(\log(\sigma))$ time). Therefore, only the text T remains in main memory, and we obtain an n-bytes semiexternal algorithm.

4.4. Phase 2

Our experiments showed that even a carefully engineered version of Algorithm 5 does not always beat the currently fastest LACA [Kärkkäinen et al. 2009]. For this reason, we will now present another algorithm that uses a modification of Algorithm 5 in Phase 1. This modified version computes each LCP-entry whose value is smaller than or equal to m, where m is a user-defined value. (All we know about the other entries is that they are greater than m.) It can be obtained from Algorithm 5 by modifying Lines 8, 10, and 14 as follows.

```
08 if $\mathsf{T}^{\mathsf{BWT}}[i] = \mathsf{T}^{\mathsf{BWT}}[i-1]$ and $\ell < m$ then
10 while $S[\mathsf{SA}[i]+\ell] = S[\mathsf{SA}[i-1]+\ell]$ and $\ell < m+1$ do
14 $\mathsf{LCP}[\mathsf{LF}[i]] \leftarrow \min\{\mathsf{LCP}[\mathsf{RMQ}(\mathsf{last\_occ}[\mathsf{T}^{\mathsf{BWT}}[i]]+1, i)]+1, m+1\}$
```

In practice, $m = 254$ is a good choice because LCP-values greater than m can be marked by the value 255 and each LCP-entry occupies only 1 byte. Because the string T must also be kept in main memory, this results in a total space consumption of $2n$ bytes.

Let $I = [i \mid 0 \leq i < n \text{ and } \mathsf{LCP}[i] \geq m]$ be an array containing the indices at which the values in the LCP-array are $\geq m$ after Phase 1. In Phase 2, we have to calculate the remaining $n_I = |I|$ many LCP-entries, and we use Algorithm 6 for this task. In essence, this algorithm is a combination of two algorithms presented in Kärkkäinen et al. [2009] that compute the PLCP-array: (i) the linear time Φ-algorithm and (ii) the $\mathcal{O}(n \log n)$ time algorithm based on the concept of irreducible LCP-values.

Inequality (6) and Equation (8) have two consequences.

—If we compute an entry $\mathsf{PLCP}[j]$ (where j varies from 1 to $n-1$), then $\mathsf{PLCP}[j-1]$ many character comparisons can be skipped. This is the reason for the linear runtime of Algorithm (6) (see Kasai et al. [2001] and Kärkkäinen et al. [2009]).

—If we know that $\mathsf{PLCP}[j]$ is reducible, then no further character comparison is needed to determine its value. At first glance, this seems to be unimportant because the next character comparison will yield a mismatch anyway. However, it turns out to be important because the character comparison may result in a cache miss.

Algorithm (6) uses a `bit_vector` b, where $b[\mathsf{SA}[i]] = 0$ if $\mathsf{LCP}[i]$ is known already (i.e., $b[j] = 0$ if $\mathsf{PLCP}[j]$ is known) and $b[\mathsf{SA}[i]] = 1$ if $\mathsf{LCP}[i]$ still must be computed (i.e., $b[j] = 1$ if $\mathsf{PLCP}[j]$ is unknown) (see Lines 01–04 of the algorithm). In contrast to the Φ-algorithm [Kärkkäinen et al. 2009], our algorithm does not compute the whole Φ-array (PLCP-array, respectively), only the n_I many entries for which the LCP-value is

ALGORITHM 6: Phase 2 of the construction of the LCP-array.

```
01 $b[0..n-1] \leftarrow [0, 0, \dots, 0]$
02 for $i \leftarrow 0$ to $n-1$ do
03   if $\mathsf{LCP}[i] > m$ then
04     $b[\mathsf{SA}[i]] \leftarrow 1$        // the b-array can be computed in Phase 1 already
05 $\Phi'[0..n_I - 1] \leftarrow [\bot, \bot, \dots, \bot]$
06 initialize a rank data structure for $b$
07 for $i \leftarrow 0$ to $n-1$ do        // stream SA, LCP, and $\mathsf{T}^{\mathsf{BWT}}$ from disk
08     if $\mathsf{LCP}[i] > m$ and $\mathsf{T}^{\mathsf{BWT}}[i] \neq \mathsf{T}^{\mathsf{BWT}}[i-1]$ then     // $\mathsf{PLCP}[\mathsf{SA}[i]]$ is irreducible
09       $\Phi'[rank_b(\mathsf{SA}[i])] \leftarrow \mathsf{SA}[i-1]$

10 $j_I \leftarrow 0$
11 $\ell \leftarrow m+1$
12 $\mathsf{PLCP}'[0..n_I - 1] \leftarrow [0, 0, \dots, 0]$
13 for $j \leftarrow 0$ to $n-1$ do     // left-to-right scan of b and T, but random access to T
14   if $b[j] = 1$ then
15     if $j \neq 0$ and $b[j-1] = 1$ then
16       $\ell \leftarrow \ell - 1$        // at least $\ell - 1$ characters match by Equation (8)
17     else
18       $\ell \leftarrow m+1$        // at least $m+1$ characters match by Phase 1

19     if $\Phi'[j_I] \neq \bot$ then        // $\mathsf{PLCP}'[j_I]$ is irreducible
20       while $\mathsf{T}[j+\ell] = \mathsf{T}[\Phi'[j_I] + \ell]$ do
21         $\ell \leftarrow \ell + 1$
22     $\mathsf{PLCP}'[j_I] \leftarrow \ell$     // if $\mathsf{PLCP}'[j_I]$ is reducible, no character comparison was needed
23     $j_I \leftarrow j_I + 1$

24 for $i \leftarrow 0$ to $n-1$ do        // stream SA and LCP from disk
25   if $\mathsf{LCP}[i] > m$ then
26     $\mathsf{LCP}[i] \leftarrow \mathsf{PLCP}'[rank_b(\mathsf{SA}[i])]$
```

still unknown (Line 05). So if we would delete the values $\Phi[j]$ ($\mathsf{PLCP}[j]$, respectively) from the original Φ-array (PLCP-array, respectively) for which $b[j] = 0$ holds, we would obtain our array $\Phi'[0..n_I - 1]$ ($\mathsf{PLCP}'[0..n_I - 1]$, respectively). We achieve a direct computation of $\Phi'[0..n_I - 1]$ with the help of a rank data structure for the bit array b. The for-loop in Lines 07–09 fills our array $\Phi'[0..n_I - 1]$, but again there is a difference to the original Φ-array: Reducible values are omitted. After initialization of the counter j_I, the number ℓ (of characters that can be skipped), and the PLCP' array, the for-loop in Lines 13–23 fills the array $\mathsf{PLCP}'[0..n_I - 1]$ by scanning the b-array and the string T from left to right. In Line 14, the algorithm tests whether the LCP-value is still unknown (this is the case if $b[j] = 1$). If so, it determines the number of characters that can be skipped in Lines 15–18.[2] If $\mathsf{PLCP}'[j_I]$ is irreducible (equivalently, $\Phi'[j_I] \neq \bot$), then its correct value is computed by character comparisons in Lines 20 and 21. Otherwise, $\mathsf{PLCP}'[j_I]$ is reducible and $\mathsf{PLCP}'[j_I] = \mathsf{PLCP}'[j_I - 1] - 1$ by Equation (8). In both cases, $\mathsf{PLCP}'[j_I]$ is assigned the correct value in Line 22. Finally, the missing entries in the LCP-array (LCP-values in SA-order) are filled with the help of the PLCP'-array in Lines 24–26.

The Phase 1 of our algorithm runs in worst-case $\mathcal{O}(nm)$ time, which is linear in n when m is a fixed value. The same is true of Phase 2, as explained earlier. Thus, the whole algorithm has a linear runtime.

[2]Note that ℓ in Line 18 in the j-th iteration of Algorithm 6 is at least $m+1$, which is greater than ℓ in Line 04 in the j-th iteration of Algorithm 3. Therefore, Algorithm 6 has also a linear time complexity.

4.5. Experimental Comparison of LACAs

We used the first author's succinct data structure library *sdsl*,[3] to implement the new algorithm called go-Φ. The library contains data structures for bit-compressed integer vectors (i.e., every integer takes exactly $\lceil \log n \rceil$ bits, not fixed 32 or 64 bits), *rank* and *select* data structures, and so on. The bit-compressed integer vector (`int_vector`) is especially useful for inputs that are slightly larger than 4 GB, because without it, almost twice the space would be required. However, the use of an `int_vector` can slow down the read and write access to each element by a factor of 2. We compared our implementation with a reimplementation of the KLAAP algorithm [Kasai et al. 2001] and with the original implementation of the Φ-algorithms of [Kärkkäinen et al. 2009].[4] Both implementations use 32-bit integer arrays and are, therefore, limited to 2GB inputs. We refer to Gog and Ohlebusch [2011] for a study in which all three algorithms are implemented with bit-compressed vectors. Here are the details of our set-up.

—Φ-optimal. This is the fastest version of the Φ-algorithm, which uses $9n$ bytes (see Algorithm 3 for pseudocode).
—Φq. This is the worst-case runtime $\mathcal{O}(qn)$ version of the Φ-algorithm, which takes only $5n + 4n/q$ bytes of space because SA is in memory and is overwritten by LCP (see Algorithm 4).
—Φq-semi. Same approach as Φq, but this time, in the last step, SA is streamed from disk and the result is written buffered to disk (see Algorithm 4). It takes only $n+4n/q$ bytes.

We used inputs from the standard *Pizza&Chili* corpus and the highly repetitive version[5] and in addition some larger DNA sequences. Since the performance of lightweight LACAs is only interesting for large test cases, we present only the results for the 200MB of the *Pizza&Chili* text corpus and a DNA sequence of 1,828MB: the file `O.Anatinus` contains the DNA sequences of the species *Ornithorhynchus anatinus*, which was preprocessed (all N symbols were removed) for an alignment tool.

The experiments were conducted on a server equipped with a 6-core AMD Opteron processor 2431, which runs on 2.4GHz on full-speed mode, and 32GB of RAM. The programs were compiled using `gcc` version 4.4.3 with options `-O9 -DNDEBUG` under a 64-bit version of Ubuntu 10.4 (Kernel 2.6.32).

Table I contains the results of the experiments on the standard corpus. It shows the runtime (average runtime of 10 runs) in seconds and peak memory consumption in bytes per input symbol for different construction algorithms and inputs. For example, algorithm Φ-optimal computes the LCP-array of the file dblp.xml in 24 seconds and uses 9 bytes per input symbol. The first two rows of Table I serve as baselines: They contain the time and memory consumption for the SA construction with `libdivsufsort` (first row) and for calculating $\mathsf{T}^{\mathsf{BWT}}$ from SA by Equation (2) (second row). In the calculation of $\mathsf{T}^{\mathsf{BWT}}$, we replaced the expensive modulo operation by a boolean expression and a look-up table of size 2. Furthermore, only SA is in memory and the result is written buffered to disk. Therefore, the space consumption is about 1 byte per input symbol.

Let us first consider the space-consuming algorithms Φ-optimal, $\Phi 1$, and KLAAP. We observe that Φ-optimal outperforms KLAAP. It is about 1.5 times faster in all cases. This can be explained by the observations made in Section 4.1. KLAAP (Algorithm 2) uses n write accesses in ISA-order in Line 09, which are substituted in Φ-optimal (Algorithm 3) by $2n$ sequential write accesses in Lines 06 and 09. So, in total, we save computing time because a sequential write access is an order of magnitude faster than a random write access. Moreover, Φ-optimal has a better locality in its main loop than KLAAP because

[3]The library is available under https://github.com/simongog/sdsl.
[4]The code was provided by Juha Kärkkäinen.
[5]Available at http://pizzachili.dcc.uchile.cl/repcorpus.html.

Table I. Results of the Experiments on the Standard Corpus

	`dblp.xml` `.200MB`		`dna` `.200MB`		`english` `.200MB`		`proteins` `.200MB`		`sources` `.200MB`		`O.Anatinus` `.1828MB`	
SA constr.	54	5.0	71	5.0	65	5.0	72	5.0	45	5.0	976	5.0
T^{BWT} constr.	14	1.0	15	1.0	14	1.0	13	1.0	13	1.0	210	1.0
go-Φ	38	2.0	37	2.0	60	2.0	56	2.0	39	2.0	421	2.0
go-Φ+T^{BWT}	52	2.0	52	2.0	74	2.0	69	2.0	52	2.0	631	2.0
Φ-optimal	24	9.0	36	9.0	30	9.0	30	9.0	22	9.0	676	9.0
Φ1	55	9.0	69	9.0	60	9.0	61	9.0	49	9.0	1018	9.0
Φ4	67	6.0	82	6.0	74	6.0	74	6.0	60	6.0	1130	6.0
Φ64	77	5.1	84	5.1	75	5.1	74	5.1	61	5.1	1074	5.1
Φ1-semi	53	5.0	71	5.0	62	5.0	62	5.0	49	5.0	938	5.0
Φ4-semi	64	2.0	77	2.0	75	2.0	73	2.0	58	2.0	996	2.0
Φ64-semi	72	1.1	76	1.1	73	1.1	70	1.1	59	1.1	907	1.1
KLAAP	34	9.0	56	9.0	50	9.0	45	9.0	33	9.0	899	9.0
reducible	86 %		39 %		66 %		48 %		77 %		47 %	

only two arrays are accessed when we use the same array for Φ and PLCP. In case of KLAAP, three arrays are accessed in the main loop. In terms of algorithm engineering, we can say that the two loops in Lines 03–07 and 08–09 in Algorithm 3 are a loop fission (see Kowarschik and Weiß [2002] for algorithm engineering techniques) of the main loop in Algorithm 2.

The next observation is that Φ-optimal and KLAAP both perform significantly better for the test cases `dblp.xml.200MB` and `sources.200MB` compared to the rest of the `Pizza&Chili` files. This can be explained as follows: Both algorithms compare in each iteration of their main loop (i) the first character of suffix $i+\ell$ with (ii) the first character of suffix $j = \Phi[i] + \ell$ (see Line 07 in Algorithm 2 and Line 05 in Algorithm 3). In Case (i), the text is accessed in almost perfect sequential order. In Case (ii), the access in two consecutive iterations $i-1$ and i is in sequential order provided that $\Phi[i] = \Phi[i-1]-1$ is true, and it has already been observed that this is the case if $\mathsf{PLCP}[i]$ is reducible. We have depicted the ratio of reducible values in the test cases at the bottom of Table I. One can see that test cases with a high ratio of reducible values are processed faster relative to the other algorithms.

The same observation is true for the sparse variants of the Φ-algorithms. Also note that Φ1 takes about twice the time of Φ-optimal because many expensive division and modulo operations are used in the sparse version of the algorithm. Therefore, the calculation of the Φ_1 array and the PLCP_1 array in Φ1—(i.e., the initialization phase in Lines 01–09 of Algorithm 4) takes about as long as Φ-optimal in total.

Now let us switch to the lightweight algorithms which use $2n$ bytes or less for the computation. Since our new solution uses T^{BWT} for the construction process, we introduced two rows. One excludes the time for the T^{BWT} computation (go-Φ) and one includes the time (go-Φ+T^{BWT}). In the context of CST construction, the T^{BWT} is needed anyway, and thus one should use the go-Φ row for a comparison with the other algorithms. If one is solely interested in the construction of the LCP-array, however, the go-Φ+T^{BWT} row should be taken for a fair comparison because the other algorithms do not need T^{BWT}. The explanation of the runtime of the new go-Φ-algorithm for the test cases is rather complex because we use many techniques to speed up the construction process. In the main loop of Phase 1 of Algorithm 5, the first accesses to the text in Line 10 are expensive because they are in SA-order. However, Line 10 is only reached in 32% to 41% of the iterations because we either have already computed the LCP-value at position i (Line 04) or $\mathsf{LF}[i] < i$ and the value is reducible. The runtime of Phase 1 is also correlated with the number of character comparisons: The algorithm

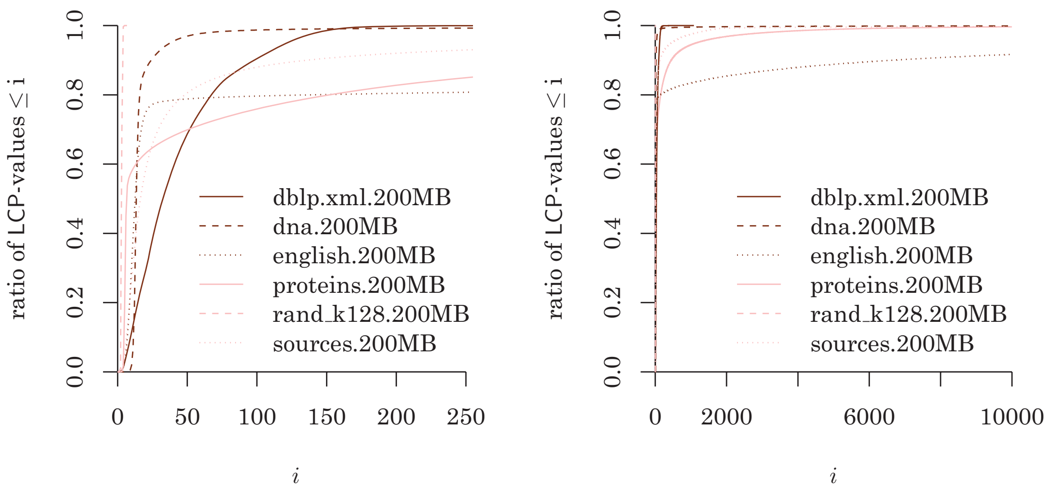


Fig. 9. Density of the LCP-values for the *Pizza&Chili* test cases of size 200MB. Instance rand_k128 is uniformly distributed text over an alphabet size of 128MB.

performs about 20 comparisons per input character for the test cases `proteins.200MB` and `english.200MB`, 6 comparisons for `dna.200MB`, and 13 comparisons for the others. Therefore, it is not surprising that Phase 1 takes about 32 seconds for `proteins.200MB` and `english.200MB` and between 24 and 26 seconds for the remaining test cases.

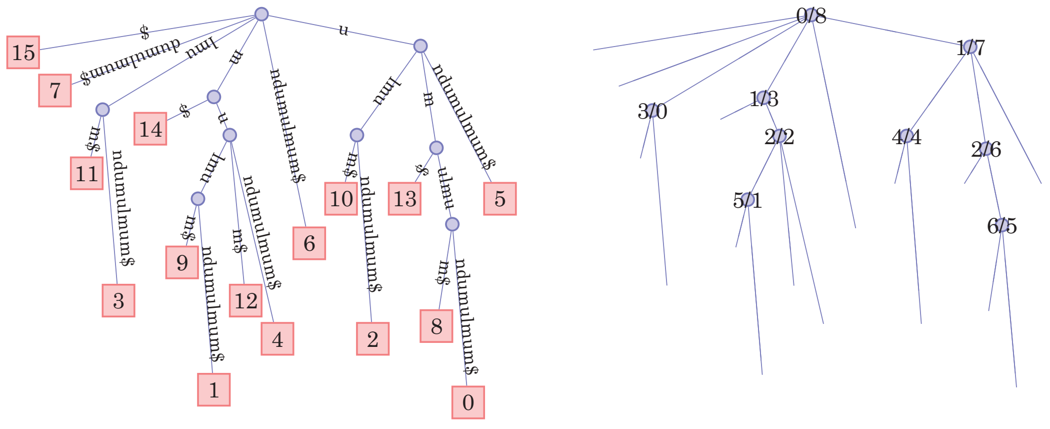
The left plot in Figure 9 depicts the ratio of small values, which are computed during Phase 1 of the algorithm. In Phase 2, the initialization of the arrays b and Φ in Lines 01 and 09 takes between 7 and 10 seconds. It might be possible to further optimize that runtime for test cases where only a small fraction of entries are left. The runtime of the remaining part of the algorithm is more input-dependent and takes less time for inputs with smaller n_I.

We can conclude that our new solution performs significantly better compared to the Φq-semi variants when the percentage of reducible values is not very high, which is often the case for DNA or protein sequences. Note that our algorithm outperforms even the Φ-optimal algorithm when the DNA sequences are large enough, as the DNA sequence of *Ornithorhynchus anatinus* in Table I. In the context of CST construction, where we do not count the time for $\mathsf{T}^{\mathsf{BWT}}$ construction, our new approach takes only 431 seconds, while Φ-optimal uses 4.5 times of our space and 676 seconds.

As expected, the picture changes if we apply the new algorithm to highly repetitive *Pizza&Chili* texts. For those inputs, the ratio of reducible values are at least 95% and the mean LCP-values vary between 3,275 (`para`, the concatenation of 36 almost identical DNA sequences) to 173,308 (`kernel`, the concatenation of 36 different Linux Kernel versions). Therefore, the execution of Phase 1 does not result in many calculated small LCP-values, and Phase 2 has to operate on the large set of unresolved LCP-values. So instead of exploiting the locality of a small set of big values, we get an extra overhead compared to the original Φ-algorithm because we have to perform the extra rank operation per array access. For this reason, the new algorithm takes between 4.2 and 7.1 times as long as the Φ-algorithm and uses space between 4.6 and 6.9 times the text size. Therefore, we advise to use the Φ-algorithm variants for highly repetitive data.

5. THE TREE-COMPRESSED LCP-REPRESENTATION

The suffix number corresponding to *ith_leaf*(i)—the i-th leaf of the suffix tree—can be found at index $i - 1$ in the suffix array because array indexing starts at 0. It is not



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5
0	0	0	3	0	1	5	2	2	0	0	4	1	2	6	1

$i =$	0	1	2	3	4	5	6	7	8
LCP$' =$	3	5	2	1	4	6	2	1	0

Fig. 10. Left: ST, SA, and LCP-array. Right: The tree topology of ST and each node v is labeled with the pair $(depth(v), po\_idx(v))$. On the bottom, we have depicted LCP$'$.

difficult to verify that LCP$[i]$ equals the string-depth of the lowest common ancestor $v = lca(ith\_leaf(i), ith\_leaf(i+1))$ of the i-th and the $(i+1)$-th leaf of the suffix tree (see Sadakane [2007]). Recall that the string-depth $depth(v)$ of an inner node v of the suffix tree is the number of characters on the path from the root to v. Hence,

$$\mathsf{LCP}[i] = depth(lca(ith\_leaf(i), ith\_leaf(i+1))).$$

It follows as a consequence that it suffices to store only the string-depths of all q inner nodes of the CST to represents all LCP-values. We will now show how these can be stored in an array LCP$'[0..q-1]$ so that LCP$[i]$ can be computed in constant time for any index $i \in [0..n-1]$ (see Figure 10).

Definition 5.1. For each inner node v of a suffix tree ST, its *postorder index* $po\_idx(v)$ is the number of inner nodes that are printed before v is printed in a postorder tree traversal of ST.

Figure 10 depicts all postorder indices in our example.

Definition 5.2. For each index $i \in [0..n-1]$ of the LCP-array, the *tree-lcp index* $tlcp\_idx(i)$ is the postorder index of the lowest common ancestor of the i-th and the $(i+1)$-th leaf of the suffix tree.

The pseudocode for the calculation of $tlcp\_idx(i)$ in BPS$_{dfs}$ is given in Algorithm 7, and Algorithm 8 depicts the pseudocode for BPS$_{sct}$. We can observe that in both cases we only need a constant number of constant time operations. Therefore, we get the following theorem.

THEOREM 5.3. *The postorder index and the tree-lcp index can be computed in constant time in the BPS$_{dfs}$ and BPS$_{sct}$.*

We will now prove the correctness of Algorithm 7.

PROOF. The i-th leaf of the suffix tree corresponds to the i-th occurrences of the pattern “10” in the BPS$_{dfs}$, where 1 stands for an opening parenthesis and 0 for a closing parenthesis. Thus, the *select* operations in Line 02 of Algorithm 7 return the

ALGORITHM 7: Calculating $tlcp\_idx(i)$ in BPS_{dfs}.

```
01 if $i > 0$
02     $jpos \leftarrow double\_enclose_{\mathsf{BPS}_{dfs}}(select_{\mathsf{BPS}_{dfs}}(i, \text{‘10’}), select_{\mathsf{BPS}_{dfs}}(i+1, \text{‘10’}))$
03 else
04     $jpos \leftarrow 0$
05 $cjpos \leftarrow find\_close_{\mathsf{BPS}_{dfs}}(jpos)$
06 return $rank_{\mathsf{BPS}_{dfs}}(cjpos, \text{‘0’}) - rank_{\mathsf{BPS}_{dfs}}(cjpos, \text{‘10’})$
```

positions of leaf i and leaf $i+1$. The position of the opening parenthesis of the lowest common ancestor v of leaves i and $i+1$ is then calculated by a *double_enclose* operation. Finally, the postorder index of v is determined by jumping to the position *cjpos* of the closing parenthesis of v and counting the number of inner nodes that were visited before the last visit of v. Observe that the number of all nodes that were visited before the last visit of v equals $rank_{\mathsf{BPS}_{dfs}}(cjpos, \text{“0”})$, the number of closing parentheses up to position *cjpos*. Thus, if we subtract $rank_{\mathsf{BPS}_{dfs}}(cjpos, \text{“10”})$, the number of all leaves encountered up to position *cjpos*, then we obtain the postorder index of v. $\square$

Leaves of the suffix tree are not explicitly represented in the BPS_{sct}, but each inner node is represented by a contiguous sequence of closing parentheses, in which all parenthesis are marked with “0” in the additional bit vector rc except for the last one (which is marked with “1”). Moreover, the rank of such a contiguous sequence of closing parentheses (from left to right) coincides with the postorder index of the inner node that is represented by this sequence. Another important observation is that the closing parenthesis corresponding to index i in the LCP-array belongs to the lowest common ancestor v of leaf i and leaf $i+1$ in the suffix tree. Note that Algorithm 8 determines the position *cipos* of this closing parenthesis in Lines 01 and 02. It follows from the earlier discussion that the postorder index of v equals the number of closing parenthesis before position *cipos*, which are marked with “1” in the bit vector rc. We find this number in Algorithm 8 as follows: In Line 03, we first determine the number x of closing parentheses, which are left to *cipos*, that is, $x = rank_{\mathsf{BPS}_{sct}}(cipos, 0)$. However, we do not have to make a *rank* query because we have calculated enough information in the previous two lines:

$$x = rank_{\mathsf{BPS}_{sct}}(cipos, 0) = \underbrace{rank_{\mathsf{BPS}_{sct}}(ipos, 0)}_{=ipos-i} + \underbrace{rank_{\mathsf{BPS}_{sct}}(cipos, 0) - rank_{\mathsf{BPS}_{sct}}(ipos, 0)}_{=\frac{cipos-ipos-1}{2}}.$$

The *select* query in Line 01 tells us that there are i ones left of position *ipos* and, therefore, $ipos - i$ zeros. In Line 02, we have calculated the matching closing parenthesis for *ipos*, so we know that $\mathsf{BPS}_{sct}[ipos..cipos]$ is a balanced parentheses sequence. Therefore, the number of zeros in it is exactly $\frac{cipos-ipos-1}{2}$. Finally, $rank_{\mathsf{rc}}(x)$ returns the number of closing parenthesis before position *cipos* that are marked with “1” in the bit vector rc. As already mentioned, this is the postorder index of v.

The tree-compressed data structure `lcp_support_tree` can now be realized as follows: First, we store the q string-depths in postorder in an array LCP'. Note that we can easily

ALGORITHM 8: Calculating $tlcp\_idx(i)$ in BPS_{sct}.

```
01 $ipos \leftarrow select_{\mathsf{BPS}_{sct}}(i+1)$
02 $cipos \leftarrow find\_close_{\mathsf{BPS}_{sct}}(ipos)$
03 return $rank_{\mathsf{rc}}(\frac{ipos+cipos-1}{2} - i)$
```

integrate the construction of LCP' into Algorithm 1 and, therefore, need only $n + o(n)$ additional bits for the succinct stack. Second, the $[i]$-operator is computed as follows: We calculate $j = \mathit{tlcp\_idx}(i)$ and return $\mathsf{LCP}'[j]$. Note that we can use every data structure of Type (1) introduced in Section 3.2.2 for the implementation of LCP'.

5.1. An Advanced Tree-Compressed LCP-Representation

The space consumption of `lcp_support_tree` depends on the number q of inner nodes of the CST and on the representation that is used for LCP'. All Type (1) representations of Section 3.2.2 have one common drawback. They cannot compress large values, as they do not use additional information from the CST. In our advanced solution, we will take a Huffman-shaped wavelet tree (`wt_huff`) to store a modified version of LCP' called $\mathsf{LCP\_SMALL_{rc}}$.

$$\mathsf{LCP\_SMALL_{rc}}[j] = \begin{cases} \mathsf{LCP}'[j] & \text{if } \mathsf{LCP}'[j] < 254 \\ 254 & \text{if } \mathsf{LCP}'[j] \geq 254 \text{ and } \mathsf{LCP}'[j] \text{ is not sampled} \\ 255 & \text{if } \mathsf{LCP}'[j] \geq 254 \text{ and } \mathsf{LCP}'[j] \text{ is sampled} \end{cases} \tag{11}$$

In the first case of Equation (11), answering $\mathsf{LCP}[i]$ is done exactly as in the implementation of `lcp_support_tree`: We calculate $j = \mathit{tlcp\_idx}(i)$ and return $\mathsf{LCP}'[j] = \mathsf{LCP\_SMALL_{rc}}[j]$ (see also Lines 01–04 in Algorithm 9). Now we store the values of the $p < q$ sampled large values in a bit-compressed `int_vector` of size p called $\mathsf{LCP\_BIG_{rc}}$. When $\mathsf{LCP\_SMALL_{rc}}[j] = 255$ indicates that there is a sampled large value, we can determine the corresponding index in $\mathsf{LCP\_BIG_{rc}}$ by a *rank* query, which is naturally supported by the Huffman-shaped wavelet tree $\mathsf{LCP\_SMALL_{rc}}$. Note that if there are many large values in LCP, then the *rank* query in a Huffman-shaped wavelet tree is faster, as we use fewer bits for the code of 255 in the wavelet tree. Therefore, the runtime of the operations adapts to the input data.

ALGORITHM 9: Calculation of $\mathsf{LCP}[i]$ in the class `lcp_support_tree2`.

```
01 $j \leftarrow cst.tlcp\_idx(i)$
02 $val \leftarrow \mathsf{LCP\_SMALL_{rc}}[j]$
03 if $val < 254$
04     return $val$
05 $offset \leftarrow 0$
06 while $val \neq 255$ do
07     $i \leftarrow cst.csa.psi(i)$        // $i = \mathsf{LF}[i]$
08     $j \leftarrow cst.tlcp\_idx(i)$
09     $val \leftarrow \mathsf{LCP\_SMALL_{rc}}[j]$
10     $offset \leftarrow offset + 1$
11 return $\mathsf{LCP\_BIG_{rc}}[rank_{\mathsf{LCP\_SMALL_{rc}}}(j, 255)] - offset$
```

Before we can explain the rest of the algorithm, we have to state how the sampling is done. First, we choose a sampling parameter $s_{\mathsf{LCP}} \geq 1$. All large values $x = \mathsf{LCP}'[j]$ with $x \equiv 0 \mod s_{\mathsf{LCP}}$ are sampled. Next, we check for each large value $x = \mathsf{LCP}'[j]$ with $x \not\equiv 0 \mod s_{\mathsf{LCP}}$ if we can "easily reconstruct" it from other values in LCP' by using the LF function. That is, for every $\mathsf{LCP}'[j]$, we take all indices i_k $(0 \leq k < t)$ with $j = \mathit{tlcp\_idx}(i_k)$ and check whether $\mathsf{T^{BWT}}[i_k] = \mathsf{T^{BWT}}[i_k - 1]$ for each $k \in [0..t-1]$. In other words, we check whether each $\mathsf{LCP}[i_k]$ is reducible. If this is true, we can calculate $\mathsf{LCP}[i_k]$ as follows: $\mathsf{LCP}[i_k] = \mathsf{LCP}[\mathsf{LF}[i_k]] - 1$ (see also Equation (7)). Therefore, storing a large value for index j in LCP' is not necessary, and we mark that with $\mathsf{LCP\_SMALL_{rc}}[j] = 254$. Otherwise, if $\mathsf{LCP}[i_k]$ is not reducible, all other large values $\mathsf{LCP}'[j]$ are added to the sampled values by setting $\mathsf{LCP\_SMALL_{rc}}[j] = 255$. It is worth mentioning that the size of the set of large values that were not "easy to reconstruct" from LF was very small in all our experiments.

Note that we can again use Algorithm 1 to calculate all the information for $\mathsf{LCP\_SMALL_{rc}}$ in linear time and only $n + o(n)$ extra space for the succinct stack, since $\mathsf{T^{BWT}}$, which is needed to decide whether a value is sampled, can be streamed from disk.

Algorithm 9 depicts the whole pseudocode for the $[i]$-operator of `lcp_support_tree2`. If $val \neq 255$ in Line 06, then $\mathsf{LCP}[i]$ is reducible and $\mathsf{LCP}[\mathsf{LF}[i]] = \mathsf{LCP}[i] + 1$ by Equation (7). Consequently, val is either 254 or 255 in the next iteration of the while-loop. So the while-loop is executed until a sampled large value is found. The variable *offset*, which is defined in Line 05, is used to keep track of how many times we have skipped a large LCP-value and used LF to go to the next larger value. Upon termination of the while-loop, *offset* is subtracted from the current LCP-value, and the correct value is returned in Line 11.

Alternatively, one could first sample all values that are not "easy to reconstruct" and then add more samples in between these.

5.2. Experimental Comparison of LCP-Implementations

In this section, we experimentally compare seven proposals of compressed LCP-representations, which are all available in the *sdsl*. To get a fair comparison, we always used the same basic data structures like *rank* or *select* data structures, CSAs, and navigation structures of the CST. Here are details of the implementations.

—`lcp_bitcompressed`. This is the representation which uses $\log n$ bits for each entry (the implementation uses a bit-compressed `int_vector`).
—`lcp_wt`. We use a Huffman-shaped wavelet tree to store the LCP-values.
—`lcp_dac`. We use the direct accessible code (presented in Brisaboa et al. [2009]) with different values for the time–space trade-off parameter b ($b \in \{4, 8, 16\}$). The compression works as follows: The b least significant bits of each LCP-value are stored in an `int_vector<`b`>` d_0 of size $n_0 = n$. We store an additional `bit_vector` $overflow_0$ of size n, which indicates whether each $\mathsf{LCP}[i]$ entry is greater than $2^b - 1$ or not. For each such $\mathsf{LCP}[i]$, we store the next b least significant bits in another `int_vector<`b`>` d_1 of length $n_1 = rank_{overflow_0}(n_0)$ at position $rank_{overflow_0}(i)$ and store an additional bit vector $overflow_1$ of size n_1, which indicates whether $\mathsf{LCP}[i] > 2^{2b} - 1$, and so on, until $n_k = 0$. So accessing an LCP-value x takes about $\log(x + 1)/b$ rank operations.
—`lcp_byte`. We use 1 byte for values < 255 and store values $\mathsf{LCP}[i] \geq 255$ as a pair $(i, \mathsf{LCP}[i])$ in a sorted array, that is, the value at position i can be recovered in $\log n$ time [Kurtz 1999].
—`lcp_support_sada`. This is the solution of Sadakane [2002], which uses $2n + o(n)$ bits on top of the suffix array. We use a CSA based on a Huffman-shaped wavelet tree and set the sample values for the SA and ISA values to 32, that is, the class `csa_wt<wt_huff<>,32,32>` in the *sdsl*.
—`lcp_support_tree`, which was presented in Section 5. We use `lcp_wt` as container for LCP' and the compressed suffix tree presented in Ohlebusch et al. [2010] (i.e., the class `cst_sct3` in the *sdsl*) and the same CSA that is used for `lcp_support_sada`.
—`lcp_support_tree2`, which was presented in Section 5.1. We set $s_{\mathsf{LCP}} \in \{2, 4, 8, 16, 32\}$ to get a time–space trade-off. For $\mathsf{LCP\_SMALL_{rc}}$, we use a Huffman-shaped wavelet tree (i.e., `wt_huff` in the *sdsl*), and for $\mathsf{LCP\_BIG_{rc}}$, we use an `int_vector`.

We measured only the space of the real LCP information, that is, we did not take the space for CSA and CST into account. The experiments were conducted on a Sun Fire X4100 M2 server equipped with a Dual-Core AMD Opteron 1222 processor, which runs with 3.0GHz on full-speed mode and 1.0GHz in idle mode. The cache size of the processor is 1MB, and the server contains 4GB of main memory. Only one core of the processor cores was used for experimental computations. We used the 64-bit version of OpenSuse 10.3 (Linux Kernel 2.6.22) as an operating system and the `gcc` compiler version 4.2.1. All programs were compiled with optimization (option `-O9`) and in case of performance tests without debugging (option `-DNDEBUG`).

Figure 11 shows the results of the following experiment. We have performed 10^7 queries to 10^6 random positions of the LCP-array of each *Pizza&Chili* test case. The diagrams show the average time for one query and the space occupied by the structures. The bit-compressed `int_vector` of the class `lcp_bitcompressed` takes the most space ($\lceil \log(200 \cdot 2^{20}) \rceil = 28$ bit) and takes about 100 nanoseconds in all cases. This is about 1.3 times the runtime that we have determined for a random access on a bit-compressed `int_vector`. For test cases with many small values < 255 like `dna.200MB` or `rank_k128.200MB`, the alternatives `lcp_byte` and `lcp_dac` performed even better (with only a fraction of the space) because small values are stored byte aligned and are accessed faster. However, for the test cases `english.200MB` and `sources.200MB`, where we have many large values $\geq$255, both solutions slow down due to the special handling of the large values. The most space-saving solution for structured texts is clearly `lcp_support_sada`. It occupies only a little bit more than $2n$ bits but has a runtime between 10,000 and 40,000 nanoseconds. This is about 100 to 400 times slower than `lcp_bitcompressed`. The access in `lcp_support_sada` basically depends on two operations: first on the access to the underlying CSA and second on a select query. A single select query takes only about 400 nanoseconds, so most of the time is spent accessing the CSA. Experiments showed that by using a larger and faster CSA, the access time of `lcp_support_sada` can be accelerated by a factor of 10, but we have to pay for this with additional space for the CSA.

Our new solutions `lcp_support_tree` and `lcp_support_tree2` take about 0.2 to 5 bits for all test cases. `lcp_support_tree` takes about 1,000 to 2,000 nanoseconds per operation. This was expected because the sum of the runtime of the operations in $tlcp\_idx(i)$ of Algorithm 8 ($1\times$ *select*, $1 \times$ *find_close*, and $(1 + 8) \times$ *rank*) is about 1,000 to 2,000 nanoseconds (see Gog [2011] for the exact runtime of each basic operation, so we have a solution that is about 10 times slower than `lcp_dac`, but reduces the space consumption significantly in all cases.

`lcp_support_tree2` further reduces space by only storing about every s_{LCP}-th large LCP-value for $s_{\mathsf{LCP}} \in \{2, 4, 8, 16, 32\}$. Clearly, for test cases with many small values (`dna.200MB`, `dblp.200MB` and `rank_k128.200MB`) the solution shows no time–space trade-off, as there are no large values that can be omitted. Otherwise, as in the case of `english.200MB`, our solution closes the gap between `lcp_support_sada` and `lcp_dac`. It takes about 3.5 bits but is about 10 times faster than `lcp_support_sada`.

In the special case of random text over an alphabet of size 128, our solution takes very little space. This is due to the fact that the CST has only few inner nodes (about $0.2n$) and we store only LCP-values at inner nodes.

In some applications the LCP-array is accessed in sequential order, for example, in the CST of Fischer et al. [2009]. Therefore, we performed a second experiment, which takes a random position and then queries the next 32 positions.[6] In total we performed $32 \cdot 10^7$ queries and took the average runtime (see Figure 12). We can observe that the type (1) LCP-representations are 10 times faster than in the previous experiment. `lcp_support_sada` *does not profit* from sequential access because it has not stored the information in sequential order. Our solutions `lcp_support_tree` and `lcp_support_tree2` *do profit* from the sequential access because we store the values in postorder of the CST and, therefore, get a good locality in some cases. Our solutions are at least two times

[6]This is exactly the same experiment that was performed by Cánovas and Navarro [2010].

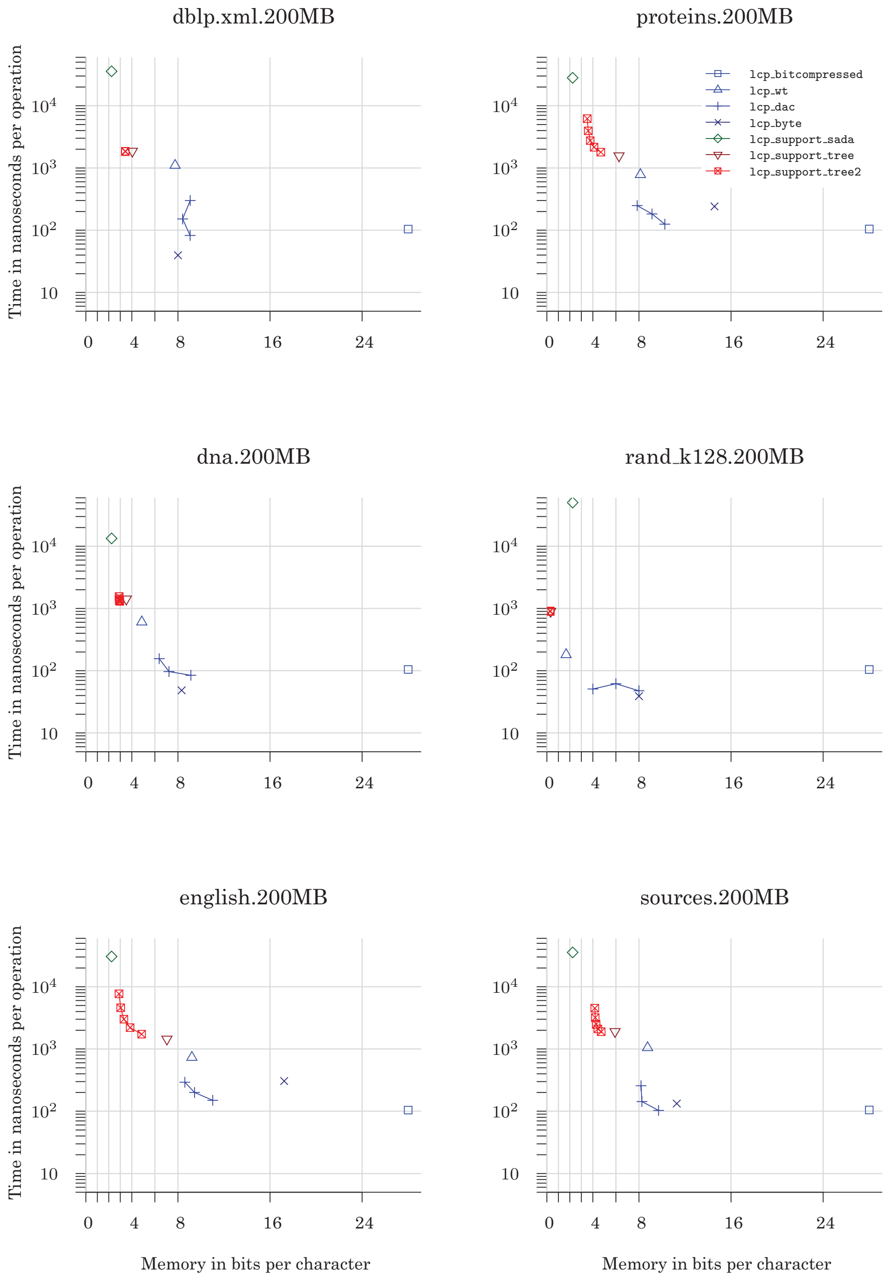


Fig. 11. Average runtime of one random LCP access for different LCP-representations included in the *sdsl*.

faster than in the previous experiment. Therefore, for practical use we can summarize the results as follows: `lcp_dac` is always a good replacement for `lcp_bitcompressed`. Its runtime is slightly better for arrays with many small values, and for the parameter $b = 8$, its runtime is about two times the runtime of `lcp_bitcompressed`, but `lcp_dac` occupies only around 8 to 10 bits for real-world texts. The access is about 10 times faster if the queries are in sequential order. `lcp_support_tree` and `lcp_support_tree2` are

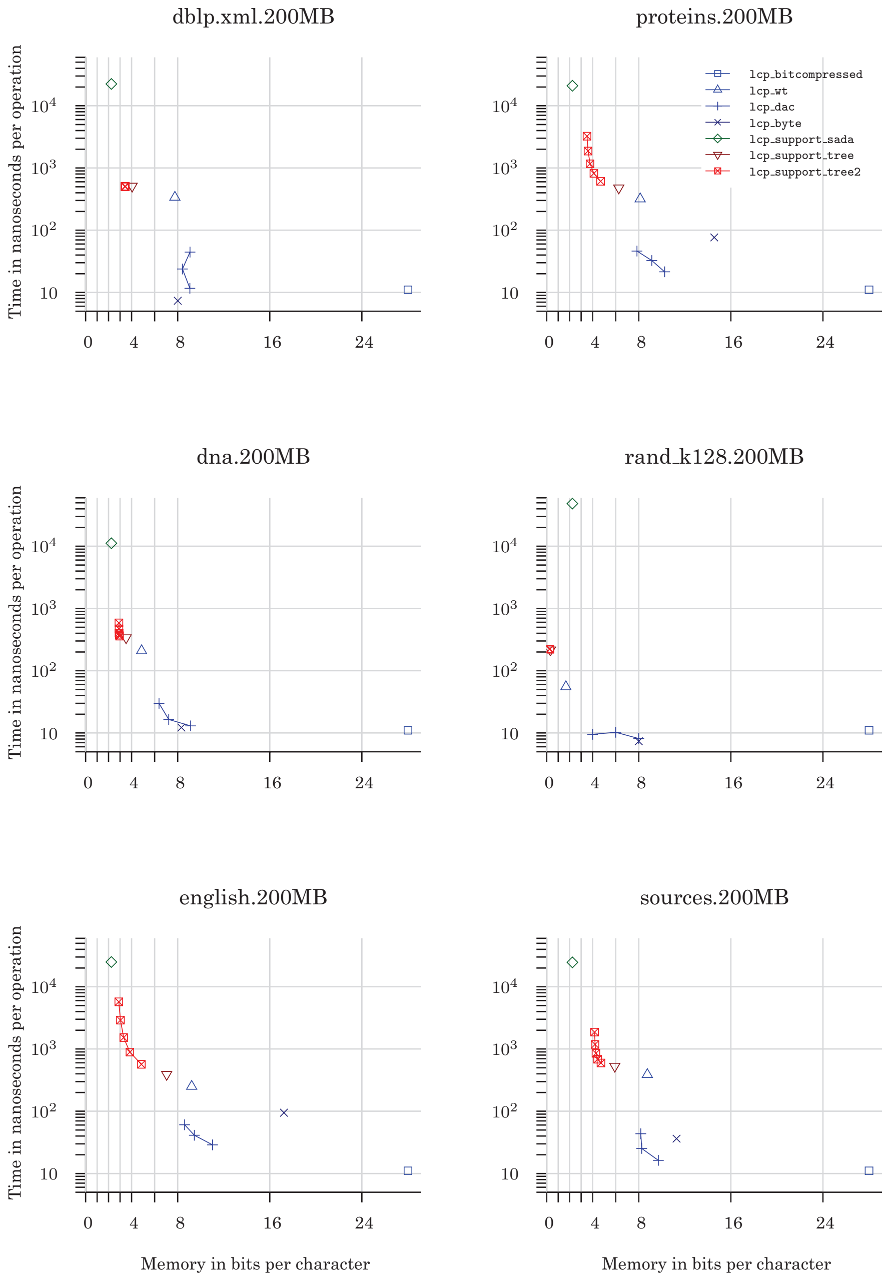


Fig. 12. Average runtime time of one random sequential LCP access for different LCP-representations included in the *sdsl*.

about 10 times slower than `lcp_dac` but occupy only around 0.2 to 5 bits. The runtime profits from sequential access by a factor of 2. `lcp_support_sada` is—for reasonably small CSAs—about 10 times slower than `lcp_support_tree` and uses in all cases a little bit more than $2n$ bits. The runtime does not profit from a sequential access order.

6. CONCLUSION

In the first part of this article, we have given an overview of recent linear time algorithms for the computation of LCP arrays (LACAs). We further proposed a practical algorithm that was designed to work in the context of compressed suffix tree construction. In this context, an LACA can also access the Burrows-Wheeler transform of the input text. Our experimental evaluation showed that the new algorithm outperforms alternatives in this scenario for non–highly repetitive texts.

In the second part, we proposed two space-efficient representations of the LCP-array in CSTs, which both make use of the compressed representation of the topology of the CST. The second variant is further compressed by also exploiting functionality of the underlying CSA of the CST. Our empirical evaluation showed that we get new relevant time–space trade-offs: Space is close to the most space-efficient existing solution, but the runtime does not depend on slow-access operations on the CSA and is, therefore, significantly faster.

All implementations are part of the open-source C++ template library called *sdsl*. The presented LCP-representations can be easily plugged into the different CSTs for further exploration of relevant time–space trade-offs. In future work, we plan to investigate the behavior of those CSTs on large datasets and on integer alphabets.

ACKNOWLEDGMENT

The authors are grateful to the anonymous reviewers. Their comments helped to improve the presentation of the results. The authors thank Timo Beller for running the experiments reported in Table I.

REFERENCES

Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. 2004. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algor. 2*, 1, 53–83.

Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. 2011. Computing the longest common prefix array based on the Burrows-Wheeler Transform. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'09)*. Lecture Notes in Computer Science, vol. 7024, Springer, 197–208.

Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. Directly addressable variable-length codes, string processing and information retrieval. In *Proceedings of the 16th International Symposium (SPIRE '09)*. Lecture Notes in Computer Science, vol. 5721, Springer, 122–130.

Rodrigo Cánovas and Gonzalo Navarro. 2010. Practical compressed suffix trees. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA '10)*. Lecture Notes in Computer Science, vol. 6049, Springer, 94–105.

David R. Clark. 1996. Compact pat trees. Ph.D. dissertation. University of Waterloo.

Jasbir Dhaliwal, Simon J. Puglisi, and Andrew Turpin. 2012. Practical efficient string mining. *IEEE Trans. Knowl. Data Eng. 24*, 4, 735–744.

Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data Structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science.* 390–398.

Johannes Fischer. 2010a. Optimal Succinctness for range minimum queries. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN '00)*. Lecture Notes in Computer Science, vol. 6034, Springer, 158–169.

Johannes Fischer. 2010b. Wee LCP. *Inform. Process. Lett. 110*, 8–9, 317–320.

Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. 2009. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci. 410*, 51, 5354–5364.

Simon Gog. 2011. Compressed suffix trees: Design, construction, and applications. Ph.D. dissertation. Ulm University.

Simon Gog and Johannes Fischer. 2010. Advantages of shared data structures for sequences of balanced parentheses. In *Proceedings of the Data Compression Conference (DCC '10)*. IEEE, 406–415.

Simon Gog and Enno Ohlebusch. 2011. Fast and lightweight LCP-array construction algorithms. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX '11)*. SIAM, 25–34.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '03).* 841–850.

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

Wing-Kai Hon and Kunihiko Sadakane. 2002. Space-economical algorithms for finding maximal unique matches. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM '02)*. Lecture Notes in Computer Science, vol. 2373, Springer, 144–152.

David A. Huffman. 1952. A Method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng. 40*, 9, 1098–1101.

Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science.* IEEE, 549–554.

Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. 2009. Permuted Longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM '09)*. Lecture Notes in Computer Science, vol. 5577, Springer, 181–192.

Juha Kärkkäinen and Peter Sanders. 2003. Simple Linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*. Lecture Notes in Computer Science, vol. 2719, Springer, 943–955.

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM '01)*. Lecture Notes in Computer Science, vol. 2089, Springer, 181–192.

Pang Ko and Srinivas Aluru. 2003. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM '03)*. Lecture Notes in Computer Science, vol. 2676, Springer, 200–210.

Markus Kowarschik and Christian Weiss. 2002. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, Lecture Notes in Computer Science, vol. 2625, Springer, 213–232.

Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.* 29, 13 (1999), 1149–1171.

Veli Mäkinen. 2003. Compact suffix array—a space-efficient full-text index. *Foundamenta Informaticae 56*, 1–2, 191–210.

Giovanni Manzini. 2004. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*. Lecture Notes in Computer Science, vol. 3111, Springer, 372–383.

Giovanni Manzini and Paolo Ferragina. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40*, 1, 33–50.

Gonzalo Navarro and Veli Mäkinen. 2007. Compressed full-text indexes. *ACM Comput. Surv. 39*, 1.

Ge Nong, Sen Zhang, and Wai Hong Chan. 2009. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the Data Compression Conference (DCC '09)*. IEEE, 193–202.

Enno Ohlebusch, Johannes Fischer, and Simon Gog. 2010. CST++. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE '10)*. Lecture Notes in Computer Science, vol. 6393, Springer, 322–333.

Enno Ohlebusch and Simon Gog. 2009. A Compressed enhanced suffix array supporting fast string matching. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE '09)*. Lecture Notes in Computer Science, vol. 5721, Springer, 51–62.

Daisuke Okanohara and Kunihiko Sadakane. 2009. A linear-time Burrows-Wheeler transform using induced sorting. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE '09)*. Lecture Notes in Computer Science, vol. 5721, Springer, 90–101.

Simon J. Puglisi, William F. Smyth, and Andrew Turpin. 2007. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv. 39*, 2.

Simon J. Puglisi and Andrew Turpin. 2008. Space-time tradeoffs for longest-common-prefix array computation. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC '09)*. Lecture Notes in Computer Science, vol. 5369, Springer, 124–135.

Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. 2008. Fully-compressed suffix trees. In *Proceedings of the 12th Latin American Symposium on Theoretical Informatics (LATIN '08)*. Lecture Notes in Computer Science, vol. 4957, Springer, 362–373.

Kunihiko Sadakane. 2002. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proceedings of the 2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02). SIAM,* 225–232.

KUNIHIKO SADAKANE. 2007. Compressed suffix trees with full functionality. *Theory Comput. Syst. 41,* 4, 589–607.

KUNIHIKO SADAKANE AND GONZALO NAVARRO. 2010. Fully-functional succinct trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. SIAM, 134–149.

JOUNI SIRÉN. 2010. Sampled longest common prefix array. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM '10)*. Lecture Notes in Computer Science, vol. 6129, Springer, 227–237.

NIKO VÄLIMÄKI, VELI MÄKINEN, WOLFGANG GERLACH, AND KASHYAP DIXIT. 2009. Engineering a compressed suffix tree implementation. *ACM J. Exp. Algor. 14,* 2.

Received March 2012; revised February 2013; accepted March 2013