LCP Array Construction Using O(sort(n)) (or Less) I/Os

Juha Kärkkäinen(✉) and Dominik Kempa

Department of Computer Science and Helsinki Institute for Information Technology HIIT, University of Helsinki, Helsinki, Finland
{juha.karkkainen,dominik.kempa}@cs.helsinki.fi

Abstract. The suffix array, one of the most important data structures in modern string processing, needs to be augmented with the longest-common-prefix (LCP) array in many applications. Their construction is often a major bottleneck especially when the data is too big for internal memory. While there are external memory algorithms that construct the suffix array and the LCP array simultaneously in the optimal I/O complexity of $\mathcal{O}(\text{sort}(n))$, for several reasons it would be desirable to construct the suffix array first and then the LCP array from the suffix array in a separate stage. In this paper we describe the first algorithm that achieves $\mathcal{O}(\text{sort}(n))$ I/O complexity for the LCP array construction stage and is not an extension of a suffix sorting algorithm. As a variant, we obtain a Monte Carlo algorithm that, given a sparse suffix array containing $m < n$ suffixes in sorted order, computes the corresponding LCP array in $\mathcal{O}(\text{scan}(n) + \text{sort}(m)\log(n/m))$ I/Os if the suffix positions are evenly spaced, and in $\mathcal{O}(\text{scan}(n) + \text{sort}(m)\log(n))$ I/Os in general.

1 Introduction

The suffix array [13,28], a lexicographically sorted list of the suffixes of a text, is one of the most important data structures in modern string processing. It is frequently augmented with the longest-common-prefix (LCP) array, which stores the lengths of the longest common prefixes between lexicographically adjacent suffixes. Together they are the basis of powerful text indexes such as enhanced suffix arrays [1] and many compressed full-text indexes [30]. Modern textbooks spend dozens of pages in describing their applications, see e.g. [27,33].

The construction of the two arrays is a bottleneck in many applications. They can be constructed either simultaneously using a single algorithm, a SLACA (suffix and LCP array construction algorithm), or separately constructing the suffix array first using a SACA (suffix array construction algorithm) and then the LCP array from the suffix array and the text using a LACA (LCP array construction algorithm). The latter option is preferred because the separate algorithms are simpler, enable separate development and optimization, and allow many different combinations. The best SACA+LACA combinations are also both faster and more space efficient than the best SLACAs in practice. This is true even in external memory computation as shown in [17,18].

S. Inenaga et al. (Eds.): SPIRE 2016, LNCS 9954, pp. 204–217, 2016.
DOI: 10.1007/978-3-319-46049-9_20

However, asymptotically the external memory LACAs are inferior to the best SLACAs. In the standard external memory (EM) model, with RAM size M and disk block size B, common I/O complexities are $\mathrm{scan}(n) = n/B$, which is the complexity of scanning a sequence of n elements, and $\mathrm{sort}(n) = (n/B)\log_{M/B}(n/B)$, which is the complexity of sorting n elements. The I/O complexity of the best SACAs and SLACAs is $\mathcal{O}(\mathrm{sort}(n))$, where n is the length of the text, which is clearly optimal since the construction involves sorting. The I/O complexity of the external memory LACAs is $\mathcal{O}\big(\mathrm{sort}(n) + (n^2/(MB\log_\sigma^2 n))\big)$ (or worse), where σ is the size of the alphabet. This leaves open the existence of a LACA with I/O complexity $\mathcal{O}(\mathrm{sort}(n))$.

Our Contribution. We describe the first LACA with I/O complexity $\mathcal{O}(\mathrm{sort}(n))$. It is based on two sampling schemes, difference covers and sparse PLCP arrays, both of which have been previously used in LCP array construction, but never together and never in the way we use them.

Difference cover sampling has been used in SACAs [15], SLACAs [21] and LACAs [34] as well as in a data structure for answering longest common extension (LCE) queries [6], which ask for $\mathrm{lcp}(i,j)$, the length of the longest common prefix of the suffixes starting at positions i and j. A difference cover sample defines a subset of text positions with specific properties. All of the above applications compute a sparse suffix array containing in lexicographical order the suffixes starting at the difference cover positions. The corresponding sparse LCP array is used in the LACA and the LCE data structure. The SACA DC3 [15] also involves $\mathcal{O}(n)$ substrings defined by recursive difference covers in the early stages of its computation, and it is these DC-substrings that form the central data structure of our new algorithm. Each DC-substring is assigned a name so that we can compare the equality of two DC-substrings by comparing their names. The names can be computed in $\mathcal{O}(\mathrm{sort}(n))$ I/Os similarly to DC3. Alternatively, in $\mathcal{O}(\mathrm{scan}(n))$ I/Os we can compute Karp-Rabin fingerprints for the DC-substrings, which results in a Monte Carlo algorithm that works correctly with high probability.

We show that given the substring names, we can answer an LCE query $\mathrm{lcp}(i,j)$ in $\mathcal{O}(\log \mathrm{lcp}(i,j))$ time. We can answer informed, approximate LCE queries even faster, where informed means that we are given lower and upper bounds on $\mathrm{lcp}(i,j)$ as input, and approximate means that the output consists of (tighter) lower and upper bounds instead of the exact value. In external memory, we can answer batches of such LCE queries efficiently. Specifically, we can answer a batch of d exact queries in $\mathcal{O}(\mathrm{scan}(n) + \mathrm{sort}(d)\log\ell)$ I/Os, where ℓ is the average value of the results, and informed approximate queries even faster.

The second sampling technique, sparse PLCP array, has been used for LCP array construction in [20]. The PLCP array is a permutation of the LCP array into the text order instead of the lexicographical order, and a sparse PLCP array is a subsequence of the full PLCP array. A sparse PLCP array allows computing lower and upper bounds for the other PLCP array entries. In [20], a simple sparse PLCP was used in a space efficient LCP array construction algorithm. Our new algorithm involves a recursive hierarchy of sparse PLCP arrays, which are used for obtaining input bounds for informed approximate LCE queries.

A careful combination of the two sampling techniques produces the full PLCP array, and thus the LCP array, using $\mathcal{O}(\text{sort}(n))$ I/Os. Furthermore, given an arbitrary sparse suffix array of size $m < n$, the associated LCP array can be computed in $\mathcal{O}(\text{scan}(n) + \text{sort}(m)\log(n))$ I/Os, excluding the computation of the DC-substring names. Using Karp-Rabin fingerprints as DC-substring names results in a Monte Carlo algorithm with the same I/O complexity. If the suffixes are evenly spaced in the text, the I/O complexity can be improved to $\mathcal{O}(\text{scan}(n) + \text{sort}(m)\log(n/m))$.

Related Work. The first SLACA appeared already in the seminal paper by Manber and Myers [28], but the LCP array did not really become popular until Kasai et al. [23] introduced the first LACA. Since then several new LACAs have been developed with a particular emphasis on reducing the space requirements [4,12,20,26,29,34,35]. Some of the algorithms are even semi-external, i.e., they keep most of the data structures on disk but need to have at least the full text in RAM [20,34].

The first I/O-optimal external memory SACA, DC3 [15], came right away with a modification into a SLACA [21, Sect. 4]. Other external memory SLACAs are eSAIS [7], which is I/O optimal, and eGSA [25], which does not have useful worst case bounds on the I/O complexity. Several recent external memory SACAs are based on induced sorting [24,31,32] and could probably be converted into SLACAs using the same technique (introduced in [9]) as eSAIS. For practical purposes, the best SACAs are probably SAscan [16] and pSAscan [19], even though their I/O complexity is a non-optimal $\mathcal{O}\big(\text{sort}(n) + (n^2/(MB\log_\sigma n))\big)$.

The external memory LACAs in [17,18] have an I/O-complexity similar to SAscan and pSAscan, $\mathcal{O}\big(\text{sort}(n) + (n^2/(MB\log_\sigma^2 n))\big)$. Despite the apparently quadratic I/O complexity, the SACA+LACA combination of these algorithms is probably the best way to construct the suffix and LCP arrays for large texts in most practical situations. Based on the analysis and experiments in [18], the text would have to be more than 100 times the size of the available RAM before the quadratic part becomes dominant, and we do not believe our new algorithm would be competitive for any smaller texts. Beyond that limit though, a well engineered implementation of the new algorithm could become the algorithm of choice.

We are not aware of previous results directly comparable to our results on LCE queries and sparse LCP array construction, but there exists tangentially related recent work on external memory range minimum queries [2,3] (since LCE queries can be answered as range minimum queries on the LCP array), as well as on LCE queries [5,11,36] and sparse suffix and LCP array construction [10,14] in internal memory.

2 Preliminaries

Throughout we consider a string $\mathsf{X} = \mathsf{X}[0\,..\,n) = \mathsf{X}[0]\mathsf{X}[1]\ldots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from the alphabet $[0\,..\,\sigma)$. Here and elsewhere we use $[i\,..\,j)$ as a shorthand for $[i\,..\,j-1]$. For $i \in [0\,..\,n]$, we write $\mathsf{X}[i\,..\,n)$ to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}[i\,..\,n) = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n-1]$. We will often refer to suffix $\mathsf{X}[i\,..\,n)$ simply as "suffix i". Similarly, we write $\mathsf{X}[0\,..\,i)$ to denote the *prefix* of X of length i. $\mathsf{X}[i\,..\,j)$ is the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j-1]$ of X that starts at position i and ends at position $j-1$.

The *suffix array* SA of X is an array $\mathrm{SA}[0\,..\,n]$ which contains a permutation of the integers $[0\,..\,n]$ such that $\mathsf{X}[\mathrm{SA}[0]\,..\,n) < \mathsf{X}[\mathrm{SA}[1]\,..\,n) < \cdots < \mathsf{X}[\mathrm{SA}[n]\,..\,n)$. In other words, $\mathrm{SA}[j] = i$ iff $\mathsf{X}[i\,..\,n)$ is the $(j+1)^{\text{th}}$ suffix of X in ascending lexicographical order. The *inverse suffix array* SA^{-1} is the inverse permutation of SA, that is $\mathrm{SA}^{-1}[i] = j$ iff $\mathrm{SA}[j] = i$. Conceptually, $\mathrm{SA}^{-1}[i]$ tells the position of suffix i in SA. Another representation of the permutation is the *Φ array* [20] $\Phi[0\,..\,n)$ defined by $\Phi[\mathrm{SA}[j]] = \mathrm{SA}[j-1]$ for $j \in [1\,..\,n]$. In other words, the suffix $\Phi[i]$ is the immediate lexicographical predecessor of the suffix i.

Let $\mathrm{lcp}(i,j)$ denote the length of the longest-common-prefix (LCP) of suffix i and suffix j. For example, in the string $\mathsf{X} = cccccatcat$, $\mathrm{lcp}(0,3) = 2 = |cc|$, and $\mathrm{lcp}(4,7) = 3 = |cat|$. The *longest-common-prefix array*, $\mathrm{LCP}[1\,..\,n]$, is defined such that $\mathrm{LCP}[i] = \mathrm{lcp}(\mathrm{SA}[i], \mathrm{SA}[i-1])$ for $i \in [1\,..\,n]$. The *permuted LCP array* [20] $\mathrm{PLCP}[0\,..\,n)$ is the LCP array permuted from the lexicographical order into the text order, i.e., $\mathrm{PLCP}[\mathrm{SA}[j]] = \mathrm{LCP}[j]$ for $j \in [1\,..\,n]$. Then $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \Phi[i])$ for all $i \in [0\,..\,n)$. The following property of the PLCP array is the basis of all efficient LACAs.

Lemma 1 ([18]). *Let $i, j \in [0\,..\,n)$. If $i \le j$, then $i + \mathrm{PLCP}[i] \le j + \mathrm{PLCP}[j]$. Symmetrically, if $\Phi[i] \le \Phi[j]$, then $\Phi[i] + \mathrm{PLCP}[i] \le \Phi[j] + \mathrm{PLCP}[j]$.*

Let p be a prime and choose $s \in [0\,..\,p-1]$ uniformly at random. The *Karp-Rabin fingerprint* [22] for a substring $\mathsf{X}[i\,..\,j]$ of X is defined as $\mathrm{FP}[i\,..\,j] = \sum_{k=i}^{j} \mathsf{X}[k] \cdot s^{j-k} \bmod p$. Clearly, if $\mathsf{X}[i\,..\,i+\ell] = \mathsf{X}[j\,..\,j+\ell]$ then $\mathrm{FP}[i\,..\,i+\ell] = \mathrm{FP}[j\,..\,j+\ell]$. On the other hand, if $\mathsf{X}[i\,..\,i+\ell] \ne \mathsf{X}[j\,..\,j+\ell]$ then $\mathrm{FP}[i\,..\,i+\ell] \ne \mathrm{FP}[j\,..\,j+\ell]$ with probability at least $1 - n^{-c}$ for any constant $c > 0$ [8] (assuming $p > n^{c+4}$). The fingerprint of a concatenation can be computed efficiently using $\mathrm{FP}[i\,..\,k] = \mathrm{FP}[i\,..\,j] \cdot s^{k-j} + \mathrm{FP}[j+1\,..\,k] \bmod p$ for any $i \le j < k$.

3 LCE Queries Using DC-substrings

In this section we develop the basic machinery that is used to compute (or approximate) the LCE queries. Assume for simplicity that n is a power of 3. Let $b_{k-1}\ldots b_0$ be the binary representation of integer b. For $k \ge 0$ let

$$S_k = \{a3^k + \sum_{i=0}^{k-1}(b_i+1)3^i \mid a \in [0\,..\,n/3^k), b \in [0\,..\,2^k)\}.$$

Note that for any $k \geq 0$, $S_k \subset [0\,..\,n)$. The set of *DC-substrings* of X is defined as

$$\bigcup_{k=0}^{\log_3 n} \{\mathsf{X}[i\,..\,i+3^k) \mid i \in S_k\}.$$

In the above definition we implicitly assume that X is followed by a sequence of infinitely repeated special symbol that is smaller than any symbol in the alphabet. From the definition of S_k we have $|S_k| = 2^k(n/3^k)$. Thus, the total number of DC-substrings is $n\sum_{k=0}^{\log_3 n}(2/3)^k = \mathcal{O}(n)$.

We want to assign a *name* to each DC-substring such that any two substrings of the same length are equal (or equal with high probability) if and only if their names are equal. We now describe a procedure for computing *deterministic names* (that when compared guarantee the equality of corresponding substrings) and *Monte-Carlo names* (that guarantee the equality with high probability) in external memory. For any $i \in S_k$ we denote the name (of any kind) of DC-substring $\mathsf{X}[i\,..\,i+3^k)$ by $\alpha_k(i)$. We will assume that names for DC-substrings of different lengths are stored in different files on disk, so that accessing the names of all DC-substrings of length 3^k takes $\mathcal{O}\big(\text{scan}\big(n(2/3)^k\big)\big)$ I/Os.

Lemma 2. *The deterministic names of all DC-substrings can be computed using $\mathcal{O}(\text{sort}(n))$ I/Os in the standard EM model.*

Proof. For $k = 0$ we sort all letters of X and assign a rank of each letter (in sorted order) as the name of the substring. For larger k we observe that $S_{k+1} \subset S_k$. Furthermore, if $i \in S_k$ and $i + 3^k < n$ then $i + 3^k \in S_k$. Thus, given the names of DC-substrings of length 3^k we can compute the names for DC-substrings of length 3^{k+1} by sorting the set of triples $\{(\alpha_k(i), \alpha_k(i+3^k), \alpha_k(i+2\cdot 3^k)) \mid i \in S_{k+1}\}$ lexicographically and again assigning a rank of each triple as the name of the corresponding substring (if either of the positions $i+3^k$ and $i+2\cdot 3^k$ are outside the range $[0..n)$ we use -1 as the name of the corresponding substring). A single sorting step takes $\mathcal{O}(\text{scan}(|S_k|) + \text{sort}(|S_{k+1}|)) = \mathcal{O}\big(\text{sort}\big(n(2/3)^k\big)\big)$ I/Os which over all lengths of DC-substrings sums up to $\mathcal{O}(\text{sort}(n))$ I/Os. □

Lemma 3. *The Monte-Carlo names of all DC-substrings can be computed using $\mathcal{O}(\text{scan}(n))$ I/Os in the standard EM model.*

Proof. The goal is to compute Karp-Rabin fingerprint for every DC-substring. The general scheme of the naming procedure follows the one from Lemma 2. However, unlike in Lemma 2 the Monte-Carlo name of substring $\mathsf{X}[i\,..\,i+3^{k+1})$ can be directly computed from the names of substrings $\mathsf{X}[i\,..\,i+3^k)$, $\mathsf{X}[i+3^k\,..\,i+2\cdot 3^k)$, and $\mathsf{X}[i+2\cdot 3^k\,..\,i+3\cdot 3^k)$ in $\mathcal{O}(1)$ time. Thus, we only need to scan the file containing the names of DC-substrings of length 3^k which takes $\mathcal{O}\big(\text{scan}\big(n(2/3)^k\big)\big)$ I/Os. Over all lengths of DC-substrings we spend $\mathcal{O}(\text{scan}(n))$ I/Os. □

Note that to efficiently collect the names during scans in the above lemmas, within a single file we need to additionally group the names of DC-substrings according to the value $i \bmod 3^k$, where i is the starting position of the substring.

We will next show how to use names of DC-substrings to efficiently compute or approximate LCE queries. For simplicity we now describe the internal-memory versions of basic procedures and later explain how to modify them for external memory. Figure 1 gives a pseudo-code of an algorithm to answer an LCE query for an arbitrary pair of suffixes. The number of iterations of the while loop in lines 3–7 is bounded using the following lemma.

Lemma 4. *Let $i, j \in S_k$ and assume that* $\max\{i, j\} + 2 \cdot 3^k < n$. *Then either* $\{i, j\} \subset S_{k+1}$ *or* $\{i + 3^k, j + 3^k\} \subset S_{k+1}$ *or* $\{i + 2 \cdot 3^k, j + 2 \cdot 3^k\} \subset S_{k+1}$.

Proof. Let a, b be such that $i = a3^k + \sum_{i=0}^{k-1}(b_i + 1)3^i$ where $a \in [0\,..\,n/3^k)$ and $b \in [0\,..\,2^k)$ (which exist from the definition of S_k). It is easy to check that $i \in S_{k+1}$ iff $a \bmod 3 \neq 0$. Thus, exactly two out of $\{i, i + 3^k, i + 2 \cdot 3^k\}$ are in S_{k+1}. Since the analogous property holds for j, the claim follows. □

```
**Function** lcp$(i, j)$
1:  $i_{\text{init}} \leftarrow i$
2:  $k \leftarrow 0$
3:  **while** $\alpha_k(i) = \alpha_k(j)$ **do**
4:      $i \leftarrow i + 3^k$
5:      $j \leftarrow j + 3^k$
6:      **if** $\{i, j\} \subset S_{k+1}$ **then**
7:          $k \leftarrow k + 1$
8:  **while** $k > 0$ **do**
    — *Invariant:* $\alpha_k(i) \neq \alpha_k(j)$
9:      $k \leftarrow k - 1$
10:     **while** $\alpha_k(i) = \alpha_k(j)$ **do**
11:         $i \leftarrow i + 3^k$
12:         $j \leftarrow j + 3^k$
13: **return** $i - i_{\text{init}}$
```

```
**Function** lcp$_h(i, j, \check{\ell}, \hat{\ell})$
1:  **if** $\hat{\ell} - \check{\ell} \leq 3^h$ **then**
2:      **return** $(\check{\ell}, \hat{\ell})$
3:  $i_{\text{init}} \leftarrow i$
4:  $k \leftarrow \min(\lfloor \log_3(\check{\ell} + 1) \rfloor, \lfloor \log_3(\hat{\ell} - \check{\ell}) \rfloor)$
5:  $\delta \leftarrow \delta_k^-(i + \check{\ell}, j + \check{\ell})$
6:  $i \leftarrow i + \check{\ell} - \delta$, $j \leftarrow j + \check{\ell} - \delta$
7:  **while** $\alpha_k(i) = \alpha_k(j)$ **do**
8:      $i \leftarrow i + 3^k$
9:      $j \leftarrow j + 3^k$
10:     **if** $\{i, j\} \subset S_{k+1}$ **then**
11:         $k \leftarrow k + 1$
12: **while** $k > h$ **do**
    — *Invariant:* $\alpha_k(i) \neq \alpha_k(j)$
13:     $k \leftarrow k - 1$
14:     **while** $\alpha_k(i) = \alpha_k(j)$ **do**
15:         $i \leftarrow i + 3^k$
16:         $j \leftarrow j + 3^k$
17: $\check{\ell} \leftarrow \max(\check{\ell}, i - i_{\text{init}})$
18: $\hat{\ell} \leftarrow \min(\hat{\ell}, i - i_{\text{init}} + 3^k)$
19: **return** $(\check{\ell}, \hat{\ell})$
```

Fig. 1. Left: Computation of lcp(i, j) using DC-substrings. Right: Approximating the value of lcp(i, j). Given $\check{\ell}$ and $\hat{\ell}$ such that $\check{\ell} \leq \text{lcp}(i, j) < \hat{\ell}$ the function lcp$_h$ returns a pair $(\check{\ell}, \hat{\ell})$ that in addition satisfies $\hat{\ell} - \check{\ell} \leq 3^h$. In both functions we assume $i \neq j$

To perform the check $\{i, j\} \subset S_{k+1}$ efficiently, we identify the DC-substrings of length 3^k starting at positions i and j using triples (k, a, b) where a, b are as in the definition of S_k. This representation supports the check in constant time. Every update of i and j (represented in this way) in Fig. 1 also takes constant time. Thus, since by Lemma 4 the lcp algorithm uses $\mathcal{O}(1)$ DC-substrings of each length it runs in $\mathcal{O}(\log n)$ time. A more careful analysis shows that the algorithm only inspects DC-substrings up to length $\Theta(\mathrm{lcp}(i,j))$, and thus its running time is in fact $\mathcal{O}(\log \mathrm{lcp}(i,j))$.

Given $h \geq 0$, $\check{\ell}$, and $\hat{\ell}$ such that $\check{\ell} \leq \mathrm{lcp}(i,j) < \hat{\ell}$ we define the *informed approximate LCE query with accuracy* 3^h as the task of refining the *slack* defined as $\hat{\ell} - \check{\ell}$, so that in addition to initial assumption, it satisfies $\hat{\ell} - \check{\ell} \leq 3^h$. We now describe a method of answering approximate LCE queries using DC-substrings. We start by introducing useful auxiliary functions.

Lemma 5. *For any $k \geq 0$ and any $i,j \in [0\,..\,n)$, $\max\{i,j\} + 3^k \leq n$, there exists $\delta \in [0\,..\,3^k)$ such that $\{i+\delta, j+\delta\} \subset S_k$. We denote such δ by $\delta_k^+(i,j)$. Symmetrically, if $i,j \geq 3^k - 1$, there exists $\delta \in [0\,..\,3^k)$ such that $\{i-\delta, j-\delta\} \subset S_k$. We denote such δ by $\delta_k^-(i,j)$.*

Proof. Clearly $\{i,j\} \subset S_0$. By Lemma 4 we can find $\delta_0 \leq 2$ such that $\{i+\delta_0, j+\delta_0\} \subset S_1$. Iteratively applying Lemma 4 gives $\delta = \delta_0 + \ldots + \delta_{k-1}$ such that $\{i+\delta, j+\delta\} \subset S_k$. Since $\delta_t \leq 2 \cdot 3^t$, we have $\delta \leq 2\sum_{t=0}^{k-1} 3^t < 3^k$. The proof for $\delta_k^-(i,j)$ is analogous. $\square$

The pseudo-code of the function approximating lcp is given in Fig. 1. It works essentially the same as the exact version except we start (lines 4–5) by computing k and δ such that $0 \leq \delta < 3^k \leq \check{\ell} + 1$ and $3^k \leq \hat{\ell} - \check{\ell}$. The first condition ensures that i and j are increased by a value in the interval $[0\,..\,\check{\ell}]$ in line 6 (which is correct from the definition of $\check{\ell}$). The second condition guarantees that the algorithm does not use DC-substrings longer than $\Theta(\hat{\ell} - \check{\ell})$. This is necessary in the case $\hat{\ell} - \check{\ell} \ll \check{\ell}$ because otherwise the algorithm would perform $\Theta(\log(\check{\ell}/(\hat{\ell} - \check{\ell})))$ comparisons of DC-substrings in the while loop in lines 12–16 which are guaranteed (from the definition of $\hat{\ell}$) to not be equal. The shortest DC-substrings used in the algorithm are of length $\Omega(\min(\check{\ell}, 3^h))$. Thus, the number of compared DC-substrings is $\mathcal{O}(\log((\hat{\ell} - \check{\ell})/\min(\check{\ell}, 3^h)))$.

In the remainder of the paper we focus on a special type of informed approximate LCE queries for which the bounds provided as input additionally satisfy $3^h \leq \check{\ell}, \hat{\ell} - \check{\ell}$, where 3^h is the required accuracy of the query. We call them 3^h-*LCE queries*. Note that from the discussion above a 3^h-LCE query can be answered using $\mathcal{O}(\log((\hat{\ell} - \check{\ell})/3^h))$ comparisons.

4 Answering Batches of LCE Queries

Assume we are given a sequence of d LCE queries $R = [(i_1, j_1), \ldots, (i_d, j_d)]$. We can answer a single LCE query (i,j) using $\mathcal{O}(\log \mathrm{lcp}(i,j))$ comparisons of DC-substrings. Thus, to answer a batch of d queries we need $\mathcal{O}(\sum_{t=1}^{d} \log \mathrm{lcp}(i_t, j_t))$ comparisons. By Jensen's inequality this is bounded by $\mathcal{O}(d \log \ell)$ where $\ell = (\sum_{t=1}^{d} \mathrm{lcp}(i_t, j_t))/d$ is the average lcp value. Thus, we obtain the following lemma.

Lemma 6. *It suffices to compare $\mathcal{O}(d \log \ell)$ DC-substrings to answer a batch of d LCE queries with an average value ℓ.*

Consider now the task of answering a batch of d 3^h-LCE queries $R = [(i_1, j_1, \check{\ell}_1, \hat{\ell}_1), \ldots, (i_d, j_d, \check{\ell}_d, \hat{\ell}_d)]$. As shown in the previous section, answering a single 3^h-LCE query takes $\mathcal{O}(\log((\hat{\ell} - \check{\ell})/3^h))$ comparisons, thus a batch of d queries needs $\mathcal{O}(\sum_{t=1}^{d} \log((\hat{\ell}_t - \check{\ell}_t)/3^h))$ comparisons. Again, by Jensen's inequality this is bounded by $\mathcal{O}\big(d \log(\ell/3^h)\big)$, where $\ell = (\sum_{t=1}^{d}(\hat{\ell}_t - \check{\ell}_t))/d$ is the average slack in R.

Lemma 7. *It suffices to compare $\mathcal{O}\big(d \log(\ell/3^h)\big)$ DC-substrings to answer a batch of d 3^h-LCE queries with an average slack of ℓ.*

Suppose now we want to answer a batch of d LCE queries in external memory. Assume that both the set of queries R and names of all DC-substrings are stored on disk. We divide the lcp algorithm in Fig. 1 into two phases corresponding to loops in lines 3–7 and 8–12.

Consider the first phase. During the algorithm we maintain $\log_3 n$ files on disk and at any given moment each LCE query is stored in exactly one of the files. The k-th file stores all triples (i_{init}, i, j) such that i_{init} corresponds to the value that we store in line 1, and $\{i, j\} \subset S_k$ stores the current state of the query. We process files in increasing order of k. To process k-th file we scan all triples and for every (i_{init}, i, j) we generate requests to retrieve the names $\alpha_k(i)$, $\alpha_k(i+3^k)$, $\alpha_k(i+2\cdot3^k)$, $\alpha_k(j)$, $\alpha_k(j+3^k)$, $\alpha_k(j+2\cdot3^k)$. By Lemma 4 these are the only names of DC-substrings of length 3^k used by the lcp algorithm. All name requests are first sorted by the starting position and then the corresponding names are retrieved with a single scan of the file containing k-level names. The name requests are then sorted back to the original order and each of the LCE queries is now updated. Depending on the result of the name comparison, the query either stays in the current file (mismatch) or is moved to the $(k+1)$-th file (match) and the values i, j are updated.

If by d_k we denote the number of triples in the k-th file then executing the k-th step takes $\mathcal{O}\big(\text{scan}\big(n(2/3)^k\big) + \text{sort}(d_k)\big)$ I/Os. Over all steps the I/O is $\mathcal{O}(\text{scan}(n) + \sum_{k=0}^{\log_3 n} \text{sort}(d_k))$. By Lemma 6 we have $\sum_{k=0}^{\log_3 n} d_k = \mathcal{O}(d \log \ell)$ where ℓ is the average lcp value. Thus by Jensen's inequality the total I/O volume is bounded by $\mathcal{O}(\text{scan}(n) + \text{sort}(d) \log \ell)$.

To execute the second stage of the algorithm (lines 8–12) the algorithm proceeds analogously, except now we process the remaining items in all files in the decreasing order of k. The I/O complexity does not change.

Lemma 8. *A batch of d LCE queries with an average value ℓ can be answered using $\mathcal{O}(\text{scan}(n) + \text{sort}(d) \log \ell)$ I/Os in the standard EM model.*

Answering a batch of 3^h-LCE queries in external memory works analogously and the result follows from Lemma 7. We don't access DC-substrings shorter than 3^h and thus the scanning time is reduced.

```
Function Reduce($k, R$)
  — Step 1: For all $i$ determine if $\mathrm{lcp}(i, \Phi(i)) \geq 3^{k+1} - 1$
1:  foreach $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \in R$ do
2:      $\delta \leftarrow \delta_k^+(i, \Phi(i))$
3:      if $\alpha_k(i + \delta) \neq \alpha_k(\Phi(i) + \delta)$ then $\hat{\ell}_i \leftarrow \delta + 3^k$
4:      elsif $\alpha_k(i + \delta + 3^k) \neq \alpha_k(\Phi(i) + \delta + 3^k)$ then $(\check{\ell}_i, \hat{\ell}_i) \leftarrow (\delta + 3^k, \delta + 2{\cdot}3^k)$
5:      elsif $\alpha_k(i + \delta + 2{\cdot}3^k) \neq \alpha_k(\Phi(i) + \delta + 2{\cdot}3^k)$ then $(\check{\ell}_i, \hat{\ell}_i) \leftarrow (\delta + 2{\cdot}3^k, \delta + 3{\cdot}3^k)$
6:      else $\check{\ell}_i \leftarrow \delta + 3{\cdot}3^k$
  — Step 2: Create a sample of $\mathcal{O}(|R|/3)$ tuples
7:  $R' \leftarrow [(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \in R \mid \check{\ell}_i \geq 3^{k+1} - 1]$ (sorted by $i$)
8:  $S \leftarrow [R'[3i] \mid i \in [0..\lceil |R'|/3 \rceil)]$
  — Step 3: Recursively reduce the sample slacks to at most $3^{k+1}$
9:  if $S \neq \emptyset$ then Reduce($k + 1$, $S$)
  — Step 4: Using sample slacks reduce the total slack to $\mathcal{O}(n)$
10: foreach $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \in R' \setminus S$ do
11:     Let $(p, \Phi(p), \check{\ell}_p, \hat{\ell}_p)$ be the predecessor (w.r.t. $i$) of $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)$ in $S$
12:     Let $(s, \Phi(s), \check{\ell}_s, \hat{\ell}_s)$ be the successor (w.r.t. $i$) of $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)$ in $S$
13:     $\check{\ell}_i \leftarrow \max(\check{\ell}_i, \check{\ell}_p - (i - p))$
14:     $\hat{\ell}_i \leftarrow \min(\hat{\ell}_i, \hat{\ell}_s + (s - i))$
  — Step 5: Reduce all slacks to at most $3^k$
15: foreach $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \in R$ do
16:     if $\hat{\ell}_i - \check{\ell}_i > 3^k$ then
17:         $(\check{\ell}_i, \hat{\ell}_i) \leftarrow \mathrm{lcp}_k(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)$
```

Fig. 2. Given $R = [(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)]$, $|R| \leq n/3^k$ such that $3^k - 1 \leq \check{\ell}_i \leq \mathrm{lcp}(i, \Phi(i)) < \hat{\ell}_i$, refine all $\check{\ell}_i, \hat{\ell}_i$ so that in addition to initial assumptions they satisfy $\hat{\ell}_i - \check{\ell}_i \leq 3^k$

Lemma 9. *A batch of d 3^h-LCE queries with an average slack ℓ can be answered using $\mathcal{O}\big(\mathrm{scan}\big(n(2/3)^h\big) + \mathrm{sort}(d)\log(\ell/3^h)\big)$ I/Os in the standard EM model.*

Note that all the complexities stated in this section exclude the time needed to compute the names of DC-substrings. Thus, to solve an arbitrary set of LCE queries we need to additionally spend $\mathcal{O}(\mathrm{sort}(n))$ I/Os to compute deterministic names (Lemma 2) or $\mathcal{O}(\mathrm{scan}(n))$ I/Os to compute Monte-Carlo names (Lemma 3).

5 LCP Array Construction

Let $k \geq 0$ and consider an arbitrary subset P of at most $n/3^k$ text positions from $[0..n)$ such that $3^k - 1 \leq \mathrm{lcp}(i, \Phi(i))$ for all $i \in P$. Let $R = \{(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \mid i \in P\}$ be such that $3^k - 1 \leq \check{\ell}_i \leq \mathrm{lcp}(i, \Phi(i)) < \hat{\ell}_i$ for all i. The main ingredient of the new LCP array construction algorithm is the procedure to reduce all slacks in R to at most 3^k. It uses batched 3^k-LCE queries in the final step after first improving the bounds with a different technique.

The pseudo-code of the procedure is given in Fig. 2. For simplicity we use the standard notation for internal-memory algorithms. Below we outline all steps and explain how to implement them in external memory using scanning and sorting.

We start by checking, for every i, whether $\mathrm{lcp}(i, \Phi(i)) \geq 3^{k+1} - 1$. This requires fetching $\mathcal{O}(1)$ names of DC-substrings of length 3^k for each tuple, and thus takes $\mathcal{O}\big(\mathrm{scan}\big(n(2/3)^k\big) + \mathrm{sort}(|R|)\big)$ I/Os. Next, we create a sample consisting of every third tuple (we assume they are sorted by i) for which $\mathrm{lcp}(i, \Phi(i)) \geq 3^{k+1} - 1$ and recursively reduce the slacks of the sample to at most 3^{k+1}. Excluding the cost of recursive call, it takes $\mathcal{O}(\mathrm{sort}(|R|))$ I/Os. In the next step we use the slacks of the sample set to reduce all remaining slacks. The correctness of the reduction follows from Lemma 1. The reduced slacks satisfy the following property.

Lemma 10. *The total slack in R after step 4 is $\mathcal{O}(n)$.*

Proof. Denote the elements of S after returning from recursion (line 9) by $(i^S_j, \Phi(i^S_j), \check{\ell}^S_j, \hat{\ell}^S_j)$, where $j \in [1\,..\,|S|]$ and assume $i^S_j < i^S_{j+1}$ for $j \in [1\,..\,|S|)$. For $j = |S|+1$ we set $i^S_j = n$, $\hat{\ell}^S_j = 0$. Let $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i) \in R' \backslash S$ be one of the elements processed in line 10. Let $(i^S_j, \Phi(i^S_j), \check{\ell}^S_j, \hat{\ell}^S_j)$ be the predecessor of $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)$ in S (which always exists since S contains the smallest item of R'). Then the successor of $(i, \Phi(i), \check{\ell}_i, \hat{\ell}_i)$ in S is $(i^S_{j+1}, \Phi(i^S_{j+1}), \check{\ell}^S_{j+1}, \hat{\ell}^S_{j+1})$. The slack of the processed tuple after the update satisfies $\hat{\ell}_i - \check{\ell}_i \leq (\hat{\ell}^S_{j+1} + (i^S_{j+1} - i)) - (\check{\ell}^S_j - (i - i^S_j)) = (\hat{\ell}^S_{j+1} - \check{\ell}^S_j) + (i^S_{j+1} - i^S_j)$. Since each tuple in S can be the predecessor of at most two elements in $R' \backslash S$ (from definition of S), the total slack in $R' \backslash S$ is bounded by

$$2\sum_{j=1}^{|S|}((\hat{\ell}^S_{j+1} - \check{\ell}^S_j) + (i^S_{j+1} - i^S_j)) \leq 2n + 2\sum_{j=1}^{|S|}(\hat{\ell}^S_j - \check{\ell}^S_j).$$

Since the total slack in S does not exceed $|S| \cdot 3^{k+1} = \mathcal{O}(n)$, the total slack in $R' \backslash S$ is also $\mathcal{O}(n)$. Finally, by Step 1 and definition of R', the slack in $R \backslash R'$ is not greater than $|R| \cdot 3^{k+1} = \mathcal{O}(n)$. □

As a last step we apply the algorithm from the previous section to answer a batch of at most $|R|$ approximate lcp queries. The average slack in R at this point is $\mathcal{O}(n/|R|)$, and thus by Lemma 9 answering all approximate lcp queries takes $\mathcal{O}\big(\mathrm{scan}\big(n(2/3)^k\big) + \mathrm{sort}(|R|)\log(n/3^k|R|)\big)$ I/Os. Excluding the cost of the recursive call, this step dominates the I/O complexity.

Consider the call of Reduce processing the sample S (i.e., the first level of recursion). The number of performed I/Os (excluding deeper recursive calls) is $\mathcal{O}\big(\mathrm{scan}\big(n(2/3)^{k+1}\big) + \mathrm{sort}(|S|)\log(n/3^{k+1}|S|)\big)$. We have $|S| \leq |R|/3$, but since $\mathcal{O}\big(\mathrm{sort}(d)\log(n/3^{k+1}d)\big)$ as a function of d is non-decreasing (assuming $d = \mathcal{O}\big(n/3^{k+1}\big)$), the I/O complexity is maximized for $|S| = |R|/3$ and hence $\mathrm{sort}(|S|)\log(n/3^{k+1}|S|) = \mathcal{O}\big(\mathrm{sort}(|R|/3)\log(n/3^k|R|)\big)$. Thus, since I/O decreases exponentially with every level of recursion, the total I/O complexity of Reduce(k,R) is not greater than the complexity at zero-level recursion.

Lemma 11. *Reduce uses* $\mathcal{O}\big(\mathrm{scan}\big(n(2/3)^k\big) + \mathrm{sort}(|R|)\log(n/3^k|R|)\big)$ *I/Os in the standard EM model.*

Using Reduce we can compute the LCP array as follows. We scan the suffix array and for every $i > 0$ we create a tuple $(i, \mathrm{SA}[i], \mathrm{SA}[i-1])$. We then sort these tuples by the second component to obtain a sequence $(\mathrm{SA}^{-1}[i], i, \Phi[i])$. Using that we create a sequence of tuples $(i, \Phi[i], 0, n)$ which is then used as an input to Reduce with $k = 0$. As a result we obtain a sequence $(i, \cdot, \mathrm{PLCP}[i], \cdot)$, which we can now permute into LCP using SA^{-1} values. The total I/O complexity (including the computation of deterministic names for DC-substrings) is $\mathcal{O}(\mathrm{sort}(n))$.

Theorem 1. *Using DC-substrings the LCP array can be computed correctly in the standard EM model from the text and its suffix array using* $\mathcal{O}(\mathrm{sort}(n))$ *I/Os.*

6 $o(\mathrm{sort}(n))$ I/Os

Finally, we look at some variants of the LCP array construction problem, where we can achieve an I/O complexity of $o(\mathrm{sort}(n))$. In all of these, we use the Monte Carlo names of DC-substrings, which can be computed in $\mathcal{O}(\mathrm{scan}(n))$ I/Os, and thus the results are correct with high probability.

A sparse suffix array contains some subset of suffixes in the lexicographical order and the associated LCP array contains lcps of suffixes that are lexicographically adjacent in that subset. The LCP array can be computed as a batch of LCE queries and thus by Lemma 8 we obtain the following result.

Theorem 2. *Given a sparse suffix array containing $m < n$ suffixes of a text of length n in sorted order it takes*

$$\mathcal{O}(\mathrm{scan}(n) + \mathrm{sort}(m)\log(\ell)) = \mathcal{O}(\mathrm{scan}(n) + \mathrm{sort}(m)\log(n))$$

I/Os in the standard EM model to compute the corresponding LCP array correctly with high probability, where ℓ is the average value in the LCP array.

We can do better in the special case of evenly spaced sparse suffix array that contains exactly every q^{th} suffix of the text for some $q \geq 1$. In this case, we can use the Reduce algorithm and give as input the set of pairs $(i, \Phi(i))$, where i and $\Phi(i)$ are divisible by q and the suffix $\Phi(i)$ is the immediate lexicographical predecessor of the suffix i among the sparse set. Notice that this approach does not work correctly for an arbitrary sparse set (because of Step 4 in Reduce).

Theorem 3. *Given an evenly spaced sparse suffix array containing every q^{th} suffix of a text of length n in sorted order it takes* $\mathcal{O}(\mathrm{scan}(n) + \mathrm{sort}(n/q)\log(q))$ *I/Os in the standard EM model to compute the corresponding LCP array correctly with high probability.*

If $q = \omega(1)$, the number of performed I/Os is $o(\text{sort}(n))$. For example, if $q = \Omega((\log_{M/B}(n/B))^2)$, the I/O complexity is $\mathcal{O}(\text{scan}(n))$.

Finally, we can sometimes compute the full PLCP array using $o(\text{sort}(n))$ I/Os by computing first a subset of so called *irreducible* lcp values, from which the other lcp values are easy to derive (see [18,20]). For highly repetitive texts, the number r of the irreducible lcp values can be much smaller than n. To identify the irreducible positions quickly we need the Burrows–Wheeler transform as an additional input. We can compute the irreducible entries of the PLCP array using Reduce in $\mathcal{O}(\text{scan}(n) + \text{sort}(r)\log(n/r))$ I/Os and then the other entries by a simple scan in $\mathcal{O}(\text{scan}(n))$ I/Os.

Theorem 4. *Given the suffix array and the Burrows-Wheeler transform of a text of length n, it takes* $\mathcal{O}(\text{scan}(n) + \text{sort}(r)\log(n/r))$ *I/Os in the standard EM model to compute the PLCP array correctly with high probability, where r is the number of irreducible lcp values.*

If $r = o(n)$, the complexity is $o(\text{sort}(n))$. Transforming the PLCP array into an LCP array still needs $\Theta(\text{sort}(n))$ I/Os.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004)
2. Afshani, P., Sitchinava, N.: I/O-efficient range minima queries. In: Ravi, R., Gørtz, I.L. (eds.) SWAT 2014. LNCS, vol. 8503, pp. 1–12. Springer, Heidelberg (2014)
3. Arge, L., Fischer, J., Sanders, P., Sitchinava, N.: On (dynamic) range minimum queries in external memory. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 37–48. Springer, Heidelberg (2013)
4. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows-Wheeler transform. J. Discrete Algorithms **18**, 22–31 (2013)
5. Bille, P., Gørtz, I.L., Knudsen, M.B.T., Lewenstein, M., Vildhøj, H.W.: Longest common extensions in sublinear space. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 65–76. Springer, Heidelberg (2015)
6. Bille, P., Gørtz, I.L., Sach, B., Vildhøj, H.W.: Time-space trade-offs for longest common extensions. J. Discrete Algorithms **25**, 42–50 (2014)
7. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. In: Sanders, P., Zeh, N. (eds.) ALENEX 2013. pp. 88–102. SIAM (2013)
8. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 235–246. Springer, Heidelberg (1992)
9. Fischer, J.: Inducing the LCP-array. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 374–385. Springer, Heidelberg (2011)
10. Fischer, J., I, T., Köppl, D.: Deterministic sparse suffix sorting on rewritable texts. In: Kranakis, E., Navarro, G., Chávez, E. (eds.) LATIN 2016. LNCS, vol. 9644, pp. 483–496. Springer, Heidelberg (2016)

11. Gawrychowski, P., Kociumaka, T., Rytter, W., Walen, T.: Faster longest common extension queries in strings over general alphabets. In: Grossi, R., Lewenstein, M. (eds.) CPM 2016. LIPIcs, vol. 54. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
12. Gog, S., Ohlebusch, E.: Fast and lightweight LCP-array construction algorithms. In: Müller-Hannemann, M., Werneck, R.F.F. (eds.) ALENEX 2011. pp. 25–34. SIAM (2011)
13. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In: Frakes, W.B., Baeza-Yates, R. (eds.) Information Retrieval: Data Structures & Algorithms, pp. 66–82. Prentice-Hall, Englewood Cliffs (1992)
14. I, T., Kärkkäinen, J., Kempa, D.: Faster sparse suffix sorting. In: Mayr, E.W., Portier, N. (eds.) STACS 2014. LIPIcs, vol. 25, pp. 386–396. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014)
15. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
16. Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. In: Iliopoulos, C.S., Langiu, A. (eds.) ICABD 2014. pp. 53–60 (2014)
17. Kärkkäinen, J., Kempa, D.: Faster external memory LCP array construction. In: Sankowski, P., Zaroliagis, C. (eds.) ESA 2016. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
18. Kärkkäinen, J., Kempa, D.: LCP array construction in external memory. J. Exp. Algorithmics **21**(1), 1.7:1–1.7:22 (2016)
19. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 329–342. Springer, Heidelberg (2015)
20. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
21. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
22. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
23. Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
24. Liu, W.J., Nong, G., Chan, W.H., Wu, Y.: Induced sorting suffixes in external memory with better design and less space. In: Iliopoulos, C., Puglisi, S., Yilmaz, E. (eds.) SPIRE 2015. LNCS, vol. 9309, pp. 83–94. Springer, Heidelberg (2015)
25. Louza, F.A., Telles, G.P., De Aguiar Ciferri, C.D.: External memory generalized suffix and LCP arrays construction. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 201–210. Springer, Heidelberg (2013)
26. Mäkinen, V.: Compact suffix array – a space efficient full-text index. Fund. Inform. **56**(1–2), 191–210 (2003)
27. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing. Cambridge University Press, Cambridge (2015)
28. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)

29. Manzini, G.: Two space saving tricks for linear time LCP array computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
30. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), 2 (2007)
31. Nong, G., Chan, W.H., Hu, S.Q., Wu, Y.: Induced sorting suffixes in external memory. ACM Trans. Inf. Syst. **33**(3), 12:1–12:15 (2015)
32. Nong, G., Chan, W.H., Zhang, S., Guan, X.F.: Suffix array construction in external memory using d-critical substrings. ACM Trans. Inf. Syst. **32**(1), 1:1–1:15 (2014)
33. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag, Bremen (2013)
34. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
35. Sirén, J.: Sampled longest common prefix array. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 227–237. Springer, Heidelberg (2010)
36. Tanimura, Y., I, T., Bannai, H., Inenaga, S., Puglisi, S.J., Takeda, M.: Deterministic sub-linear space LCE data structures with efficient construction. In: Grossi, R., Lewenstein, M. (eds.) CPM 2016. LIPIcs, vol. 54, pp. 1:1–1:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)