Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Journal of Discrete Algorithms 2 (2004) 53–86

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

Replacing suffix trees with enhanced suffix arrays

Mohamed Ibrahim Abouelhoda [a], Stefan Kurtz [b], Enno Ohlebusch [a,*]

[a] *Faculty of Computer Science, University of Ulm, 89069 Ulm, Germany*
[b] *Center for Bioinformatics, University of Hamburg, 20146 Hamburg, Germany*

Abstract

The suffix tree is one of the most important data structures in string processing and comparative genomics. However, the space consumption of the suffix tree is a bottleneck in large scale applications such as genome analysis. In this article, we will overcome this obstacle. We will show how *every* algorithm that uses a suffix tree as data structure can systematically be replaced with an algorithm that uses an enhanced suffix array and solves the same problem in the same time complexity. The generic name *enhanced suffix array* stands for data structures consisting of the suffix array and additional tables. Our new algorithms are not only more space efficient than previous ones, but they are also faster and easier to implement.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Suffix array; Suffix tree; Repeat analysis; Genome comparison; Pattern matching

1. Introduction

The suffix tree is undoubtedly one of the most important data structures in string processing. This is particularly true if the sequences to be analyzed are very large and do not change. An example of prime importance from the field of bioinformatics is genome analysis, where the sequences under consideration are whole genomes (the human genome, for example, contains more than $3 \cdot 10^9$ base pairs).

The suffix tree of a sequence S is an index structure that can be computed and stored in $O(n)$ time and space [32], where $n = |S|$. Once constructed, it can be used to efficiently

[*] Corresponding author.
E-mail address: eo@informatik.uni-uni-ulm.de (E. Ohlebusch).

Table 1
The suffix tree applications from [15] and the kinds of traversals they require

Application	Type of tree traversal		
	Bottom-up	Top-down	Suffix-links
Supermaximal repeats	√		
Maximal repeats	√		
Maximal repeated pairs	√		
Longest common substring	√		
All-pairs suffix-prefix matching	√		
Ziv–Lempel decomposition	√		
Common substrings of multiple strings	√	√	
Exact string matching		√	
Exact set matching		√	
Matching statistics		√	√
Construction of DAWGs		√	√

solve a "myriad" of string processing problems [3], and Gusfield devotes about 70 pages of his book [15] to applications of suffix trees. These applications can be classified into the following kinds of tree traversals:

- a bottom-up traversal of the complete suffix tree,
- a top-down traversal of a subtree of the suffix tree,
- a traversal of the suffix tree using suffix links.

Table 1 shows some of the suffix-tree applications discussed in [15] plus the kind of traversal they use.

While suffix trees play a prominent role in algorithmics, they are not as widespread in actual implementations of software tools as one should expect. There are two major reasons for this:

(i) Although being asymptotically linear, the space consumption of a suffix tree is quite large; even recently improved implementations of linear time constructions still require 20 bytes per input character in the worst case; see, e.g., [25].
(ii) In most applications, the suffix tree suffers from a poor locality of memory reference, which causes a significant loss of efficiency on cached processor architectures, and renders it difficult to store in secondary memory.

These problems have been identified in several large scale applications like the repeat analysis of whole genomes [27] and the comparison of complete genomes [8,17].

More space efficient data structures than the suffix tree exist. The most prominent one is the *suffix array*, which was introduced by Manber and Myers [29] and independently by Gonnet et al. [13] under the name PAT array. The suffix array requires only $4n$ bytes in its basic form and it can be constructed in $O(n)$ time in the worst case by first constructing the suffix tree of S; see [15]. Very recently, it was shown independently and contemporaneously in [19,21,23] that a direct linear time construction of the suffix array is possible. However, at first glance, it seems that the suffix array has a disadvantage over the suffix tree: It is not clear that (and how) every algorithm using a suffix tree can be replaced with an algorithm based on a suffix array solving the same problem in the same time complexity. For example, using only the basic suffix array, it takes $\mathrm{O}(m \log n)$ time in the worst case to answer decision queries of the type "Is P a substring of S?", where $m = |P|$. In this paper, we will show that every algorithm using a suffix tree can be replaced with an equivalent algorithm based on a suffix array and additional information. It will be demonstrated how to efficiently solve all problems with enhanced suffix arrays that are usually solved by a bottom-up or a top-down traversal of the suffix tree. Moreover, we will show how traversals of the suffix tree that use suffix links can be simulated over an enhanced suffix array.

In Section 3, we treat applications (such as computing supermaximal repeats and maximal unique matches) that are solely based on the properties of the enhanced suffix array.

In Section 4, we will take the approach of Kasai et al. [20] one step further. They showed that every bottom-up traversal of a suffix tree can be simulated on a suffix array enhanced with the longest common prefix (lcp) information, but they did not take the information of the child nodes of an internal node of the suffix tree into account. We will introduce the concept of *lcp-interval trees* to remedy this. The lcp-interval tree of an enhanced suffix array is only conceptual (i.e., it is not really built) but it allows us to simulate all kinds of suffix tree traversals very efficiently.

With the help of the lcp-interval tree, it will be shown in Section 5 how to solve all problems with enhanced suffix arrays that are usually solved by a bottom-up traversal of the suffix tree. As examples, we show how to compute all maximal repeated pairs and the Ziv–Lempel decomposition of a string. These application use the suffix array and the lcp-table, both of which can be stored in $4n$ bytes.

In Section 6, we are concerned with problems that are usually solved by a top-down traversal of the suffix tree. A prime example is exact pattern matching. Using an additional table, Manber and Myers [29] showed that decision queries can be answered in $\mathrm{O}(m + \log n)$ time in the worst case. However, no $\mathrm{O}(m)$ time algorithm based on the suffix array was known for this task. In this paper, we will show how decision queries can be answered in optimal $\mathrm{O}(m)$ time and how to find all z occurrences of a pattern P in optimal $\mathrm{O}(m + z)$ time. This new result is achieved by using the basic suffix array enhanced with the lcp-table and an additional table, called the child-table, that requires $4n$ bytes. Our new approach is not confined to exact pattern matching. In general, we can simulate any top-down traversal of the suffix tree by means of the enhanced suffix array. To further exemplify this, we will show how to efficiently compute all shortest unique substrings of S.

In Section 7 we show how to incorporate the concept of suffix links (known from suffix trees) into enhanced suffix arrays. To this end, we further enhance the suffix array with an additional table, called the suffix link table, that stores the left and right boundaries of suffix link intervals. This table can be stored in $8n$ bytes. As a corresponding application we show how to compute matching statistics in $\mathrm{O}(m)$ time for a string of length m, using the enhanced suffix array.

Section 8 presents implementation details that considerably reduce the space requirement. It will be shown that in practice both the lcp-table and the child-table can be stored in n bytes, whereas the suffix link table requires $2n$ bytes. This space reduction entails no loss of performance.

Section 9 presents experimental results that show the practical usefulness of our algorithms.

The last section concludes with a brief summary of the contributions of this article, provides pointers to related work, and outlines an alternative approach to simulate a top down traversal of the suffix tree.

Parts of this article appeared in [1] and [2].

2. Basic notions

Let Σ be a finite ordered *alphabet*. Σ^* is the *set of all strings over* Σ. We use Σ^+ to denote the set $\Sigma^* \setminus \{\varepsilon\}$ of non-empty strings. Let S be a string of length $|S| = n$ over Σ. To simplify analysis, we suppose that the size of the alphabet is a constant, and that $n < 2^{32}$. The latter implies that an integer in the range $[0, n]$ can be stored in 4 bytes. We assume that the special symbol \$ is an element of Σ (which is larger then all other elements) but does not occur in S. $S[i]$ denotes the *character at position* i in S, for $0 \leqslant i < n$. For $i \leqslant j$, $S[i..j]$ denotes the *substring* S starting with the character at position i and ending with the character at position j. The substring $S[i..j]$ is also denoted by the *pair* (i, j) *of positions*.

A *suffix tree* for the string S is a rooted directed tree with exactly $n+1$ leaves numbered 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $S\$$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the string S_i, where $S_i = S[i..n-1]\$$ denotes the ith nonempty suffix of the string $S\$$, $0 \leqslant i \leqslant n$. Fig. 1 shows the suffix tree for the string $S = acaaacatat$.

The *suffix array* suftab of the string S is an array of integers in the range 0 to n, specifying the lexicographic ordering of the $n+1$ suffixes of the string $S\$$. That is, $S_{\mathsf{suftab}[0]}, S_{\mathsf{suftab}[1]}, \ldots, S_{\mathsf{suftab}[n]}$ is the sequence of suffixes of $S\$$ in ascending lexicographic order. The suffix array requires $4n$ bytes.

The *inverse suffix array* suftab^{-1} is a table of size $n+1$ such that $\mathsf{suftab}^{-1}[\mathsf{suftab}[q]] = q$ for any $0 \leqslant q \leqslant n$. suftab^{-1} can be computed in linear time from the suffix array and needs $4n$ bytes.

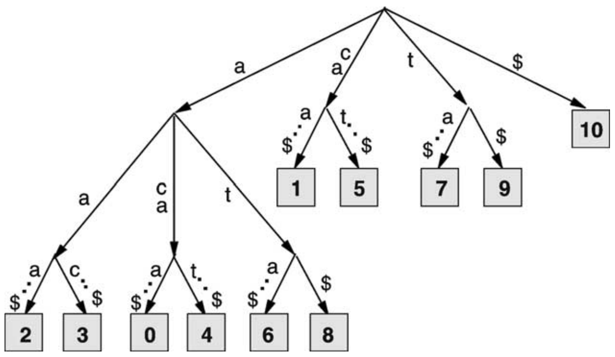

Fig. 1. The suffix tree for $S = acaaacatat$.

The table bwttab contains the *Burrows and Wheeler* transformation [6] known from data compression. It is a table of size $n+1$ such that for every i, $0 \leqslant i \leqslant n$, $\mathsf{bwttab}[i] = S[\mathsf{suftab}[i]-1]$ if $\mathsf{suftab}[i] \neq 0$. $\mathsf{bwttab}[i]$ is undefined if $\mathsf{suftab}[i] = 0$. The table bwttab is stored in n bytes and constructed in one scan over the suffix array in $O(n)$ time.

The *lcp-table* lcptab is an array of integers in the range 0 to n. We define $\mathsf{lcptab}[0] = 0$ and $\mathsf{lcptab}[i]$ is the length of the longest common prefix of $S_{\mathsf{suftab}[i-1]}$ and $S_{\mathsf{suftab}[i]}$, for $1 \leqslant i \leqslant n$. Since $S_{\mathsf{suftab}[n]} = \$$, we always have $\mathsf{lcptab}[n] = 0$. The lcp-table can be computed as a by-product during the construction of the suffix array, or alternatively, in linear time from the suffix array [20]. The lcp-table requires $4n$ bytes in the worst case.

3. Algorithms based on lcp-intervals

3.1. Motivation: repeat analysis and genome comparison

To start with, we will shed some light on the underlying problem. Repeat analysis plays a key role in the study, analysis, and comparison of complete genomes. In the analysis of a single genome, a basic task is to characterize and locate the repetitive elements of the genome. In the comparison of two or more genomes, a basic task is to find similar subsequences of the genomes. This problem can also be reduced to the computation of certain types of repeats of the string that consists of the concatenated genomes; cf. [8,17].

The repetitive elements of the human genome can be generally classified into two large groups: dispersed repetitive DNA and tandemly repeated DNA. Dispersed repetitions vary in size and content and fall into two basic categories: transposable elements and segmental duplications [28]. Transposable elements belong to one of the following four classes: SINEs (short interspersed nuclear elements), LINEs (long interspersed nuclear elements), LTR (long terminal repeats), and transposons. Segmental duplications, which might contain complete genes, have been divided into two classes: chromosome-specific and trans-chromosome duplications [30]. Tandemly repeated DNA can also be classified into two categories: simple sequence repetitions (relatively short k-mers such as micro and minisatellites) and larger ones, which are called blocks of tandemly repeated segments.

While bacterial genomes usually do not contain large parts of redundant DNA, the genomes of higher organisms are often very repetitive. For example, 50% of the 3 billion basepairs of the human genome consist of repeats. Repeats also comprise 11% of the mustard weed genome, 7% of the worm genome and 3% of the fly genome [28]. Clearly, one needs extensive algorithmic support for a systematic study of repetitive DNA on a genomic scale. The algorithms for this task usually use the suffix tree to locate repetitive structures such as maximal or supermaximal repeats; see [15]. In this section we show how to locate all supermaximal repeats, while Section 5.1 treats maximal repeated pairs. Let us recall the definitions of these notions.

A pair of substrings $R = ((i_1, j_1), (i_2, j_2))$ is a *repeated pair* if and only if $(i_1, j_1) \neq (i_2, j_2)$ and $S[i_1..j_1] = S[i_2..j_2]$. The length of R is $j_1 - i_1 + 1$. A repeated pair

$((i_1, j_1), (i_2, j_2))$ is called *left maximal* if $S[i_1 - 1] \neq S[i_2 - 1]$[1] and *right maximal* if $S[j_1 + 1] \neq S[j_2 + 1]$. A repeated pair is called *maximal* if it is left and right maximal. A substring ω of S is a (*maximal*) *repeat* if there is a (maximal) repeated pair $((i_1, j_1), (i_2, j_2))$ such that $\omega = S[i_1..j_1]$. A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat.

3.2. The lcp-intervals

We start this subsection with the introduction of the first essential concept of this article, namely lcp-intervals. Then we will derive two new algorithms that solely exploit the properties of lcp-intervals. The algorithms are much simpler than the corresponding ones based on suffix trees.

Definition 3.1. An interval $[i..j]$, $0 \leqslant i < j \leqslant n$, is an *lcp-interval of lcp-value* ℓ if

1. $\mathsf{lcptab}[i] < \ell$,
2. $\mathsf{lcptab}[k] \geqslant \ell$ for all k with $i + 1 \leqslant k \leqslant j$,
3. $\mathsf{lcptab}[k] = \ell$ for at least one k with $i + 1 \leqslant k \leqslant j$,
4. $\mathsf{lcptab}[j + 1] < \ell$.

We will also use the shorthand ℓ-interval (or even ℓ-$[i..j]$) for an lcp-interval $[i..j]$ of lcp-value ℓ. Every index k, $i + 1 \leqslant k \leqslant j$, with $\mathsf{lcptab}[k] = \ell$ is called ℓ-index. The set of all ℓ-indices of an ℓ-interval $[i..j]$ will be denoted by $\ell Indices(i, j)$. If $[i..j]$ is an ℓ-interval such that $\omega = S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \ell - 1]$ is the longest common prefix of the suffixes $S_{\mathsf{suftab}[i]}, S_{\mathsf{suftab}[i+1]}, \ldots, S_{\mathsf{suftab}[j]}$, then $[i..j]$ is called the ω-interval.

i	suftab	lcptab	bwttab	$S_{\mathsf{suftab}[i]}$
0	2	0	c	$aaacatat\$$
1	3	2	a	$aacatat\$$
2	0	1		$acaaacatat\$$
3	4	3	a	$acatat\$$
4	6	1	c	$atat\$$
5	8	2	t	$at\$$
6	1	0	a	$caaacatat\$$
7	5	2	a	$catat\$$
8	7	0	a	$tat\$$
9	9	1	a	$t\$$
10	10	0	t	$\$$

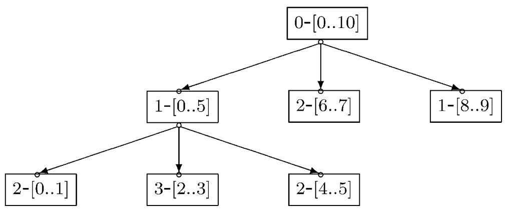

Fig. 2. The enhanced suffix array of the string $S = acaaacatat$ and its lcp-interval tree.

[1] This definition has to be extended to the cases $i_1 = 0$ or $i_2 = 0$, but throughout the paper we do not explicitly state boundary cases like these.

As an example, consider the table on the left side of Fig. 2. [0..5] is a 1-interval because $\mathsf{lcptab}[0] = 0 < 1$, $\mathsf{lcptab}[5+1] = 0 < 1$, $\mathsf{lcptab}[k] \geqslant 1$ for all k with $1 \leqslant k \leqslant 5$, and $\mathsf{lcptab}[2] = 1$. Furthermore, 1-[0..5] is the a-interval and $\ell Indices(0,5) = \{2,4\}$. We shall see later that lcp-intervals correspond to internal nodes of the suffix tree.

3.3. A new algorithm for finding supermaximal repeats

Definition 3.2. An ℓ-interval $[i..j]$ is called a *local maximum* in the lcp-table if $\mathsf{lcptab}[k] = \ell$ for all $i+1 \leqslant k \leqslant j$.

For instance, in the lcp-table of Fig. 2, the local maxima are the intervals [0..1], [2..3], [4..5], [6..7], and [8..9].

Lemma 3.3. *A string ω is a supermaximal repeat if and only if there is an ℓ-interval $[i..j]$ such that*

- *$[i..j]$ is a local maximum in the lcp-table and $[i..j]$ is the ω-interval,*
- *the characters $\mathsf{bwttab}[i], \mathsf{bwttab}[i+1], \ldots, \mathsf{bwttab}[j]$ are pairwise distinct.*

Proof. (If) Since ω is a common prefix of the suffixes $S_{\mathsf{suftab}[i]}, \ldots, S_{\mathsf{suftab}[j]}$ and $i < j$, it is certainly a repeat. The characters $S[\mathsf{suftab}[i]+\ell], S[\mathsf{suftab}[i+1]+\ell], \ldots, S[\mathsf{suftab}[j]+\ell]$ are pairwise distinct because $[i..j]$ is a local maximum in the lcp-table. By the second condition, the characters $\mathsf{bwttab}[i], \mathsf{bwttab}[i+1], \ldots, \mathsf{bwttab}[j]$ are also pairwise distinct. It follows that ω is a maximal repeat and that there is no repeat in S which contains ω. In other words, ω is a supermaximal repeat.

(Only if) Let ω be a supermaximal repeat of length $|\omega| = \ell$. Furthermore, suppose that $\mathsf{suftab}[i], \mathsf{suftab}[i+1], \ldots, \mathsf{suftab}[j]$, $0 \leqslant i < j \leqslant n$, are the consecutive entries in suftab such that ω is a common prefix of $S_{\mathsf{suftab}[i]}, S_{\mathsf{suftab}[i+1]}, \ldots, S_{\mathsf{suftab}[j]}$ but neither of $S_{\mathsf{suftab}[i-1]}$ nor of $S_{\mathsf{suftab}[j+1]}$. Because ω is supermaximal, the characters $S[\mathsf{suftab}[i] + \ell], S[\mathsf{suftab}[i+1] + \ell], \ldots, S[\mathsf{suftab}[j] + \ell]$ are pairwise distinct. Hence $\mathsf{lcptab}[k] = \ell$ for all k with $i+1 \leqslant k \leqslant j$. Furthermore, $\mathsf{lcptab}[i] < \ell$ and $\mathsf{lcptab}[j+1] < \ell$ hold because otherwise ω would also be a prefix of $S_{\mathsf{suftab}[i-1]}$ or $S_{\mathsf{suftab}[j+1]}$. All in all, $[i..j]$ is a local maximum of the array lcptab and $[i..j]$ is the ω-interval. Finally, the characters $\mathsf{bwttab}[i], \mathsf{bwttab}[i+1], \ldots, \mathsf{bwttab}[j]$ are pairwise distinct because ω is supermaximal. $\square$

The preceding lemma does not only imply that the number of supermaximal repeats is smaller than n, but it also suggests a simple linear time algorithm to compute all supermaximal repeats of a string S: Find all local maxima in the lcp-table of S. For every local maximum $[i..j]$ check whether $\mathsf{bwttab}[i], \mathsf{bwttab}[i+1], \ldots, \mathsf{bwttab}[j]$ are pairwise distinct characters. If so, report the string $S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \mathsf{lcptab}[i] - 1]$ as supermaximal repeat. The reader is invited to compare our simple algorithm with the suffix-tree based algorithm of [15, p. 146].

3.4. Computation of maximal unique matches

Next, we tackle a problem that has its origin in genome comparisons. Nowadays, the DNA sequences of entire genomes are being determined at a rapid rate. For example, the genomes of several strains of the bacteria *E. coli* and *S. aureus* have already been completely sequenced. When the genomic DNA sequences of closely related organisms become available, one of the first questions researchers ask is how the genomes align. This alignment may help, for example, in understanding why a strain of a bacterium is pathogenic or resistant to antibiotics while another is not. The software tool *MUMmer* [8] has been developed to efficiently align two sufficiently similar genomic DNA sequences. In the first phase of its underlying algorithm, a maximal unique match (*MUM*) decomposition of two genomes S_1 and S_2 is computed. Using the suffix tree of $S_1\#S_2$, *MUM*s can be computed in $O(n)$ time and space, where $n = |S_1\#S_2|$ and # is a symbol neither occurring in S_1 nor in S_2. However, the space consumption of the suffix tree has been identified to be a major problem when comparing large genomes; see [8]. We will solve this problem by using the suffix array enhanced with the lcp-table.

Definition 3.4. Given two sequences S_1 and S_2, a *MUM* is a sequence that occurs exactly once in S_1 and once in S_2, and is not contained in any longer such sequence.

Lemma 3.5. *Let # be a unique separator symbol not occurring in S_1 and S_2 and let $S = S_1\#S_2$. The string u is a MUM of S_1 and S_2 if and only if u is a supermaximal repeat in S such that*

(1) *there is only one maximal repeated pair $((i_1, j_1), (i_2, j_2))$ with*

$$u = S[i_1..j_1] = S[i_2..j_2],$$

(2) *$j_1 < p < i_2$, where $p = |S_1|$ is the position of # in S.*

Proof. (If) It is a consequence of conditions (1) and (2) that u occurs exactly once in S_1 and once in S_2. Because the repeated pair $((i_1, j_1), (i_2, j_2))$ is maximal, u is a *MUM*.

(Only if) If u is a *MUM* of the sequences S_1 and S_2, then it occurs exactly once in S_1 (say, $u = S_1[i_1..j_1]$) and once in S_2 (say, $u = S_2[i_2..j_2]$), and is not contained in any longer such sequence. Clearly, $((i_1, j_1), (p + 1 + i_2, p + 1 + j_2))$ is a repeated pair in $S = S_1\$S_2$, where $p = |S_1|$. Because u occurs exactly once in S_1 and once in S_2, and is not contained in any longer such sequence, it follows that u is a supermaximal repeat in S satisfying conditions (1) and (2). □

The first version of *MUMmer* [8] computed *MUM*s in $O(|S|)$ time and space with the help of the suffix tree of $S = S_1\#S_2$. Using an enhanced suffix array, this task can be done more time and space economically as follows: Find all local maxima in the lcp-table of $S = S_1\#S_2$. For every local maximum $[i..j]$ check whether $i + 1 = j$, $\mathsf{bwttab}[i] \neq \mathsf{bwttab}[j]$, and $\mathsf{suftab}[i] < p < \mathsf{suftab}[j]$. If so, report $S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \mathsf{lcptab}[i] - 1]$ as *MUM*. This simple algorithm was found independently by Hon and Sadakane [18] and the authors of this article [1]. In Section 9, we compare the performance of *MUMmer* with the implementation of the preceding algorithm.

Recently, Delcher et al. [9] presented a new version of *MUMmer*, called *MUMmer* 2. It constructs the suffix tree of S_1 and computes matches by streaming S_2 against it. A similar, but more space efficient algorithm can be implemented based on the enhanced suffix array of S_1. See [26] for details of this algorithm and for an experimental comparison with *MUMmer* 2.

The algorithms to compute supermaximal repeats and *MUM*s require tables suftab, lcptab, and bwttab, but do not access the input sequence. More precisely, instead of the input string, we use table bwttab without increasing the total space requirement. This is because the tables suftab, lcptab, and bwttab can be accessed in sequential order, thus leading to an improved cache coherence and in turn considerably reduced running time; see Section 9. The same technique is applied in the computation of maximal repeated pairs in Section 5.1.

4. The lcp-interval tree of a suffix array

Kasai et al. [20] presented a linear time algorithm to simulate the bottom-up traversal of a suffix tree with a suffix array and its lcp-information. The following algorithm is a slight modification of their algorithm TraverseWithArray. It computes all lcp-intervals of the lcp-table with the help of a stack. The elements on the stack are lcp-intervals represented by tuples $\langle lcp, lb, rb\rangle$, where *lcp* is the lcp-value of the interval, *lb* is its left boundary, and *rb* is its right boundary. In Algorithm 4.1, *push* (pushes an element onto the stack) and *pop* (pops an element from the stack and returns that element) are the usual stack operations, while *top* provides a pointer to the topmost element of the stack. Furthermore, $\perp$ stands for an undefined value.

Algorithm 4.1 (Computation of lcp-intervals (adapted from Kasai et al. [20])).

```
$push(\langle 0, 0, \perp\rangle)$
for $i := 1$ to $n$ do
    $lb := i - 1$
    while $\mathsf{lcptab}[i] < top.lcp$
        $top.rb := i - 1$
        $interval := pop$
        $report(interval)$
        $lb := interval.lb$
    if $\mathsf{lcptab}[i] > top.lcp$ then
        $push(\langle \mathsf{lcptab}[i], lb, \perp\rangle)$
```

Here, we will take the approach of Kasai et al. [20] one step further and introduce the second essential concept of this article—the lcp-interval tree.

Definition 4.2. An m-interval $[l..r]$ is said to be *embedded* in an ℓ-interval $[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leqslant l < r \leqslant j$) and $m > \ell$.[2] The ℓ-interval $[i..j]$ is then called the interval *enclosing* $[l..r]$. If $[i..j]$ encloses $[l..r]$ and there is no interval embedded in $[i..j]$ that also encloses $[l..r]$, then $[l..r]$ is called a *child interval* of $[i..j]$.

This parent-child relationship constitutes a conceptual (or virtual) tree which we call the *lcp-interval tree* of the suffix array. The root of this tree is the 0-interval $[0..n]$; see Fig. 2. The lcp-interval tree is basically the suffix tree without leaves (more precisely, there is a one-to-one correspondence between the nodes of the lcp-interval tree and the internal nodes of the suffix tree). These leaves are left implicit in our framework, but every leaf in the suffix tree, which corresponds to the suffix $S_{\mathsf{suftab}[l]}$, can be represented by a *singleton interval* $[l..l]$. The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ with $l \in [i..j]$. For instance, continuing the example of Fig. 2, the child intervals of [0..5] are [0..1], [2..3], and [4..5]. The next theorem shows how the parent-child relationship of the lcp-intervals can be determined from the stack operations in Algorithm 4.1.

Theorem 4.3. *Consider the for-loop of Algorithm* 4.1 *for some index i. Let top be the topmost interval on the stack and top_{-1} be the interval next to it* (*note that $top_{-1}.lcp < top.lcp$*). *If* $\mathsf{lcptab}[i] < top.lcp$*, then before top will be popped from the stack in the while-loop, the following holds*:

(1) *If* $\mathsf{lcptab}[i] \leqslant top_{-1}.lcp$*, then top is the child interval of top_{-1}.*
(2) *If* $top_{-1}.lcp < \mathsf{lcptab}[i] < top.lcp$*, then top is the child interval of the* $\mathsf{lcptab}[i]$*-interval that contains i.*

Proof. We will show (1). The other case follows similarly. First, we show that *top* is embedded in top_{-1}. The following invariant is maintained in the for-loop of Algorithm 4.1: if $\langle \ell_1, lb_1, rb_1\rangle, \ldots, \langle \ell_k, lb_k, rb_k\rangle$ are the intervals on the stack, where $top = \langle \ell_k, lb_k, rb_k\rangle$ then $lb_i \leqslant lb_j$ and $\ell_i < \ell_j$ for all $1 \leqslant i < j \leqslant k$. Furthermore, because $\langle \ell_j, lb_j, rb_j\rangle$ will be popped from the stack before $\langle \ell_i, lb_i, rb_i\rangle$, it follows that $rb_j \leqslant rb_i$. Thus, the ℓ_j-interval $[lb_j..rb_j]$ is embedded in the ℓ_i-interval $[lb_i..rb_i]$. In particular, *top* is embedded in top_{-1}.

If *top* was not the child interval of top_{-1}, then there would be an lcp-interval $\langle lcp', lb', rb'\rangle$ such that *top* is embedded in $\langle lcp', lb', rb'\rangle$ and $\langle lcp', lb', rb'\rangle$ is embedded in top_{-1}. This, however, can only happen if $\langle lcp', lb', rb'\rangle$ is an interval on the stack that is above top_{-1}. This contradiction proves the claim. □

An important consequence of Theorem 4.3 is the correctness of Algorithm 4.4. There, the lcp-interval tree is traversed in a bottom-up fashion by a linear scan of the lcp-table, while storing needed information on a stack. We stress that the lcp-interval tree is not really build: whenever an ℓ-interval is processed by the generic function *process*, only its child intervals have to be known. These are determined solely from the lcp-

[2] Note that we cannot have both $i = l$ and $r = j$ because $m > \ell$.

information, i.e., there are no explicit parent-child pointers in our framework. In contrast to Algorithm 4.1, Algorithm 4.4 computes all lcp-intervals of the lcp-table *with* the child information. Here, the elements on the stack are lcp-intervals represented by quadruples $\langle lcp, lb, rb, childList\rangle$, where *lcp* is the lcp-value of the interval, *lb* is its left boundary, *rb* is its right boundary, and *childList* is a list of its child intervals. Furthermore, $add([c_1, \ldots, c_k], c)$ appends the element c to the list $[c_1, \ldots, c_k]$ and returns the result.

Algorithm 4.4 (Traverse and process the lcp-interval tree).

```
$lastInterval := \bot$
$push(\langle 0, 0, \bot, [\,]\rangle)$
for $i := 1$ to $n$ do
  $lb := i - 1$
  while $\mathsf{lcptab}[i] < top.lcp$
    $top.rb := i - 1$
    $lastInterval := pop$
    $process(lastInterval)$
    $lb := lastInterval.lb$
    if $\mathsf{lcptab}[i] \leqslant top.lcp$ then
      $top.childList := add(top.childList, lastInterval)$
      $lastInterval := \bot$
  if $\mathsf{lcptab}[i] > top.lcp$ then
    if $lastInterval \neq \bot$ then
      $push(\langle \mathsf{lcptab}[i], lb, \bot, [lastInterval]\rangle)$
      $lastInterval := \bot$
    else $push(\langle \mathsf{lcptab}[i], lb, \bot, [\,]\rangle)$
```

In Section 5, we will show how to solve several problems merely by specifying the function *process* called in line 8 of Algorithm 4.4.

5. Bottom-up traversals

In this section, we show how to efficiently solve all problems with enhanced suffix arrays that are usually solved by a bottom-up traversal of the suffix tree. As examples, we show how to compute all maximal repeated pairs and the Ziv–Lempel decomposition of a string.

5.1. An efficient implementation of an optimal algorithm for finding maximal repeated pairs

The computation of maximal repeated pairs plays an important role in the analysis of a genome. The algorithm of Gusfield [15, p. 147] computes maximal repeated pairs of a sequence S of length n in $O(|\Sigma|n + z)$ time, where z is the number of maximal repeated pairs. This running time is optimal. To the best of our knowledge, Gusfield's algorithm was first implemented in the *REPuter*-program [27], based on space efficient suffix trees described in [25]. The software tool *REPuter* uses maximal repeated pairs as seeds for finding degenerate (or approximate) repeats. In this section, we show how to implement Gusfield's algorithm using enhanced suffix arrays. This considerably reduces the space requirements, thus removing a bottle neck in the algorithm. As a consequence, much larger genomes can be searched for repetitive elements. As in the algorithms in Section 3.3, the implementation requires tables suftab, lcptab, and bwttab, but does not access the input sequence. The accesses to the three tables are in sequential order, thus leading to an improved cache coherence and in turn to a considerably reduced running time; this is verified in Section 9.

We begin by introducing some notation: Let $\perp$ stand for the undefined character. We assume that it is different from all characters in Σ. Let $[i..j]$ be an ℓ-interval and $u = S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \ell - 1]$. Define $\mathcal{P}_{[i..j]}$ to be the set of positions p such that u is a prefix of S_p, i.e., $\mathcal{P}_{[i..j]} = \{\mathsf{suftab}[r] \mid i \leqslant r \leqslant j\}$. We divide $\mathcal{P}_{[i..j]}$ into disjoint and possibly empty sets according to the characters to the left of each position: For any $a \in \Sigma \cup \{\perp\}$ define

$$\mathcal{P}_{[i..j]}(a) = \begin{cases} \{0 \mid 0 \in \mathcal{P}_{[i..j]}\} & \text{if } a = \perp, \\ \{p \mid p \in \mathcal{P}_{[i..j]},\ p > 0, \text{ and } S[p-1] = a\} & \text{otherwise.} \end{cases}$$

The algorithm computes position sets in a bottom-up strategy. In terms of an lcp-interval tree, this means that the lcp-interval $[i..j]$ is processed only after all child intervals of $[i..j]$ have been processed.

Suppose $[i..j]$ is a singleton interval, i.e., $i = j$. Let $p = \mathsf{suftab}[i]$. Then $\mathcal{P}_{[i..j]} = \{p\}$ and

$$\mathcal{P}_{[i..j]}(a) = \begin{cases} \{p\} & \text{if } p > 0 \text{ and } S[p-1] = a \text{ or } p = 0 \text{ and } a = \perp, \\ \emptyset & \text{otherwise.} \end{cases}$$

Now suppose that $i < j$. For each $a \in \Sigma \cup \{\perp\}$, $\mathcal{P}_{[i..j]}(a)$ is computed step by step while processing the child intervals of $[i..j]$. These are processed from left to right. Suppose that they are numbered, and that we have already processed q child intervals of $[i..j]$. By $\mathcal{P}^q_{[i..j]}(a)$ we denote the subset of $\mathcal{P}_{[i..j]}(a)$ obtained after processing the qth child interval of $[i..j]$. Let $[i'..j']$ be the $(q+1)$th child interval of $[i..j]$. Due to the bottom-up strategy, $[i'..j']$ has been processed and hence the position sets $\mathcal{P}_{[i'..j']}(b)$ are available for any $b \in \Sigma \cup \{\perp\}$.

The interval $[i'..j']$ is processed in the following way: First, maximal repeated pairs are output by combining the position set $\mathcal{P}^q_{[i..j]}(a)$, $a \in \Sigma \cup \{\perp\}$, with position sets $\mathcal{P}_{[i'..j']}(b)$, $b \in \Sigma \cup \{\perp\}$. In particular, $((p, p+\ell-1), (p', p'+\ell-1))$, $p < p'$, are output for all $p \in \mathcal{P}^q_{[i..j]}(a)$ and $p' \in \mathcal{P}_{[i'..j']}(b)$, $a, b \in \Sigma \cup \{\perp\}$ and $a \neq b$.

It is clear that u occurs at position p and p'. Hence $((p, p+\ell-1), (p', p'+\ell-1))$ is a repeated pair. By construction, only those positions p and p' are combined for which the characters immediately to the left, i.e., at positions $p-1$ and $p'-1$ (if they exist), are different. This guarantees left-maximality of the output repeated pairs.

The position sets $\mathcal{P}^q_{[i..j]}(a)$ were inherited from child intervals of $[i..j]$ that are different from $[i'..j']$. Hence the characters immediately to the right of u at positions $p+\ell$ and $p'+\ell$ (if they exist) are different. As a consequence, the output repeated pairs are maximal.

Once the maximal repeated pairs for the current child interval $[i'..j']$ have been output, the union $\mathcal{P}^{q+1}_{[i..j]}(e) := \mathcal{P}^{q}_{[i..j]}(e) \cup \mathcal{P}_{[i'..j']}(e)$ is computed for all $e \in \Sigma \cup \{\perp\}$. That is, the position sets are inherited from $[i'..j']$ to $[i..j]$.

In Algorithm 4.4, if the function *process* is applied to an lcp-interval, then all its child intervals are available. Hence the maximal repeated pair algorithm can be implemented by a bottom-up traversal of the lcp-interval tree. To this end, the function *process* in Algorithm 4.4 outputs maximal repeated pairs and further maintains position sets on the stack (which are added as a fifth component to the quadruples). The bottom-up traversal requires $O(n)$ time.

There are two operations performed when processing an lcp-interval $[i..j]$. Output of maximal repeated pairs by combining position sets and union of position sets. Each combination of position sets means to compute their Cartesian product. This delivers a list of position pairs, i.e., maximal repeated pairs. Each repeated pair is computed in constant time from the position lists. Altogether, the combinations can be computed in $O(z)$ time, where z is the number of repeats. The union operation for the position sets can be implemented in constant time, if we use linked lists. For each lcp-interval, we have $O(|\Sigma|)$ union operations. Since $O(n)$ lcp-intervals have to be processed, the union and add operations require $O(|\Sigma|n)$ time. Altogether, the algorithm runs in $O(|\Sigma|n + z)$ time.

Next, we analyze the space consumption of the algorithm. A position set $\mathcal{P}_{[i..j]}(a)$ is the union of position sets of the child intervals of $[i..j]$. If the child intervals of $[i..j]$ have been processed, the corresponding position sets are obsolete. Hence it is not required to copy position sets. Moreover, we only have to store the position sets for those lcp-intervals which are on the stack used for the bottom-up traversal of the lcp-interval tree. So it is natural to store references to the position sets on the stack together with other information about the lcp-interval. Thus the space required for the position sets is determined by the maximal size of the stack. Since this is $O(n)$, the space requirement is $O(|\Sigma|n)$. In practice, however, the stack size is much smaller. Altogether the algorithm is optimal, since its space and time requirement is linear in the size of the input plus the output.

5.2. Computing the Ziv–Lempel decomposition

As a second application of the bottom-up traversal of the lcp-interval tree, we will very briefly describe how to compute the Ziv–Lempel decomposition [33,34] of a string. The Ziv–Lempel decomposition plays an important role in data compression, and recently it was used in linear time algorithms for the detection of all tandem repeats of a string [16, 24].

For each position i of S, let l_i denote the length of the longest prefix of $S[i..n]$ that also occurs as a substring of S starting at some position $j < i$. Let s_i denote the starting position of the leftmost occurrence of this substring in S if $l_i > 0$, and $s_i = 0$, otherwise; see Fig. 3.

The Ziv–Lempel decomposition of S is the list of indices $i_1, i_2, \ldots, i_k$, defined inductively by $i_1 = 0$ and $i_{B+1} = i_B + \max\{1, l_{i_B}\}$ for $B \geqslant 1$ and $i_B \leqslant n$. The substring $S[i_B..i_{B+1} - 1]$, $1 \leqslant B \leqslant k$, obtained in this way is called the Bth block of the Ziv–Lempel decomposition of S.

The Ziv–Lempel decomposition of a string S can also be computed *off-line* in linear time by a bottom-up traversal of the lcp-interval tree; see Algorithm 4.4. To this end, we add

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	2	0	1	0	0	6	7	0
l_i	0	0	1	2	3	2	1	0	2	1	0

B	1	2	3	4	5	6	7	8
i_B	0	1	2	3	5	7	8	10
B-th block	a	c	a	aa	ca	t	at	\$

Fig. 3. The values of s_i and l_i (left) and the Ziv–Lempel decomposition (right).

another value *min* of type integer to the quadruples stored on the stack. This value is initially set to $\perp$ and will be updated by the *process* function. At any stage, when the function *process* is applied to an ℓ-interval $[i..j]$, all its child intervals are known and have already been processed (note that $[i..j] \neq [0..n]$ must hold). Let $[l_1..r_1]$, $[l_2..r_2], \ldots, [l_k..r_k]$ be the k child intervals of $[i..j]$, stored in its *childList*. Let $min_1, \ldots, min_k$ be the respective *min*-values of the child intervals. Let

$$M = \{min_1, \ldots, min_k\} \cup \bigl\{\mathsf{suftab}[q] \mid q \in [i..j] \text{ and } q \notin [l_p..r_p] \text{ for all } 1 \leqslant p \leqslant k\bigr\}.$$

Compute $min := \min M$ and assign for all $q \in M$ with $q \neq min$: $s_q := min$ and $l_q := \ell$. Finally, for the root $[0..n]$ of the *lcp*-interval tree, we assign for all $q \in M$: $s_q := 0$ and $l_q := 0$.

6. Top-down traversals

Based on the analogy between the lcp-interval tree and the suffix tree, it is desirable to enhance the suffix array with additional information to determine, for any ℓ-interval $[i..j]$, all its child intervals in constant time. We achieve this goal by enhancing the suffix array with the lcp-table and an additional table: the child-table childtab; see Fig. 4. The child-table is a table of size $n + 1$ indexed from 0 to n and each entry contains three values: *up*, *down*, and *nextℓIndex*. Each of these three values requires 4 bytes in the worst case. We

			childtab			
i	suftab	lcptab	1.	2.	3.	$S_{\mathsf{suftab}[i]}$
0	2	0		②	6	aaacatat\$
1	3	2			↗	aacatat\$
2	0	1	1	③	4	acaaacatat\$
3	4	3			↗	acatat\$
4	6	1	3	5	→	atat\$
5	8	2			↗	at\$
6	1	0	2	⑦	8	caaacatat\$
7	5	2			↗	catat\$
8	7	0	7	⑨	10	tat\$
9	9	1			↗	t\$
10	10	0	9			\$

Fig. 4. Suffix array of the string $S = acaaacatat$ enhanced with the lcptab and childtab. The fields 1, 2, and 3 of the childtab denote the *up*, *down*, and *nextℓIndex* field. The encircled entries are redundant because they also occur in the *up* field. The arcs point to the field where the *up*-value is stored.

shall see later that it is possible to store the same information in only one field. Formally, the values of each childtab-entry are defined as follows (we assume that $\min\emptyset = \max\emptyset = \bot$):

$$\begin{aligned}
&\mathsf{childtab}[i].up = \min\bigl\{q \in [0..i-1] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i] \text{ and}\\
&\qquad\qquad \forall k \in [q+1..i-1] : \mathsf{lcptab}[k] \geqslant \mathsf{lcptab}[q]\bigr\},\\
&\mathsf{childtab}[i].down = \max\bigl\{q \in [i+1..n] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i] \text{ and}\\
&\qquad\qquad \forall k \in [i+1..q-1] : \mathsf{lcptab}[k] > \mathsf{lcptab}[q]\bigr\},\\
&\mathsf{childtab}[i].next\ell Index\\
&\quad = \min\bigl\{q \in [i+1..n] \mid \mathsf{lcptab}[q] = \mathsf{lcptab}[i] \text{ and}\\
&\qquad\qquad \forall k \in [i+1..q-1] : \mathsf{lcptab}[k] > \mathsf{lcptab}[i]\bigr\}.
\end{aligned}$$

In essence, the child-table stores the parent-child relationship of lcp-intervals. Roughly speaking, for an ℓ-interval $[i..j]$ whose ℓ-indices are $i_1 < i_2 < \cdots < i_k$, the $\mathsf{childtab}[i].down$ or $\mathsf{childtab}[j+1].up$ value is used to determine the first ℓ-index i_1. The other ℓ-indices $i_2, \ldots, i_k$ can be obtained from $\mathsf{childtab}[i_1].next\ell Index, \ldots, \mathsf{childtab}[i_{k-1}].next\ell Index$, respectively. Once these ℓ-indices are known, one can determine all the child intervals of $[i..j]$ according to the following lemma.

Lemma 6.1. *Let $[i..j]$ be an ℓ-interval. If $i_1 < i_2 < \cdots < i_k$ are the ℓ-indices in ascending order, then the child intervals of $[i..j]$ are $[i..i_1-1]$, $[i_1..i_2-1], \ldots, [i_k..j]$ (note that some of them may be singleton intervals).*

Proof. Let $[l..r]$ be one of the intervals $[i..i_1-1]$, $[i_1..i_2-1], \ldots, [i_k..j]$. If $[l..r]$ is a singleton interval, then it is a child interval of $[i..j]$. Suppose that $[l..r]$ is an m-interval. Since $[l..r]$ does not contain an ℓ-index, it follows that $[l..r]$ is embedded in $[i..j]$. Because

$$\mathsf{lcptab}[i_1] = \mathsf{lcptab}[i_2] = \cdots = \mathsf{lcptab}[i_k] = \ell,$$

there is no interval embedded in $[i..j]$ that encloses $[l..r]$. That is, $[l..r]$ is a child interval of $[i..j]$. Finally, it is not difficult to see that $[i..i_1-1]$, $[i_1..i_2-1], \ldots, [i_k..j]$ are all the child intervals of $[i..j]$, i.e., there cannot be any other child interval. □

As an example, consider the enhanced suffix array in Fig. 4. The 1-[0..5] interval has the 1-indices 2 and 4. The first 1-index 2 is stored in $\mathsf{childtab}[0].down$ and $\mathsf{childtab}[6].up$. The second 1-index is stored in $\mathsf{childtab}[2].next\ell Index$. Thus, the child intervals of [0..5] are [0..1], [2..3], and [4..5]. In Section 6.2, it will be shown in detail how the child-table can be used to determine the child intervals of an lcp-interval in constant time.

6.1. Construction of the child-table

The child-table can be computed in linear time by a bottom-up traversal of the lcp-interval tree as in Algorithm 4.4. For clarity of presentation, however, we introduce *two* algorithms to separately construct the *up*/*down* values and the *nextℓIndex* value of the child-table. Similar to Algorithm 4.4, Algorithm 6.2 scans the lcp-table in linear order and pushes the current index on the stack if its lcp-value is greater than or equal to the lcp-value of *top*. Otherwise, elements of the stack are popped as long as their lcp-value is greater than that of the current index. Based on a comparison of the lcp-values of *top* and the current index, the *up* and *down* fields of the child-table are filled with elements that are popped from the stack during the scan.

Algorithm 6.2 (Construction of the *up* and *down* values).

```
$lastIndex := -1$
$push(0)$
**for** $i := 1$ **to** $n$ **do**
    **while** $\mathsf{lcptab}[i] < \mathsf{lcptab}[top]$
        $lastIndex := pop$
        **if** $(\mathsf{lcptab}[i] \leqslant \mathsf{lcptab}[top])$ **and** $(\mathsf{lcptab}[top] \neq \mathsf{lcptab}[lastIndex])$ **then**
            $\mathsf{childtab}[top].down := lastIndex$
    /* now $\mathsf{lcptab}[i] \geqslant \mathsf{lcptab}[top]$ holds */
    **if** $lastIndex \neq -1$ **then**
        $\mathsf{childtab}[i].up := lastIndex$
        $lastIndex := -1$
    $push(i)$
```

For a correctness proof, we need the following lemma.

Lemma 6.3. *The following invariants are maintained in the for-loop of Algorithm* 6.2*: If $i_1, \ldots, i_p$ are the indices on the stack* (*where i_p is the topmost element*), *then $i_1 < \cdots < i_p$ and $\mathsf{lcptab}[i_1] \leqslant \cdots \leqslant \mathsf{lcptab}[i_p]$. Furthermore, if $\mathsf{lcptab}[i_j] < \mathsf{lcptab}[i_{j+1}]$, then for all k with $i_j < k < i_{j+1}$ we have $\mathsf{lcptab}[k] > \mathsf{lcptab}[i_{j+1}]$.*

Proof. The lemma holds before the for-loop is executed for the first time. By induction, we assume that the lemma holds after the for-loop was executed m times, where $m < n$. Consider the $(m+1)$th execution of the for-loop. Suppose there is an index q with $1 \leqslant q < p$ such that $\mathsf{lcptab}[i_1] \leqslant \cdots \leqslant \mathsf{lcptab}[i_q] \leqslant \mathsf{lcptab}[m+1] < \mathsf{lcptab}[i_{q+1}] \leqslant \cdots \leqslant \mathsf{lcptab}[i_p]$. (The cases, where $\mathsf{lcptab}[m+1] < \mathsf{lcptab}[i_1]$ or $\mathsf{lcptab}[i_p] < \mathsf{lcptab}[m+1]$ are proven similarly.) In the while-loop, $i_{q+1}, \ldots, i_p$ are popped from the stack and in the if-statement immediately after the while-loop, $m+1$ is pushed onto the stack. That is, after the $(m+1)$th execution of the for-loop, $i_1, \ldots, i_q, m+1$ are on the stack with $m+1$ being the topmost element. Clearly, $i_1 < \cdots < i_q < m+1$ and $\mathsf{lcptab}[i_1] \leqslant \cdots \leqslant \mathsf{lcptab}[i_q] \leqslant \mathsf{lcptab}[m+1]$. Suppose that $\mathsf{lcptab}[i_q] < \mathsf{lcptab}[m+1]$. By the inductive hypothesis, for every $j \in \{1, \ldots, p\}$ with $\mathsf{lcptab}[i_j] < \mathsf{lcptab}[i_{j+1}]$, we have $\mathsf{lcptab}[k] > \mathsf{lcptab}[i_{j+1}]$ for all k with $i_j < k < i_{j+1}$. It is not difficult to see that $\mathsf{lcptab}[k] > \mathsf{lcptab}[m+1]$ for all k with $i_q < k < m+1$ is a consequence, and hence the lemma follows. □

Theorem 6.4. *Algorithm* 6.2 *correctly fills the up and down fields of the child-table.*

Proof. If the $\mathsf{childtab}[top].down := lastIndex$ statement is executed, then we have $\mathsf{lcptab}[i] \leqslant \mathsf{lcptab}[top] < \mathsf{lcptab}[lastIndex]$ and $top < lastIndex < i$. Recall that $\mathsf{childtab}[top].down$

is the maximum of the set $M = \{q \in [top + 1..n] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[top]$ and $\forall k \in [top + 1..q - 1] : \mathsf{lcptab}[k] > \mathsf{lcptab}[q]\}$. Clearly, $lastIndex \in [top + 1..n]$ and $\mathsf{lcptab}[lastIndex] > \mathsf{lcptab}[top]$. Furthermore, according to Lemma 6.3, for all k with $top < k < lastIndex$ we have $\mathsf{lcptab}[k] > \mathsf{lcptab}[lastIndex]$. In other words, *lastIndex* is an element of M. Suppose that *lastIndex* is not the maximum of M. Then there is an element q' in M with $lastIndex < q' < i$. According to the definition of M, it follows that $\mathsf{lcptab}[lastIndex] > \mathsf{lcptab}[q']$. This, however, implies that *lastIndex* must have been popped from the stack when index q' was considered. This contradiction shows that *lastIndex* is the maximum of M.

If the $\mathsf{childtab}[i].up := lastIndex$ statement is executed, then $\mathsf{lcptab}[top] \leqslant \mathsf{lcptab}[i] < \mathsf{lcptab}[lastIndex]$ and $top < lastIndex < i$. Recall that $\mathsf{childtab}[i].up$ is the minimum of the set $M' = \{q \in [0..i - 1] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i]$ and $\forall k \in [q + 1..i - 1] : \mathsf{lcptab}[k] \geqslant \mathsf{lcptab}[q]\}$. Clearly, we have $lastIndex \in [0..i - 1]$ and $\mathsf{lcptab}[lastIndex] > \mathsf{lcptab}[i]$. Moreover, for all k with $lastIndex < k < i$ we have $\mathsf{lcptab}[k] \geqslant \mathsf{lcptab}[lastIndex]$ because otherwise *lastIndex* would have been popped earlier from the stack. In other words, $lastIndex \in M'$. Suppose that *lastIndex* is not the minimum of M'. Then there is a $q' \in M'$ with $top < q' < lastIndex$. According to the definition of M', it follows that $\mathsf{lcptab}[lastIndex] \geqslant \mathsf{lcptab}[q'] > \mathsf{lcptab}[i] \geqslant \mathsf{lcptab}[top]$. Hence, index q' must be an element between *top* and *lastIndex* on the stack. This contradiction shows that *lastIndex* is the minimum of M'. □

The construction of the *nextℓIndex* field is easier. One merely has to check whether $\mathsf{lcptab}[i] = \mathsf{lcptab}[top]$ holds true. If so, then i is assigned to the field $\mathsf{childtab}[top].next\ell Index$. It is not difficult to see that Algorithms 6.2 and 6.5 construct the child-table in linear time and space.

Algorithm 6.5 (Construction of the *nextℓIndex* value).

```
push(0)
for i := 1 to n do
    while lcptab[i] < lcptab[top]
        pop
    if lcptab[i] = lcptab[top] then
        lastIndex := pop
        childtab[lastIndex].nextℓIndex := i
    push(i)
```

To reduce the space requirement of the child-table, only one field is used in practice. The *down* field is needed only if it does not contain the same information as the *up* field. Fortunately, for an ℓ-interval, only one *down* field is required because an ℓ-interval $[i..j]$ with k ℓ-indices has at most $k + 1$ child intervals. Suppose $[l_1..r_1]$, $[l_2..r_2], \ldots, [l_k..r_k], [l_{k+1}..r_{k+1}]$ are the $k + 1$ child intervals of $[i..j]$, where $[l_q..r_q]$ is an ℓ_q-interval and i_q denotes its first ℓ_q-index for any $1 \leqslant q \leqslant k + 1$. In the *up* field of $\mathsf{childtab}[r_1 + 1], \mathsf{childtab}[r_2 + 1], \ldots, \mathsf{childtab}[r_k + 1]$ we store the indices $i_1, i_2, \ldots, i_k$, respectively. Thus, only the remaining index i_{k+1} must be stored in the *down* field of $\mathsf{childtab}[r_k + 1]$. This value can be stored in $\mathsf{childtab}[r_k + 1].next\ell Index$ because $r_k + 1$ is the last ℓ-index and hence $\mathsf{childtab}[r_k + 1].\mathit{next\ell Index}$ is empty; see Fig. 4. However, if we do this, then for a given index i we must be able to decide whether $\mathsf{childtab}[i].\mathit{next\ell Index}$ contains the next ℓ-index or the $\mathsf{childtab}[i].\mathit{down}$ value. This can be accomplished as follows. $\mathsf{childtab}[i].\mathit{next\ell Index}$ contains the next ℓ-index if $\mathsf{lcptab}[\mathsf{childtab}[i].\mathit{next\ell Index}] = \mathsf{lcptab}[i]$, whereas it stores the $\mathsf{childtab}[i].\mathit{down}$ value if $\mathsf{lcptab}[\mathsf{childtab}[i].\mathit{next\ell Index}] > \mathsf{lcptab}[i]$. This follows directly from the definition of the *nextℓIndex* and *down* field, respectively. Moreover, the memory cells of $\mathsf{childtab}[i].\mathit{next\ell Index}$, which are still unused, can store the values of the *up* field. To see this, note that $\mathsf{childtab}[i + 1].\mathit{up} \neq \perp$ if and only if $\mathsf{lcptab}[i] > \mathsf{lcptab}[i + 1]$. In this case, we have $\mathsf{childtab}[i].\mathit{next\ell Index} = \perp$ and $\mathsf{childtab}[i].\mathit{down} = \perp$. In other words, $\mathsf{childtab}[i].\mathit{next\ell Index}$ is empty and can store the value $\mathsf{childtab}[i + 1].\mathit{up}$; see Fig. 4. Finally, for a given index i, one can decide whether $\mathsf{childtab}[i].\mathit{next\ell Index}$ contains the value $\mathsf{childtab}[i + 1].\mathit{up}$ by testing whether $\mathsf{lcptab}[i] > \mathsf{lcptab}[i + 1]$. To sum up, although the child-table theoretically uses three fields, only space for one field is actually required.

6.2. Determining child intervals in constant time

Given the child-table, the first step to locate the child intervals of an ℓ-interval $[i..j]$ in constant time is to find the first ℓ-index in $[i..j]$, i.e., the minimum of the set $\ell\mathit{Indices}(i, j)$. This is possible with the help of the *up* and *down* fields of the child-table:

Lemma 6.6. *For every ℓ-interval $[i..j]$ the following statements hold*:

(1) $i < \mathsf{childtab}[j + 1].\mathit{up} \leqslant j$ *or* $i < \mathsf{childtab}[i].\mathit{down} \leqslant j$.
(2) $\mathsf{childtab}[j + 1].\mathit{up}$ *stores the first ℓ-index in $[i..j]$ if $i < \mathsf{childtab}[j + 1].\mathit{up} \leqslant j$.*
(3) $\mathsf{childtab}[i].\mathit{down}$ *stores the first ℓ-index in $[i..j]$ if $i < \mathsf{childtab}[i].\mathit{down} \leqslant j$.*

Proof. (1) First, consider index $j + 1$. Suppose $\mathsf{lcptab}[j + 1] = \ell'$ and let I' be the corresponding ℓ'-interval. If $[i..j]$ is a child interval of I', then $\mathsf{lcptab}[i] = \ell'$ and there is no ℓ-index in $[i + 1..j]$. Therefore, $\mathsf{childtab}[j + 1].\mathit{up} = \min \ell\mathit{Indices}(i, j)$, and consequently $i < \mathsf{childtab}[j + 1].\mathit{up} \leqslant j$. If $[i..j]$ is not a child interval of I', then we consider index i. Suppose $\mathsf{lcptab}[i] = \ell''$ and let I'' be the corresponding ℓ''-interval. Because $\mathsf{lcptab}[j + 1] = \ell' < \ell'' < \ell$, it follows that $[i..j]$ is a child interval of I''. We conclude that $\mathsf{childtab}[i].\mathit{down} = \min \ell\mathit{Indices}(i, j)$. Hence, $i < \mathsf{childtab}[i].\mathit{down} \leqslant j$.

(2) If $i < \mathsf{childtab}[j + 1].\mathit{up} \leqslant j$, then the claim follows from

$$\begin{aligned}\mathsf{childtab}[j + 1].\mathit{up} &= \min\bigl\{q \in [i + 1..j] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[j + 1],\\ &\qquad \mathsf{lcptab}[k] \geqslant \mathsf{lcptab}[q]\ \forall k \in [q + 1..j]\bigr\}\\ &= \min\bigl\{q \in [i + 1..j] \mid \mathsf{lcptab}[k] \geqslant \mathsf{lcptab}[q]\ \forall k \in [q + 1..j]\bigr\}\\ &= \min \ell\mathit{Indices}(i, j).\end{aligned}$$

(3) Let i_1 be the first ℓ-index of $[i..j]$. Then $\mathsf{lcptab}[i_1] = \ell > \mathsf{lcptab}[i]$ and for all $k \in [i + 1..i_1 - 1]$ the inequality $\mathsf{lcptab}[k] > \ell = \mathsf{lcptab}[i_1]$ holds. Moreover, for any other index $q \in [i + 1..j]$, we have $\mathsf{lcptab}[q] \geqslant \ell > \mathsf{lcptab}[i]$ but *not* $\mathsf{lcptab}[i_1] > \mathsf{lcptab}[q]$. □

Once the first ℓ-index i_1 of an ℓ-interval $[i..j]$ is found, the remaining ℓ-indices $i_2 < i_3 < \cdots < i_k$ in $[i..j]$, where $1 \leqslant k \leqslant |\Sigma|$, are obtained successively from the *nextℓIndex* field of childtab$[i_1]$, childtab$[i_2], \ldots,$ childtab$[i_{k-1}]$. It follows that the child intervals of $[i..j]$ are the intervals $[i..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_k..j]$; see Lemma 6.1. The pseudo-code implementation of the following function *getChildIntervals* takes a pair (i, j) representing an ℓ-interval $[i..j]$ as input and returns a list containing the pairs $(i, i_1 - 1)$, $(i_1, i_2 - 1), \ldots, (i_k, j)$.

Algorithm 6.7 (*getChildIntervals*, applied to an lcp-interval $[i..j] \neq [0..n]$).

```
intervalList = [ ]
if $i <$ childtab$[j + 1].up \leqslant j$ then
    $i_1 :=$ childtab$[j + 1].up$
else $i_1 :=$ childtab$[i].down$
add(intervalList, $(i, i_1 - 1)$)
while childtab$[i_1].next\ell Index \neq \perp$ do
    $i_2 :=$ childtab$[i_1].next\ell Index$
    add(intervalList, $(i_1, i_2 - 1)$)
    $i_1 := i_2$
add(intervalList, $(i_1, j)$)
```

The function *getChildIntervals* runs in time $O(|\Sigma|)$. Since we assume that $|\Sigma|$ is a constant, *getChildIntervals* runs in constant time. Using *getChildIntervals* one can simulate every top-down traversal of a suffix tree on an enhanced suffix array. To this end, one can easily modify the function *getChildIntervals* to a function *getInterval* which takes an ℓ-interval $[i..j]$ and a character $a \in \Sigma$ as input and returns the child interval $[l..r]$ of $[i..j]$ (which may be a singleton interval) whose suffixes have the character a at position ℓ. Note that all the suffixes in $[l..r]$ share the same ℓ-character prefix because $[l..r]$ is a subinterval of $[i..j]$. If such an interval $[l..r]$ does not exist, *getInterval* returns $\perp$. Clearly, *getInterval* has the same time complexity as *getChildIntervals*.

With the help of Lemma 6.6, it is also easy to implement a function *getlcp*(i, j) that determines the lcp-value of an lcp-interval $[i..j]$ in constant time as follows: If $i <$ childtab$[j + 1].up \leqslant j$, then *getlcp*(i, j) returns the value lcptab[childtab$[j + 1].up$], otherwise it returns lcptab[childtab$[i].down$].

6.3. *Answering queries in optimal time*

As already mentioned in the introduction, given the basic suffix array, it takes $O(m \log n)$ time in the worst case to answer decision queries of length m. By using an additional table (similar to the lcp-table), this time complexity can be improved to $O(m + \log n)$; see [29]. The logarithmic terms are due to binary searches, which locate P in the suffix array of S. In this section, we show how enhanced suffix arrays allow us to answer decision queries of the type “Is P a substring of S?” in optimal $O(m)$ time. Moreover, enumeration queries of the type “Where are all z occurrences of P in S?” can be answered in optimal $O(m + z)$ time, totally independent of the size of S.

Algorithm 6.8 (Answering decision queries).

```
$c := 0$
$queryFound := True$
$(i, j) := getInterval(0, n, P[c])$
**while** $(i, j) \neq \perp$ and $c < m$ and $queryFound = True$
  **if** $i \neq j$ **then**
    $\ell := getlcp(i, j)$
    $min := \min\{\ell, m\}$
    $queryFound := S[\mathsf{suftab}[i] + c..\mathsf{suftab}[i] + min - 1] = P[c..min - 1]$
    $c := min$
    $(i, j) := getInterval(i, j, P[c])$
  **else** $queryFound := S[\mathsf{suftab}[i] + c..\mathsf{suftab}[i] + m - 1] = P[c..m - 1]$
**if** $queryFound$ **then**
  $Report(i, j)$    /* the $P$-interval */
**else** print "pattern $P$ not found"
```

The algorithm starts by determining with $getInterval(0, n, P[0])$ the lcp or singleton interval $[i..j]$ whose suffixes start with the character $P[0]$. If $[i..j]$ is a singleton interval, then pattern P occurs in S if and only if $S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + m - 1] = P$. Otherwise, if $[i..j]$ is an lcp-interval, then we determine its lcp-value ℓ by the function *getlcp*; see end of Section 6.2. Let $\omega = S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \ell - 1]$ be the longest common prefix of the suffixes $S_{\mathsf{suftab}[i]}, S_{\mathsf{suftab}[i+1]}, \ldots, S_{\mathsf{suftab}[j]}$. If $\ell \geqslant m$, then pattern P occurs in S if and only if $\omega[0..m-1] = P$. Otherwise, if $\ell < m$, then we test whether $\omega = P[0..\ell - 1]$. If not, then P does not occur in S. If so, we search with $getInterval(i, j, P[\ell])$ for the ℓ'- or singleton interval $[i'..j']$ whose suffixes start with the prefix $P[0..\ell]$ (note that the suffixes of $[i'..j']$ have $P[0..\ell - 1]$ as a common prefix because $[i'..j']$ is a subinterval of $[i..j]$). If $[i'..j']$ is a singleton interval, then pattern P occurs in S if and only if $S[\mathsf{suftab}[i'] + \ell..\mathsf{suftab}[i'] + m - 1] = P[\ell..m - 1]$. Otherwise, if $[i'..j']$ is an ℓ'-interval, let $\omega' = S[\mathsf{suftab}[i']..\mathsf{suftab}[i'] + \ell' - 1]$ be the longest common prefix of the suffixes $S_{\mathsf{suftab}[i']}, S_{\mathsf{suftab}[i'+1]}, \ldots, S_{\mathsf{suftab}[j']}$. If $\ell' \geqslant m$, then pattern P occurs in S if and only if $\omega'[\ell..m - 1] = P[\ell..m - 1]$ (or equivalently, $\omega[0..m - 1] = P$). Otherwise, if $\ell' < m$, then we test whether $\omega[\ell..\ell' - 1] = P[\ell..\ell' - 1]$. If not, then P does not occur in S. If so, we search with $getInterval(i', j', P[\ell'])$ for the next interval, and so on.

Enumerative queries can be answered in optimal $O(m + z)$ time as follows. Given a pattern P of length m, we search for the P-interval $[l..r]$ using the preceding algorithm. This takes $O(m)$ time. Then we can report the start position of every occurrence of P in S by enumerating $\mathsf{suftab}[l], \ldots, \mathsf{suftab}[r]$. In other words, if P occurs z times in S, then reporting the start position of every occurrence requires $O(z)$ time in addition.

6.4. Finding all shortest unique substrings

As a second application of a top-down traversal of the lcp-interval tree, we will briefly describe how to find all shortest unique substrings in optimal time. The problem is relevant when designing primers for DNA sequences.

A substring of S is *unique* if it occurs only once in S. The *shortest unique substring problem* is to find all shortest unique substrings of S. For example, ca is the only shortest unique substring in $acac$. It is easy to verify that a unique substring in S corresponds to a singleton interval. In particular, if u is a shortest unique substring of S, then there is an ℓ-interval $[i..j]$ and a singleton child interval $[k..k]$ of $[i..j]$ such that u is a prefix of length $\ell + 1$ of $S_{\mathsf{suftab}[k]}$ and $u[\ell] \neq \$$. As a consequence, we solve the shortest unique substring problem by enumerating lcp-intervals. Since we are interested in lcp-intervals of minimal lcp-value, we perform a breadth-first traversal of the lcp-interval tree, using a queue. Of course, we do not construct the lcp-interval tree. Instead we use the enhanced suffix array to generate the lcp-intervals. Besides the queue, we maintain a set M of unique substrings, represented by their length and their start position in S. The length q of the unique substrings in M is minimal over all unique substrings detected so far. Initially, M is empty and $q = \infty$.

Suppose that $[i..j]$ is the current ℓ-interval to be processed during the traversal. We compute all child intervals of $[i..j]$ according to Algorithm 6.7. For each singleton child interval $[k..k]$ of $[i..j]$ with $S_{\mathsf{suftab}[k]}[\ell] \neq \$$, the prefix of $S_{\mathsf{suftab}[k]}$ of length $\ell + 1$ is a unique substring of S. If M is empty or $q > \ell + 1$, then M is updated by $\{(\ell + 1, \mathsf{suftab}[k])\}$ and q is assigned $\ell + 1$. If M is not empty and $q = \ell + 1$, then we add $(\ell + 1, \mathsf{suftab}[k])$ to M. Otherwise, M and q are left unchanged.

Each child interval $[i'..j']$ of $[i..j]$ with lcp-value ℓ' is added to the back of the queue, whenever $\ell' + 1 \leqslant q$. Then we proceed with the next lcp-interval at the front of the queue, as described above, until the queue is empty.

Computing the child intervals of an lcp-interval takes constant time. Verifying the uniqueness and maintaining the queue as well as the set M takes time proportional to the number of processed lcp-intervals. In the worst case, this is $O(n)$. Thus the algorithm runs in $O(n)$ time. However, in practice only a small number of lcp-intervals is processed; see Section 9.

7. Incorporating suffix links

In this section, we incorporate suffix links into our framework. As an application, we will show how to efficiently compute matching statistics by a traversal of the lcp-interval tree that uses suffix links. Let us first recall the definition of suffix links. In the following, we denote a node u in the suffix tree by $\overline{\omega}$ if and only if the concatenation of the edge-labels on the path from the root to u spells out the string ω. It is a property of suffix trees that for any internal node $\overline{a\omega}$, there is also an internal node $\overline{\omega}$. A pointer from $\overline{a\omega}$ to $\overline{\omega}$ is called a *suffix link*.

Recall that the inverse suffix array suftab^{-1} is a table such that $\mathsf{suftab}^{-1}[\mathsf{suftab}[q]] = q$ for every $0 \leqslant q \leqslant n$; see Fig. 5.

Definition 7.1. Let $S_{\mathsf{suftab}[i]} = a\omega$. If index j, $0 \leqslant j < n$, satisfies $S_{\mathsf{suftab}[j]} = \omega$, then we denote j by $\mathsf{link}[i]$ and call it the suffix link (index) of i.

Lemma 7.2. *If* $\mathsf{suftab}[i] < n$, *then* $\mathsf{link}[i] = \mathsf{suftab}^{-1}[\mathsf{suftab}[i] + 1]$.

			childtab			suflink			
i	suftab	lcptab	1.	2.	3.	l	r	suftab^{-1}	$S_{\mathsf{suftab}[i]}$
0	2	0		②	6			2	$aaacatat\$$
1	3	2			↗	0	5	6	$aacatat\$$
2	0	1	1	③	4	0	10	0	$acaaacatat\$$
3	4	3			↗	6	7	1	$acatat\$$
4	6	1	3	5 →				3	$atat\$$
5	8	2			↗	8	9	7	$at\$$
6	1	0	2	⑦	8			4	$caaacatat\$$
7	5	2			↗	0	5	8	$catat\$$
8	7	0	7	⑨	10			5	$tat\$$
9	9	1			↗	0	10	9	$t\$$
10	10	0	9					10	$\$$

Fig. 5. Suffix array of the string $S = acaaacatat$ enhanced with the lcp-table, the child-table, and the suffix link table. The inverse suffix array is used only in the construction of the suffix link table.

Proof. Let $S_{\mathsf{suftab}[i]} = a\omega$. Since $\omega = S_{\mathsf{suftab}[i]+1}$, $\mathsf{link}[i]$ must satisfy $\mathsf{suftab}[\mathsf{link}[i]] = \mathsf{suftab}[i] + 1$. This immediately proves the lemma. □

Under a different name, the function link appeared already in [14].

Definition 7.3. Given ℓ-interval $[i..j]$, the smallest lcp-interval $[l..r]$ satisfying $l \leqslant \mathsf{link}[i] < \mathsf{link}[j] \leqslant r$ is called the *suffix link interval* of $[i..j]$.

Suppose that the ℓ-interval $[i..j]$ corresponds to an internal node $\overline{a\omega}$ in the suffix tree. Then there is a suffix link from node $\overline{a\omega}$ to the internal node $\overline{\omega}$. The following lemma states that node $\overline{\omega}$ corresponds to the suffix link interval of $[i..j]$.

Lemma 7.4. *Given the $a\omega$-interval ℓ-$[i..j]$, its suffix link interval is the ω-interval, which has lcp-value $\ell - 1$.*

Proof. Let $[l..r]$ be the suffix link interval of $[i..j]$. Because the lcp-interval $[i..j]$ is the $a\omega$-interval, $a\omega$ is the longest common prefix of $S_{\mathsf{suftab}[i]}, \ldots, S_{\mathsf{suftab}[j]}$. Consequently, ω is the longest common prefix of $S_{\mathsf{suftab}[\mathsf{link}[i]]}, \ldots, S_{\mathsf{suftab}[\mathsf{link}[j]]}$. It follows that ω is the longest common prefix of $S_{\mathsf{suftab}[l]}, \ldots, S_{\mathsf{suftab}[r]}$, because $[l..r]$ is the smallest lcp-interval satisfying $l \leqslant \mathsf{link}[i] < \mathsf{link}[j] \leqslant r$. That is, $[l..r]$ is the ω-interval and thus it has lcp-value $\ell - 1$. □

7.1. Construction of the suffix link table

In order to incorporate suffix links into the enhanced suffix array, we proceed as follows. In a preprocessing step, we compute for every ℓ-interval $[i..j]$ its suffix link interval $[l..r]$ and store the left and right boundaries l and r at the first ℓ-index of $[i..j]$. The corresponding table, indexed from 0 to n is denoted by suflink; see Fig. 5 for an example. Note that the lcp-value of $[l..r]$ need not be stored because it is known to be $\ell - 1$. Thus, the space requirement for suflink is $2 \cdot 4n$ bytes in the worst case. To compute the suffix link table suflink, the lcp-interval tree is traversed in a breadth first left-to-right manner. For every lcp-value encountered, we hold a list of intervals of that lcp-value, which is initially empty. Whenever an ℓ-interval is computed, it is appended to the list of ℓ-intervals; this list is called ℓ-list in what follows. In the example of Fig. 2, this gives

0-list:	[0..10]
1-list:	[0..5], [8..9]
2-list:	[0..1], [4..5], [6..7]
3-list:	[2..3]

Note that the ℓ-lists are automatically sorted in increasing order of the left-boundary of the intervals and that the total number of ℓ-intervals in the ℓ-lists is at most n. For every lcp-value $\ell > 0$ and every ℓ-interval $[i..j]$ in the ℓ-list, we proceed as follows. We first compute $\mathsf{link}[i]$ according to Lemma 7.2. Then, by a binary search in the $(\ell - 1)$-list, we search in $O(\log n)$ time for the interval $[l..r]$ such that l is the largest left boundary of all $(\ell-1)$-intervals with $l \leqslant \mathsf{link}[i]$. This interval is the suffix link interval of $[i..j]$. Finally, we determine in constant time the first ℓ-index of $[i..j]$ according to Lemma 6.6 and store l and r there. Because there are less than n lcp-intervals and for each interval the binary search takes $O(\log n)$ time, the preprocessing phase requires $O(n \log n)$ time. Table suftab^{-1} and the ℓ-lists require $O(n)$ space, but they are only used in the preprocessing phase and can be deleted after the computation of the suffix link table.

Theoretically, it is possible to compute the suffix link intervals in time $O(n)$ via the construction of the suffix tree. But it is also possible to give a linear time algorithm without intermediate construction of the suffix tree. We achieve this by avoiding the binary search over the ℓ-lists and reducing the problem of computing the suffix link intervals to the problem of answering range minimum queries. In contrast to the previous $O(n \log n)$-time algorithm, we store the boundaries i and j of an ℓ-interval $[i, j]$ at *every* ℓ-index (again, these values can be deleted once the suffix link table suflink is created).

Next, we will show that it is possible to compute the suffix link interval $[l..r]$ of an ℓ-interval $[i, j]$ in constant time. To this end, we need the following lemma:

Lemma 7.5. *Let $[i, j]$ be an ℓ-interval $[i, j]$ and let $[l..r]$ be its suffix link interval. Since there is an ℓ-index q with $i + 1 \leqslant q \leqslant j$, there is also an index k such that k is an $(\ell - 1)$-index of $[l..r]$ and $\mathsf{link}[i] + 1 \leqslant k \leqslant \mathsf{link}[j]$.*

Proof. Follows from the proof of Lemma 7.4. □

Because $l \leqslant \mathsf{link}[i] + 1 \leqslant \mathsf{link}[j] \leqslant r$ and $\ell - 1$ is the length of the longest common prefix of $\mathsf{link}[i]$ and $\mathsf{link}[j]$, the minimum value of the lcp-table in the range $[\mathsf{link}[i] + 1..\mathsf{link}[j]]$ is $\ell - 1$. Therefore, one can locate an $(\ell - 1)$-index k of $[l..r]$ with $\mathsf{link}[i] + 1 \leqslant k \leqslant \mathsf{link}[j]$ by answering the range minimum query in the range $[\mathsf{link}[i] + 1..\mathsf{link}[j]]$. The range minimum query is defined as follows.

Definition 7.6. Let L be an integer-array of size n whose elements are in the range $[0, n - 1]$. Let $0 \leqslant i < j \leqslant n - 1$. The *range minimum query RMQ(i, j)* asks for an index k such that $i \leqslant k \leqslant j$ and $L[k] = \min\{L[q] \mid i \leqslant q \leqslant j\}$.

An *RMQ* can be answered in constant time provided that the array L is appropriately preprocessed. Fortunately, the preprocessing of L requires only linear time and space; see [4,21,31].

For the computation of suffix link intervals, one solves *RMQ*s for $L = \mathsf{lcptab}$. As in the previous algorithm, the lcp-interval tree is traversed in breadth-first order. Thus the ℓ-intervals are processed in ascending order of their lcp-value. Suppose ℓ-interval $[i..j]$ is to be processed and all intervals of lcp-value $\ell - 1$ have already been processed. First, we store the ℓ-interval boundaries i and j at every ℓ-index of $[i..j]$. Second, we compute $\mathsf{link}[i]$ and $\mathsf{link}[j]$ according to Lemma 7.2, and evaluate $k = RMQ(\mathsf{link}[i] + 1, \mathsf{link}[j])$. k is an $(\ell - 1)$-index of the suffix link interval of $[i..j]$, and thus we can look up the boundaries l and r of this suffix link interval at index k. Finally, we store l and r in the suffix link table at the first *ℓ-index* of $[i..j]$. Because every step in this procedure takes constant time and space, the overall complexity of computing the suffix link intervals is $O(n)$.

The following subsection describes the application of suffix link intervals to compute matching statistics.

7.2. Computing matching statistics

Matching statistics were introduced in [7] to solve the approximate string matching problem in sublinear expected time.

Let T be a string of length m. A *matching statistics* of T w.r.t. S is a table of pairs (l_j, p_j), where $0 \leqslant j \leqslant m - 1$, such that the following holds:

1. $T[j..j + l_j - 1]$ is the longest prefix of $T[j..m - 1]$ which occurs as a substring of S.
2. $T[j..j + l_j - 1] = S[p_j..p_j + l_j - 1]$.

If $T[j..j + l_j - 1]$ occurs more than once as a substring of S, then there are several choices for p_j. Here it is merely required that one such p_j is determined. Let $S = cacaccc$ and $T = caacacacca$. Then the following table shows a matching-statistics of T w.r.t. S:

j	0	1	2	3	4	5	6	7	8	9
(l_j, p_j)	$(2,0)$	$(1,1)$	$(4,1)$	$(6,0)$	$(5,1)$	$(4,2)$	$(3,3)$	$(2,4)$	$(2,2)$	$(1,3)$

Chang and Lawler [7] provided an algorithm to compute matching statistics in $O(n + m)$ time. This algorithm traverses the suffix tree of S in a single left-to-right scan of T utilizing suffix links. In each step of the algorithm, the suffix $T[j..m - 1]$ of T is matched against the suffix tree until a mismatch occurs or all characters in T have been completely matched. This determines a location in the suffix tree and delivers the length l_j of the longest matching prefix of $T[j..m - 1]$. p_j is the starting position of a suffix of $S\$$ in the subtree below the location. If $l_j > 0$, then $l_{j+1} \geqslant l_j - 1$, because $T[j + 1..j + l_j - 1] = S[p_j + 1..p_j + l_j - 1]$. Using suffix links one determines the location for $T[j + 1..j + l_j - 1]$ in the suffix tree in constant amortized time and continues to match $T[j + l_j..m]$ against the tree.

Using the methods described in previous sections, we can adapt this algorithm to enhanced suffix arrays. Given the enhanced suffix array for S with tables suftab, lcptab, childtab, and suflink, a *location in the enhanced suffix array* is a triple $([i..j], q, [l..r])$ where $[i..j]$ is an ℓ-interval, and either $q = \ell$ and $[i..j] = [l..r]$ or the following holds: $[l..r]$ is a child interval of $[i..j]$ and either $[l..r]$ is an m-interval and $\ell < q < m$ or $[l..r]$ is a singleton interval and $\ell < q \leqslant n - \mathsf{suftab}[l]$. Each location $([i..j], q, [l..r])$ in the enhanced suffix array corresponds to exactly one substring of S, namely $S[\mathsf{suftab}[l]..\mathsf{suftab}[l] + q - 1]$.

Algorithm 6.8 can easily be modified such that

- it greedily matches a string character by character until there is no child interval for the current character or all characters have been matched, and
- it starts matching at any location and delivers a location as a result.

The resulting algorithm is called *greedymatch*. To compute the matching statistics, *greedymatch* is applied to each suffix $T[j..m-1]$ of T, from longest to shortest. In each step, *greedymatch* determines a location $([i..j], q, [l..r])$ corresponding to the longest prefix of $T[j..m-1]$ occurring as a substring of S, and we assign $l_j := q$ and $p_j := \mathsf{suftab}[z]$ for some $z \in [l, r]$. If $j = 0$ or $l_j = 0$, then the matching process starts at location $([0..n], 0, [0..n])$. Otherwise, we look up the suffix link interval $[i'..j']$ of $[i..j]$ in $\mathsf{suflink}[\min \ell \mathit{Indices}(i, j)]$. If $q = \ell$ and $[i..j] = [l..r]$, then we proceed with $[i'..j']$. Otherwise, we first have to "rescan" $S[\mathsf{suftab}[l] + \ell..\mathsf{suftab}[l] + q - 1]$ from location $[i'..j']$. This can easily be achieved in constant time per visited lcp-interval by a modification of *greedymatch*. In this way, we obtain an algorithm that determines the matching statistics in $O(n + m)$ time.

8. Implementation details

In this section, we present implementation details that considerably reduce the space requirement. Our experiments show that this entails no loss of performance, albeit the worst case time complexities of the algorithms may be affected.

8.1. The lcp-table

It has already been mentioned that the lcp-table requires $4n$ bytes in the worst case. In practice, however, the lcp-table can be implemented in little more than n bytes. More precisely, we store most of the values of table lcptab in a table lcptab_1 using n bytes. That is, for any $i \in [1, n]$, $\mathsf{lcptab}_1[i] = \max\{255, \mathsf{lcptab}[i]\}$. There are usually only few entries in lcptab that are larger than or equal to $\geqslant 255$; see Section 9. To access these efficiently, we store them in an extra table llvtab. This contains all pairs $(i, \mathsf{lcptab}[i])$ such that $\mathsf{lcptab}[i] \geqslant 255$, ordered by the first component. Each entry in llvtab requires 8 bytes. If $\mathsf{lcptab}_1[i] = 255$, then the correct value of lcptab is found in llvtab. If we scan the values in lcptab_1 in consecutive order and find a value 255, then we access the correct value in lcptab in the next entry of table llvtab. If we access the values in lcptab_1 in arbitrary order and find a value 255 at index i, then we perform a binary search in llvtab using i as the key. This delivers $\mathsf{lcptab}[i]$ in $O(\log_2 |\mathsf{llvtab}|)$ time.

8.2. The child-table

As the lcp-table, the child-table requires $4n$ bytes but in practice it can be stored in n bytes without loss of performance. To achieve this goal, we store relative indices in childtab. For example, if $j = \mathsf{childtab}[i].\mathit{next}\ell\mathit{Index}$, then we store $j - i$. The relative indices are almost always smaller than 255. Hence we use only one byte for storing a value of table childtab. The values $\geqslant 255$ are not stored. Instead, if we encounter the value 255 in childtab, then we use a function that is equivalent to *getInterval*, except that it determines a child interval by a binary search, similar to the algorithm of [29, p. 937]. Consequently, instead of 4 bytes per entry of the child-table, only 1 byte is needed. The overall space consumption for tables suftab, lcptab, and childtab is thus only $6n$ bytes.

For a given parameter d, we additionally use an extra bucket table bcktab_d. This table stores for each string w of length d the smallest integer i, such that $S_{\mathsf{suftab}[i]}$ is a prefix of w. In this way, we can answer small queries of length $m \leqslant d$ in time $O(m)$. For larger queries, this bucket table allows us to locate the interval containing the d-character prefix $P[0..d-1]$ of the query P in constant time. Then our algorithm, which searches for the pattern P in S, starts with this interval instead of the interval $[0..n]$. d is chosen to be the maximal value such that table bcktab_d never requires more than n bytes. The advantage of this hybrid method is that only a small part of the suffix array is actually accessed. Moreover, we only rarely access the values 255 in childtab.

8.3. The suffix link table

In the algorithm of Section 7.2 we compute for the d-length prefix w of each suffix of length at least d, a unique integer code $\varphi(w)$ in the range $[0, |\Sigma|^d - 1]$. These integer codes can be computed in $O(m)$ additional time, and they are used to access table bcktab_d. Now suppose we want to compute the suffix link interval of some ℓ-interval $[i..j]$. If $\ell \leqslant d+1$, then this can be done in constant time by some integer arithmetic and looking up appropriate values in table bcktab_d. Now let $\ell > d+1$. In this case, we access table suflink as described at the beginning of Section 7.1. However, in suftab we have stored the left boundary value we are looking for relative to $\mathsf{bcktab}_d[\varphi(w)]$. This relative value is usually very small, and therefore we use 1 byte to store it. Similarly, the right boundary value is stored relative to the left boundary value, which also allows to reduce the corresponding space to 1 byte. Altogether, the suffix link table suflink requires only $2n$ bytes in our implementation.

9. Experimental results

For our experiments, we collected a set of files of different sizes and types:

E. coli:	The complete genome of the bacterium *Escherichia coli*, strain K12. This is a DNA sequence of length 4,639,221. The alphabet size is 4.
Yeast:	The complete genome of the baker's yeast *Saccharomyces cerevisiae*, i.e., a DNA sequence of length 12,156,300. The alphabet size is 4.

Hs21: The complete sequence of chromosome 21 of homo sapiens. The length is 33,917,895. The alphabet size is 4.
Swissprot: The complete collection of protein sequences from the Swissprot database (release 38). The total size of all sequences is 29,165,964. The alphabet size is 20.
Shaks: A collection of the complete works of William Shakespeare. The total size is 5,582,655 bytes. The alphabet size is 92.

In addition we collected four different pairs of similar genomes:

Streptococuss 2: The complete genomes of two strains of *Streptococcus pneumoniae* (length 2,160,837 and 2,038,615).
E. coli 2: *E. coli* (see above) and the complete genome of a different strain of this bacterium (*E. coli* O157:H7, length 5,528,445).
Yeast 2: *Yeast*(see above) and the complete genome of a different kind of yeast (*S. pombe*, length 12,534,386).
Human 2: *Hs21*(see above) and chromosome 22 of homo sapiens (length 33,821,705).

Prior to all computations described below, we constructed the enhanced suffix array for all input sequences. Each of the tables comprising the index is stored on a different file. The construction was done by a program that is based on the suffix sorting algorithm of [5]. This program uses about 50% less space than the best programs constructing suffix trees (see below). The enhanced suffix array is constructed in about the same time as the suffix tree. We do not give more details here, since we want to focus on the application of enhanced suffix arrays.

The running times reported here are for a SUN-Sparc computer equipped with 32 gigabytes RAM and a 950 Mhz CPU. For our tests, we only needed at most 3165 megabytes of memory; see Table 3.

9.1. Computing repeats and maximal unique matches

In our first experiment we ran different programs computing repeats and maximal matches. The name of a program based on enhanced suffix arrays always begins with the prefix *esa*.

- *REPuter* and *esarep* implement the algorithm of Gusfield (see Section 5.1) to compute maximal repeated pairs. *REPuter* is based on suffix trees (improved linked list representation of [25]).
- *esasupermax* computes supermaximal repeats. It implements the algorithm described in Section 3.3.
- *unique-match* and *esamum* compute *MUM*s. *unique-match* is part of the original distribution of *MUMmer* (version 1.0) [8]. It is based on suffix trees. *unique-match* as well as *REPuter* construct the suffix tree in main memory (using $O(n)$ time). *esamum* uses the algorithm described at the end of Section 3.4.

Table 2
Running times (in seconds) and space requirement (in megabytes) for computing maximal repeated pairs and supermaximal repeats. The column titled #reps gives the number of repeats of length $\geqslant \ell$. The space requirement is independent of ℓ, hence it is given in a separate table

ℓ	Running time for *E. coli* ($n = 4{,}639{,}221$) in sec.					Running time for *Yeast* ($n = 12{,}156{,}300$) in sec.				
	maximal repeated pairs			*esasupermax*		*maximal repeated pairs*			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps		#reps	*REPuter*	*esarep*	#reps	
20	7799	3.28	0.79	899	0.16	175455	9.71	2.23	6432	0.47
23	5206	3.28	0.78	642	0.15	84115	9.63	2.16	4069	0.47
27	3569	3.31	0.79	500	0.15	41400	9.72	2.14	2813	0.45
30	2730	3.30	0.80	456	0.15	32199	9.69	2.14	2374	0.46
40	840	3.29	0.79	281	0.15	20767	9.57	2.13	1674	0.44
50	607	3.29	0.79	196	0.14	16209	9.64	2.12	1354	0.44

ℓ	Running time for Hs21 ($n = 33{,}917{,}895$) in sec.				
	maximal repeated pairs			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps	
20	40193973	54.63	24.00	188695	1.50
23	19075117	51.78	14.62	138523	1.44
27	8529120	47.97	9.88	98346	1.39
30	4787086	46.54	8.15	77695	1.34
40	732822	45.06	6.21	35719	1.23
50	149482	44.33	5.85	16392	1.19

Space requirement in megabytes			
	REPuter	*esarep*	*esasupermax*
E. coli	61	31	31
Yeast	160	83	83
Hs21	446	227	227

Table 3
Running times (in seconds) and space consumption (in megabytes) for computing *MUM*s of length $\geqslant 20$. The column titled #*MUM*s gives the number of *MUM*s. The time given for *unique-match* does not include suffix tree construction. *esamum* reads the enhanced suffix array from different files via memory mapping

Genome pair	Total size	#*MUM*s	*unique-match*		*esamum*	
			time	*space*	*time*	*space*
Streptococuss 2	4,199,453	6613	9.0	196	0.33	30
E. coli 2	10,107,957	10817	30.7	472	0.69	62
Yeast 2	24,690,687	2536	118.2	1154	0.66	144
Human 2	67,739,601	217014	430.1	3165	2.34	389

All programs based on suffix arrays use memory mapping to access the enhanced suffix array from the different files. Of course, a file is mapped into main memory only if the table it stores is required for the particular algorithm. We applied the three programs for the detection of repeats to *E. coli*, *Yeast*, and *Hs21*. Additionally, we applied *unique-match* and *esamum* to the pairs of genomes listed above.

The results of applying the different programs to the different data sets are shown in Tables 2 and 3. For a fair comparison, we report the running time of *REPuter* and of *unique-match* without suffix tree construction.

The running time of *esasupermax* is almost independent of the minimal length of the supermaximal repeats computed. Since the algorithm is so simple, the main part of the running time is the input and output. The strmat-package of [22] implements a more complicated algorithm than ours for the same task. For example, when applied to *E. coli*, it requires 19 sec. (without suffix tree construction) to compute all 944,546 supermaximal repeats of length at least 2. For this task *esasupermax* requires 0.82 seconds due to the large size of the output.

The comparison of *esarep* and *REPuter* underlines the advantages of the enhanced suffix array over the suffix tree. *esarep* used about halve of the space of *REPuter*. If there are many repeats, then the computation is dominated by the postprocessing of the repeats (e.g., computing E-values), which is identical in both programs. Hence *esarep* is only 2–3 times faster than *REPuter* in these cases. In general, *esarep* is 4–5 times faster than *REPuter*. This is due to the improved cache behavior achieved by the linear scanning of the tables suftab, lcptab, and bwttab.

The running times and space results shown in Table 3 reveal that *esamum* is much faster than *unique-match*, using at most 15% of the space.

All in all, the experiments show that our programs based on enhanced suffix arrays define the state-of-the-art in computing different kinds of repeats and maximal matches. The programs *esarep*, *esasupermax*, and *esamum* are available as part of the *Vmatch*-software package, see http://www.vmatch.de.

9.2. Searching for patterns

For our second experiment, we ran three different programs for answering enumeration queries:

- *streematch* is based on the improved linked list representation of suffix trees, as described in [25].
- *mamy* is based on suffix arrays and uses the algorithm of [29] with additional buckets to speedup the searches. We used the original program code developed by Gene Myers.
- *esamatch* is based on enhanced suffix arrays (tables suftab, lcptab, childtab) and uses Algorithm 6.8.

The programs *streematch* and *mamy* first construct the index in main memory and then perform pattern searches. *esamatch* accesses the enhanced suffix array from the different files via memory mapping.

Table 4 shows the running times in seconds for the different programs when searching for one million patterns. This seems to be a large number of queries to be answered. However, at least in the field of genomics, it is relevant; see [15]. The shortest running times in Table 4 are shown in bold face. The time for index construction is not included. Patterns were generated according to the following strategy: For each input string S of length n we randomly sampled $p = 1{,}000{,}000$ substrings $s_1, s_2, \ldots, s_p$ of different lengths from S. The lengths were evenly distributed over different intervals $[minpl, maxpl]$, where $(minpl, maxpl) \in \{(20, 30), (30, 40), (40, 50)\}$. For $i \in [1, p]$, the programs were called to search for pattern p_i, where $p_i = s_i$, if i is even, and p_i is the reverse of s_i, if i is odd. Reversing a string s_i simulates the case that a pattern search is often unsuccessful.

Table 4
Running times (in seconds) and space requirement (in megabytes) for one million enumeration queries searching for exact patterns in the input strings. *minpl* and *maxpl* are the minimal and maximal size of the patterns searched for

File	Running time for *minpl* = 20, *maxpl* = 30			Running time for *minpl* = 30, *maxpl* = 40		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.47	5.56	**4.48**	9.63	5.70	**4.69**
Yeast	12.42	8.26	**5.37**	12.56	8.46	**5.80**
Hs21	20.15	12.50	**7.23**	20.43	12.69	**7.30**
Swissprot	41.78	9.55	**6.22**	40.80	10.09	**6.25**
Shaks	15.61	**4.29**	72.44	15.78	**4.37**	66.60
	Running time for *minpl* = 40, *maxpl* = 50			Space requirement		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.86	5.87	**4.85**	56	40	47
Yeast	13.34	8.63	**5.74**	146	106	120
Hs21	21.22	12.88	**7.61**	407	296	327
Swissprot	42.96	9.83	**6.39**	320	288	281
Shaks	15.88	**4.49**	67.16	52	48	60

As expected, the running times of *streematch* and *esamatch* depend on the alphabet size. This is not true for *mamy*. For *Shaks*, *mamy* is much faster than the other programs, which we explain by the large alphabet. For the other files, *esamatch* is always more than twice as fast as *streematch* and slightly faster than *mamy*. All in all, this experiment shows that for small alphabets *esamatch* can compete with the other programs and is not only of theoretical interest.

9.3. Searching for minimal unique substrings

For our third experiment, we implemented the breadth first traversal algorithm of Section 6.4 to find shortest unique substrings. We applied it to *E. coli* and *Yeast*. For *E. coli* our program computed three shortest unique substrings, each of length 7, in 0.09 seconds. It processed 11,392 lcp-intervals (0.38% of all 2,978,098 lcp-intervals in the corresponding lcp-interval tree). For *Yeast* our program computed 383 shortest unique substrings, each of length 9, in 0.75 seconds. It processed 92,863 lcp-intervals (1.2% of all 7,904,703 lcp-intervals in the corresponding lcp-interval tree). To demonstrate the efficiency of our solution to the shortest unique substring problem, we implemented a straightforward method to solve the same problem by enumerating all lcp-intervals. For *E. coli*, the straightforward method delivers the result in 0.79 seconds, while it takes 3.47 seconds for *Yeast*.

9.4. Computing matching statistics

For our final experiment, we applied two programs computing matching statistics to the pairs of genomes listed at the beginning of this section (Table 5). The program *streems* is based on the improved linked list implementation of suffix trees, while our program *esams* uses the enhanced suffix arrays as described in Section 7.2. The experiments show a

Table 5
Running times (in seconds) and space consumption (in megabytes) for computing matching statistics. The time given for *streems* does not include suffix tree construction. *esams* reads the enhanced suffix array from different files via memory mapping. The space requirement for the matching statistics is not included

Genome pair	Total length	*streems*		*esams*	
		time	*space*	*time*	*space*
Streptococuss 2	4,199,453	4.1	30	11.1	21
E. coli 2	10,107,957	13.3	65	18.9	43
Yeast 2	24,690,687	41.0	170	43.4	109
Human 2	67,739,601	169.2	472	314.0	294

trade-off between time and space consumption: While *esams* uses 30–40% less space than *streems*, the latter program is up to three times faster. We explain this by the slow lookup of the suffix link interval in the enhanced suffix array. It remains an open problem to find an alternative way to locate suffix link intervals more efficiently.

10. Conclusions and related work

The contribution of this article is twofold: First, it has been shown that every algorithm that uses a suffix tree as data structure can systematically be replaced with an algorithm that uses an enhanced suffix array and solves the same problem in the *same* time complexity. This shows that our new approach to solving string processing problems is interesting from a theoretical point of view. Second, we have shown that the space requirement in large scale applications such as the comparison of whole genomes can drastically be reduced by using enhanced suffix arrays instead of suffix trees. This makes the algorithms very valuable in practice.

All the algorithms presented in this article and others such as the computation of all tandem repeats of a string (see [1]) have been carefully implemented and the space consumption has been reduced to a few bytes per input character. The precise space consumption depends on the application; see Table 6 for an overview. Although the practical implementation does not always achieve the worst case time complexity that is possible without space reduction, we did not observe any loss of performance. In fact, our experiments show that the programs can handle large data sets very efficiently. Some of the algorithms described here are implemented in the software tool *Vmatch*; see http://www.vmatch.de.

We would like to mention that the very recent results concerning *RMQ*s [4,21,31] (see Section 7.1) can be used to obtain a different method to simulate top-down traversals of a suffix tree, i.e., without the construction of the childtab. In order to compute the child intervals of an ℓ-interval $[i..j]$, it suffices to compute the ℓ-indices of $[i..j]$; see Lemma 6.1. By Definition 3.1, the ℓ-indices $i_1 < i_2 < \cdots < i_k$ of $[i..j]$ are the indices with minimum *lcp*-value in the range $[i+1..j]$. Suppose that every *RMQ* returns the first index k such that lcptab$[k]$ is minimum in the given range (according to [21], one such *RMQ* can be answered in constant time). Then the ℓ-indices of $[i..j]$ can be found by successively computing $i_1 := RMQ(i+1, j)$, $i_2 := RMQ(i_1+1, j), \ldots, i_k := RMQ(i_{k-1}+1, j)$, until $RMQ(i_k+1, j)$

Table 6
Summary of the tables required for the applications mentioned in the paper. The program *esamum*, for example, requires an enhanced suffix array consisting of the tables suftab, lcptab, and bwttab

Application	Enhanced suffix array					
	suftab $4n$ bytes	lcptab n bytes	childtab n bytes	suflink $2n$ bytes	S $n \log\lvert\Sigma\rvert$ bits	bwttab $n \log\lvert\Sigma\rvert$ bits
esasupermax	$\surd$	$\surd$				$\surd$
esamum	$\surd$	$\surd$				$\surd$
esarep	$\surd$	$\surd$				$\surd$
Ziv–Lempel	$\surd$	$\surd$				
esamatch	$\surd$	$\surd$	$\surd$		$\surd$	
shortest unique sub.	$\surd$	$\surd$	$\surd$			
esams	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	

returns a value q such that lcptab$[q] \neq \ell$. Future work will show whether this approach is also of practical interest.

Clearly, it would be desirable to further reduce the space requirement of the suffix array. Recently, interesting results in this direction have been obtained. The most notable ones are the compressed suffix array introduced by Grossi and Vitter [14] and the so-called opportunistic data structure devised by Ferragina and Manzini [10]. These data structures reduce the space consumption considerably. Because the papers cited above solely focus on pattern matching, we can only compare their pattern matching results with ours. Due to the compression, the above-mentioned approaches do not allow to answer enumeration queries in $O(m + z)$ time; instead they require $O(m + z \log^{\varepsilon} n)$ time, where $\varepsilon > 0$ is a constant.[3] Worse, experimental results [11] show that the gain in space reduction has to be paid by considerably slower pattern matching; this is true even for decision queries. According to [11], the opportunistic index is 8–13 times more space efficient than the suffix array but string matching based on the opportunistic index is 16–35 times slower than their implementation based on the suffix array. So there is a trade-off between time and space consumption. In contrast to that, suffix arrays can be queried at speeds comparable to suffix trees, while being much more space efficient than these. Let us briefly compare our retrieval times with those of an implementation of the opportunistic data structure [11]. According to [11], it takes 7.6 seconds to answer 1000 enumerative queries searching for random patterns of length between 8 and 15 in *E. coli* (on a Pentium 600 Mhz). By contrast, our program *esamatch* requires only 0.003 seconds for the same task (on a Pentium 933 Mhz). Under the (conservative) assumption that a 933 MHz processor is 1.5 times faster than a 600 Mhz processor, a comparison of the preceding running times shows that our program is more than 1650 times faster than that of [11]. However, a closer look at the experimental results of [11] reveals some inconsistencies with our results. For example, [11] report that their program based on suffix arrays requires 0.6 seconds to answer 1000 enumerative queries searching for random patterns of length between 8 and 15 in *E. coli*

[3] Ferragina and Manzini [12] also proposed a compressed data structure that removes the $\log^{\varepsilon} n$ factor from the search time at the cost of adding a $\log^{\varepsilon} n$ factor to the space. However, no experiments with this data structure are reported.

(on a Pentium 600 Mhz). By contrast, *mamy* takes only 0.02 seconds for the same task. It is not clear where these differences come from. The authors of [11] may have used a different algorithm than *mamy*, or they may have implemented the same algorithm less efficiently than Gene Myers did.

More recently, Hon and Sadakane [18] and Sadakane [31] showed that compressed suffix arrays can be used to solve string processing tasks like computing all *MUM*s of two sequences. However, it remains an open problem to develop a software tool based on compressed suffix arrays that can compete with *MUMmer* or ours. Moreover, a systematic approach like ours has not yet been developed for compressed suffix arrays.

Acknowledgements

We thank Dirk Strothmann, who observed that the values of the *up* field can also be stored in the *nextℓIndex* field of the child-table. Gene Myers provided the original code for programs to construct and search suffix arrays. M.I.A. and E.O. were partly supported by DFG-grant Oh 54/4-1 and S.K. was partly supported by the DFG-grant Ku 1257/1-3.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, in: Proc. Workshop on Algorithms in Bioinformatics, in: Lecture Notes in Computer Science, vol. 2452, Springer-Verlag, Berlin, 2002, pp. 449–463.

[2] M.I. Abouelhoda, E. Ohlebusch, S. Kurtz, Optimal exact string matching based on suffix arrays, in: Proc. International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 2476, Springer-Verlag, Berlin, 2002, pp. 31–43.

[3] A. Apostolico, The myriad virtues of subword trees, in: Combinatorial Algorithms on Words, Springer-Verlag, Berlin, 1985, pp. 85–96.

[4] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proc. Latin American Theoretical Informatics, 2000, pp. 88–94.

[5] J. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, in: Proc. ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 360–369.

[6] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital Systems Research Center, 1994.

[7] W.I. Chang, E.L. Lawler, Sublinear approximate string matching and biological applications, Algorithmica 12 (4–5) (1994) 327–344.

[8] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucl. Acids Res. 27 (1999) 2369–2376.

[9] A.L. Delcher, A. Phillippy, J. Carlton, S.L. Salzberg, Fast algorithms for large-scale genome alignment and comparison, Nucl. Acids Res. 30 (11) (2002) 2478–2483.

[10] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. IEEE Symposium on Foundations of Computer Science, 2000, pp. 390–398.

[11] P. Ferragina, G. Manzini, An experimental study of an opportunistic index, in: Proc. ACM-SIAM Symposium on Discrete Algorithms, 2001, pp. 269–278.

[12] P. Ferragina, G. Manzini, On compressing and indexing data, Technical Report TR-02-01, Dipartimento di Informatica, Università di Pisa, 2002.

[13] G. Gonnet, R. Baeza-Yates, T. Snider, New indices for text: PAT trees and PAT arrays, in: W. Frakes, R.A. Baeza-Yates (Eds.), Information Retrieval: Algorithms and Data Structures, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 66–82.

[14] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: Proc. ACM Symposium on the Theory of Computing, ACM Press, 2000, pp. 397–406.
[15] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, New York, 1997.
[16] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, Report CSE-98-4, Computer Science Division, University of California, Davis, 1998.
[17] M. Höhl, S. Kurtz, E. Ohlebusch, Efficient multiple genome alignment, Bioinformatics 18 (2002) S312–S320.
[18] W.-K. Hon, K. Sadakane, Space-economical algorithms for finding maximal unique matches, in: Proc. Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2373, Springer-Verlag, Berlin, 2002, pp. 144–152.
[19] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. International Colloquium on Automata, Languages and Programming, in: Lecture Notes in Computer Science, vol. 2719, Springer-Verlag, Berlin, 2003, pp. 943–955.
[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, Berlin, 2001, pp. 181–192.
[21] D.K. Kim, J.S. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: Proc. Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2676, Springer-Verlag, Berlin, 2003, pp. 186–199.
[22] J. Knight, D. Gusfield, J. Stoye, The strmat software-package, 1998, http://www.cs.ucdavis.edu/~gusfield/strmat.tar.gz.
[23] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: Proc. Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2676, Springer-Verlag, Berlin, 2003, pp. 200–210.
[24] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: Proc. Symposium on Foundations of Computer Science, IEEE Computer Society, 1999, pp. 596–604.
[25] S. Kurtz, Reducing the space requirement of suffix trees, Software—Practice and Experience 29 (13) (1999) 1149–1171.
[26] S. Kurtz, A time and space efficient algorithm for the substring matching problem, Technical Report, Zentrum für Bioinformatik, Universität Hamburg, 2003.
[27] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich, REPuter: the manifold applications of repeat analysis on a genomic scale, Nucl. Acids Res. 29 (22) (2001) 4633–4642.
[28] E.S. Lander, L.M. Linton, B. Birren, C. Nusbaum, M.C. Zody, J. Baldwin, K. Devon, K. Dewar, et al., Initial sequencing and analysis of the human genome, Nature 409 (2001) 860–921.
[29] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[30] C. O'Keefe, E. Eichler, The pathological consequences and evolutionary implications of recent human genomic duplications, in: Comparative Genomics, Kluwer, Dordrecht, 2000, pp. 29–46.
[31] K. Sadakane, Succinct representations of *lcp* information and improvements in the compressed suffix arrays, in: Proc. ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 225–232.
[32] P. Weiner, Linear pattern matching algorithms, in: Proc. IEEE Annual Symposium on Switching and Automata Theory, The University of Iowa, 1973, pp. 1–11.
[33] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.
[34] J. Ziv, A. Lempel, Compression of individual sequences via variable length coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.