In Proceedings of the Ninth International Symposium
on String Processing and Information Retrieval,
Springer-Verlag, Lecture Notes in Computer Science, 2002

Optimal exact string matching based on suffix arrays

Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz

Faculty of Technology, University of Bielefeld, P.O. Box 10 01 31, 33501 Bielefeld, Germany. Email: {mibrahim,enno,kurtz}@TechFak.Uni-Bielefeld.DE

Abstract. Using the suffix tree of a string S, decision queries of the type "Is P a substring of S?" can be answered in $O(|P|)$ time and enumeration queries of the type "Where are all z occurrences of P in S?" can be answered in $O(|P|+z)$ time, totally independent of the size of S. However, in large scale applications as genome analysis, the space requirements of the suffix tree are a severe drawback. The suffix array is a more space economical index structure. Using it and an additional table, Manber and Myers (1993) showed that decision queries and enumeration queries can be answered in $O(|P|+\log|S|)$ and $O(|P|+\log|S|+z)$ time, respectively, but no optimal time algorithms are known. In this paper, we show how to achieve the optimal $O(|P|)$ and $O(|P|+z)$ time bounds for the suffix array. Our approach is not confined to exact pattern matching. In fact, it can be used to efficiently solve all problems that are usually solved by a top-down traversal of the suffix tree. Experiments show that our method is not only of theoretical interest but also of practical relevance.

1 Introduction

The suffix tree of a sequence S can be computed and stored in $O(n)$ time and space [13], where $n = |S|$. Once constructed, it allows one to answer queries of the type "Is P a substring of S?" in $O(m)$ time, where $m = |P|$. Furthermore, all z occurrences of a pattern P can be found in $O(m+z)$ time, totally independent of the size of S. Moreover, typical string processing problems like searching for all repeats in S can be efficiently solved by a bottom-up traversal of the suffix tree of S. These properties are most convenient in a "myriad" of situations [2], and Gusfield devotes about 70 pages of his book [8] to applications of suffix trees.

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 the recently improved implementations (see, e.g., [10]) of linear time constructions still require $20n$ bytes in the worst case. (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. On the other hand, the suffix array (introduced in [12] and in [6] under the name PAT array) is a more space efficient data structure than the suffix tree. It requires only $4n$ bytes in its basic form. However, at first glance, it seems that the suffix array has two disadvantages over the suffix tree:

(1) The direct construction of the suffix array takes $O(n \cdot \log n)$ time.
(2) 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 $O(m \cdot \log n)$ time in the worst case to answer decision queries.

Let us briefly comment on the two seemingly drawbacks:

(1) The $O(n \cdot \log n)$ time bound for the direct construction of the suffix array is not a real drawback, neither from a theoretical nor from a practical point of view. The suffix array of S can be constructed in $O(n)$ time in the worst case by first constructing the suffix tree of S; see [8]. However, in practice the improved $O(n \cdot \log n)$ time algorithm of [11] to directly construct the suffix array is reported to be more efficient than building it indirectly in $O(n)$ time via the suffix tree.

(2) We strongly believe that every algorithm using a suffix tree can be replaced with an equivalent algorithm based on a suffix array and additional information. As an example. let us look at the exact pattern matching problem. Using an additional table, Manber and Myers [12] showed that decision queries can be answered in $O(m + \log n)$ time in the worst case. However, no $O(m)$ 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 $O(m)$ time and how to find all z occurrences of a pattern P in optimal $O(m+z)$ time. This new result is achieved by using the basic suffix array enhanced with two additional tables; each can be computed in linear time and requires only $4n$ bytes. In practice each of these tables can even be stored in n bytes without loss of performance. 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. Thus, our method can efficiently solve all problems that are usually solved by a top-down traversal of the suffix tree. By taking the approach of Kasai et al. [9] one step further, it is also possible to efficiently solve all problems with enhanced suffix arrays that are usually solved by a bottom-up traversal of the suffix tree; see Abouelhoda et al. [1] for details.

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 [7] and the so-called opportunistic data structure devised by Ferragina and Manzini [4]. These data structures reduce the space consumption considerably. However, due to the compression, these 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. Worse, experimental results [5] 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 [5], 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. Moreover, experimental results show that our method can compete with the method of [12]. In case of DNA sequences, it is even 1.5 times faster than the method of [12]. Therefore, it is not only of theoretical interest but also of practical relevance.

2 Basic notions

In order to fix notation, we briefly recall some basic concepts. Let S be a string of length $|S| = n$ over an ordered alphabet 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 $0 \leq i < n$. For $i \leq j$, $S[i..j]$ denotes the substring of S starting with the character at position i and ending with the character at position j.

The *suffix array* suftab 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, where $S_i = S[i..n-1]\$$ denotes the ith nonempty suffix of the string $S\$$, $0 \leq i \leq n$. The suffix array requires $4n$ bytes. The direct construction of the suffix array takes $O(n \cdot \log n)$ time [12], but it can be build in $O(n)$ time via the construction of the suffix tree; see, e.g., [8].

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 $1 \leq i \leq n$. Since $S_{\mathsf{suftab}[n]} = \$$, we always have $\mathsf{lcptab}[n] = 0$; see Fig. 1. 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 [9]. The lcp-table requires $4n$ bytes. However, in practice it can be implemented in little more than n bytes; see section 8.

3 The lcp-intervals of a suffix array

To achieve the goals outlined in the introduction, we need the following concepts.

Definition 1. *Interval $[i..j]$, $0 \leq i < j \leq n$, is an lcp-interval of lcp-value ℓ if*

1. $\mathsf{lcptab}[i] < \ell$,
2. $\mathsf{lcptab}[k] \geq \ell$ *for all k with $i + 1 \leq k \leq j$,*
3. $\mathsf{lcptab}[k] = \ell$ *for at least one k with $i + 1 \leq k \leq j$,*
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, $i + 1 \leq k \leq j$, 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 also called ω-interval.

	suf-	lcp-	cldtab			
i	tab	tab	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			`$`

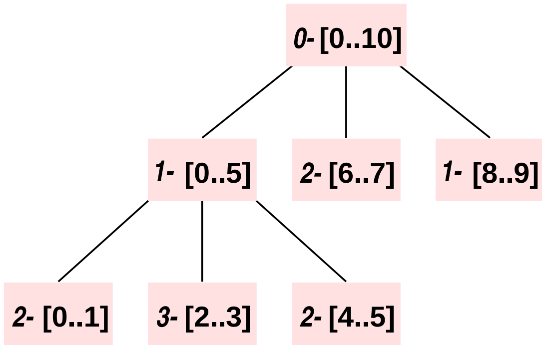

Fig. 1. Enhanced suffix array of the string $S =$ `acaaacatat$` and its lcp-interval tree. The fields 1, 2, and 3 of the cldtab denote the *up*, *down*, and *nextℓIndex* field, respectively; see Section 4. The encircled entries are redundant because they also occur in the *up* field.

Definition 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., $i \le l < r \le j$) and $m > \ell$.*[1] *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. 1. The lcp-interval tree is basically the suffix tree without leaves (note, however, that it is not our intention to build this 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 $l \in [i..j]$. The child intervals of an ℓ-interval can be computed according to the following lemma.

Lemma 3. *Let $[i..j]$ be an ℓ-interval. If $i_1 < i_2 < \ldots < i_k$ 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 $\mathsf{lcptab}[i_1] = \mathsf{lcptab}[i_2] = \ldots = \mathsf{lcptab}[i_k] = \ell$, 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.

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

Based on the analogy between the suffix array 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 two tables. In order to distinguish our new data structure from the basic suffix array, we call it the *enhanced suffix array.*

4 The enhanced suffix array

Our new data structure consists of the suffix array, the lcp-table, and an additional table: the child-table cldtab; see Fig. 1. The lcp-table was already presented in Section 2. 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 shall see later that it is possible to store the same information in only one field. Formally, the values of each cldtab-entry are defined as follows (we assume that $\min\emptyset = \max\emptyset = \perp$):

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

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 $i_1 < i_2 < \ldots < i_k$, the $\mathsf{cldtab}[i].down$ or $\mathsf{cldtab}[j+1].up$ value is used to determine the first ℓ-index i_1. The other ℓ-indices $i_2, \ldots i_k$ can be obtained from $\mathsf{cldtab}[i_1].next\ell Index$, $\ldots \mathsf{cldtab}[i_{k-1}].next\ell Index$, respectively. Once these ℓ-indices are known, one can determine all the child intervals of $[i..j]$ according to Lemma 3. As an example, consider the enhanced suffix array in Fig. 1. The 1-[0..5] interval has the 1-indices 2 and 4. The first 1-index 2 is stored in $\mathsf{cldtab}[0].down$ and $\mathsf{cldtab}[6].up$. The second 1-index is stored in $\mathsf{cldtab}[2].next\ell Index$. Thus, the child intervals of [0..5] are [0..1], [2..3], and [4..5]. In Section 6, 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.

5 Construction of the child-table

For clarity of presentation, we introduce *two* algorithms to construct the *up*/*down* values and the *nextℓIndex* value of the child-table separately. It is not difficult, however, to devise an algorithm that constructs the whole child-table in one scan of the lcptab. Both algorithms use a stack whose elements are indices of the lcptab. *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* is the topmost element of the stack. Algorithm 4 scans the lcptab 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 during the scan.

Algorithm 4 *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 $(\mathsf{lcptab}[i] \leq \mathsf{lcptab}[top]) \wedge (\mathsf{lcptab}[top] \neq \mathsf{lcptab}[lastIndex])$ **then**
        $\mathsf{cldtab}[top].down := lastIndex$
  if $\mathsf{lcptab}[i] \geq \mathsf{lcptab}[top]$ **then**
    if $lastIndex \neq -1$ **then**
      $\mathsf{cldtab}[i].up := lastIndex$
      $lastIndex := -1$
    $push(i)$

For a correctness proof, we need the following lemma.

Lemma 5. *The following invariants are maintained in the* **while***-loop of Algorithm 4: 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 $\mathsf{lcptab}[i_1] \leq \cdots \leq \mathsf{lcptab}[i_p]$. Moreover, if $\mathsf{lcptab}[i_j] < \mathsf{lcptab}[i_{j+1}]$, then for all k with $i_j < k < i_{j+1}$ we have $\mathsf{lcptab}[k] > \mathsf{lcptab}[i_{j+1}]$.*

Theorem 6. *Algorithm 4 correctly fills the up and down fields of the child-table.*

Proof. If the $\mathsf{cldtab}[top].down := lastIndex$ statement is executed, then we have $\mathsf{lcptab}[i] \leq \mathsf{lcptab}[top] < \mathsf{lcptab}[lastIndex]$ and $top < lastIndex < i$. Recall that $\mathsf{cldtab}[top].down$ 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 5, 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 $\mathsf{cldtab}[i].up := lastIndex$ statement is executed, then $\mathsf{lcptab}[top] \leq \mathsf{lcptab}[i] < \mathsf{lcptab}[lastIndex]$ and $top < lastIndex < i$. Recall that $\mathsf{cldtab}[i].up$ is the minimum of the set $M' = \{q \in [0..i-1] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[i]$ and $\forall k \in [q+1..i-1] : \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\}$. 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 $\mathsf{lcptab}[k] \geq \mathsf{lcptab}[lastIndex]$ 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 $\mathsf{lcptab}[lastIndex] \geq \mathsf{lcptab}[q'] > \mathsf{lcptab}[i] \geq \mathsf{lcptab}[top]$. 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 $next\ell Index$ field is easier. One merely has to check whether $\mathsf{lcptab}[i] = \mathsf{lcptab}[top]$ holds true. If so, then index i is assigned to the field $\mathsf{cldtab}[top].next\ell Index$. It is not difficult to see that Algorithms 4 and 7 construct the child-table in linear time and space.

Algorithm 7 *Construction of the $next\ell Index$ value.*

```
push(0)
for i := 1 to n do
  while lcptab[i] < lcptab[top]
    pop
  if lcptab[i] = lcptab[top] then
    lastIndex := pop
    cldtab[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 $1 \leq q \leq k+1$. In the up field of $\mathsf{cldtab}[r_1+1]$, $\mathsf{cldtab}[r_2+1], \ldots, \mathsf{cldtab}[r_k+1]$ 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 $\mathsf{cldtab}[r_k+1]$. This value can be stored in $\mathsf{cldtab}[r_k+1].next\ell Index$ because r_k+1 is the last ℓ-index and hence $\mathsf{cldtab}[r_k+1].next\ell Index$ is empty; see Fig. 1. However, if we do this, then for a given index i we must be able to decide whether $\mathsf{cldtab}[i].next\ell Index$ contains the next ℓ-index or the $\mathsf{cldtab}[i].down$ value. This can be accomplished as follows. $\mathsf{cldtab}[i].next\ell Index$ contains the next ℓ-index if $\mathsf{lcptab}[\mathsf{cldtab}[i].next\ell Index] = \mathsf{lcptab}[i]$, whereas it stores the $\mathsf{cldtab}[i].down$ value if $\mathsf{lcptab}[\mathsf{cldtab}[i].next\ell Index] > \mathsf{lcptab}[i]$. This follows directly from the definition of the $next\ell Index$ and $down$ field, respectively. Moreover, the memory cells of $\mathsf{cldtab}[i].next\ell Index$, which are still unused, can store the values of the up field. To see this, note that $\mathsf{cldtab}[i+1].up \neq \perp$ if and only if $\mathsf{lcptab}[i] > \mathsf{lcptab}[i+1]$. In this case, we have $\mathsf{cldtab}[i].next\ell Index = \perp$ and $\mathsf{cldtab}[i].down = \perp$. In other words, $\mathsf{cldtab}[i].next\ell Index$ is empty and can store the value $\mathsf{cldtab}[i+1].up$; see Fig. 1. Finally, for a given index i, one can decide whether $\mathsf{cldtab}[i].next\ell Index$ contains the value $\mathsf{cldtab}[i+1].up$ 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 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., $\min \ell Indices(i,j)$. This is possible with the help of the *up* and *down* fields of the child-table:

Lemma 8. *For every ℓ-interval $[i..j]$ the following statements hold:*

1. $i < \mathsf{cldtab}[j+1].up \leq j$ *or* $i < \mathsf{cldtab}[i].down \leq j$.
2. $\mathsf{cldtab}[j+1].up$ *stores the first ℓ-index in $[i..j]$ if* $i < \mathsf{cldtab}[j+1].up \leq j$.
3. $\mathsf{cldtab}[i].down$ *stores the first ℓ-index in $[i..j]$ if* $i < \mathsf{cldtab}[i].down \leq j$.

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, $\mathsf{cldtab}[j+1].up = \min \ell Indices(i,j)$, and consequently $i < \mathsf{cldtab}[j+1].up \leq j$. 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 $\mathsf{cldtab}[i].down = \min \ell Indices(i,j)$. Hence, $i < \mathsf{cldtab}[i].down \leq j$.
(2) If $i < \mathsf{cldtab}[j+1].up \leq j$, then the claim follows from $\mathsf{cldtab}[j+1].up = \min\{q \in [i+1..j] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[j+1], \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\ \forall k \in [q+1..j]\} = \min\{q \in [i+1..j] \mid \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\ \forall k \in [q+1..j]\} = \min \ell Indices(i,j)$.
(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 $\mathsf{lcptab}[q] \geq \ell > \mathsf{lcptab}[i]$ 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 $i_2 < i_3 < \ldots < i_k$ in $[i..j]$, where $1 \leq k \leq |\Sigma|$, are obtained successively from the *nextℓIndex* field of $\mathsf{cldtab}[i_1], \mathsf{cldtab}[i_2], \ldots, \mathsf{cldtab}[i_{k-1}]$. 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 3. 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 9 *getChildIntervals, applied to an lcp-interval* $[i..j] \neq [0..n]$.

```
intervalList = [ ]
if i < cldtab[j+1].up ≤ j then
   i1 := cldtab[j+1].up
else i1 := cldtab[i].down
add(intervalList, (i, i1 − 1))
while cldtab[i1].nextℓIndex ≠ ⊥ do
   i2 := cldtab[i1].nextℓIndex
   add(intervalList, (i1, i2 − 1))
   i1 := i2
add(intervalList, (i1, j))
```

The function *getChildIntervals* runs in constant time, provided the alphabet size is constant. 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 $\perp$.

With the help of Lemma 8, 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 $i < \mathsf{cldtab}[j+1].up \leq j$, then $getlcp(i, j)$ returns the value $\mathsf{lcptab}[\mathsf{cldtab}[j+1].up]$, otherwise it returns the value $\mathsf{lcptab}[\mathsf{cldtab}[i].down]$.

7 Answering queries in optimal time

As already mentioned in the introduction, given the basic suffix array, it takes $O(m \cdot \log n)$ time in the worst case to answer decision queries. By using an additional table (similar to the lcp-table), this time complexity can be improved to $O(m+\log n)$; see [12]. 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 and enumeration queries for P in optimal $O(m)$ and $O(m + z)$ time, respectively, where z is the number of occurrences of P in S.

Algorithm 10 *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. 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 $\ell \geq m$, 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 $\ell' \geq m$, 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.

8 Implementation details

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 ≥ 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 $\mathsf{lcptab}[i] \geq 255$, ordered by the first component. At index i of table lcptab_1 we store 255 whenever, $\mathsf{lcptab}[i] \geq 255$. This tells us that 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.

In cldtab we store relative indices. For example, if $j = \mathsf{cldtab}[i].next\ell Index$, 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 cldtab. The values ≥ 255 are not stored. Instead, if we encounter the value 255 in cldtab, 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 [12, page 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 cldtab is thus only $6n$ bytes.

Additionally, we use an extra bucket table. For a given parameter q, we store for each string w of length q 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 $m \leq q$ in constant time. For larger queries, this bucket table allows us to locate the interval containing the q-character prefix $P[0..q-1]$ 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 $[0..n]$. The advantage of this hybrid method is that only a small part of the suffix array is actually accessed. In particular, we only rarely access a field with value 255 in cldtab.

9 Experimental results

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

1. *ecoli* is the complete genome of the bacterium *Escherichia coli*, i.e., a DNA sequence of length 4,639,221. The alphabet size is 4.
2. *yeast* is 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.
3. *swiss* is a collection of protein sequences from the Swissprot database. The total size of all protein sequences is 2,683,054. The alphabet size is 20.
4. *shaks* is a collection of the complete works of William Shakespeare. The total size is 5,582,655 bytes. The alphabet size is 92.

We use the algorithm of [3] to sort suffixes, i.e., to compute table suftab. Table lcptab is constructed as a by-product of the sorting. The construction of the enhanced suffix array (including storage on file) requires: 6.6 sec. and 21 MB RAM for *ecoli*, 27 sec. and 51 MB RAM for *yeast*, 7 sec. and 13 MB for *swiss*, 7 sec. and 32 MB for *shaks*. These and all other timings include system time and refer to a computer with a 933 MHz Pentium PIII Processor and 512 MB RAM, running Linux. We ran three different programs for answering enumeration queries:

1. *stree* is based on an improved linked list suffix tree representation as described in [10]. Searching for a pattern and enumerating the z occurrences takes $O(m+z)$ time. The space requirement is $12.6n$ bytes for *ecoli* and *yeast*, $11.6n$ bytes for *swiss*, and $9.6n$ bytes for *shaks*.
2. *mamy* is based on suffix arrays and uses the algorithm of [12, page 937]. We used the original program code developed by Gene Myers. Searching for a pattern and enumerating its occurrences takes $O(m \log n + z)$ time. The space requirement is $4n$ bytes for all files.
3. *esamatch* is based on enhanced suffix arrays (tables suftab, lcptab, cldtab) and uses Algorithm 10. Searching a pattern takes $O(m+z)$ time. The space requirement is $6n$ bytes.

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

Table 1 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

	minpl = 20, *maxpl* = 30			*minpl* = 30, *maxpl* = 40			*minpl* = 40, *maxpl* = 50		
	stree	*mamy*	*esamatch*	*stree*	*mamy*	*esamatch*	*stree*	*mamy*	*esamatch*
file	*time*	*time*	*time*	*time*	*time*	*time*	*time*	*time*	*time*
ecoli	7.40	4.86	$\underline{3.09}$	7.47	5.00	$\underline{3.23}$	7.63	5.12	$\underline{3.35}$
yeast	8.97	5.18	$\underline{3.41}$	9.16	5.35	$\underline{3.53}$	9.20	5.43	$\underline{3.66}$
swiss	10.53	3.40	$\underline{3.34}$	10.47	3.53	$\underline{3.40}$	10.55	3.65	$\underline{3.45}$
shaks	44.55	$\underline{3.43}$	28.54	18.45	$\underline{3.47}$	27.14	13.15	$\underline{3.58}$	27.00

Table 1. Running times (in seconds) for one million enumeration queries searching for exact patterns in the input strings.

be answered. However, at least in the field of genomics, it is relevant; see [8]. For example, when comparing two genomes it is necessary to match all substrings of one genome against all substrings of the other genome, and this requires to answer millions of enumeration queries in very short time.

The smallest running times in Table 1 are underlined. 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.

The running time of all three programs is only slightly dependent on the size of the input strings and the length of the pattern. The only exception is *stree* applied to *shaks*, where the running time increases by a factor of about 2.5, when searching for smaller patterns. This is due to the fact that there are many patterns of length between 20 and 30 that occur very often in *shaks* (for example, lines that consist solely of white spaces). Enumerating their occurrences requires to traverse substantial parts of the suffix tree, which are often far apart in main memory. This slows down the enumeration. In contrast, in the suffix array the positions to be enumerated are stored in one consecutive memory area. As a consequence, for *esamatch* and *mamy* enumerating occurrences requires virtually no extra time. As expected, the running times of *stree* and *esamatch* depend on the alphabet size, while *mamy* shows basically the same speed for all files. For *shaks* it is much faster than the other programs, due to the large alphabet. For the other files, *esamatch* is always more than twice as fast as *stree* and slightly faster than *mamy* (1.5 times faster for DNA). This shows that *esamatch* is not only of theoretical interest.

Acknowledgments. We thank Dirk Strothmann, who observed that the values of the *up* field can also be stored in the $next\ell Index$ field of the child-table. Gene Myers provided his code for constructing and searching suffix arrays.

References

1. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The Enhanced Suffix Array and its Applications to Genome Analysis. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics*. Springer Verlag, Lecture Notes in Computer Science, accepted for publication, 2002.
2. A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words, Springer Verlag*, pages 85–96, 1985.
3. J. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.
4. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.
5. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Symposium on Discrete Algorithms*, pages 269–278, 2001.
6. G. Gonnet, R. Baeza-Yates, and T. Snider. New Indices for Text: PAT trees and PAT arrays. In W. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval: Algorithms and Data Structures*, pages 66–82. Prentice-Hall, Englewood Cliffs, NJ, 1992.
7. R. Grossi and J.S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *ACM Symposium on the Theory of Computing (STOC 2000)* , pages 397–406. ACM Press, 2000.
8. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.
9. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, July 2001, Lecture Notes in Computer Science 2089, Springer Verlag*, pages 181–192, 2001.
10. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.
11. N.J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, 1999.
12. U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5):935–948, 1993.
13. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The University of Iowa, 1973.