Proceedings of the Second Workshop on Algorithms in Bioinformatics,
Springer-Verlag, Lecture Notes in Computer Science, 2002

The enhanced suffix array and its applications to genome analysis

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch

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

Abstract. In large scale applications as computational genome analysis, the space requirement of the suffix tree is a severe drawback. In this paper, we present a uniform framework that enables us to systematically replace every string processing algorithm that is based on a bottom-up traversal of a suffix tree by a corresponding algorithm based on an enhanced suffix array (a suffix array enhanced with the lcp-table). In this framework, we will show how maximal, supermaximal, and tandem repeats, as well as maximal unique matches can be efficiently computed. Because enhanced suffix arrays require much less space than suffix trees, very large genomes can now be indexed and analyzed, a task which was not feasible before. Experimental results demonstrate that our programs require not only less space but also much less time than other programs developed for the same tasks.

1 Introduction

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.

The repetitive elements 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 [14]. 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 [17]. 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.

The repeat contents of a genome can be very large. For example, 50% of the 3 billion bp 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 [14]. Clearly, one needs extensive algorithmic support for a systematic study of repetitive DNA on a genomic scale. The software tool *REPuter* has

originally been developed for this task [13]. The core algorithm of *REPuter* locates maximal exact repeats of a sequence in linear space and time. Although it is based on an efficient and compact implementation of suffix trees [12], the space consumption is still quite large: it requires 12.5 bytes per input character in practice. Moreover, in large scale applications, the suffix tree suffers from a poor locality of memory reference, which causes a significant loss of efficiency on cached processor architectures.

The same problem has been identified in the context of genome comparison. Nowadays, the DNA sequences of entire genomes are being determined at a rapid rate. For example, the genomes of several strains of *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* [5] 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. A *MUM* is a sequence that occurs exactly once in genome S_1 and once in genome S_2, and is not contained in any longer such sequence. (We will show in this paper that *MUM*s are special supermaximal repeats.) 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 is a major problem when comparing large genomes; see [5].

To sum up, although the suffix tree is undoubtedly one of the most important data structures in sequence analysis [2, 7], its space consumption is the bottleneck in genome analysis tasks, whenever the sequences under consideration are large genomes.

More space efficient data structures than the suffix tree exist. The most prominent one is the *suffix array* [16] which requires only $4n$ bytes (4 bytes per input character) in its basic form. In most practical applications, it can be stored in secondary memory because no random access to it is necessary. The direct construction of the suffix array takes $O(n \cdot \log n)$ time in the worst case, but in practice it can be constructed as fast as a suffix tree [15]. Furthermore, it is known that pattern matching based on suffix arrays can compete with pattern matching based on suffix trees; see [1, 16] for details. However, efficient algorithms on suffix arrays that solve typical string processing problems like searching for all repeats in a string have not yet been devised.

In this paper, we present a uniform framework that enables us to systematically replace a string processing algorithm that is based on a bottom-up traversal of a suffix tree by a corresponding algorithm that is based on an enhanced suffix array (a suffix array enhanced with the lcp-table).[1] This approach has several advantages:

[1] It is also possible to replace every string processing algorithm based on a top-down traversal of a suffix tree by an algorithm on an enhanced suffix array; see [1].

1. The algorithms are more space efficient than the corresponding ones based on the suffix tree because our implementation of the enhanced suffix array requires only $5n$ bytes.
2. Experiments show that the running times of the algorithms are much better than those based on the suffix tree.
3. The algorithms are easier to implement on enhanced suffix arrays than on suffix trees.

First, we introduce the unified framework: the lcp-interval tree of a suffix array. Based on this framework, we then concentrate on algorithms that efficiently compute various kinds of repeats. To be precise, we will show how to efficiently compute maximal, supermaximal, and tandem repeats of a string S using a bottom-up traversal of the lcp-interval tree of S. We stress that this tree is only conceptual, in the sense that it is not really built during the bottom-up traversal (i.e., at any stage, only a representation of a small part of the lcp-interval tree resides in main memory). We implemented the algorithms and applied them to several genomes. Experiments show that our programs require less time and space than other programs for the same task. We would like to point out that our framework is not confined to the above-mentioned applications. For example, the off-line computation of the Lempel-Ziv decomposition of a string (which is important in data compression) can also be efficiently implemented using this framework.

2 Basic notions

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 substring $S[i..j]$ is also denoted by the *pair of positions* (i, j).

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 $S[i_1 - 1] \neq S[i_2 - 1]$[2] 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. A string is a *tandem repeat* if it can be written as $\omega\omega$ for some nonempty string ω. An *occurrence* of a tandem repeat $\omega\omega = S[p..p + |2\omega| - 1]$ is represented by the pair $(|\omega|, p)$. Such an occurrence $(|\omega|, p)$ is *branching* if $S[p + |\omega|] \neq S[p + 2|\omega|]$.

[2] 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.

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

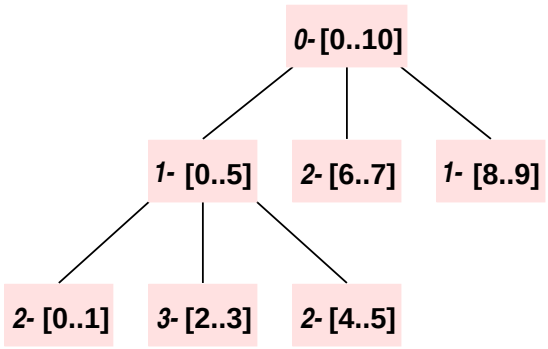

Fig. 1. Enhanced suffix array of the string $S =$ acaaacatat\$ and its lcp-interval tree. Table suftab^{-1} is introduced in Section 6.2.

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, $S_{\text{suftab}[0]}, S_{\text{suftab}[1]}, \ldots, S_{\text{suftab}[n]}$ 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 $S\$$, $0 \le i \le n$. The suffix array requires $4n$ bytes. The direct construction of the suffix array takes $O(n \cdot \log n)$ time [16], but it can be build in $O(n)$ time via the construction of the suffix tree; see, e.g., [7].

bwtab is a table of size $n+1$ such that for every $i, 0 \le i \le n$, $\text{bwtab}[i] = S[\text{suftab}[i]-1]$ if $\text{suftab}[i] \neq 0$. $\text{bwtab}[i]$ is undefined if $\text{suftab}[i] = 0$. bwtab is the *Burrows and Wheeler* transformation [4] known from data compression. It is stored in n bytes and constructed in one scan over suftab in $O(n)$ time.

The lcp-table lcptab is an array of integers in the range 0 to n. We define $\text{lcptab}[0] = 0$ and $\text{lcptab}[i]$ is the length of the longest common prefix of $S_{\text{suftab}[i-1]}$ and $S_{\text{suftab}[i]}$, for $1 \le i \le n$. Since $S_{\text{suftab}[n]} = \$$, we always have $\text{lcptab}[n] = 0$; 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 in the worst case. However, in practice it 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 $i \in [1, n]$, $\text{lcptab}_1[i] = \max\{255, \text{lcptab}[i]\}$. There are usually only few entries in lcptab that are larger than or equal to ≥ 255; see Section 7. To access these efficiently, we store them in an extra table llvtab. This contains all pairs $(i, \text{lcptab}[i])$ such that $\text{lcptab}[i] \ge 255$, ordered by the first component. At index i of table lcptab_1 we store 255 whenever $\text{lcptab}[i] \ge 255$. 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 $\text{lcptab}[i]$ in $O(\log_2 |\text{llvtab}|)$ time.

3 The lcp-interval tree of a suffix array

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.*

As an example, consider the table in Fig. 1. $[0..5]$ is a 1-interval because $\mathsf{lcptab}[0] = 0 < 1$, $\mathsf{lcptab}[5+1] = 0 < 1$, $\mathsf{lcptab}[k] \geq 1$ for all k with $1 \leq k \leq 5$, and $\mathsf{lcptab}[2] = 1$. Furthermore, $\ell Indices(0,5) = \{2,4\}$.

Kasai et al. [9] presented a linear time algorithm to simulate the bottom-up traversal of a suffix tree with a suffix array combined with the 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 2, *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.

Algorithm 2 *Computation of lcp-intervals (adapted from Kasai et al. [9]).*

```
push(⟨0, 0, ⊥⟩)
for i := 1 to n do
  lb := i − 1
  while lcptab[i] < top.lcp
    top.rb := i − 1
    interval := pop
    report(interval)
    lb := interval.lb
  if lcptab[i] > top.lcp then
    push(⟨lcptab[i], lb, ⊥⟩)
```

Here, we will take the approach of Kasai et al. [9] one step further and introduce the concept of an lcp-interval tree.

Definition 3. *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.,* $i \leq l < r \leq j$*) and* $m > \ell$*.*[3] *The* ℓ*-interval*

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

$[i..j]$ *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 $l \in [i..j]$. For instance, continuing the example of Fig. 1, 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 2.

Theorem 4. *Consider the* **for***-loop of Algorithm 2 for some index* i. *Let top be the topmost interval on the stack and* $(top-1)$ *be the interval next to it* (*note that* $(top-1).lcp < top.lcp$).

1. *If* $\mathsf{lcptab}[i] \leq (top-1).lcp$, *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 those* $\mathsf{lcptab}[i]$*-interval which contains* i.

An important consequence of Theorem 4 is the correctness of Algorithm 5. There, 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 $[c_1, \ldots, c_k]$.

Algorithm 5 *Traverse and process the lcp-interval tree*

```
lastInterval := ⊥
push(⟨0, 0, ⊥, [ ]⟩)
for i := 1 to n do
  lb := i − 1
  while lcptab[i] < top.lcp
    top.rb := i − 1
    lastInterval := pop
    process(lastInterval)
    lb := lastInterval.lb
    if lcptab[i] ≤ top.lcp then
      top.childList := add(top.childList, lastInterval)
      lastInterval := ⊥
  if lcptab[i] > top.lcp then
    if lastInterval ≠ ⊥ then
      push(⟨lcptab[i], lb, ⊥, [lastInterval]⟩)
      lastInterval := ⊥
    else push(⟨lcptab[i], lb, ⊥, [ ]⟩)
```

In Algorithm 5, the lcp-interval tree is traversed in a bottom-up fashion by a linear scan of the lcptab, while needed information is stored 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-information, i.e., there are no explicit parent-child pointers in our framework. In contrast to Algorithm 2, Algorithm 5 computes all lcp-intervals of the lcp-table *with* the child information. In the rest of the paper, we will show how to solve several problems merely by specifying the function *process* called on line 8 of Algorithm 5.

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

The algorithm of Gusfield [7, page 147] computes maximal repeated pairs in a sequence S. It runs in $O(kn + z)$ time where $k = |\Sigma|$ and 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 [13], based on space efficient suffix trees as described in [12]. In this section, we show how to implement the 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. The implementation requires tables suftab, lcptab, bwtab, 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 considerably reduced running time. This is verified in Section 7.

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., $\mathcal{P}_{[i..j]} = \{\mathsf{suftab}[r] \mid i \leq r \leq j\}$. 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

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

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 $b \in \Sigma \cup \{\bot\}$.

$[i'..j']$ is processed in the following way: First, maximal repeated pairs are output by combining the position set $\mathcal{P}^q_{[i..j]}(a)$, $a \in \Sigma \cup \{\bot\}$, with position sets $\mathcal{P}_{[i'..j']}(b)$, $b \in \Sigma \cup \{\bot\}$. 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 $p' \in \mathcal{P}_{[i'..j']}(b)$, $a, b \in \Sigma \cup \{\bot\}$ 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 maximal 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 these exist) are different. As a consequence, the output repeated pairs are maximal.

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

In Algorithm 5 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 5 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.

Algorithm 5 accesses the lcp-table in sequential order. Additionally, the maximal repeated pair algorithm does so with table suftab. Now consider the accesses to the input string S: Whenever suftab$[i]$ is processed, the input character $S[\mathsf{suftab}[i]-1]$ is accessed. Since bwtab$[i] = S[\mathsf{suftab}[i]-1]$, whenever suftab$[i] > 0$, the access to S can be replaced by an access to bwtab. Since suftab is accessed in sequential order, this also holds for bwtab. In other words, we do not need the input string when computing maximal repeated pairs. Instead we use table bwtab without increasing the total space requirement. The same technique is applied when computing supermaximal repeats; see Section 5.

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 $O(k)$ union operations, where $k = |\Sigma|$. Since $O(n)$ lcp-intervals have to be processed, the union and

add operations require $O(kn)$ time. Altogether, the algorithm runs in $O(kn+z)$ 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, which is 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 A new algorithm for finding supermaximal repeats

An ℓ-interval $[i..j]$ is called a *local maximum* in the lcp-table if $\mathsf{lcptab}[k] = \ell$ for all $i+1 \leq k \leq j$. It is not difficult to see that there is a one-to-one correspondence between the local maxima in the lcp-table and the leaves of the lcp-interval tree.

Lemma 6. *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 $\mathsf{bwtab}[i], \mathsf{bwtab}[i+1], \ldots, \mathsf{bwtab}[j]$ are pairwise distinct.*

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 $\mathsf{bwtab}[i], \mathsf{bwtab}[i+1], \ldots, \mathsf{bwtab}[j]$ are pairwise distinct characters. If so, report $S[\mathsf{suftab}[i]..\mathsf{suftab}[i] + \mathsf{lcptab}[i] - 1]$ as supermaximal repeat. The reader is invited to compare our simple algorithm with the one described in [7, page 146]. An experimental comparison of the two algorithms can be found in Section 7.

As an application of Lemma 6, we will show how to efficiently compute maximal unique matches. This is an important subtask in the software tool *MUMmer* [5], which aligns DNA sequences of whole genomes, and in the identification of X-patterns, which appear in the comparison of two bacterial genomes [6].

Definition 7. *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 8. *Let # be a unique separator symbol not occurring in S_1 and S_2 and let $S = S_1 \# S_2$. 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* $p = |S_1|$.

In *MUMmer*, *MUM*s are computed 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 $i+1 = j$ and $\mathsf{bwtab}[i] \neq \mathsf{bwtab}[j]$. If so, report $S[\mathsf{suftab}[i]..\mathsf{suftab}[i]+\mathsf{lcptab}[i]-1]$ as *MUM*. In Section 7, we also compare the performance of *MUMmer* with the implementation of the preceding algorithms.

6 Efficient detection of branching tandem repeats

This section is devoted to the computation of tandem repeats via enhanced suffix arrays. Stoye and Gusfield [18] described how all tandem repeats can be derived from branching tandem repeats by successive left-rotations. For this reason, we restrict ourselves to the computation of all branching tandem repeats.

6.1 A brute force method

The simplest method to find branching tandem repeats is to process all lcp-intervals. For a given ω-interval $[i..j]$ one checks whether there is a child interval $[l..r]$ (which may be a singleton interval) of $[i..j]$ such that $\omega\omega$ is a prefix of $S_{\mathsf{suftab}[q]}$ for each $q \in [l..r]$. Such a child interval can be detected in $O(|\omega| \log(j-i))$ time (if it exists), using the algorithm of [16] that searches for ω in $[i..j]$. It turns out that the running time of this algorithm is $O(n^2)$ (take, e.g., $S = a^n$). However, in practice this method is faster and more space efficient than other methods; see Section 7.

6.2 The optimized basic algorithm

The *optimized basic algorithm* of [18] computes all branching tandem repeats in $O(n \log n)$ time. It is based on a traversal of the suffix tree, in which each branching node α is annotated by its leaf list, i.e., by the set of leaves in the subtree below α. The leaf list of a branching node α corresponds to an lcp-interval in the lcp-interval tree. As a consequence, it is straightforward to implement the optimized basic algorithm via a traversal of the lcp-interval tree.

Algorithm 9 *For each ℓ-interval $[i..j]$ determine the child interval $[i_{\max}..j_{\max}]$ of maximal width among all child intervals of $[i..j]$. Then, for each q such that $i \leq q \leq i_{\max} - 1$ or $j_{\max} + 1 \leq q \leq j$, let $p = \mathsf{suftab}[q]$ and verify the following:*

(1) $p + 2\ell = n$ *or* $S[p+\ell] \neq S[p+2\ell]$ (*character-check*)
(2) $p + \ell = \mathsf{suftab}[h]$ *for some* $h, i \leq h \leq j$ (*forward-check*)
(3) $p - \ell = \mathsf{suftab}[h]$ *for some* $h, i \leq h \leq j$ (*backward-check*)

If (1) *and either* (2) *or* (3) *is satisfied, then output* (ℓ, p).

Algorithm 9 computes all branching tandem repeats. To analyze the efficiency, we determine the number of character-, backward-, and forward-checks. Since the largest child interval is always excluded, Algorithm 9 only performs verification steps for $O(n \log n)$ suffixes; see [18]. A character-check can easily be performed in constant time. For the forward- and backward-check the inverse suffix table suftab^{-1} is used. This is a table of size $n+1$, such that for any $q, 0 \leq q \leq n$, $\mathsf{suftab}^{-1}[\mathsf{suftab}[q]] = q$. Obviously, $p + \ell = \mathsf{suftab}[h]$, for some $h, i \leq h \leq j$, if and only if $i \leq \mathsf{suftab}^{-1}[p + \ell] \leq j$. Similarly, $p - \ell = \mathsf{suftab}[h]$ for some $h, i \leq h \leq j$, if and only if $i \leq \mathsf{suftab}^{-1}[p - \ell] \leq j$. Hence, every forward-check and backward-check requires constant time. The lcp-interval tree is processed in $O(n)$ time. For each interval, a child interval of maximal width can be determined in constant extra time. Therefore, Algorithm 9 runs in $O(n \log n)$ time. Processing the lcp-interval tree requires tables lcptab and suftab plus some space for the stack used during the bottom-up traversal. To perform the forward- and backward-checks in constant time, one also needs table suftab^{-1}, which requires $4n$ bytes. Hence our implementation of the optimized basic algorithm requires $9n$ bytes.

6.3 The improved $O(n \log n)$-algorithm

We improve Algorithm 9 by getting rid of the character-checks and reducing the number of forward- and backward checks. That is, we exploit the fact that for an occurrence $(|\omega|, p)$ of a branching tandem repeat $\omega\omega$, we have $S[p] = S[p + |\omega|]$. As a consequence, if $p + |\omega| = \mathsf{suftab}[q]$ for some q in the ω-interval $[i..j]$, p must occur in the child interval $[l_a..r_a]$ storing the suffixes of S that have ωa as a prefix, where $a = S[\mathsf{suftab}[i]] = S[p]$. This is formally stated in the following lemma.

Lemma 10. *The following statements are equivalent:*

(1) $(|\omega|, p)$ *is an occurrence of a branching tandem repeat* $\omega\omega$.
(2) $p + |\omega| = \mathsf{suftab}[q]$ *for some* q *in the* ω*-interval* $[i..j]$, *and* $p = \mathsf{suftab}[q_a]$ *for some* q_a *in the child interval* $[i_a..j_a]$ *representing the suffixes of* S *that have* ωa *as a prefix, where* $a = S[p]$.

This lemma suggests the following algorithm:

Algorithm 11 *For each* ℓ*-interval* $[i..j]$, *let* $a = S[\mathsf{suftab}[i]]$ *and determine the child interval* $[i_a..j_a]$ *of* $[i..j]$ *such that* $a = S[\mathsf{suftab}[i_a] + \ell]$. *Proceed according to the following cases:*

(1) *If* $j_a - i_a + 1 \leq i - j + 1 - (j_a - i_a + 1)$, *then for each* q, $i_a \leq q \leq j_a$, *let* $p = \mathsf{suftab}[q]$. *If* $p + \ell = \mathsf{suftab}[r]$ *for some* r, $i \leq r \leq i_a - 1$ *or* $j_a + 1 \leq r \leq j$, *then output* (ℓ, p).

(2) *If* $j_a - i_a + 1 > i - j + 1 - (j_a - i_a + 1)$*, then for each* q*,* $i \le q \le i_a - 1$ *or* $j_a + 1 \le q \le j$*, let* $p = \mathsf{suftab}[q]$*. If* $p - \ell = \mathsf{suftab}[r]$ *for some* r*,* $i_a \le r \le j_a$*, then output* (ℓ, p)*.*

One easily verifies for each interval $[l..r]$ that $l - r + 1 - (r_{\max} - l_{\max} + 1) \ge \min\{r_a - l + 1, l - r + 1 - (r_a - l_a + 1)\}$. Hence Algorithm 11 checks not more suffixes than Algorithm 9. Thus the number of suffixes checked is $O(n \log n)$. For each suffix either a forward-check or backward-check is necessary. This takes constant time. The lcp-interval tree is processed in $O(n)$ time. Additionally, for each interval the algorithm determines the child interval $[l_a..r_a]$. This takes constant extra time. Hence the running time of Algorithm 11 is $O(n \log n)$.

7 Experiments

7.1 Programs and data

In our experiments we applied the following programs:

- *REPuter* and *esarep* implement the algorithm of Gusfield (see Section 4) to compute maximal repeated pairs. While *REPuter* is based on suffix trees, *esarep* uses enhanced suffix arrays, as described in Section 4.
- *supermax* computes supermaximal repeats. It implements the algorithm described in Section 5.
- *unique-match* and *esamum* compute *MUM*s. *unique-match* is part of the original distribution of *MUMmer* (version 1.0) [5]. It is based on suffix trees. *esamum* is based on enhanced suffix arrays and uses the algorithm described at the end of Section 5.
- *BFA*, *OBA*, and *iOBA* compute branching tandem repeats. *BFA* implements the brute force algorithm described in Section 6. It uses the binary search algorithm of [16] to check for the appropriate child interval. *OBA* is an implementation of the optimized basic algorithm, and *iOBA* incorporates the improvements described in Section 6.

We applied the six programs for the detection of repeats to the genome of *E. coli* (4,639,221 bp) and the genome of *yeast* (12,156,300 bp). Additionally, we applied *unique-match* and *esamum* to the following pairs of genomes:

Mycoplasma 2: The complete genomes of *Mycoplasma pneumoniae* (816,394 bp) and of *Mycoplasma genitalium* (580,074 bp).

Streptococcus 2: The complete genomes of two strains of *Streptococcus pneumoniae* (2,160,837 bp and 2,038,615 bp).

***E. coli* 2:** The complete genomes of two strains of *E. coli* (4,639,221 bp and 5,528,445 bp).

For *E. coli* and *yeast* and for all pairs of genomes we constructed the corresponding enhanced suffix array (tables suftab, lcptab, bwtab, suftab^{-1}) once and stored each table on a separate file. The construction was done by a program

that is based on the suffix sorting algorithm of [3]. *REPuter* and *unique-match* construct the suffix tree in main memory (using $O(n)$ time) before they search for maximal repeated pairs and *MUM*s, respectively. All other programs 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.

Our method to construct the enhanced suffix array uses about 30% of the space required by *REPuter*, and 15% of the space required by *unique-match*. The construction time is about the same as the time to construct the suffix tree in *REPuter* and in *unique-match*.

7.2 Main experimental results

The results of applying the different programs to the different data sets are shown in Tables 1–3. The running times reported are for an Intel Pentium III computer with a 933 MHz CPU and 500 MB RAM running Linux. For a fair comparison, we report the running time of *REPuter* and of *unique-match* without suffix tree construction.

The running time of *supermax* 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 [10] 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 *supermax* requires 1.3 sec due to the large size of the output.

The comparison of *esarep* and *REPuter* underline the advantages of the enhanced suffix array over the suffix tree. *esarep* used about halve of the space of *REPuter* and it is 5 to 10 times faster. The performance gain is due to the improved cache behavior achieved by the linear scanning of the tables suftab, lcptab, and bwtab.

	E. coli (n = 4,639,221)					*yeast* (n = 12,156,300)				
ℓ	*maximal repeated pairs*			*supermax*		*maximal repeated pairs*			*supermax*	
	#reps	*REPuter*	*esarep*	#reps		#reps	*REPuter*	*esarep*	#reps	
18	11884	9.68	0.83	1676	0.15	306931	27.81	5.85	12939	0.42
20	7800	9.65	0.77	890	0.15	175456	27.70	4.07	6372	0.41
23	5207	9.68	0.74	635	0.14	84116	27.62	2.91	4041	0.40
27	3570	9.66	0.72	493	0.14	41401	27.62	2.39	2804	0.40
30	2731	9.66	0.71	449	0.14	32200	27.64	2.27	2367	0.40
40	841	9.67	0.69	280	0.14	20768	27.69	2.12	1669	0.40
50	608	9.66	0.68	195	0.14	16210	27.64	2.05	1349	0.40

Table 1. Running times (in sec.) for computing maximal repeated pairs and supermaximal repeats. The column titled #reps gives the number of repeats of length $\geq \ell$.

	E. coli ($n = 4{,}639{,}221$)				*yeast* ($n = 12{,}156{,}300$)			
ℓ	*#reps*	*BFA*	*OBA*	*iOBA*	*#reps*	*BFA*	*OBA*	*iOBA*
2	298853	1.32	7.83	1.60	932971	3.59	22.77	4.38
5	5996	1.32	4.73	1.47	32034	3.56	14.80	4.07
8	136	1.30	2.87	1.38	4107	3.54	8.72	3.87
11	20	0.84	1.20	1.13	1326	2.83	4.74	3.47
14	17	0.38	0.46	0.52	576	1.20	1.59	1.64

Table 2. Running times (in sec.) for computing branching tandem repeats. The column titled #reps gives the number of branching tandem repeats of length $\geq \ell$.

			unique-match		*esamum*	
genome pair	ℓ	*#MUM*s	*time*	*space*	*time*	*space*
Mycoplasma 2	20	10	1.85	65.26	0.03	7.99
Streptococcus 2	50	6613	6.76	196.24	0.29	29.71
E. coli 2	100	10817	17.67	472.47	0.46	62.59

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

The most surprising result of our experiments is the superior running time of the brute force algorithm to compute branching tandem repeats. *BFA* is always faster than the other two algorithms. *iOBA* is faster than *OBA* if $\ell \leq 11$, and slightly slower if $\ell = 14$. This is due to the fact that the additional access to the text, which is necessary to find the appropriate child interval $[i_a..j_a]$, outweighs the efficiency gain due to the reduced number of suffix checks.

We have also measured the running time of a program implementing the linear time algorithm of [8] to compute all tandem repeats. It is part of the strmat-package. For *E. coli* the program needs about 21 sec. (without suffix tree construction) and 490 MB main memory to deliver all tandem repeats of length at least 2. The linear time algorithm of [11] takes 4.7 sec. using 63 MB of main memory. For the same task our fastest program *BFA* (with an additional post processing step to compute non-branching tandem repeats from branching tandem repeats) requires 1.7 sec and 27 MB of main memory.

The running times and space results shown in Table 3 reveal that *esamum* is at least 20 times faster than *unique-match*, using only 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.

References

1. M.I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal Exact String Matching based on Suffix Arrays. In *Proceedings of the Ninth International Symposium on String Processing and Information Retrieval*. Springer-Verlag, Lecture Notes in Computer Science, 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. M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Systems Research Center, 1994.
5. A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Res.*, 27:2369–2376, 1999.
6. J. A. Eisen, J. F. Heidelberg, O. White, and S.L. Salzberg. Evidence for Symmetric Chromosomal Inversions Around the Replication Origin in Bacteria. *Genome Biology*, 1(6):1–9, 2000.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
8. D. Gusfield and 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.
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*, pages 181–192. Lecture Notes in Computer Science 2089, Springer-Verlag, 2001.
10. J. Knight, D. Gusfield, and J. Stoye. The Strmat Software-Package, 1998. http://www.cs.ucdavis.edu/ gusfield/strmat.tar.gz.
11. R. Kolpakov and G. Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *Symposium on Foundations of Computer Science*, pages 596–604. IEEE Computer Society, 1999.
12. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.
13. S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. *Nucleic Acids Res.*, 29(22):4633–4642, 2001.
14. E.S. Lander, L.M. Linton, B. Birren, C. Nusbaum, M.C. Zody, J. Baldwin, K. Devon, and K. Dewar, et. al. Initial Sequencing and Analysis of the Human Genome. *Nature*, 409:860–921, 2001.
15. N.J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, 1999.
16. 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.
17. C. O'Keefe and E. Eichler. The Pathological Consequences and Evolutionary Implications of Recent Human Genomic Duplications. In *Comparative Genomics*, pages 29–46. Kluwer Press, 2000.
18. J. Stoye and D. Gusfield. Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002.