1

Enhanced Suffix Arrays and Applications

Mohamed I. Abouelhoda
University of Ulm, Germany

Stefan Kurtz
University of Hamburg, Germany

Enno Ohlebusch
University of Ulm, Germany

1.1 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 [34], where $n = |S|$. Once constructed, it can be used to efficiently solve a "myriad" of string processing problems [3]. Table 1.1 shows the applications discussed in this chapter. 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

Therefore, Table 1.1 also shows which kind of traversal is used for the respective application.

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., [22].

0-8493-8597-0/01/$0.00+$1.50

	Type of tree traversal		
Application	bottom-up	top-down	suffix-links
supermaximal repeats	√		
maximal repeats	√		
maximal unique matches	√		
maximal multiple exact matches	√		
tandem repeats	√		
tandem repeats (brute force)		√	
exact pattern matching		√	
maximal exact matches (space efficient)		√	√

TABLE 1.1 The suffix tree applications discussed here and the kinds of traversals they require.

(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 [23] and the comparison of complete genomes [11, 16].

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 [27] and independently by Gonnet et al. [12] 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 [14]. Recently, it was shown independently and contemporaneously in [17, 19, 20] that a direct linear time construction of the suffix array is possible. However, the suffix array has less structure than the suffix tree, so that it is not clear that (and how) an algorithm using a suffix tree can be replaced with an algorithm based on a suffix array. In this chapter, we will show that the problems listed in Table 1.1 can also be solved with suffix arrays plus additional information. The algorithms presented here are not only more space efficient than previous ones, but they are also faster and easier to implement. In many applications the above-mentioned "additional information" consists of the longest common prefix (lcp) information. Kasai et al. [18] coined the name *virtual suffix tree* for a suffix array enhanced with the lcp information. However, there are also other applications that cannot be solved with this virtual suffix tree data structure. Thus, we will use the generic name *enhanced suffix array* for data structures consisting of the suffix array and additional tables representing the required information.

In Section 1.3, we treat applications (computation of supermaximal repeats and maximal unique matches) that are solely based on the properties of the enhanced suffix array.

In Section 1.4, we will introduce the concept of *lcp-interval tree*. 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. As examples of a bottom-up traversal, we will show how to compute all maximal repeated pairs of a string as well as all maximal multiple exact matches of a set of strings. These applications use the suffix array and the lcp-table, both of which can be stored in $4n$ bytes. In order to compute tandem repeats efficiently, one further needs the inverse suffix array; see Section 1.6.

In Section 1.5, we consider the exact pattern matching problem, which consists of finding all occurrences of a pattern P of length m in a string S of length n. We suppose that the enhanced suffix array of S has been constructed. It is well-known that the exact pattern matching problem can be solved by a binary search in the suffix array: A decision query "Is P a substring of S?" can be answered in $O(m \log n)$ time. Manber and Myers [27] showed how this can be improved to $O(m + \log n)$ running time using an additional table. This result is, however, only of theoretical interest, and therefore we only describe the

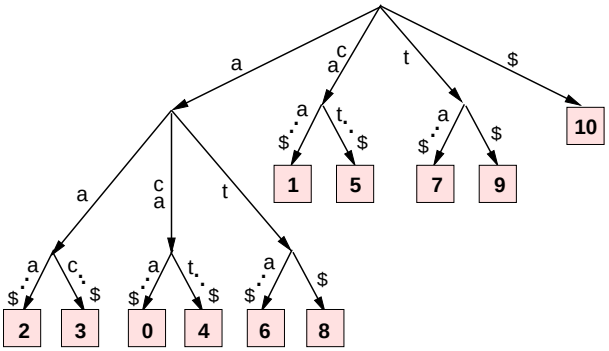

FIGURE 1.1: The suffix tree for $S = acaaacatat$.

$O(m \log n)$-method. Furthermore, we will show how to answer decision queries in optimal $O(m)$ time and how to find all z occurrences of a pattern P in optimal $O(m + z)$ time. These results are achieved by using the basic suffix array enhanced with the lcp-table and an additional table, called the child-table, that requires $4n$ bytes.

A space efficient algorithm for computing maximal exact matches of two strings (genomes) is presented in Section 1.7. Because it uses suffix links—a well-known feature of suffix trees—we will show how to incorporate the concept of suffix links 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.

We would like to point out 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; see [1] for implementation details. Experiments revealed that this space reduction entails no loss of performance; see [1].

1.2 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 sentinel 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 $i \leq j$, $S[i \ldots j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. The substring $S[i \ldots j]$ 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 $S_i = S[i \ldots n-1]\$$ denotes the i-th nonempty suffix of the string $S\$$, $0 \leq i \leq n$. Figure 1.1 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,

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

FIGURE 1.2: Suffix array of the string $S = acaaacatat$ enhanced with the lcp-table, the child-table, the suffix link table, the inverse suffix array, and the Burrows and Wheeler table bwttab. The child-table and the suffix link table will be explained later.

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; see Figure 1.2. The suffix array requires $4n$ bytes.

The *lcp-table* lcptab is an array of integers in the range 0 to n. We define $\mathsf{lcptab}[0] = 0$ and $\mathsf{lcptab}[i]$ to be 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 $\mathsf{lcptab}[n] = 0$. The lcp-table can be computed as a by-product during the construction of the suffix array (see,e.g., [17]), or alternatively, in linear time from the suffix array [18]. The lcp-table requires $4n$ bytes in the worst case.

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

The table bwttab contains the *Burrows and Wheeler transformation* [8] known from data compression. It is a table of size $n+1$ such that for every $i, 0 \leq i \leq n$, $\mathsf{bwttab}[i] = S[\mathsf{suftab}[i]-1]$ 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.

1.3 Computation of Supermaximal Repeats and Maximal Unique Matches

Motivation: Repeat Analysis

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. As we shall see later, this problem can also be reduced to the computation of certain types of repeats of the string that consists of the concatenated genomes.

The repetitive elements of the human genome can be 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

[25]. Transposable elements belong to one of the following four classes: SINEs (short interspersed nuclear elements), LINEs (long interspersed nuclear elements), LTRs (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*, a considerable portion of the genomes of higher organisms is composed of repeats. 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 [25]. 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 [14]. In this section we show how to locate all supermaximal repeats, while Section 1.4 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 $S[i_1 \ldots j_1] = S[i_2 \ldots j_2]$. 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 $\omega = S[i_1 \ldots j_1]$. A *supermaximal repeat* is a maximal repeat that does not occur as a substring of any other maximal repeat.

The lcp-Intervals of an Enhanced Suffix Array

We start this subsection with the introduction of the first essential concept of this chapter, 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 1.1 An 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 $\omega = S[\mathsf{suftab}[i] \ldots \mathsf{suftab}[i] + \ell - 1]$ 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. The size of an interval $[i..j]$ is $j - i + 1$.

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

As an example, consider the table in Figure 1.2. [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, 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.

Computation of Supermaximal Repeats

Next, we present an algorithm that computes all supermaximal repeats of a string. The reader is invited to compare our simple algorithm with the suffix-tree based algorithm of [14, page 146].

DEFINITION 1.2 An ℓ-interval $[i..j]$ is called a *local maximum* in the lcp-table if $\mathsf{lcptab}[k] = \ell$ for all $i+1 \leq k \leq j$.

For instance, in the lcp-table of Figure 1.2, the local maxima are the intervals [0..1], [2..3], [4..5], [6..7], and [8..9].

LEMMA 1.1 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 $\mathsf{suftab}[i], \mathsf{suftab}[i+1], \ldots, \mathsf{suftab}[j]$, $0 \leq i < j \leq n$, 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 $i+1 \leq k \leq j$. 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.

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.

Algorithm 1.3 *Computation of the supermaximal repeats of string S*

find all local maxima in the lcp-table of S
for *each local maximum $[i..j]$ in the lcp-table of S* **do**
  if $\mathsf{bwttab}[i], \mathsf{bwttab}[i+1], \ldots, \mathsf{bwttab}[j]$ *are pairwise distinct characters*
  then *report the string* $S[\mathsf{suftab}[i] \ldots \mathsf{suftab}[i] + \mathsf{lcptab}[i] - 1]$ *as supermaximal repeat*

Motivation: Comparison of Whole Genomes

To date (mid 2003), Genbank contains "complete" genomes for more than 1,000 viruses, over 100 microbes, and about 100 eukariota. This abundance of complete genomic *DNA*-sequences has boosted the field of comparative genomics. Comparative studies are emerging as a powerful tool for the identification of genes and regulatory elements. Comparing the genomes of related species gives us new insights into the complex structure of organisms at the *DNA*-level and protein-level.

The first step in the comparison of genomes is to produce an alignment, i.e., a colinear arrangement of sequence similarities. Alignment of nucleic or amino acid sequences has been one of the most important methods in sequence analysis. Nowadays, many sophisticated algorithms are available for aligning sequences with similar regions. These require to score all possible alignments (typically, the score is the sum of the similarity/identity values for the aligned symbols, minus a penalty for the introduction of gaps), in conjunction with a dynamic programming method to find optimal or near-optimal alignments according to this scoring scheme (see, e.g., [29]). The dynamic programming algorithms run in time proportional to the product of the lengths of the sequences. Hence they are not suitable for aligning entire genomes. Recently, several genome alignment programs have been developed, all using an anchor-based method to compute an alignment (for an overview see [9]). The anchor-based methods are composed of the following three phases:

(1) Computation of all potential anchors (exact or approximate matches).
(2) Computation of an optimal colinear sequence of non-overlapping potential anchors: these are the anchors that form the basis of the alignment.
(3) Closure of the gaps in between the anchors by applying the same method recursively—yielding a divide and conquer method—or, e.g., by a standard dynamic programming method.

In this chapter, we will focus on phase (1) and explain several algorithms to compute exact matches. The first one is used in the software tool *MUMmer* [11]. It is based on a maximal unique match (*MUM*) decomposition of two genomes S_1 and S_2. The implementation of *MUMmer* uses the suffix tree of $S_1\#S_2$ to compute *MUMs* 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.

Computation of Maximal Unique Matches

The space consumption of the suffix tree has been identified to be a major problem in the computation of the maximal unique match decomposition of two large genomes; see [11]. We will solve this problem by using the suffix array enhanced with the lcp-table.

DEFINITION 1.4 Given two strings S_1 and S_2, a *MUM* is a string that occurs exactly once in S_1 and once in S_2, and is not contained in any longer such string.

LEMMA 1.2 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 $u = S[i_1 \dots j_1] = S[i_2 \dots j_2]$,
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, $u = S_1[i_1 \dots j_1]$) and once in S_2 (say, $u = S_2[i_2 \dots j_2]$), 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* [11] computed *MUMs* 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.

Algorithm 1.5 *Computation of maximal unique matches of two strings S_1 and S_2*

find all local maxima in the lcp-table of $S = S_1 \# S_2$
for *each local maximum $[i..j]$ in the lcp-table of S* **do**
  if $i + 1 = j$ **and** $\mathsf{bwttab}[i] \neq \mathsf{bwttab}[j]$ **and** $\mathsf{suftab}[i] < p < \mathsf{suftab}[j]$
  then *report the string* $S[\mathsf{suftab}[i] \dots \mathsf{suftab}[i] + \mathsf{lcptab}[i] - 1]$ *as MUM*

The algorithms that compute supermaximal repeats and *MUMs* 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 [1] for details and experimental results.

1.4 Computation of Maximal Repeats and Maximal (Multiple) Exact Matches

Next, we introduce the second essential concept of this chapter—the lcp-interval tree.

DEFINITION 1.6 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 $m > \ell$.[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 Figure 1.3. 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 Figure 1.2, the child intervals of $[0..5]$ are $[0..1]$, $[2..3]$, and $[4..5]$.

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

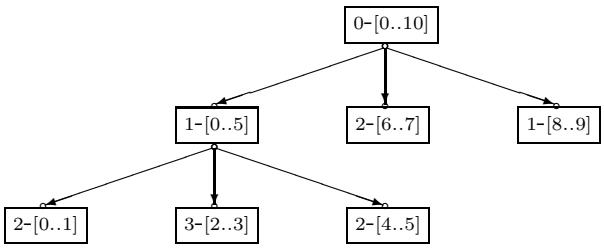

FIGURE 1.3: The lcp-interval tree of the string $S = acaaacatat$.

In Algorithm 1.7, 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-information, i.e., there are no explicit parent-child pointers in our framework. In Algorithm 1.7, 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 1.7 *Traverse and process the lcp-interval tree*

```
1  lastInterval := ⊥
2  push(⟨0, 0, ⊥, [ ]⟩)
3  for i := 1 to n do
4    lb := i − 1
5    while lcptab[i] < top.lcp
6      top.rb := i − 1
7      lastInterval := pop
8      process(lastInterval)
9      lb := lastInterval.lb
10     if lcptab[i] ≤ top.lcp then
11       top.childList := add(top.childList, lastInterval)
12       lastInterval := ⊥
13   if lcptab[i] > top.lcp then
14     if lastInterval ≠ ⊥ then
15       push(⟨lcptab[i], lb, ⊥, [lastInterval]⟩)
16       lastInterval := ⊥
17     else push(⟨lcptab[i], lb, ⊥, [ ]⟩)
```

In this chapter, several problems will be solved merely by specifying the function *process* called in line 8 of Algorithm 1.7.

Computation of Maximal Repeated Pairs

The computation of maximal repeated pairs plays an important role in the analysis of a genome. The algorithm of Gusfield [14, page 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 [23], based on space efficient suffix trees described in [22]. 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 1.3, 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; see [1].

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 $u = S[\mathsf{suftab}[i] \ldots \mathsf{suftab}[i] + \ell - 1]$. 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

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

The algorithm computes position sets in a bottom-up traversal. 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 q-th 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 positions 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 1.7, 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 1.7 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.

Let us 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.

Computation of Maximal (Multiple) Exact Matches

In this section, we come back to the problem of aligning genomic sized sequences. In Section 1.3, we have seen that the software tool *MUMmer* uses *MUMs* as potential anchors in the first phase of its anchor-based method. Delcher et al. [11] wrote: "The crucial principle behind this step is the following: if a long, perfectly matching sequence occurs exactly once in each genome, it is almost certain to be part of the global alignment." It has been argued in [16] that the restriction to *MUMs* as anchors seems unnecessary because exact matches occurring more than once in a genome may also be meaningful. This consideration lead to the notion of maximal exact matches.

DEFINITION 1.8 An *exact match* between two strings S_1 and S_2 is a triple (l, p_1, p_2) such that $p_1 \in [0, |S_1|-l]$, $p_2 \in [0, |S_2|-l]$, and $S_1[p_1 \ldots p_1+l-1] = S_2[p_2 \ldots p_2+l-1]$. An exact match is *left maximal* if $S[p_1-1] \neq S[p_2-1]$ and *right maximal* if $S[p_1+l] \neq S[p_2+l]$. A *maximal exact match* (*MEM*) is a left and right maximal exact match.

Of course, computing maximal exact matches between two strings S_1 and S_2 boils down to computing maximal repeated pairs of the string $S = S_1 \# S_2$. This is made precise in the following lemma.

LEMMA 1.3 Let # be a unique separator symbol not occurring in the strings S_1 and S_2 and let $S = S_1 \# S_2$. (l, p_1, p_2) is a *MEM* if and only if $((p_1, p_1 + l - 1), (p_2, p_2 + l - 1))$ is a maximal repeated pair of the string S such that $p_1 + l - 1 < p < p_2$, where $p = |S_1|$ is the position of # in S.

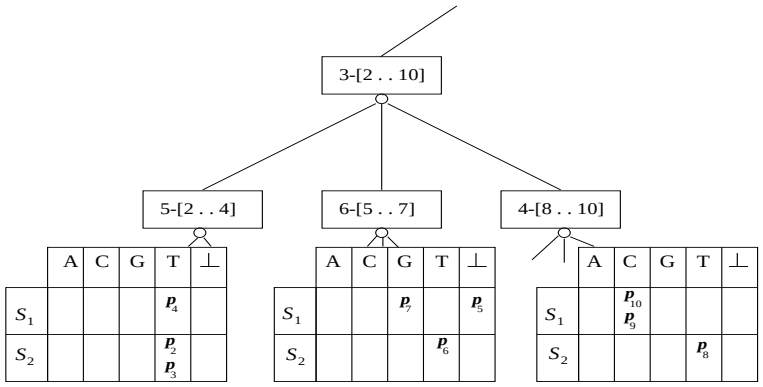

FIGURE 1.4: The position sets of a part of an lcp-interval tree. p_i denotes the position $\mathsf{suftab}[i]$.

Therefore, one can use the maximal repeated pair algorithm with the following modification to compute maximal exact matches. The position sets are divided into two disjoint and possibly empty subsets: One subsets contains all positions that correspond to S_1 (these are smaller than $|S_1|$). Another subset contains all positions that correspond to S_2 (these are greater than $|S_1|$). Let $\mathcal{P}_{[i..j]}(S_1, a)$ denote the set of all positions $p \in \mathcal{P}_{[i..j]}$ such that p corresponds to a position in S_1 and $S[p-1] = a \in \Sigma \cup \{\bot\}$. $\mathcal{P}_{[i..j]}(S_2, a)$ is defined analogously. (Figure 1.4 shows an example.) To compute maximal exact matches, the Cartesian product is build from each position set $\mathcal{P}_{[i..j]}(S_1, a)$, $a \in \Sigma \cup \{\bot\}$ and the position sets $\mathcal{P}_{[i..j]}(S_2, b)$, where $a \neq b \in \Sigma \cup \{\bot\}$. It is not difficult to see that this modification does not affect the time and space complexity of the algorithm.

However, if one uses *MEMs* instead of *MUMs* in a global alignment tool, then one is faced with the problem that the number of *MEMs* can be very large. (This is because the number of *MEMs* is determined by a Cartesian product of position sets.) Needless to say that this phenomenon occurs especially in the comparison of highly repetitive genomes. Clearly, *MEMs* that occur too many times should be excluded from the set of potential anchors. In our opinion, the exact definition of "too many times" should be left to the user of the global alignment tool. In other words, it should be a parameter of the program. The next definition makes this precise.

DEFINITION 1.9 Suppose (l, p_1, p_2) is a *MEM* and let $u = S_1[p_1 \ldots p_1 + l - 1]$ be the corresponding sequence. Define the set

$$MP_u = \{(p_1', p_2') \mid (l, p_1', p_2') \text{ is a } \textit{MEM} \text{ and } u = S_1[p_1' \ldots p_1' + l - 1]\}$$

of position pairs where the *MEMs* with sequence u start. Given a threshold $t \in \mathbb{N}$, the *MEM* (l, p_1, p_2) is called

- *infrequent in* S_1 if $r_1 := |\{p_1' \mid (p_1', p_2') \in MP_u\}|$ satisfies $r_1 \leq t$,
- *infrequent in* S_2 if $r_2 := |\{p_2' \mid (p_1', p_2') \in MP_u\}|$ satisfies $r_2 \leq t$,
- *infrequent* if $r := |MP_u|$ satisfies $r \leq t$.

It is our next goal to calculate the values r, r_1, and r_2. To this end, the following notation will be useful. For a position set $\mathcal{P}$, let $C_{\mathcal{P}}(S_1, a) = |\mathcal{P}(S_1, a)|$ and $C_{\mathcal{P}}(S_1) =$

$\sum_{a\in\Sigma\cup\{\perp\}} C_{\mathcal{P}}(S_1,a)$ (the values $C_{\mathcal{P}}(S_2,a)$ and $C_{\mathcal{P}}(S_2)$ are defined similarly). The value $C_{\mathcal{P}}(S_1)$ represents the number of repeats in S_1 from which the *MEMs* are derived. One can also impose constraints on this value.

For any lcp-interval $[i..j]$ that contains k child intervals, the value r can be calculated according to the following formula, where $q, q' \in [1,k]$:

$$r = \frac{1}{2} \sum_{a\in\Sigma\cup\{\perp\}} \sum_{q\neq q'} C_{\mathcal{P}^q_{[i..j]}}(S_1,a) \cdot (C_{\mathcal{P}^{q'}_{[i..j]}}(S_2) - C_{\mathcal{P}^{q'}_{[i..j]}}(S_2,a))$$

In the example of Figure 1.4, the calculation of the value r for the interval [2..10] yields $r = (0*1+0*1)+(2*2+2*1)+(1*2+1*1)+(1*0+1*0)+(1*2+1*1) = 12$.

For any lcp-interval $[i..j]$ that contains k child intervals, the value r_1 can be calculated by the formula:

$$r_1 = \sum_{a\in\Sigma\cup\{\perp\}} \sum_{q\in[1,k]} C'_{\mathcal{P}^q_{[i..j]}}(S_1,a)$$

where

$$C'_{\mathcal{P}^q_{[i..j]}}(S_1,a) = \begin{cases} 0, & \text{if } \forall q' \in [1,k], q' \neq q : C_{\mathcal{P}^{q'}_{[i..j]}}(S_2) = C_{\mathcal{P}^{q'}_{[i..j]}}(S_2,a) \\ C_{\mathcal{P}^q_{[i..j]}}(S_1,a) & \text{otherwise} \end{cases}$$

In the example of Figure 1.4, we have $r_1 = 0+2+1+0+1 = 4$. It is obvious how the algorithm for the computation of *MEMs* has to be modified to compute infrequent *MEMs*. Choosing an appropriate threshold on r (or a threshold on the combination of r_1, r_2, $C_{\mathcal{P}}(S_1)$, and $C_{\mathcal{P}}(S_2)$), the user can fine tune the computation of infrequent *MEMs*. Thus the concept of infrequent *MEMs* provides a reasonable compromise between *MUMs* and *MEMs*.

We would also like to comment on methods for multiple genome comparisons. As already mentioned, the *DNA* sequences of entire genomes are being determined at a rapid rate. Nowadays, it is quite common for a project to sequence the genome of an organism that is very closely related to another completed genome. For example, the genomes of several strains of the bacteria *E. coli* and *S. aureus* have already been completely sequenced. A global alignment of the genomes may help, for example, in understanding why a strain of a bacterium is pathogenic or resistant to antibiotics while another is not. Current software tools for multiple alignment of genomic sized sequences build a multiple alignment from pairwise alignments; see [6, 7, 28, 32]. However, if the organisms under consideration are closely related, then it makes sense to build a multiple alignment directly from exact matches that occur in each genome. The software tool *MGA* [16] is capable of aligning three or more closely related genomes. In the first phase of its anchor-based method, all *maximal multiple exact matches* (*multiMEMs*) longer than a specified minimum length are computed. The notion of a *multiMEM* is the natural extension of maximal exact matches to more than two genomes. Roughly speaking, a *multiMEM* is a sequence that occurs in all genomes to be aligned and cannot simultaneously be extended to the left or right in every genome. The maximal repeated pair algorithm can also be modified such that it computes (infrequent) *multiMEMs*. A detailed description of the resulting algorithm can be found in [24].

1.5 Exact Pattern Matching

In this section we consider the exact pattern matching problem. We suppose that the suffix array for S is available. We consider two methods: A practical method based on binary

searches in the suffix array (running in $O(m \log n)$ time), and an optimal method requiring only $O(m)$ time.

A Method Based on Binary Searches in the Suffix Array

Consider a pattern P of length m. To find all occurrences of P in S, we need to find all the suffixes of S that have P as a prefix. Since the suffix array for S stores the suffixes of $S\$$ in lexicographic order, all these suffixes are consecutive in the suffix array. Using a binary search, one can therefore compute the leftmost (i.e., smallest) position $l, 0 \leq l \leq n$, in the suffix array such that P is a prefix of $S_{\mathsf{suftab}[l]}$. If such an l does not exist, then P does not occur as a substring of S. Otherwise, one computes the rightmost (i.e., largest) position $r, 0 \leq r \leq n$, in the suffix array such that P is a prefix of $S_{\mathsf{suftab}[r]}$. Given l and r, all occurrences of P in S are computed as follows: Output $\mathsf{suftab}[j]$ for all $j, l \leq j \leq r$. This obviously requires $O(r - l)$ time.

Algorithm 1.10 presents a pseudo-code implementation for this method. In particular, the function *findleftmost* delivers the leftmost position l and the function *findrightmost* delivers the rightmost position r, given some boundaries l_0 and r_0 in the suffix array and some value h_0, such that all suffixes in the range l_0 to r_0 have a common prefix of length at least h_0. To reduce the number of character comparisons, the length of the longest common prefix of all suffixes between the current boundaries is computed from h_l and h_r. These two values are the lengths of the longest common prefix of P with $S_{\mathsf{suftab}[l]}$ and $S_{\mathsf{suftab}[r]}$, respectively. Hence $S_{\mathsf{suftab}[mid]}$ and P have a common prefix of length $\min\{h_l, h_r\}$, where $mid = \lfloor (l + r)/2 \rfloor$ is the midpoint between the current boundaries l and r. The longest common prefix can be skipped when comparing $S_{\mathsf{suftab}[mid]}$ and P using the function *compare*; see Algorithm 1.10.

The function *compare* performs the comparison of a suffix of $S\$$ with some string w of length m. More precisely, let $0 \leq i \leq n$ and $v = S_{\mathsf{suftab}[i]}$ and suppose that w and v have a common prefix of length q. Then $\mathit{compare}(w, m, i, q)$ delivers a pair (c, f_c) such that the following holds:

- c is the length of the longest common prefix of w and v. (Note that $c \geq q$.)
- If w is a prefix of v, i.e., $c = m$ holds, then $f_c = 0$.
- Otherwise, if w is not a prefix of v, then $w[c] \neq v[c]$. There are two cases
  - If $w[c] < v[c]$, then $f_c = -1$.
  - If $w[c] > v[c]$, then $f_c = 1$.

The function *sasearch* calls *findleftmost* and *findrightmost*. If the left boundary l is smaller than or equal to the right boundary r, then all start positions of the suffixes between these boundaries are reported. If (h_0, l_0, r_0) is set to $(0, 0, n)$, then both searches take $O(m \log_2 n)$ time. That is, the running time of *sasearch* is $O(m \log_2 n + z)$ where z is the number of occurrences of P in S.

Algorithm 1.10 *Search all occurrences of P in S using the suffix array for S.*

```
function findleftmost(P, m, h, l, r)
(h_l, f_l) := compare(P, m, l, h)
if f_l ≤ 0 then
    return l
(h_r, f_r) := compare(P, m, r, h)
if f_r > 0 then
    return r + 1
while r > l + 1 do
    mid := ⌊(l + r)/2⌋
    (c, f_c) := compare(P, m, mid, min{h_l, h_r})
    if f_c ≤ 0 then
        (h_r, r) := (c, mid)
    else
        (h_l, l) := (c, mid)
return r
```

```
function compare(w, m, i, q)
v := S_suftab[i]
c := q
while c < min{m, |v|} do
    if w[c] < v[c] then
        return (c, −1)
    else
        if w[c] > v[c] then
            return (c, 1)
        else
            c := c + 1
if c = m then
    return (c, 0)
else
    return (c, −1)
```

```
function findrightmost(P, m, h, l, r)
(h_l, f_l) := compare(P, m, l, h)
if f_l < 0 then
    return −1
(h_r, f_r) := compare(P, m, r, h)
if f_r ≥ 0 then
    return r
while r > l + 1 do
    mid := ⌊(l + r)/2⌋
    (c, f_c) := compare(P, m, mid, min{h_l, h_r})
    if f_c ≥ 0 then
        (h_l, l) := (c, mid)
    else
        (h_r, r) := (c, mid)
return l
```

```
function sasearch(P, m, h_0, l_0, r_0)
l := findleftmost(P, m, h_0, l_0, r_0)
r := findrightmost(P, m, h_0, l_0, r_0)
if l ≤ r then
    for j := l to r do
        print "match at pos" suftab[j]
```

Note that we do not always have to start the binary searches with the boundaries 0 and n. One can divide the suffix array into buckets such that each bucket contains all suffixes having the same prefix of length d, where d is a given parameter. Formally, one precomputes a table bcktab_d storing for each string $u \in \Sigma^*$ of length d a pair $\mathsf{bcktab}_d(u) := (l_d, r_d)$ of integer values. l_d and r_d are the leftmost and rightmost positions of all suffixes in the suffix array having prefix u. Table bcktab_d can be precomputed in $O(n)$ time in one scan over tables suftab and lcptab. Given a string u of length d, the pair $\mathsf{bcktab}_d(u)$ can easily be accessed by computing a unique integer code for u in the range 0 to $|\Sigma|^d - 1$. This requires $O(d)$ time.

Now suppose $m \geq d$ and let $(l_d, r_d) := \mathsf{bcktab}_d(P[0 \ldots d-1])$. If $l_d > r_d$, then P does not occur in S. Otherwise, all suffixes of S having prefix $P[0 \ldots d-1]$ occur between the boundaries l_d and r_d. Moreover, all suffixes between these boundaries have a common prefix of length d. Hence one can start the binary search with these boundaries and with prefix length d to search for the remaining suffix $P[d \ldots m-1]$ of P. In this way, the overall search time for P using *sasearch* becomes $O(d + (m-d)\log_2(r_d - l_d + 1))$ with an additional space consumption of $O(|\Sigma|^d)$ for table bcktab_d. In practice one chooses d such that bcktab_d can be stored in n bytes, so that the space requirement only increases by 25%.

The algorithm presented here can be improved to $O(m + \log n)$ by employing an extra

table containing the length of the longest common prefix for each pair of suffix array boundaries which may occur in a binary search; for details, see [27]. However, the asymptotic improvement does not lead to an improved running time in practice.

An Optimal Method with Additional Tables

In this subsection we will demonstrate how to answer decision queries "Is P a substring of S?" in optimal $O(m)$ time. The same method allows one to find all z occurrences of a pattern P in optimal $O(m+z)$ time. To achieve this time complexity, one must be able 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 Figure 1.2. The child-table is a table of size $n+1$ indexed from 0 to n and each entry contains three values: *up*, *down*, and $next\ell Index$. Each of these three values requires 4 bytes in the worst case, but it is possible to store the same information in only one byte; see [1] for details. Formally, the values of each childtab-entry are defined as follows (we assume that $\min\emptyset = \max\emptyset = \bot$):

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

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 $\mathsf{childtab}[i].down$ 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 1.4 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.

As an example, consider the enhanced suffix array in Figure 1.2. 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]. Before we show in detail how the child-table can be used to determine the child intervals of an lcp-interval in constant time, we address the problem of building the child-table efficiently.

Construction of the Child-Table

The childtab can be computed in linear time by a bottom-up traversal of the lcp-interval tree (Algorithm 1.7) as follows. 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*. If $k = 0$, then $[i..j]$ is a leaf in the lcp-interval tree. In this case, the ℓ-indices of $[i..j]$ are the indices $i+1, i+2, \ldots, j$. Otherwise, if $k > 0$, then the ℓ-indices of $[i..j]$ are the indices $l_2, \ldots, l_k$ plus all those indices from $[i..j]$ that are not contained in any of the child intervals (these indices correspond to singleton intervals). In our example from Figure 1.3, when the function *process* is applied to the 1-interval [0..5], its child intervals are [0..1], [2..3], and [4..5]; hence 2 and 4 are the 1-indices of [0..5]. Let $i_1, \ldots, i_p$ be the ℓ-indices of $[i..j]$, in ascending order. The first ℓ-index i_1 is assigned to $\mathsf{childtab}[j+1].up$ and $\mathsf{childtab}[i].down$. The other ℓ-indices $i_2, \ldots, i_p$ are stored in the $next\ell Index$ field of $\mathsf{childtab}[i_1]$, $\mathsf{childtab}[i_2], \ldots, \mathsf{childtab}[i_{p-1}]$, respectively. Finally, one has to determine the 0-indices $j_1, \ldots, j_q$ (in ascending order) of the interval $[0..n]$ (which remained on the stack) and store them in the $next\ell Index$ field of $\mathsf{childtab}[j_1]$, $\mathsf{childtab}[j_2], \ldots, \mathsf{childtab}[j_{q-1}]$. The resulting child-table is shown in Figure 1.2, where the fields 1, 2, and 3 of the childtab denote the *up*, *down*, and $next\ell Index$ field. However, the space requirement of the child-table can be reduced in two steps. First, the three fields *up*, *down*, and $next\ell Index$ of the childtab can be stored in one field of 4 bytes. Second, only one byte is used in practice; see [1] for details.

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 $\ell Indices(i, j)$. This is possible with the help of the *up* and *down* fields of the child-table:

LEMMA 1.5 For every ℓ-interval $[i..j]$, the following statements hold:

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

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, $\mathsf{childtab}[j+1].up = \min \ell Indices(i, j)$, and consequently $i < \mathsf{childtab}[j+1].up \leq j$. 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 $\mathsf{childtab}[i].down = \min \ell Indices(i, j)$. Hence, $i < \mathsf{childtab}[i].down \leq j$.
(2) If $i < \mathsf{childtab}[j+1].up \leq j$, then the claim follows from $\mathsf{childtab}[j+1].up = \min\{q \in [i+1..j] \mid \mathsf{lcptab}[q] > \mathsf{lcptab}[j+1], \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\ \forall k \in [q+1..j]\} = \min\{q \in [i+1..j] \mid \mathsf{lcptab}[k] \geq \mathsf{lcptab}[q]\ \forall k \in [q+1..j]\} = \min \ell Indices(i, j)$.
(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 $next\ell Index$ field of $\mathsf{childtab}[i_1]$, $\mathsf{childtab}[i_2], \ldots, \mathsf{childtab}[i_{k-1}]$. 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 1.4. 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 1.11 *getChildIntervals, applied to an lcp-interval* $[i..j] \neq [0..n]$.

```
intervalList = [ ]
if i < childtab[j + 1].up ≤ j then
   i1 := childtab[j + 1].up
else i1 := childtab[i].down
add(intervalList, (i, i1 − 1))
while childtab[i1].nextℓIndex ≠ ⊥ do
   i2 := childtab[i1].nextℓIndex
   add(intervalList, (i1, i2 − 1))
   i1 := i2
add(intervalList, (i1, 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 1.5, 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.

Algorithm 1.12 *Function* $getlcp(i, j)$

```
if i < childtab[j + 1].up ≤ j
then return lcptab[childtab[j + 1].up]
else return lcptab[childtab[i].down]
```

Answering Queries in Optimal Time

Now we are in a position to show how enhanced suffix arrays can be used 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 1.13 *Answering decision queries.*

```
c := 0
queryFound := True
(i, j) := getInterval(0, n, P[c])
while (i, j) ≠ ⊥ and c < m and queryFound = True
   if i ≠ j then
```

```
      ℓ := getlcp(i, j)
      min := min{ℓ, m}
      queryFound := S[suftab[i] + c ... suftab[i] + min − 1] = P[c ... min − 1]
      c := min
      (i, j) := getInterval(i, j, P[c])
    else queryFound := S[suftab[i] + c ... 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 $S[\mathsf{suftab}[i] \ldots \mathsf{suftab}[i] + m - 1] = P$. Otherwise, if $[i..j]$ is an lcp-interval, then we determine its lcp-value ℓ by the function *getlcp*. Let $\omega = S[\mathsf{suftab}[i] \ldots \mathsf{suftab}[i] + \ell - 1]$ 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 $\omega[0 \ldots m-1] = P$. Otherwise, if $\ell < m$, then we test whether $\omega = P[0 \ldots \ell - 1]$. 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 $P[0 \ldots \ell]$ (note that the suffixes of $[i'..j']$ have $P[0 \ldots \ell - 1]$ 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 $S[\mathsf{suftab}[i'] + \ell \ldots \mathsf{suftab}[i'] + m - 1] = P[\ell \ldots m - 1]$. Otherwise, if $[i'..j']$ is an ℓ'-interval, let $\omega' = S[\mathsf{suftab}[i'] \ldots \mathsf{suftab}[i'] + \ell' - 1]$ 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 $\omega'[\ell \ldots m-1] = P[\ell \ldots m-1]$ (or equivalently, $\omega[0 \ldots m-1] = P$). Otherwise, if $\ell' < m$, then we test whether $\omega[\ell \ldots \ell' - 1] = P[\ell \ldots \ell' - 1]$. 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.

We would like to mention that the very recent results concerning *RMQs* [5, 19, 31] can be used to obtain a different method to simulate top-down traversals of a suffix tree.

1.6 Computation of Tandem Repeats

As already mentioned at the beginning of Section 1.3, repeats play an important role in molecular biology. If the repeated segments are occurring adjacent to each other, then we speak of tandem repeats. Large tandem repeats can span hundreds of thousands of base pairs. There are two types of large tandem repeats: Those that contain genes and those that do not. If a large tandem repeat contains a gene, then it also contains a second copy of that gene. These genes are called paralogous genes. (By contrast, orthologous genes result from speciation rather than duplication.) An example of such tandemly duplicated genes is the human TRGV Locus, which is a region that contains nine repeated genes [26]. Examples of large tandem repeats that do not contain genes are those that are located in or near the centromeres and telomeres of the chromosomes, the so-called (*macro*) *satellites*.

Let us recall the mathematical definition of tandem repeats. A substring of S is a *tandem repeat* if it can be written as $\omega\omega$ for some nonempty string ω. An *occurrence* of a tandem

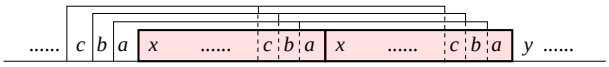

FIGURE 1.5: Chain of non-branching tandem repeats *axcb*, *baxc*, and *cbax*, derived by successively shifting a window one character to the left, starting from a branching tandem repeat *xcba*.

repeat $\omega\omega = S[p \ldots p + 2|\omega| - 1]$ is represented by the pair $(p, |\omega|)$. Such an occurrence $(p, |\omega|)$ is *branching* if $S[p + |\omega|] \neq S[p + 2|\omega|]$.

There is an abundance of papers dealing with the efficient computation of tandem repeats; see, e.g., [33] for references. It is known that tandem repeats of a string S can be computed in $O(n)$ time in the worst case; see [15, 21]. Because these algorithms are quite complicated, we will present simpler algorithms, albeit with non-optimal worst case time complexities.

Stoye and Gusfield [33] described how all tandem repeats can be derived from branching tandem repeats by successively shifting a window to the left; see Figure 1.5. For this reason, we restrict ourselves to the computation of all branching tandem repeats.

A Brute Force Method

The simplest method to find branching tandem repeats is to process all lcp-intervals by top-down traversals of the lcp-interval tree. 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 $q \in [l..r]$. Using the child-table, such a child interval can be detected in $O(|\omega|)$ time (if it exists) with the algorithm of the previous section. Without the child-table, such a child interval can be found in $O(|\omega| \log(j - i))$ time by the algorithm of Manber and Myers [27] described in Section 1.5. This algorithm searches for ω in $[i..j]$. It turns out that the running time of the brute force algorithm is $O(n^2)$ (take, e.g., $S = a^n$). However, the expected length of the longest repeated subword is $O(\log n)$ according to [4]. As a consequence, in practice the brute force method is faster and more space efficient than other methods; see the experimental results in [2].

The Optimized Basic Algorithm

The *optimized basic algorithm* of [33] 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 is annotated by its leaf list, i.e., by the set of leaves in the subtree below it. The leaf list of a branching node corresponds to an lcp-interval in the lcp-interval tree. As a consequence, it is not difficult to implement the optimized basic algorithm via a traversal of the lcp-interval tree. For didactic reasons, we start with the *basic algorithm* of [33], which is justified by the following lemma.

LEMMA 1.6 Let $\omega = S[p \ldots p + \ell - 1]$, where $\ell = |\omega| > 0$, and let $[i..j]$ be the ω-interval. The following statements are equivalent.

1. (p, ℓ) is an occurrence of a branching tandem repeat.
2. $i \leq \mathsf{suftab}^{-1}[p] \leq j$ and $i \leq \mathsf{suftab}^{-1}[p + \ell] \leq j$ and $S[p + \ell] \neq S[p + 2\ell]$.

Proof (1) ⇒ (2): If (p, ℓ) is an occurrence of the tandem repeat ω, then $\omega = S[p \ldots p+\ell-1] = S[p+\ell \ldots p+2\ell-1]$. Since ω is the longest common prefix of $S_{\mathsf{suftab}[i]}, \ldots, S_{\mathsf{suftab}[j]}$, it follows that $i \leq \mathsf{suftab}^{-1}[p] \leq j$ and $i \leq \mathsf{suftab}^{-1}[p+\ell] \leq j$. Furthermore, $S[p+\ell] \neq S[p+2\ell]$ because (p, ℓ) is branching.
(2) ⇒ (1): If $i \leq \mathsf{suftab}^{-1}[p] \leq j$ and $i \leq \mathsf{suftab}^{-1}[p+\ell] \leq j$, then the longest common prefix of S_p and $S_{p+\ell}$ has length at least ℓ. Thus (p, ℓ) is an occurrence of a tandem repeat. Since $S[p+\ell] \neq S[p+2\ell]$, this occurrence is branching.

Algorithm 1.14 *Basic algorithm for the computation of tandem repeats*

for *each ℓ-interval $[i..j]$ with $\ell > 0$* **do**
  for $q := i$ **to** j **do**
    $p := \mathsf{suftab}[q]$
    if $i \leq \mathsf{suftab}^{-1}[p+\ell] \leq j$ **and** $S[p+\ell] \neq S[p+2\ell]$
    then *report (p, ℓ) as an occurrence of a branching tandem repeat*

The basic algorithm finds all occurrences of branching tandem repeats in time proportional to the sum of the sizes of all ℓ-intervals, which is $O(n^2)$ (take, e.g., $S = a^n$). A simple modification of the basic algorithm yields the optimized basic algorithm of [33] as follows. We saw that if (p, ℓ) is an occurrence of the tandem repeat ω, then $\mathsf{suftab}^{-1}[p]$ and $\mathsf{suftab}^{-1}[p+\ell]$ are elements of the ω-interval $[i..j]$. The modification relies on the following observation: If the occurrence is branching, then $\mathsf{suftab}^{-1}[p]$ and $\mathsf{suftab}^{-1}[p+\ell]$ belong to different child intervals of $[i..j]$. Thus, we can omit all indices of one child $[l..r]$ of $[i..j]$ in the second for-loop of the basic algorithm, provided that for each $q \in [i..j] \setminus [l..r]$ we do not only look forward from $p := \mathsf{suftab}[q]$ (i.e., consider $p+\ell$) but we also look backward from it (i.e., we must also consider $p-\ell$). This is made precise in the next algorithm.

Algorithm 1.15 *Optimized basic algorithm for the computation of tandem repeats*

for *each ℓ-interval $[i..j]$* **do**
  determine the child interval $[l..r]$ of maximum size among all children of $[i..j]$
  for $q \in [i..j] \setminus [l..r]$ **do**
    $p := \mathsf{suftab}[q]$
    if $i \leq \mathsf{suftab}^{-1}[p+\ell] \leq j$ **and** $S[p+\ell] \neq S[p+2\ell]$
    then *report (p, ℓ) as an occurrence of a branching tandem repeat*
    if $i \leq \mathsf{suftab}^{-1}[p-\ell] \leq j$ **and** $S[p-\ell] \neq S[p+\ell]$
    then *report $(p-\ell, \ell)$ as an occurrence of a branching tandem repeat*

Algorithm 1.15 can be implemented by a bottom-up traversal of the lcp-interval tree. Then, for each ℓ-interval, a child interval of maximum size can be determined in constant time. Since the largest child interval is always excluded in the second for-loop of Algorithm 1.15, the algorithm runs in $O(n \log n)$ time; see [33]. Algorithm 1.15 requires the tables lcptab, suftab, and suftab^{-1} plus some space for the stack used during the bottom-up traversal of the lcp-interval tree.

It is possible to further improve Algorithm 1.15 by exploiting the fact that $S[p] = S[p+|\omega|]$ for an occurrence $(p, |\omega|)$ of a branching tandem repeat $\omega\omega$. Namely, if $p + |\omega| = \mathsf{suftab}[q]$ for some q in the ω-interval $[i..j]$, then p must occur in the child interval $[l_a..r_a]$ storing the suffixes of S which have ωa as a prefix, where $a = S[\mathsf{suftab}[i]] = S[p]$. The interested reader is referred to [2] for details and for experimental results.

1.7 Space Efficient Computation of Maximal Exact Matches

It is our next goal to compute the maximal exact matches of two genomes in a very space efficient manner. Because the algorithm is based on suffix links, we show how to incorporate them into our framework. 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*.

DEFINITION 1.16 Let $S_{\mathsf{suftab}[i]} = a\omega$. If index j, $0 \leq j < n$, satisfies $S_{\mathsf{suftab}[j]} = \omega$, then we denote j by $\mathsf{link}[i]$ and call it the suffix link (index) of i.

The suffix link of i can be computed with the help of the inverse suffix array as follows.

LEMMA 1.7 If $\mathsf{suftab}[i] < n$, then $\mathsf{link}[i] = \mathsf{suftab}^{-1}[\mathsf{suftab}[i] + 1]$.

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 [13].

DEFINITION 1.17 Given ℓ-interval $[i..j]$, the smallest lcp-interval $[l..r]$ satisfying the inequality $l \leq \mathsf{link}[i] < \mathsf{link}[j] \leq r$ 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 1.8 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 $l \leq \mathsf{link}[i] < \mathsf{link}[j] \leq r$. That is, $[l..r]$ is the ω-interval and thus it has lcp-value $\ell - 1$.

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 called suffix link table and denoted by suflink; see Figure 1.2 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 Figure 1.3, 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 1.7. 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 $l \leq \mathsf{link}[i]$. 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 1.5 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. 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, it is also possible to give a linear time algorithm without intermediate construction of the suffix tree; see [1] for details.

Space Efficient Computation of MEMs for two Genomes

In a previous section we have described an algorithm that uses the enhanced suffix array of the concatenation of S_1 and S_2 to compute *MEMs*. The following algorithm only needs the enhanced suffix array of S_1. It computes all *MEMs* of length at least ℓ by matching S_2 against the enhanced suffix array of S_1. The matching process delivers substrings of S_2 represented by *locations* in the enhanced suffix array. This notion is defined as follows: Suppose that u occurs as a substring of S_1 and consider the enhanced suffix array of S_1. Let ω be the maximal prefix of u such that there is an ω-interval $[l..r]$ and $u = \omega t$ for some string t. We distinguish between two cases:

- If $t = \varepsilon$, then $u = \omega$ and $[l..r]$ is defined to be the *location of* u.
- If $t \neq \varepsilon$, then $([l..r], t)$ is defined to be the *location of* u.

The *location of* u is denoted by *loc*(u).

The enhanced suffix array represents all suffixes $S_1[i \ldots |S_1|-1]\$$ of $S_1\$$. The algorithm processes S_2 suffix by suffix from longest to shortest. In the jth step, for $0 \leq j \leq |S_2|-1$, the algorithm processes suffix $R_j = S_2[j \ldots |S_2|-1]$ and computes the locations of two prefixes minp_j and maxp_j of R_j defined as follows:

- maxp_j is the longest prefix of R_j that is a substring of S_1.
- minp_j is the prefix of maxp_j of length $\min\{\ell, |\mathrm{maxp}_j|\}$.

If $|\mathrm{minp}_j| < \ell$, then less than ℓ of the first characters of R_j match a substring of S_1. If $|\mathrm{minp}_j| = \ell$, then at least the first ℓ characters of R_j match a substring of S_1. Now determine an lcp-interval $[l'..r']$ as follows: If the location of minp_j is the lcp-interval $[l..r]$ then let

$[l'..r'] := [l..r]$. If the location of minp_j is of the form $([l..r], at)$ for some lcp-interval $[l..r]$, some character a, and some string t, then let $[l'..r'] := \mathit{getInterval}([l..r], a)$. By construction, at least one suffix in the lcp-interval $[l'..r']$ matches at least the first ℓ characters of R_j. To extract the *MEMs*, $[l'..r']$ is traversed in a depth first order. The depth first traversal maintains for each ω-interval encountered the length of the longest common prefix of ω and R_j. Each time a leaf-interval $[l''..l'']$ is visited, one first checks whether $S_1[i-1] \neq S_2[j-1]$, where $i = \mathsf{suftab}[l'']$. If this is the case, then $(\perp, i, j)$ is a left maximal exact match and one determines the length c of the longest common prefix of $S_{\mathsf{suftab}[i]}$ and R_j. By construction, $c \geq \ell$ and $S_1[i+c] \neq S_2[j+c]$. Hence (c, i, j) is a *MEM*. Now consider the different steps of the algorithm in more detail:

Computation of $loc(\text{minp}_j)$: For $j = 0$, one computes $loc(\text{minp}_j)$ by greedily matching $S_2[0 \ldots \ell-1]$ against the suffix array, using the algorithms described in Section 1.5. For each $j, 1 \leq j \leq |S_2| - 1$, one considers two cases: (a) follow the suffix link of $loc(\text{minp}_{j-1})$, if this is an lcp-interval. (b) follow the suffix link of $[l..r]$ if $loc(\text{minp}_{j-1})$ is of the form $([l..r], w)$. This shortcut via the suffix link leads to an lcp-interval on the path from the root of the lcp-interval tree to $loc(\text{minp}_j)$. Starting from this location, one matches the next characters of R_j. The method is similar to the matching-statistics computation of [10], and one can show that its overall running time for the computation of all $loc(\text{minp}_j)$, $0 \leq j \leq |S_2| - 1$, is $O(|S_2|)$.

Computation of $loc(\text{maxp}_j)$: Starting from $loc(\text{minp}_j)$ one computes $loc(\text{maxp}_j)$ by greedily matching $S_2[|\text{minp}_j| \ldots |S_2| - 1]$ against the enhanced suffix array of S_1. To facilitate the computation of longest common prefixes, one keeps track of the list of lcp intervals on the path from $loc(\text{minp}_j)$ to $loc(\text{maxp}_j)$. This list is called the *match path*. Since $|\text{maxp}_{j-1}| \geq 1$ implies $|\text{maxp}_j| \geq |\text{maxp}_{j-1}| - 1$, we do not always have to match the edges of the lcp-interval tree completely against the corresponding substring of S_2. Instead, to reach $loc(\text{maxp}_j)$, one rescans most of the edges by only looking at the first character of the edge label to determine the appropriate edge to follow. Thus the total time for this step is $O(|S_2| + \alpha)$ where α is the total length of all match paths. α is upper bounded by the total size β of the subtrees below $loc(\text{minp}_j)$, $0 \leq j \leq |S_2| - 1$. β is upper bounded by the number z_r of right maximal exact matches between S_1 and S_2. Hence the running time for this step of the algorithm is $O(|S_2| + z_r)$.

Depth first traversal: This maintains an *lcp*-stack which stores for each visited lcp-interval, say the ω-interval $[l..r]$, a pair of values $(\mathit{onmatchpath}, \mathit{lcpvalue})$, where the boolean value *onmatchpath* is true if and only if $[l..r]$ is on the match path, and *lcpvalue* stores the length of the longest common prefix of ω and R_j. Given the match path, the *lcp*-stack can be maintained in constant time for each branching node visited. For each leaf-interval $[l..l]$ visited during the depth first traversal, the *lcp*-stack allows to determine in constant time the length of the longest common prefix of $S_{\mathsf{suftab}[l]}$ and R_j. As a consequence, the depth first traversal requires time proportional to the size of the traversed subtree. As exploited above, this is bounded by the number of right maximal matches in the traversed subtree. Thus the total time for all depth first traversals of the subtrees below $loc(\text{minp}_j)$, $0 \leq j \leq |S_2| - 1$, is $O(z_r)$.

Altogether, the algorithm described here runs in $O(|S_1|+|S_2|+z_r)$ time and $O(|S_1|)$ space, where z_r is the number of right maximal matches. Table 1.2 shows the values computed when running the algorithm for two concrete sequences.

Acknowledgement

j	minp_j:$loc(\text{minp}_j)$	remainder of matchpath	depth first traversal	linking locations
0	ac:([0..5],c)		S(0) C(0,0) M(2) S(4) C(4,0) M(2)	ac:([0..5],c)→c:([0..10],c)
1	c:([0..10],c)			c:([0..10],c)→:[0..10]
2	t:[8..9]			t:[8..9]→:[0..10]
3	ta:([8..9],a)		S(7) C(7,3) M(2)	ta:([8..9],a)→a:[0..5]
4	aa:[0..1]	aaaca:([0..1],aca)	S(2) C(2,4) M(5) S(3) C(3,4) M(2)	aa:[0..1]→a:[0..5]
5	aa:[0..1]	aaca:([0..1],ca)	S(2) C(2,5) M(2) S(3) C(3,5)	aa:[0..1]→a:[0..5]
6	ac:([0..5],c)	[2..3] acaaac:([2..3],aac)	S(0) C(0,6) M(6) S(4) C(4,6)	ac:([0..5],c)→c:([0..10],c)
7	ca:[6..7]	caaac:([6..7],aac)	S(1) C(1,7) S(5) C(5,7)	ca:[6..7]→a:[0..5]
8	aa:[0..1]	aaac:([0..1],ac)	S(2) C(2,8) S(3) C(3,8) M(2)	aa:[0..1]→a:[0..5]
9	aa:[0..1]	aac:([0..1],c)	S(2) C(2,9) M(2) S(3) C(3,9)	aa:[0..1]→a:[0..5]
10	ac:([0..5],c)		S(0) C(0,10) M(2) S(4) C(4,10)	ac:([0..5],c)→c:([0..10],c)
11	c:([0..10],c)			

TABLE 1.2 Computation of MEMs of length $\geq \ell = 2$ between $S_1 = acaaacatat$ and $S_2 = acttaaacaaact$, using the enhanced suffix array of S_1 (see Figure 1.2). For each $j \in [0, |S_2| - \ell]$, we show the values minp_j and $loc(\text{minp}_j)$ separated by a colon. Furthermore, for the case that $|\text{minp}_j| \geq \ell$, we show the remainder of the matchpath, i.e., all elements, except for $loc(\text{minp}_j)$. The fourth column shows the actions performed during the depth first traversal of the subtree below minp_j. $S(i)$ means that the suffix $S_1[i \ldots n-1]\$$ of S_1 is visited, where $n = |S_1|$. $C(i,j)$ means that $S_1[i \ldots n-1]\$$ and R_j are checked for left maximality. If this is the case, then $M(q)$ means that a MEM of length q is output. The final column shows the link which is followed to obtain a location representing a prefix of minp_{j+1}. Consider the situation for $j = 6$. Then $\text{minp}_j = ac$ is obtained by scanning $R_j = acaaact$ starting at the location $[0..5]$ of a. Further scanning $aaact$ starting at location $loc(ac) = ([0..5], c)$ visits the intermediate lcp-interval $[2..3]$ end ends at $loc(acaaac) = ([2..3], aac) = \text{maxp}_j$. The depth first traversal of subtree below the lcp-interval $[2..3]$ visits the suffixes of S_1 starting position 0 and 4, respectively. For both suffixes, it is checked whether their left context is different than the left context of R_j. This is the case for $S_1[0 \ldots n-1]\$$, but not for $S_1[4 \ldots n-1]\$$. The length 6 of the longest common prefix of $S_1[0 \ldots n-1]\$$ is determined from the depth of maxp_j.

M.I.A. and E.O. were supported by DFG-grant Oh 54/4-1. S.K. was supported in part by DFG-grant Ku 1257/3-1.

References

[1] Abouelhoda, M., Kurtz, S., and Ohlebusch, E. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms 2* (2004), 53–86.

[2] Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. The Enhanced Suffix Array and its Applications to Genome Analysis. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics* (2002), Lecture Notes in Computer Science 2452, Springer-Verlag, pp. 449–463.

[3] Apostolico, A. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words, Springer Verlag* (1985), pp. 85–96.

[4] Apostolico, A., and Szpankowski, W. Self-Alignments in Words and Their Applications. *Journal of Algorithms 13* (1992), 446–467.

[5] Bender, M. A., and Farach-Colton, M. The LCA Problem Revisited. In *Latin American Theoretical Informatics* (2000), pp. 88–94.

[6] Bray, N., Dubchak, I., and Pachter, L. AVID: A global alignment program. *Genome Res. 13* (2003), 97–102.

[7] Brudno, M., Do, C. B., Cooper, G. M., Kim, M. F., Davydov, E., Green, E. D., Sidow, A., and Batzoglou, S. LAGAN and Multi-LAGAN: Efficient Tools for large-scale Multiple Alignment of Genomic DNA. *Genome Res. 13* (2003), 721–731.

[8] Burrows, M., and Wheeler, D. J. A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Systems Research Center, 1994.

[9] Chain, P., Kurtz, S., Ohlebusch, E., and Slezak, T. R. An Applications-Focused Review of Comparative Genomics Tools: Capabilities, Limitations and Future Challenges. *Briefings in Bioinformatics 4*, 2 (2003), 105–123.

[10] Chang, W. I., and Lawler, E. L. Sublinear Approximate String Matching and Biological Applications. *Algorithmica 12*, 4/5 (1994), 327–344.

[11] Delcher, A. L., Kasif, S., Fleischmann, R. D., Peterson, J., White, O., and Salzberg, S. L. Alignment of Whole Genomes. *Nucleic Acids Res. 27* (1999), 2369–2376.

[12] Gonnet, G., Baeza-Yates, R., and Snider, T. New Indices for Text: PAT trees and PAT arrays. In *Information Retrieval: Algorithms and Data Structures*, W. Frakes and R. A. Baeza-Yates, Eds. Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 66–82.

[13] Grossi, R., and Vitter, J. S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *ACM Symposium on the Theory of Computing* (2000), ACM Press, pp. 397–406.

[14] Gusfield, D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[15] Gusfield, D., and Stoye, J. 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.

[16] Höhl, M., Kurtz, S., and Ohlebusch, E. Efficient Multiple Genome Alignment. *Bioinformatics 18*, Suppl. 1 (2002), S312–S320.

[17] Kärkkäinen, J., and Sanders, P. Simple Linear Work Suffix Array Construction. In *Proc. International Colloquium on Automata, Languages and Programming* (2003), Lecture Notes in Computer Science 2719, Springer Verlag, pp. 943–955.

[18] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching* (2001), Lecture Notes in Computer Science 2089, Springer-Verlag, pp. 181–192.

[19] Kim, D. K., Sim, J. S., Park, H., and Park, K. Linear-Time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)* (2003), vol. 2676 of *LNCS*, Springer-Verlag, pp. 186–199.

[20] Ko, P., and Aluru, S. Space Efficient Linear Time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)* (2003), vol. 2676 of *LNCS*, Springer-Verlag, pp. 200–210.

[21] Kolpakov, R., and Kucherov, G. Finding Maximal Repetitions in a Word in Linear Time. In *Symposium on Foundations of Computer Science* (1999), IEEE Computer Society, pp. 596–604.

[22] Kurtz, S. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience 29*, 13 (1999), 1149–1171.

[23] Kurtz, S., Choudhuri, J. V., Ohlebusch, E., Schleiermacher, C., Stoye, J., and Giegerich, R. REPuter: The Manifold Applications of Repeat Analysis

on a Genomic Scale. *Nucleic Acids Res. 29*, 22 (2001), 4633–4642.

[24] Kurtz, S., and Lonardi, S. Computational Biology. In *Handbook on Data Structures and Applications*, Dinesh P Mehta, Ed. CRC Press, accepted, 2003.

[25] Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., and K. Dewar, et. al. Initial Sequencing and Analysis of the Human Genome. *Nature 409* (2001), 860–921.

[26] LeFranc, M. P., Forster, A., and Rabbitts, T. H. Rearrangement of two distinct T-cell Gamma-Chain Variable-Region genes in human DNA. *Nature 319*, 6052 (1986), 420–422.

[27] Manber, U., and Myers, E. W. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing 22*, 5 (1993), 935–948.

[28] Morgenstern, B. A Space Efficient Algorithm for Aligning Large Genomic Sequences. *Bioinformatics 16* (2000), 948–949.

[29] Needleman, S. B., and Wunsch, C. D. A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins. *J. Mol. Biol. 48* (1970), 443–453.

[30] O'Keefe, C., and Eichler, E. The Pathological Consequences and Evolutionary Implications of Recent Human Genomic Duplications. In *Comparative Genomics* (2000), Kluwer Press, pp. 29–46.

[31] Sadakane, K. Succinct Representations of lcp Information and Improvements in the Compressed Suffix Arrays. In *Proceedings of ACM-SIAM SODA* (2002), pp. 225–232.

[32] Schwartz, S., Zhang, Z., Frazer, K. A., Smit, A., Riemer, C., Bouck, J., Gibbs, R., Hardison, R., and Miller, W. PipMaker–a web server for aligning two genomic DNA sequences. *Genome Research 10*, 4 (2000), 577–586.

[33] Stoye, J., and Gusfield, D. Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree. *Theoretical Computer Science 270*, 1-2 (2002), 843–856.

[34] Weiner, P. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory* (Iowa City, 1973), The University of Iowa, pp. 1–11.

References

Index