A Compressed Enhanced Suffix Array Supporting Fast String Matching

Enno Ohlebusch and Simon Gog

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
Enno.Ohlebusch@uni-ulm.de, Simon.Gog@uni-ulm.de

Abstract. Index structures like the suffix tree or the suffix array are of utmost importance in stringology, most notably in exact string matching. In the last decade, research on compressed index structures has flourished because the main problem in many applications is the space consumption of the index. It is possible to simulate the matching of a pattern against a suffix tree on an enhanced suffix array by using range minimum queries or the so-called *child table*. In this paper, we show that the Super-Cartesian tree of the LCP-array (with which the suffix array is enhanced) very naturally explains the child table. More important, however, is the fact that the balanced parentheses representation of this tree constitutes a very natural compressed form of the child table which admits to locate all *occ* occurrences of pattern P of length m in $O(m \log |\Sigma| + occ)$ time, where Σ is the underlying alphabet. Our compressed child table uses less space than previous solutions to the problem. An implementation is available.

1 Introduction

The suffix tree of a string S is an index structure that can be computed and stored in $O(n)$ time and space [1], where $n = |S|$. Once constructed, it can be used to efficiently solve a "myriad" of string processing problems [2,3]. Although being asymptotically linear, the space consumption of a suffix tree is quite large. This is a drawback in actual implementations. Thus, nowadays many string algorithms are based on suffix arrays and not on suffix trees. The suffix array specifies the lexicographic ordering of all suffixes of S, and it was introduced by Manber and Myers [4]. They showed that all *occ* occurrences of a pattern P of length m can be found in $O(m \log n + occ)$ time by binary search. Using additional information, this worst-case time complexity can be improved to $O(m + \log n + occ)$; see [4]. The suffix array can be compressed; see e.g. [5,6].

In another line of research, Abouelhoda et al. [7] introduced the concept of lcp-intervals in the LCP-array (the LCP-array stores the lengths of longest common prefixes of consecutive suffixes in the suffix array; it can also be compressed [8,9]) and showed that these form a virtual tree (called lcp-interval tree) which directly corresponds to the suffix tree of the string under consideration. To simulate the string matching of pattern P against a suffix tree, one must be able to solve the following problem efficiently: Given an lcp-interval $[i..j]$, find its child interval

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 51–62, 2009.

(if it exists) that "starts" with a certain character a. The solution of Abouelhoda et al. [7] uses a so-called *child table*, which can be precomputed in linear time. Exact pattern matching then takes $O(m|\Sigma|+occ)$ time, where Σ is the underlying alphabet. Other researchers improved this result:

- Kim et al. [10] showed that pattern matching can be done in $O(m \log |\Sigma| + occ)$ time. This time can even be achieved with a compressed child table; see Kim and Park [11].
- Fischer and Heun [12] pointed out that the child table can be replaced by constant time range minimum queries, yielding an $O(m|\Sigma| + occ)$ time pattern search algorithm. Very recently, they showed that an improvement to $O(m \log |\Sigma| + occ)$ time is possible by finding range medians of minima queries, building on the new data structure *Super-Cartesian tree*; see [13].

In this paper, we show that the Super-Cartesian tree of the LCP-array naturally explains the child table (i.e., the introduction of the child table in [7] was not as simple as it could have been). More important, however, is the fact that the balanced parentheses representation of this tree constitutes a very natural compressed form of the child table which admits to locate all *occ* occurrences of pattern P in $O(m \log |\Sigma| + occ)$ time. Our compressed child table requires only $2n + o(n)$ bits, while Fischer and Heun's [13] approach takes $2.54n + o(n)$ bits and Kim and Park's [11] compressed child table requires $5n + o(n)$ bits.

As a matter of fact, the combination of a compressed enhanced suffix array (i.e., both the suffix array [5,6] and the LCP-array are compressed [8,9]) with the balanced parentheses representation of the Super-Cartesian tree of the LCP-array yields yet another compressed suffix tree with full functionality; see e.g. [8,9] and [14] for an overview of this field.

2 Preliminaries

Let S be a string of length n over the alphabet Σ. For every i, $1 \leq i \leq n$, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$. As already mentioned, the suffix array was introduced by Manber and Myers [4]. In 2003, it was shown independently and contemporaneously by Kärkkäinen & Sanders [15], Kim et al. [16], and Ko & Aluru [17] that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see the taxonomy by Puglisi et al. [18].

The *inverse suffix array* SA^{-1} is an array of size n such that for any q with $1 \leq q \leq n$ the equality $\mathsf{SA}^{-1}[\mathsf{SA}[q]] = q$ holds. Moreover, ψ is defined by $\psi[i] = \mathsf{SA}^{-1}[\mathsf{SA}[i] + 1]$ for all i with $1 \leq i \leq n$ if $\mathsf{SA}[i] \neq n$ and $\mathsf{SA}^{-1}[1]$ otherwise.

Let $\mathsf{lcp}(u, v)$ denote the longest common prefix between two strings u and v. The suffix array is often enhanced with the so-called LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA. Formally, the LCP-array is an array such that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n + 1]$ and $\mathsf{LCP}[i] =$

						CLD	
i	SA	LCP	PSV	NSV	$S_{\mathsf{SA}[i]}$	L	R
1	3	-1			*aaacatat*		11
2	4	2	1	3	*aacatat*		
3	1	1	1	7	*acaaacatat*	2	5
4	5	3	3	5	*acatat*		
5	9	1	1	7	*at*	4	6
6	7	2	5	7	*atat*		
7	2	0	1	11	*caaacatat*	3	9
8	6	2	7	9	*catat*		
9	10	0	1	11	*t*	8	10
10	8	1	9	11	*tat*		
11		-1				7	

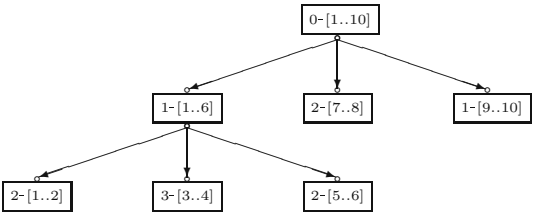

Fig. 1. The enhanced suffix array of the string $S = acaaacatat$ consists of the arrays SA and LCP. The corresponding lcp-interval tree is shown on the right.

$|\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \le i \le n$. Kasai et al. [19] showed that the LCP-array can be computed in linear time from the suffix array and its inverse.

According to [7], an interval $[i..j]$, where $1 \le i < j \le n$, in an LCP-array is called an *lcp-interval of lcp-value* ℓ (denoted by ℓ-$[i..j]$) if

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

Every index k, $i+1 \le k \le j$, with $\mathsf{LCP}[k] = \ell$ is called ℓ-index. Note that each lcp-interval has at most $|\Sigma| - 1$ many ℓ-indices.

An lcp-interval m-$[p..q]$ is said to be *embedded* in an lcp-interval ℓ-$[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \le p < q \le j$) and $m > \ell$. The interval $[i..j]$ is then called the interval *enclosing* $[p..q]$. If $[i..j]$ encloses $[p..q]$ and there is no interval embedded in $[i..j]$ that also encloses $[p..q]$, then $[p..q]$ is called a *child interval* of $[i..j]$. This parent-child relationship constitutes a tree which we call the *lcp-interval tree* (without singleton intervals). An interval $[k..k]$ is called *singleton interval*. The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ which contains k.

The child intervals of an lcp-interval can be determined as follows. If $i_1 < i_2 < \ldots < i_k$ are the ℓ-indices of an lcp-interval ℓ-$[i..j]$ 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); see [7] for details. With range minimum queries on the LCP-array, ℓ-indices can be computed easily [12]: $\mathsf{RMQ}_{\mathsf{LCP}}(i+1, j)$ returns the smallest index k such that $\mathsf{LCP}[k] = \min\{\mathsf{LCP}[q] \mid i+1 \le q \le j\}$. Therefore, it returns the first ℓ-index i_1. Analogously, $\mathsf{RMQ}_{\mathsf{LCP}}(i_1+1, j)$ yields the second ℓ-index i_2, etc. In this way, one can simulate a top-down traversal of the lcp-interval tree, and exact string matching takes $O(m|\Sigma|)$ time in the worst case [7,12] because an array can be preprocessed in linear time so that range minimum queries can be answered in constant time [12,20,21].

The next lemma states how the parent interval of an lcp-interval can be determined; cf. [9]. For any index $2 \leq i \leq n$, define

$$\mathsf{PSV}[i] = \max\{j \mid 1 \leq j < i \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$
$$\mathsf{NSV}[i] = \min\{j \mid i < j \leq n+1 \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$

Lemma 1. *Let ℓ-$[i..j]$ be an lcp-interval with $\mathsf{LCP}[i] = p$ and $\mathsf{LCP}[j+1] = q$. If $p \geq q$, then the parent of $[i..j]$ is the lcp-interval $[\mathsf{PSV}[i]..\mathsf{NSV}[i]-1]$. Otherwise, if $p < q$, the parent of $[i..j]$ is the lcp-interval $[\mathsf{PSV}[j+1]..\mathsf{NSV}[j+1]-1]$.*

Given an lcp-interval $[i..j]$, the smallest lcp-interval $[p..q]$ satisfying $p \leq \psi[i] < \psi[j] \leq q$ is called the *suffix link interval* of $[i..j]$. For every lcp-interval ℓ-$[i..j]$ the suffix link interval exists and it has lcp-value $\ell - 1$; see [7] for details.

3 Finding Child Intervals without Range Minimum Queries

In the top-down traversal of the lcp-interval tree we actually do not need the rather complex machinery of constant-time range minimum queries. To see this, we first recall the definition of the Super-Cartesian tree from [13].

Definition 1. *Let $A[l..r]$ be an array of elements of a totally ordered set $(S, <)$ and suppose that the minima of $A[l..r]$ appear at positions $p_1, p_2, \ldots, p_k$ for some $k \geq 1$. The* Super-Cartesian tree $\mathcal{C}^{sup}(A[l..r])$ *of $A[l..r]$ is recursively constructed as follows:*

- *If $l > r$, then $\mathcal{C}^{sup}(A[l..r])$ is the empty tree.*
- *Otherwise create k nodes $v_1, v_2, \ldots, v_k$, label each v_j with p_j, and for each j with $1 < j \leq k$ the node v_j is the right sibling of node v_{j-1} (in Fig. 2, node v_{j-1} is connected with v_j by a* horizontal *edge). Node v_1 is the root of $\mathcal{C}^{sup}(A[l..r])$. Recursively construct $\mathcal{C}_1 = \mathcal{C}^{sup}(A[l..p_1-1])$, $\mathcal{C}_2 = \mathcal{C}^{sup}(A[p_1+1..p_2-1]), \ldots, \mathcal{C}_{k+1} = \mathcal{C}^{sup}(A[p_k+1..r])$. For each j with $1 \leq j < k$, the left child of v_j is the root of $\mathcal{C}_j$. The left and right children of v_k are the roots of $\mathcal{C}_k$ and $\mathcal{C}_{k+1}$, respectively.*

We would like to emphasize that a node in a Super-Cartesian tree has either a right sibling or a right child but not both. The Super-Cartesian tree $\mathcal{C}^{sup}(A)$ of an array A can be build incrementally in $O(n)$ time; see [13] for details. As an example, consider the enhanced suffix array of the string $S = acaaacatat$ in Fig. 1. The Super-Cartesian tree of this LCP-array is depicted in Fig. 2.

We store the Super-Cartesian tree in an additional table CLD, which we call *child table* because it can be used to determine child intervals. For didactic reasons, we will first store the child table CLD in two arrays $\mathsf{CLD}.L$ and $\mathsf{CLD}.R$. We shall see in a moment, however, that one array suffices. By definition, a node in a Super-Cartesian tree has either a right sibling or a right child but not both. Therefore, for each node i, we store its left child in $\mathsf{CLD}[i].L$ and its right

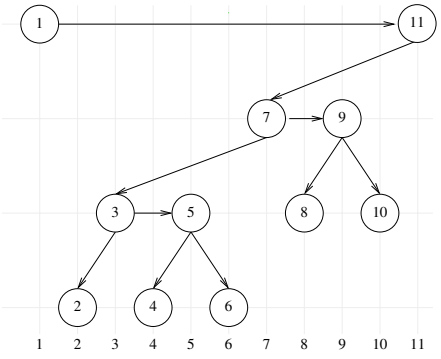

Fig. 2. The Super-Cartesian tree of the LCP-array from Fig. 1

sibling/right child in $\mathsf{CLD}[i].R$. For example, the child table CLD of the string $S = acaaacatat$ is depicted in Fig. 1.

As already mentioned, finding all child intervals of an lcp-interval ℓ-$[i..j]$ boils down to finding all ℓ-indices of that interval. The following theorem shows where the first ℓ-index of an lcp-interval ℓ-$[i..j]$ can be found in the child table.

Proposition 1. *For every lcp-interval ℓ-$[i..j]$ we have:*

1. *If $\mathsf{LCP}[i] \leq \mathsf{LCP}[j+1]$, then $\mathsf{CLD}[j+1].L$ stores the first ℓ-index of the lcp-interval $[i..j]$.*
2. *If $\mathsf{LCP}[i] > \mathsf{LCP}[j+1]$, then $\mathsf{CLD}[i].R$ stores the first ℓ-index of the lcp-interval $[i..j]$.*

Now we have all ingredients to realize a top-down traversal of the lcp-interval tree without range minimum queries. Proposition 1 tells us where the first ℓ-index, say i_1, of $[i..j]$ can be found. Using the child table, we find the second ℓ-index i_2 by $i_2 = \mathsf{CLD}[i_1].R$, the third ℓ-index i_3 by $i_3 = \mathsf{CLD}[i_2].R$, and so on. The index i_k is the last ℓ-index if $\mathsf{LCP}[i_{k+1}] \neq \ell$. Algorithm 1 implements this approach. The procedure *getChildIntervals* applied to an lcp-interval $[i..j]$ returns the list of all child intervals of $[i..j]$.

Of course, the Super-Cartesian tree is only conceptual, i.e., we can construct the child table without it. Algorithm 2 uses a stack to do this. The procedures *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. Moreover, *top*().*idx* denotes the first component of the topmost element of the stack, while *top*().*lcp* denotes the second component.

To reduce the space requirement of the child table, only one array is used in practice. As a matter of fact, the memory cells of $\mathsf{CLD}[i].R$, which are unused, can store the values of the $\mathsf{CLD}.L$ array. To see this, note that $\mathsf{CLD}[i+1].L \neq \bot$ if and only if $\mathsf{LCP}[i] > \mathsf{LCP}[i+1]$. In this case, however, we have $\mathsf{CLD}[i].R = \bot$. In other words, $\mathsf{CLD}[i].R$ is empty and can store the value $\mathsf{CLD}[i+1].L$; see

Algorithm 1. *getChildIntervals* applied to an lcp-interval $[i..j]$.

```
intervalList = [ ]
k ← i
if LCP[i] ≤ LCP[j + 1] then
  m ← CLD[j + 1].L
else m ← CLD[i].R
ℓ ← LCP[m]
repeat
  add(intervalList, [k..m − 1])
  k ← m
  m ← CLD[m].R
until m = ⊥ or LCP[m] ≠ ℓ
add(intervalList, [k..j])
```

Algorithm 2. Construction of the child table.

```
push(⟨1, −1⟩)          /* an element on the stack has the form ⟨idx, lcp⟩ */
for k ← 2 to n + 1 do
  while LCP[k] < top().lcp do
    last ← pop()
    while top().lcp = last.lcp do
      CLD[top().idx].R ← last.idx
      last ← pop()
    if LCP[k] < top().lcp then
      CLD[top().idx].R ← last.idx
    else CLD[k].L ← last.idx
  push(⟨k, LCP[k]⟩)
```

Fig. 1. Finally, for a given index i, one can decide whether $\mathsf{CLD}[i].R$ contains the value $\mathsf{CLD}[i+1].L$ by testing whether $\mathsf{LCP}[i] > \mathsf{LCP}[i+1]$. To sum up, although the child table conceptually uses two arrays, only space for one array is actually required.

4 Balanced Parentheses Representation of the Tree

The Super-Cartesian tree of the LCP-array can be represented by a sequence of balanced parentheses; see Fig. 3. Again, it turns out that the Super-Cartesian tree is only conceptual. To be precise, its balanced parentheses representation can be obtained solely based on the LCP-array; see Algorithm 3.

Each node k, $1 \leq k \leq n$, in the Super-Cartesian tree is represented by the k-th opening parenthesis (and the matching closing parenthesis). Node $n+1$ is not represented. Consequently, the sequence of balanced parentheses has $2n$ parentheses, and it can be represented with $2n$ bits.

(	(	)	(	(	)	(	(	)	)	)	(	(	)	(	(	)	)	)	)
1	2	2	3	4	4	5	6	6	5	3	7	8	8	9	10	10	9	7	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Fig. 3. Balanced parentheses representation of the Super-Cartesian tree of Fig. 2

Algorithm 3. Construction of the balanced parentheses representation of the Super-Cartesian tree of the LCP-array.

```
push(−1)          /* LCP[1] = −1 */
write an opening parenthesis
for k ← 2 to n + 1 do
  while LCP[k] < top() do
    pop() and write a closing parenthesis
  if k ≠ n + 1 then
    push(LCP[k]) and write an opening parenthesis
  else
    write a closing parenthesis
```

Given a balanced parentheses sequence, the following operations can be supported in constant time with only $o(n)$ bits of extra space [22,23,24,25]:

- $rank_(i)$: returns the number of opening parentheses up to and including position i; see Jacobson [22]. $rank_)(i)$ is defined analogously.
- $select_(i)$: returns the position of the i-th opening parenthesis; see Clark [23]. $select_)(i)$ is defined analogously.
- $findclose(i)$: returns the position of the closing parenthesis matching the opening parenthesis at position i; see Munro & Raman [24]. $findopen(i)$ is defined analogously.
- $enclose(i)$: given a parenthesis pair whose opening parenthesis is at position i, it returns the position of the opening parenthesis corresponding to the closest matching parenthesis pair enclosing i; see Munro & Raman [24].

Geary et al. [25] provide a simpler $o(n)$ extra space solution for $findclose$, $findopen$, and $enclose$. In our implementation, we use a data structure which is similar to that of [25]. That is, our implementation needs $2n + o(n)$ bits to support all operations in constant time.

As we have seen, determining the child intervals of an ℓ-interval $[i..j]$ boils down to finding the ℓ-indices of $[i..j]$ in ascending order. On the balanced parentheses representation of the Super-Cartesian tree of the LCP-array these can be found as follows.

Lemma 2. *With the balanced parentheses representation, the first ℓ-index k of an lcp-interval $[i..j]$ can be determined in constant time by*

$$k = \begin{cases} rank_(findopen(select_(j+1)-1)) & , \text{if } \mathsf{LCP}[i] \le \mathsf{LCP}[j+1] \\ rank_(findopen(findclose(select_(i))-1)), & \text{if } \mathsf{LCP}[i] > \mathsf{LCP}[j+1] \end{cases}$$

Proof. If $\mathsf{LCP}[i] \leq \mathsf{LCP}[j+1]$, then k is the left child of $j+1$ in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the k-th opening parenthesis is directly followed by the $(j+1)$-th opening parenthesis. Therefore, $k = rank_{(}(findopen(select_{(}(j+1)-1))$. If $\mathsf{LCP}[i] > \mathsf{LCP}[j+1]$, then k is the right child of i in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the k-th opening parenthesis is directly followed by the closing parenthesis matching the i-th opening parenthesis. Thus, $k = rank_{(}(findopen(findclose(select_{(}(i))-1))$.

If we know an ℓ-index k of an lcp-interval $[i..j]$, then the next ℓ-index m (if it exists) is the right sibling of k in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the m-th opening parenthesis is directly followed by the closing parenthesis matching the k-th opening parenthesis. So $m = rank_{(}(findopen(findclose(select_{(}(k))-1))$.

Given an ℓ-index k of an lcp-interval $[i..j]$, an algorithm that computes the next ℓ-index m (if it exists) works as follows. First it tests whether $select_{(}(k) \neq findclose(select_{(}(k))-1$. If this is the case, then k is not a leaf in the Super-Cartesian tree of the LCP-array (i.e., the k-th opening parenthesis is not directly followed by a closing parenthesis) and the algorithm further computes $m = rank_{(}(findopen(findclose(select_{(}(k))-1))$. If $\mathsf{LCP}[k] = \mathsf{LCP}[m]$, then it returns m as the next ℓ-index of $[i..j]$. Otherwise, there is no next ℓ-index. It follows as a consequence that all child intervals of an lcp-interval $[i..j]$ can be determined in $O(|\Sigma|)$ time solely based on the balanced parentheses representation of the Super-Cartesian tree of the LCP-array. That is, one neither needs range minimum queries nor the child table.

To exemplify our method, we search for the first ℓ-index k of the lcp-interval 1-$[1,6]$ (see Fig. 1): As $\mathsf{LCP}[1] = -1 \leq \mathsf{LCP}[7] = 0$, k is the left child of node 7 in the Super-Cartesian tree of Fig. 2. In the balanced parentheses sequence, we obtain the position of the 7th opening parenthesis by $select_{(}(7) = 12$; see Fig. 3. The left child of node 7 is represented by the opening parenthesis matching the closing parenthesis at position $12-1 = 11$, and this opening parenthesis is found at position $findopen(11) = 4$. Since $rank_{(}(4) = 3$, we conclude that $k = 3$. The next ℓ-index (if it exists) corresponds to the right sibling of node $k = 3$ in the Super-Cartesian tree. If node $k = 3$ is not a leaf, then the parenthesis directly left to the closing parentheses corresponding to k is also a closing parenthesis. In our example this is the case and therefore $rank_{(}(findopen(10)) = 5$ tells us that node 5 is either a sibling or a child of node $k = 3$. Because $\mathsf{LCP}[5] = \mathsf{LCP}[3]$, node 5 is the right sibling. Hence 5 is the next ℓ-index.

In string matching, we search for a specific child interval. To be precise, if $[i..j]$ is an lcp-interval that represents a string ω, we wish to compute the lcp-interval $[i'..j']$ that represents the string ωa for some character $a \in \Sigma$. Clearly, we can enumerate all child intervals $[i'..j']$ of $[i..j]$ until the one with $S[\mathsf{SA}[i'] + |\omega|] = \ldots = S[\mathsf{SA}[j'] + |\omega|] = a$ is found. This takes $O(|\Sigma|)$ time in the worst case. As a matter of fact, the balanced parentheses representation allows us to determine such an interval in $O(\log|\Sigma|)$ time. This goes as follows. The left

boundary of the interval we are searching for is either i or one of the ℓ-indices of the lcp-interval $[i..j]$. Thus, if $S[\mathsf{SA}[i] + |\omega|] = a$, we are done. Otherwise, one determines the first ℓ-index k as in Lemma 2. If $S[\mathsf{SA}[k] + |\omega|] = a$, we are done. If not, we determine the position p of the closing parenthesis matching the k-th opening parenthesis by $p = findclose(select_(k))$. The key observation is that the remaining ℓ-indices of $[i..j]$ (there are at most $|\Sigma| - 2$ many) are the siblings of node k in the Super-Cartesian tree and—by construction—the closing parentheses immediately preceding the closing parenthesis at position p correspond to these ℓ-indices. Therefore, a binary search on the matching opening parentheses of the first $|\Sigma| - 2$ closing parentheses immediately preceding the closing parenthesis at position p (if there are so many closing parentheses at all), can be used to find the desired interval. First check whether the index m under consideration satisfies $\mathsf{LCP}[m] = \mathsf{LCP}[k]$. If not, we have to search in the right half. If so, m is another ℓ-index and one further compares $S[\mathsf{SA}[m] + |\omega|]$ with the character a. If the characters coincide, then m is the left boundary of the interval we are searching for. If $S[\mathsf{SA}[m] + |\omega|] < a$, we have to search in the right half, and if $S[\mathsf{SA}[m] + |\omega|] > a$, we have to search in the left half.

5 Full Functionality

The balanced parentheses representation of the Super-Cartesian tree of the LCP-array supports all operations of a suffix tree as listed e.g. in [8,9]. Here we show how the following two crucial operations can be implemented (the other operations are rather straightforward; cf. [9]):

- $parent([i..j])$: returns the parent interval of the lcp-interval $[i..j]$.
- $slink([i..j])$: returns the suffix link interval of the lcp-interval $[i..j]$.

According to Lemma 1, the parent interval of an lcp-interval can be determined with the help of PSV and NSV-values. The next lemma shows how these values can be computed on the balanced parentheses representation.

Lemma 3. *Let i be an index with $\mathsf{LCP}[i] \neq -1$. With the balanced parentheses representation of the Super-Cartesian tree of the* LCP*-array, $\mathsf{NSV}[i]$ can be determined in constant time by $\mathsf{NSV}[i] = rank_(findclose(select_(i))) + 1$ and $\mathsf{PSV}[i]$ can be computed in $O(|\Sigma|)$ time by*

$j \leftarrow rank_(enclose(select_(i)))$
while $\mathsf{LCP}[j] = \mathsf{LCP}[i]$ **do**
  $j \leftarrow rank_(enclose(select_(j)))$
$\mathsf{PSV}[i] \leftarrow j$

It is also possible to compute $\mathsf{PSV}[i]$ in $O(\log |\Sigma|)$ time by a binary search on the balanced parentheses sequence (similar to the method described above). Consequently, the parent interval of an lcp-interval can be found in $O(\log |\Sigma|)$ time by Lemma 1.

The suffix link interval $[p..q]$ of an lcp-interval ℓ-$[i..j] \neq 0$-$[1..n]$ can be determined as follows: First, the range minimum query $\mathsf{RMQ}_{\mathsf{LCP}}(\psi[i] + 1, \psi[j])$ yields

an $(\ell-1)$-index of $[p..q]$ (see [7] for details), say index k, and then the boundaries of the suffix link interval are determined by $p = \mathsf{PSV}[k]$ and $q = \mathsf{NSV}[k]-1$ (this is a direct consequence of the definition of an lcp-interval). Thus, $[p..q]$ can be computed in $O(\log|\Sigma|)$ time provided that range minimum queries can be answered in constant time. However, the ability to answer range minimum queries in constant time requires a different data structure, and this is disadvantageous in practice. To compute suffix links on our data structure, we introduce the new operation *range-restricted-enclose* (*rr-enclose* for short) on balanced parentheses sequences: Given two opening parentheses at positions i and j such that $findclose(i) < j$, the operation $rr\text{-}enclose(i,j)$ returns the smallest position, say k, of an opening parenthesis such that $findclose(i) < k < j$ and $findclose(j) < findclose(k)$. If such a k does not exist, it returns $\perp$.

Theorem 1. *Given the balanced parentheses representation of the Super-Cartesian tree of the* LCP*-array and two indices* i *and* j *with* $1 \le i \le j \le n$*, let* $i' = select_{(}(i)$ *and* $j' = select_{(}(j)$*. Then*

$$\mathsf{RMQ}_{\mathsf{LCP}}(i,j) = \begin{cases} i, & \text{if } j' < findclose(i') \\ j, & \text{if } findclose(i') < j' \text{ and } rr\text{-}enclose(i',j') = \perp \\ rank_{(}(rr\text{-}enclose(i',j')), & \text{otherwise} \end{cases}$$

Proof. We use a case differentiation. If $j' < findclose(i')$, then $i' < j' < findclose(j') < findclose(i')$, i.e., the parenthesis pair with opening parenthesis at position i encloses the other parenthesis pair. This, in turn, means that $\mathsf{LCP}[i] \le \mathsf{LCP}[q]$ for all q with $i < q \le j$. Hence $\mathsf{RMQ}_{\mathsf{LCP}}(i,j) = i$. Otherwise, we have $findclose(i') < j'$. If $rr\text{-}enclose(i',j') = \perp$, then there is no parenthesis pair with opening parenthesis at a position $> findclose(i)$ that encloses the parenthesis pair with opening parenthesis at position j. This means that the parenthesis pairs are "siblings". In other words, $\mathsf{LCP}[i]$ is a successor of $\mathsf{LCP}[j]$ on a "left path" in the Super-Cartesian tree of the LCP-array. It follows as a consequence that $\mathsf{LCP}[q] > \mathsf{LCP}[j]$ for all q with $i \le q < j$. Thus, $\mathsf{RMQ}_{\mathsf{LCP}}(i,j) = j$. In the last case $rr\text{-}enclose(i',j') = k$, where k is the smallest position of an opening parenthesis such that $findclose(i') < k < j'$ and $findclose(j') < findclose(k)$. So we have $i' < findclose(i') < k < j' < findclose(j') < findclose(k)$. In the Super-Cartesian tree of the LCP-array, this corresponds to $\mathsf{LCP}[q] > \mathsf{LCP}[k]$ for all q with $i \le q < k$ and $\mathsf{LCP}[k] \le \mathsf{LCP}[q]$ for all q with $k < q \le j$. Therefore, $\mathsf{RMQ}_{\mathsf{LCP}}(i,j) = k$.

6 Conclusions

The methods described above have been implemented in *C++*, and the implementation is available under the GNU General Public License. In our opinion, the main advantage of the balanced parentheses representation of the Super-Cartesian tree of the LCP-array is that child intervals can be computed efficiently. Thus, it would be natural to compare our implementation experimentally with

the related methods of Kim and Park [11] and of Fischer and Heun [13]. Unfortunately, for both methods no implementation is available.

On the other hand, our compressed enhanced suffix array has full functionality, i.e., it supports all operations of a suffix tree and it is natural to compare it experimentally with implementations of compressed suffix trees with such a full functionality. Välimäki et al. [26] implemented Sadakane's compressed suffix tree [8], and to the best of our knowledge this is the sole implementation which is available. For a fair comparison, however, both implementations should use the same compressed suffix array and the same compressed LCP-array. We are currently working on our own implementation of Sadakane's compressed suffix tree [8] and an experimental comparison is forthcoming.

First experiments with texts of size 50MB show that the $O(\log|\Sigma|)$-time version of the method that determines a specific child interval is two times faster than the $O(|\Sigma|)$-time version for $20 < |\Sigma| < 30$ and up to 10 times faster for $90 < |\Sigma| < 230$. Unsurprisingly, for $|\Sigma| = 4$ (DNA-alphabet) the $O(\log|\Sigma|)$-time version is (slightly) slower than the $O(|\Sigma|)$-time version.

References

1. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)
2. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words, pp. 85–96. Springer, Heidelberg (1985)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
4. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
5. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symposium on the Theory of Computing, pp. 397–406. ACM Press, New York (2000)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
7. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
8. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41, 589–607 (2007)
9. Fischer, J., Navarro, G., Mäkinen, V.: An(other) entropy bounded compressed suffix tree. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 152–165. Springer, Heidelberg (2008)
10. Kim, D., Jeon, J., Park, H.: An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 138–149. Springer, Heidelberg (2004)
11. Kim, D., Jeon, J., Park, H.: A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 33–44. Springer, Heidelberg (2005)

12. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
13. Fischer, J., Heun, V.: Range median of minima queries, super-cartesian trees, and text indexing. In: Proc. 19th International Workshop on Combinatorial Algorithms, pp. 239–252. College Publications (2008)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39 (2007)
15. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
16. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
17. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
18. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
19. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
20. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13, 338–355 (1984)
21. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing 17, 1253–1262 (1988)
22. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE, Los Alamitos (1989)
23. Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo (1996)
24. Munro, J., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
25. Geary, R., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
26. Välimäki, N., Gerlach, W., Dixit, K., Mäkinen, V.: Engineering a compressed suffix tree implementation. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 217–228. Springer, Heidelberg (2007)