A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array[⋆]

Johannes Fischer and Volker Heun

Inst. für Informatik, Ludwig-Maximilians-Universität München
Amalienstr. 17, D-80333 München
{Johannes.Fischer|Volker.Heun}@bio.ifi.lmu.de

Abstract. The Range-Minimum-Query-Problem is to preprocess an array of length n in $O(n)$ time such that all subsequent queries asking for the position of a minimal element between two specified indices can be obtained in constant time. This problem was first solved by Berkman and Vishkin [1], and Sadakane [2] gave the first *succinct* data structure that uses $4n + o(n)$ bits of additional space. In practice, this method has several drawbacks: it needs $O(n \log n)$ bits of intermediate space when constructing the data structure, and it builds on previous results on succinct data structures. We overcome these problems by giving the first algorithm that *never* uses more than $2n + o(n)$ bits, and does not rely on rank- and select-queries or other succinct data structures. We stress the importance of this result by simplifying and reducing the space consumption of the Enhanced Suffix Array [3], while retaining its capability of simulating top-down-traversals of the suffix tree, used, e.g., to locate all *occ* positions of a pattern p in a text in optimal $O(|p| + occ)$ time (assuming constant alphabet size). We further prove a lower bound of $2n - o(n)$ bits, which makes our algorithm asymptotically optimal.

1 Introduction

Given an array A of n real numbers or other elements from a totally ordered set, a natural question is to ask for the position of a minimal element between two specified indices l and r. Queries of this form are known under the name of *range minimum queries* (RMQ), and the result is denoted by $\textsc{rmq}_A(l, r)$. There are several variants of the problem, the most prominent being the one where the array is static and known in advance, which is the issue of this article. It has been shown by Berkman and Vishkin [1] that a preprocessing in $O(n)$ time is sufficient to answer all RMQs in $O(1)$ time. This algorithm has been rediscovered and simplified by Bender and Farach-Colton [4]. The main motivation of both papers is the strong connection between RMQs and a different fundamental algorithmic problem in trees, namely the task of computing the *lowest common ancestor* (LCA) of two specified nodes in constant time, first noted by Gabow

[⋆] This work was partially funded by the German Research Foundation (DFG).

et al. [5]. They showed that LCA-queries on a tree T correspond to a *restricted* version of RMQ on an array that is obtained by writing down the heights of the nodes visited during an Euler-Tour through T (the details will be explained later). The fact that the general RMQ-problem can be reduced to an instance of LCA, which is in turn reduced to the restricted version of RMQ, is the basis for their RMQ-algorithms.

A drawback of these methods is that, because they employ arrays of size n that store numbers up to the computer's word size, their space consumption is $O(n \log n)$ bits. It has been noted that in many cases this is far from optimal, and a flood of algorithms under the term *succinct* (meaning that their space complexity is $O(n)$ *bits* instead of *words*) has been developed, starting with Jacobson's succinct representation for labeled trees [6].

The only known result on succinct data structures for the RMQ-problem is due to Sadakane [2]. His solution requires $4n+o(n)$ bits of space and makes heavy use of succinct data structures for rank- and select-queries on binary sequences (see [7]), and of succinct data structures for related problems [8]. Another drawback is that the solution in [2] requires the intermediate construction of several labeled trees, which use $O(n \log n)$ bits of space in the worst case.

1.1 Contributions of Our Work

We present a new succinct data structure for the RMQ-problem that uses $2n + o(n)$ bits of extra space and answers range minimum queries in $O(1)$ time. It is a practicable and direct method, meaning that (1) at each stage of its construction it never requires more space than the final data structure, (2) it does not rely on any other results on succinct data structures, and (3) it is easy to implement.

As a direct application of our new storage scheme for RMQ, we show that it leads to simplifications and improvements in the Enhanced Suffix Array (ESA) [3], a collection of arrays which can be used as a space-conscious alternative to the suffix tree, e.g., for locating all *occ* occurrences of a pattern p in a text in optimal $O(|p| + \mathit{occ})$ time. Note that despite the significant progress that has been made in the field of compressed indexes [7, 9], none of these indexes achieves this time bound, even if the size $|\Sigma|$ of the alphabet is assumed to be constant. So the ESA is still the method of choice. We show that one of the arrays from the ESA (previously occupying $n \log n$ bits) can be replaced by our storage scheme for RMQ, thus reducing its space to $2n + o(n)$ bits. We further emphasize the practicability of our method by performing tests on realistically-sized input arrays which show that our method uses less space than the most space-conscious *non*-succinct data structure [10], and that the query time is competitive.

Finally, a lower bound of $2n - o(n)$ bits is shown, making our data structure asymptotically optimal.

1.2 Applications of RMQ

We briefly sketch the most important applications of RMQ.

Computing Lowest Common Ancestors in Trees. For a static tree T to be preprocessed, $\textsc{lca}_T(v,w)$ returns the deepest node in T that is an ancestor of both v and w. It has been noted by Gabow et al. [5] that this problem can be reduced to RMQ as follows: store the heights of the nodes in an array H in the order in which they are visited during an in-order tree traversal of T. Also, in $I[j]$ remember the node from which the height $H[j]$ has come. Finally, let R be the inverse array of I, i.e., $I[R[j]] = j$. Then $\textsc{lca}_T(v,w)$ is given by $I[\textsc{rmq}_H(R[v], R[w])]$. This is simply because $\textsc{lca}_T(v,w)$ is the shallowest node visited between v and w during the in-order tree traversal.

Computing Longest Common Extensions of Suffixes. This problem has numerous applications in approximate pattern matching and is defined for a static string t of size n: given two indices i and j, $\textsc{lce}_t(i,j)$ returns the length of the longest common prefix of t's suffixes starting at position i and j; i.e., $\textsc{lce}_t(i,j) = \max\{k : t_{i,\ldots,i+k-1} = t_{j,\ldots,j+k-1}\}$. It is well-known that $\textsc{lce}_t(i,j)$ is given $\mathsf{LCP}[\textsc{rmq}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i]+1, \mathsf{SA}^{-1}[j])]$, where LCP is the *LCP-array* [11] for t.

Document Retrieval Queries. The setting of document retrieval problems is as follows: For a static collection of n text documents, on-line queries like "return all d documents containing pattern p" are posed to the system. Muthukrishnan [12] gave elegant algorithms that solve this and related tasks in optimal $O(|p| + d)$ time. The idea behind these algorithms is to "chain" suffixes from the same document and use RMQs to ensure that each document containing p is visited at most twice. Sadakane [2] continued this line of research towards succinctness, again using RMQs.

Maximum-Sum Segment Queries. Given a static array A of n real numbers, on-line queries of the form "return the sub-interval of $[l,r]$ with the highest sum" are to be answered; i.e., $\mathrm{MSSQ}(l,r)$ returns the index pair (x,y) such that $(x,y) = \arg\max_{l\le x\le y\le r} \sum_{i=x}^{y} A[i]$. This problem and extensions thereof have very elegant optimal solutions based on RMQs due to Chen and Chao [13]. The fundamental connection between RMQ and MSSQ can be seen as follows: compute an array of prefix sums $C[i] = \sum_{k=0}^{i} A[k]$ and prepare it for range minimum and maximum queries. Then if $C[x]$ is the minimum and $C[y]$ the maximum of all $C[i]$ in $[l-1,r]$ and $x < y$, then $(x+1,y)$ is the maximum-sum segment in $[l,r]$. The more complicated case where $x > y$ is also broken down to RMQs.

2 Definitions and Previous Results

The *Range Minimum Query* (RMQ) problem is formally defined as follows: given an array $A[0,n-1]$ of elements from a totally ordered set (with order relation "$\le$"), $\textsc{rmq}_A(l,r)$ returns the index of a smallest element in $A[l,r]$, i.e., $\textsc{rmq}_A(l,r) = \arg\min_{k\in\{l,\ldots,r\}}\{A[k]\}$. (The subscript A will be omitted if the context is clear.) The most naive algorithm for this problem searches the array from l to r each time a query is presented, resulting in a $\Theta(n)$ query time in the worst case. As mentioned in the introduction, we consider the variant where A is first preprocessed in order to answer future queries faster. The following definition will be central for both our algorithm and that of [1].

Definition 1. *A* Cartesian Tree *of an array $A[l,r]$ is a binary tree $\mathcal{C}(A)$ whose root is a minimum element of A, labeled with the position i of this minimum. The left child of the root is the Cartesian Tree of $A[l,i-1]$ if $i > l$, otherwise it has no left child. The right child is defined analogously for $A[i+1,r]$.*

Note that $\mathcal{C}(A)$ is not necessarily unique if A contains equal elements. To overcome this problem, we impose a *strong* total order "$\prec$" on A by defining $A[i] \prec A[j]$ iff $A[i] < A[j]$, or $A[i] = A[j]$ and $i < j$. The effect of this definition is just to consider the 'first' occurrence of equal elements in A as being the 'smallest'. Defining a Cartesian Tree over A using the $\prec$-order gives a *unique* tree $\mathcal{C}^{\text{can}}(A)$, which we call the *Canonical Cartesian Tree*. Note also that this order results in unique answers for the RMQ-problem, because the minimum under "$\prec$" is unique. A linear-time algorithm for constructing $\mathcal{C}^{\text{can}}(A)$ is given in [4].

In the rest of this paper the space is analyzed in *bit*-complexity. For the sake of clarity we write $O(f(n) \cdot \log(g(n)))$ for the number of bits needed by a table, where $f(n)$ denotes the number of entries in the table, and $g(n)$ is their maximal size. For example, a normal integer array of size n which takes values up to n uses $O(n \cdot \log n)$ bits of space.

2.1 Berkman and Vishkin's Algorithm

This section describes the solution to the general RMQ-problem as a combination of the results obtained in [1,4,5]. We follow the simplified presentation from [4].

$\pm$1RMQ is a special case of the RMQ-problem, where consecutive array elements differ by exactly 1. The solution starts by reducing RMQ to $\pm$1RMQ as follows: given an array $A[0,n-1]$ to be preprocessed for RMQ, build $\mathcal{C}^{\text{can}}(A)$. Then perform an Euler Tour in this tree, storing the labels of the visited nodes in an array $E[0,2n-2]$, and their respective heights in $H[0,2n-2]$. Further, store the position of the first occurrence of $A[i]$ in the Euler Tour in a representative array $R[0,n-1]$. The Cartesian Tree is not needed anymore once the arrays E, H and R are filled, and can thus be deleted. The paper then shows that $\text{RMQ}_A(l,r) = E[\pm 1\text{RMQ}_H(R[l],R[r])]$. Note in particular the doubling of the input when going from A to H; i.e., H has $n' := 2n-1$ elements.

To solve $\pm$1RMQ on H, partition H into *blocks* of size $\frac{\log n'}{2}$.[1] Define two arrays A' and B of size $\frac{2n'}{\log n'}$, where $A'[i]$ stores the minimum of the ith block in H, and $B[i]$ stores the position of this minimum in H.[2] We now want to preprocess A' such that out-of-block queries (i.e., queries that span over several blocks in H) can be answered in $O(1)$. The idea is to precompute all RMQs whose length is a power of two. For every $0 \le i < 2n'/\log n'$ and every $1 \le j \le \log(2n'/\log n')$ compute the position of the minimum in the sub-array $A'[i, i+2^j-1]$ and store the result in $M[i][j]$. Table M occupies $O(\frac{2n'}{\log n'} \log \frac{2n'}{\log n'} \cdot \log \frac{2n'}{\log n'}) = O(n \cdot \log n)$ bits of space and can be filled in $O(n)$

[1] For a simpler presentation we often omit floors and ceilings from now on.

[2] Array A' is just conceptual because $A'[i]$ is given by $H[B[i]]$.

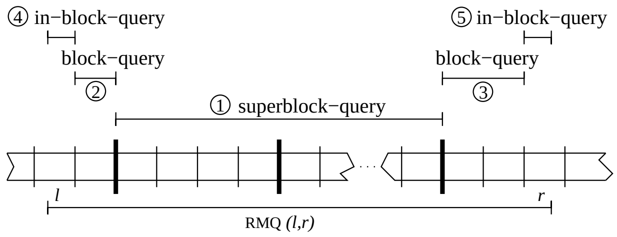


Fig. 1. How a range-minimum query $\text{RMQ}(l, r)$ can be decomposed into at most five different sub-queries. Thick lines denote the boundaries between superblocks, thin lines denote the boundaries between blocks.

time by using the formula $M[i][j] = \arg\min_{k\in\{M[i][j-1],M[i+2^{j-1}][j-1]\}}\{A'[k]\}$. To answer $\text{RMQ}_{A'}(i, j)$, select two *overlapping* blocks that exactly cover the interval $[i, j]$, and return the position where the overall minimum occurs. Precisely, let $l = \lfloor\log(j - i)\rfloor$. Then $\text{RMQ}(i, j) = \arg\min_{k\in\{M[i][l],M[j-2^l+1][l]\}}\{A'[k]\}$.

It remains to show how in-block-queries are handled. This is done with the so-called Four-Russians-Trick, where one precomputes the answers to all possible queries when the number of possible instances is sufficiently small. The authors of [4] noted that due to the ±1-property there are only $O(\sqrt{n'})$ blocks to be precomputed: we can virtually subtract the initial value of a block from each element without changing the answers to the RMQs; this enables us to describe a block by a ±1-vector of length $2^{1/2\log n'-1}$. For each such block precompute all $\frac{1}{2}\frac{\log n'}{2}(\frac{\log n'}{2} + 1) = O(\log^2 n')$ possible RMQs and store them in a table $P[1, 2^{1/2\log n'-1}][1, \frac{\log n'}{4}(\frac{\log n'}{2}+1)]$ with a total size of $O(\sqrt{n'}\log^2 n' \cdot \log\frac{\log n'}{2}) = o(n)$ bits. To index table P, precompute the *type* of each block and store it in array $T[1, \frac{2n'}{\log n'}]$. The block type is simply the binary number obtained by comparing subsequent elements in the block, writing a 0 at position i if $H[i+1] = H[i] + 1$ and 1 otherwise. Because tables M, E and R are of size $O(n)$, the total space needed is $O(n \cdot \log n)$ bits.

Now, to answer $\text{RMQ}(l, r)$, if l and r occur in different blocks, compute (1) the minimum from l to the end of l's block using arrays T and P, (2) the minimum of all blocks between l's and r's block using the precomputed queries on A' stored in table M, and (3) the minimum from the beginning of r's block to r, again using T and P. Finally, return the position where the overall minimum occurs, possibly employing B. If l and r occur in the same block, just answer an in-block-query from l to r. In both cases, the time needed for answering the query is constant.

3 Our New Algorithm

This section describes our new algorithm for the RMQ-problem. The array A to be preprocessed is (conceptually) divided into superblocks $B'_1, \ldots, B'_{n/s'}$ of size $s' := \log^{2+\epsilon} n$, where B'_i spans from $A[(i - 1)s']$ to $A[is' - 1]$. Here, ϵ is an arbitrary constant greater than 0. Likewise, A is divided into (conceptual) blocks $B_1, \ldots, B_{n/s}$ of size $s := \log n/(2+\delta)$. Again, $\delta > 0$ is a constant. For the sake of simplicity we assume that s' is a multiple of s. We will preprocess long queries by a two-level step due to Sadakane [8], and short queries will be precomputed by a combination of the Four-Russians-Trick (as presented in [15]) with the method from [10]. The general idea is that a query from l to r can be divided into at most five sub-queries (see also Fig. 1): one *superblock-query* that spans several superblocks, two *block-queries* that span the blocks to the left and right of the superblock-query, and two *in-block-queries* to the left and right of the block-queries. From now on, we assume that the $\prec$-relation is used for answering RMQs, such that the answers become unique.

3.1 A Succinct Data Structure for Handling Long Queries

We first wish to precompute the answers to all RMQs that span over at least one superblock. Define a table $M'[0, n/s'-1][0, \log(n/s')]$. $M'[i][j]$ stores the position of the minimum in the sub-array $A[is', (i+2^j)s'-1]$. As in Sect. 2.1, $M'[i][0]$ can be filled by a linear pass over the array, and for $j > 0$ we use a dynamic programming approach by setting $M'[i][j] = \arg\min_{k \in \{M'[i][j-1], M'[i+2^{j-1}][j-1]\}}\{A[k]\}$.

In the same manner we precompute the answers to all RMQs that span over at least one block, but *not* over a superblock. These answers are stored in a table $M[0, n/s-1][0, \log(s'/s)]$, where $M[i][j]$ stores the minimum of $A[is, (i+2^j)s-1]$. Again, dynamic programming can be used to fill table M in optimal time.

3.2 A Succinct Data Structure for Handling Short Queries

We now show how to store all necessary information for answering in-block-queries in a table P.

Theorem 1 ([15]). *Let A and B be two arrays, both of size s. Then* $\text{RMQ}_A(l, r) = \text{RMQ}_B(l, r)$ *for all $0 \leq l \leq r < s$ if and only if $\mathcal{C}^{can}(A) = \mathcal{C}^{can}(B)$.* $\square$

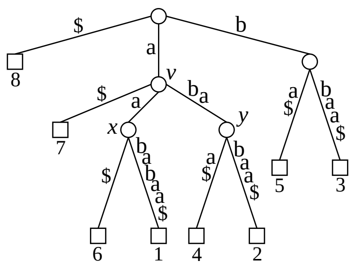
It is well known that the number of binary trees with s nodes is C_s, where C_s is the s'th *Catalan Number* defined by $C_s = \frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1 + O(s^{-1}))$. Because the Cartesian tree is a binary tree, this means that table P does not have to store the in-block-queries for all n/s *occurring* blocks, but only for $4^s/(\sqrt{\pi}s^{3/2})(1+O(s^{-1}))$ *possible* blocks. We use the method described in [10] to represent the answers to all RMQ-queries inside one block.

Computing the Block Types In order to index table P, it remains to show how to fill array T; i.e., how to compute the types of the blocks B_i occurring in A in linear time. Thm. 1 implies that there are only C_s different types of arrays of size s, so we are looking for a surjection

$$t : \mathcal{A}_s \to \{0, \ldots, C_s - 1\}, \text{ and } t(B_i) = t(B_j) \text{ iff } \mathcal{C}^{\text{can}}(B_i) = \mathcal{C}^{\text{can}}(B_j) \ , \qquad (1)$$

```
Input: block $B_j$
Output: $t(B_j)$, the type of $B_j$
let $R[0, s-1]$ be an array
$R[0] \leftarrow -\infty$
$q \leftarrow s, N \leftarrow 0$
for $i \leftarrow 1, \ldots, s$ do
    while
    $R[q+i-s-1] > B_j[i-1]$ do
        $N \leftarrow N + C_{(s-i)q}$
        $q \leftarrow q - 1$
    end
    $R[q+i-s] \leftarrow B_j[i-1]$
end
return $N$
```

Fig. 2. An algorithm to compute the type of a block.



	1	2	3	4	5	6	7	8
SA=	8	7	6	1	4	2	5	3
LCP=	−1	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{1}$	$\underline{3}$	0	2

Fig. 3. The suffix tree (top) and the suffix- and LCP-array (bottom) for string $t = aababaa\$$.

where $\mathcal{A}_s$ is the set of arrays of size s. The reason for requiring that B_i and B_j have the same Canonical Cartesian Tree is given by Thm. 1 which tells us that in such a case both blocks share the same RMQs. The algorithm in Fig. 2 shows how to compute the block type directly. It makes use of the so-called *ballot numbers* C_{pq}, defined by

$$C_{00} = 1, C_{pq} = C_{p(q-1)} + C_{(p-1)q}, \text{ if } 0 \le p \le q \ne 0, \text{ and } C_{pq} = 0 \text{ otherwise.} \quad (2)$$

The basic idea of the algorithm in Fig. 2 is that it simulates a walk in a certain graph, as explained in [15].

Theorem 2 ([15]). *The algorithm in Fig. 2 computes a function satisfying the conditions given in* (1) *in* $O(s)$ *time.* □

3.3 Space Analysis

Table M' has dimensions $n/s' \times \log(n/s') = n/\log^{2+\epsilon} n \times \log(n/\log^{2+\epsilon} n)$ and stores values up to n; the total number of bits needed for M' is therefore $n/\log^{2+\epsilon} n \times \log(n/\log^{2+\epsilon} n) \cdot \log n = o(n)$. Table M has dimensions $n/s \times \log(s'/s) = (2+\delta)n/\log n \times \log((2+\delta)\log^{1+\epsilon} n)$. If we just store the offsets of the minima then the values do not become greater than s'; the total number of bits needed for M is therefore $O(n/\log n \times \log(\log^{1+\epsilon} n) \cdot \log(\log^{2+\epsilon} n)) = o(n)$. To store the type of each block, array T has length $n/s = (2+\delta)n/\log n$, and because of Thm. 1 the numbers do not get bigger than $O(4^s/s^{3/2})$. This means that the number of bits to encode T is

$$\frac{n}{s} \cdot \log(O(4^s/s^{3/2})) = \frac{n}{s}(2s - O(\log s)) = 2n - O(n/\log n \log\log n) = 2n - o(n) \ .$$

Finally, by Sect. 3.2, and as [10] allows to store all queries inside of one block in $O(s \cdot s)$ bits, table P can be stored in

$$O\left(\frac{4^s}{s^{3/2}} s \cdot s\right) = O\left(n^{2/(2+\delta)}\sqrt{\log n}\right) = O\left(n^{1-\frac{1}{2/\delta+1}}\sqrt{\log n}\right) = o(n/\log n)$$

bits. Thus, the total space needed is $2n + o(n)$ bits. It is interesting that the leading term ($2n$) comes from table T, i.e., from remembering the type of all blocks occurring in A. One can wonder if this is really necessary. The following theorem says that, asymptotically, one cannot do better than this, provided that the array is *only* used for minimum evaluations. To model this situation, we introduce the so-called *min-probe model*, where we only count evaluations of $\arg\min\{A[i], A[j]\}$, and all other computations and accesses to additional data structures (but *not* to A) are free (note the analogy to the cell-probe model [16]).

Theorem 3. *For an array A of size n one needs at least $2n - o(n)$ additional bits to answer $\text{RMQ}_A(l, r)$ in $O(1)$ for all $0 \le l \le r < n$ in the min-probe model.*

Proof. Let $\mathcal{D}_A$ be the additional data structure. To evaluate $\text{RMQ}_A(l, r)$ in $O(1)$, the algorithm can make $O(k)$ argmin-evaluations (constant k). The algorithm's decision on which $i \in [l, r]$ is returned as the minimum is based on the outcome of the $K := 2^k$ possible outcomes of these argmin-evaluations, and possibly other computations in $\mathcal{D}_A$. Now suppose there are less than $C_n/(K+1)$ different such $\mathcal{D}_A$'s. Then there exists a set $\{A_0, \ldots, A_K\}$ of $K + 1$ arrays of length n with $\mathcal{D}_{A_i} = \mathcal{D}_{A_j}$ for all i, j, but with pairwise different Cartesian Trees. Because the algorithm can return different answers for at most K of these arrays, for at least two of them (say A_x and A_y) it gives the same answer to $\text{RMQ}(l, r)$ for all l, r. But A_x and A_y have different Cartesian trees, so there must be at least one pair of indices l', r' for which $\text{RMQ}_{A_x}(l', r') \neq \text{RMQ}_{A_y}(l', r')$. Contradiction. So there must be at least $C_n/(K+1)$ different choices for $\mathcal{D}_A$; thus the space needed to represent $\mathcal{D}_A$ is at least $\log(C_n/(K+1)) = 2n - o(n) - O(1)$ bits. □

4 Improvements in the Enhanced Suffix Array

Suffix trees are a very powerful tool for many tasks in pattern matching. Because of their large space consumption, a recent trend in text indexing tries to replace them by adequate array-based data structures. Kasai et al. [17] showed how algorithms based on a bottom-up traversal of a suffix tree can be simulated by a parallel scan of the suffix- and LCP-array. The Enhanced Suffix Array (ESA) [3] takes this approach one step further by also being capable of simulating top-down traversals of the suffix tree. This, however, requires the addition of another array to the suffix- and LCP-array, the so-called *child-table*. Essentially, the child table captures the information on how to move from an internal node to its children. This table requires $O(n)$ words (or $O(n \cdot \log n)$ bits). We show in this section that the RMQ-information on the LCP-array can be used as an alternative representation of the child-table, thus reducing the space requirement to $O(n/\log n)$ words (precisely, to $2n + o(n)$ bits). Note that our representation is not only less space-consuming than [3], but also much simpler. Throughout this section, t is a text of length n.

4.1 Enhanced Suffix Arrays

In its simplest form, the ESA consists of the suffix- and LCP-array for t. The basic idea of the ESA is that internal nodes of the suffix tree correspond to certain intervals (so-called ℓ-intervals) in the LCP-array (recall the definition of SA and LCP in Sect. 1.2):

Theorem 4 ([3]). *Let T, SA, LCP be t's suffix tree, suffix- and LCP-array, respectively. Then the following is equivalent:*

1. *There is an internal node in T representing a sub-word ϕ of t.*
2. *There exist $1 \leq l < r \leq n$ s.t. (a) $\mathsf{LCP}[l] < |\phi|$ and $\mathsf{LCP}[r+1] < |\phi|$, (b) $\mathsf{LCP}[i] \geq |\phi|$ for all $l < i \leq r$ and $\phi = t_{\mathsf{SA}[q]..\mathsf{SA}[q]+|\phi|-1}$, and (c) $\exists\, q \in \{l+1, \dots, r\}$ with $\mathsf{LCP}[q] = |\phi|$.*

The pair of indices satisfying point 2 of the above theorem are said to form a $|\phi|$-interval $[l:r]$ (denoted as $|\phi|$-$[l:r]$), and each position q satisfying (c) is called a $|\phi|$-index. For example, in the tree in Fig. 3, node v corresponds to the 1-interval $[2{:}6]$ in LCP and has 1-indices 3 and 5.

Let ℓ-$[l{:}r]$ be any such interval, corresponding to node v in T. Then if there exists an $\ell' > \ell$ such that there is an ℓ'-interval $[l'{:}r']$ contained in $[l{:}r]$, and no super-interval of $[l'{:}r']$ has this property, then ℓ'-$[l'{:}r']$ corresponds to an internal child of v in T. E.g., in Fig. 3, the two child-intervals of 1-$[2{:}6]$ representing internal nodes in T are 2-$[3:4]$ and 3-$[5:6]$, corresponding to nodes x and y. The connection between ℓ-indices and child-intervals is as follows [3, Lemma 6.1]:

Lemma 1. *Let $[l{:}r]$ be an ℓ-interval. If $i_1 < i_2 < \dots < i_k$ are the ℓ-indices in ascending order, then the child intervals of $[l{:}r]$ are $[l{:}i_1 - 1], [i_1{:}i_2], \dots, [i_k{:}r]$. (Singleton intervals are leaves!)*

With the help of Lemma 1 it is possible to simulate top-down traversals of the suffix tree: start with the interval 0-$[1:n]$ (representing the root), and at each interval calculate the child-intervals by enumerating their ℓ-indices. To find the ℓ-indices in constant time, the authors of [3] introduce a new array $C[1, n]$, the so-called child-table, occupying $n \log n$ bits of space.

4.2 An RMQ-based Representation of the Child-Table

The following lemma is the key to our new technique:

Lemma 2. *Let $[l : r]$ be an ℓ-interval. Then its ℓ-indices can be obtained in ascending order by $i_1 = \textsc{rmq}_{\mathsf{LCP}}(l+1, r)$, $i_2 = \textsc{rmq}_{\mathsf{LCP}}(i_1 + 1, r), \dots$, as long as $\mathsf{LCP}[\textsc{rmq}_{\mathsf{LCP}}(i_k + 1, r)] = \ell$.*

```
    Input: pattern $p = p_{1..m}$ to be found in $t$
    Output: interval of $p$ in $\mathsf{SA}$ or negative answer
1   $c \leftarrow 0$, $found \leftarrow$ true, $l \leftarrow 1$, $r \leftarrow n$
2   repeat
3       $[l, r] \leftarrow$ getChild$(l, r, p_{c+1})$
4       if $[l : r] = \emptyset$ then return "not found"
5       if $l = r$ then $M \leftarrow m$
6         else $M \leftarrow \min\{\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(l+1, r)], m\}$
7       $found \leftarrow (p_{c+1..M-1} = t_{\mathsf{SA}[l]+c..\mathsf{SA}[l]+M-1})$
8       $c \leftarrow M$
9   until $l = r \vee c = m \vee found =$ false
10  if $found$ then return $[l : r]$
11    else return "not found"
```

```
11  function getChild $(l, r, a)$
12  $r_{old} \leftarrow r$
13  $r \leftarrow \text{RMQ}_{\mathsf{LCP}}(l+1, r_{old})$
14  $\ell \leftarrow \mathsf{LCP}[r]$
15  repeat
16      if $t_{\mathsf{SA}[l]+\ell} = a$ then
17          return $[l : r-1]$
18      end
19      $l \leftarrow r$
20      $r \leftarrow \text{RMQ}_{\mathsf{LCP}}(l+1, r_{old})$
21  until $l = r_{old} \vee \mathsf{LCP}[r] > \ell$
22  if $t_{\mathsf{SA}[l]+\ell} = a$ then
23      return $[l : r_{old}]$
24  else return $\emptyset$
```

Fig. 4. How to locate a pattern of length m in a text in $O(m)$ time.

Proof. Because of point 2(b) in Thm. 4, the LCP-values in $[l+1{:}r]$ cannot be less than ℓ. Thus any position in this interval with a minimal LCP-value must be an ℓ-index of $[l{:}r]$. On the other hand, if $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(i_k+1, r)] > \ell$ for some k, then there cannot be another ℓ-index in $[i_k+1{:}r]$. Because RMQ always yields the position of the *leftmost* minimum if this is not unique, we get the ℓ-indices in ascending order. □

With Lemma 1 this allows us to compute the child-intervals by preprocessing the LCP-array for RMQ. As an example, we can retrieve the 1-indices of 1-$[2{:}6]$ as $i_1 = \text{RMQ}_{\mathsf{LCP}}(3, 6) = 3$ giving interval $[2{:}2]$ (corresponding to leaf 7 in Fig. 3), $i_2 = \text{RMQ}_{\mathsf{LCP}}(4, 6) = 5$ giving $[3{:}4]$ (corresponding to node x). Because $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(6, 6)] = \mathsf{LCP}[6] = 3 > 1$, there are no more 1-indices to be found, so the last child-interval is $[5{:}6]$ (corresponding to y).

Theorem 5. *Any algorithm based on a top-down traversal of a suffix tree can be replaced by data structure using* $|\mathsf{SA}| + 4n + o(n)$ *bits without affecting its time bounds, where* $|\mathsf{SA}|$ *denotes the space consumed by the suffix array.*

Proof. With Lemmas 1 & 2, preparing the LCP-array for RMQ is enough for retrieving the child-intervals. Because of Thm. 3, this requires $2n + o(n)$ bits. [8] has shown that the LCP-array can be stored in $2n + o(n)$ bits as well, while retaining constant access to $\mathsf{LCP}[i]$. With Thm. 4, the claim follows. □

4.3 Application to Pattern Matching

For a given pattern p of length m, the task is to answer in $O(m)$ time whether p is a substring of t, and to locate all *occ* occurrences of p in $O(m+occ)$ time. With a plain suffix array these time bounds cannot be achieved (they are $O(m \log n)$ and $O(m \log n + occ)$, respectively). Note that the ESA is still the method of choice for this task, as all known compressed indexes [7,9] have asymptotic worse matching or locating times, even for the alphabet size $|\Sigma|$ being a constant.

The algorithm to locate a pattern p is shown in Fig. 4. The invariant of the algorithm is that *found* is `true` if $p_{1..c}$ occurs in t and has the interval $[l:r]$ in SA. In each step of the loop, method `getChild`(l, r, a) is used to find the subinterval of $[l:r]$ that stores the suffixes having $p_{1..c+1}$ as a prefix. This is done exactly as described in Sect. 4.2. Because $|\Sigma|$ is a constant, function `getChild` takes constant time. (Note that for large $|\Sigma| \in \omega(\log n)$ one would actually drop back to binary search, so even for large alphabets `getChild` never takes more than $O(\log n)$ time.) The if-statement in lines 5–6 distinguishes between internal nodes ($l > r$) and leaves ($l = r$). The actual pattern matching is done in line 7 of the algorithm. Because c is increased by at least 1 in each step of the loop, the running time of the whole algorithm is $O(m)$.

To retrieve all *occ* occurrences of p, get p's interval $[l:r]$ in SA by the method explained above, and then return the set of positions $\{\mathsf{SA}[l], \mathsf{SA}[l+1], \ldots, \mathsf{SA}[r]\}$. This takes $O(occ)$ additional time.

5 Implementation Details

We implemented the algorithm from Sect. 3 in C++ (available at the first author's home page). A good trade-off between time and space is to fix the block size s to 2^3 and the superblock size s' to 2^8, and to introduce an "intermediate" block-division of size $s'' = 2^4$. As the intermediate blocks consist of two blocks of size s, their RMQs can be answered with two look-ups to table P, and therefore do not have to be stored at all. The advantage of this intermediate layer is that it reduces the space of table M'. In total, our implementation uses $\frac{7}{8}n$ bytes.

We compared our algorithm with the most space-conscious algorithm for RMQ due to Alstrup et al. [10], which uses only $2n + o(n)$ additional *words* of space. We could not compare with the succinct method [2] because it is not yet publicly available. We performed some tests on random arrays of length up to $n = 2^{27} \approx 1.34 \times 10^6$ (i.e., the array uses space up to $2^{27} \times 4 = 2^{29} = 512$MB). We found that our method is not only much less space consuming than [10] (by a factor of ≈ 8), but also faster in constructing the index (by a factor of ≈ 2.5). This shows that the extra work from Sect. 3 pays off. The query time for short queries ($\log n/2$) is about the same for both methods (with a slight advantage for our method), whereas for long queries ($n/100$) our method is slowed down by only a factor of two, which can be coped with given the reduction in space.

6 Conclusions

We have presented a direct and easy-to-implement data structure for constant-time RMQ-retrieval that uses $2n + o(n)$ bits of additional space, which is asymptotically optimal. This led to direct improvements in the Enhanced Suffix Array. Tests confirmed the practical utility of our method. We finally note that our algorithm is also easy to implement on PRAMs (or real-world shared-memory machines), where with n/t processors the preprocessing runs in time $\Theta(t)$ if $t = \Omega(\log n)$, which is work-optimal.

References

1. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2) (1993) 221–242
2. Sadakane, K.: Space-efficient data structures for flexible text retrieval systems. In: Proc. ISAAC. Volume 2518 of LNCS, Springer (2002) 14–24
3. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1) (2004) 53–86
4. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2) (2005) 75–94
5. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. of the ACM Symp. on Theory of Computing. (1984) 135–143
6. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. FOCS, IEEE Computer Society (1989) 549–554
7. Navarro, G., Mkinen, V.: Compressed full-text indexes. ACM Computing Surveys (to appear 2007) Preliminary version available at http://www.dcc.uchile.cl/˜gnavarro/ps/acmcs06.ps.gz
8. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proc. SODA, ACM/SIAM (2002) 225–237
9. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems (to appear 2007) Preliminary version available at http://tcslab.csce.kyushu–u.ac.jp/˜sada/papers/cst.ps
10. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: A survey and a new distributed algorithm. In: Proc. SPAA, ACM Press (2002) 258–264
11. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5) (1993) 935–948
12. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. SODA, ACM/SIAM (2002) 657–666
13. Chen, K.Y., Chao, K.M.: On the range maximum-sum segment query problem. In: Proc. ISAAC. Volume 3341 of LNCS, Springer (2004) 294–305
14. Tarjan, R.E., Vishkin, U.: An efficient parallel biconnectivity algorithm. SIAM J. Comput. **14**(4) (1985) 862–874
15. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Proc. CPM. Volume 4009 of LNCS, Springer (2006) 36–48
16. Yao, A.C.C.: Should tables be sorted? J. ACM **28**(3) (1981) 615–628
17. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. CPM. Volume 2089 of LNCS, Springer (2001) 181–192