Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE

Johannes Fischer and Volker Heun

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

Abstract. The Range-Minimum-Query-Problem is to preprocess an array such that the position of the minimum element between two specified indices can be obtained efficiently. We present a direct algorithm for the general RMQ-problem with linear preprocessing time and constant query time, without making use of any dynamic data structure. It consumes less than half of the space that is needed by the method by Berkman and Vishkin. We use our new algorithm for RMQ to improve on LCA-computation for binary trees, and further give a constant-time LCE-algorithm solely based on arrays. Both LCA and LCE have important applications, e.g., in computational biology. Experimental studies show that our new method is almost twice as fast in practice as previous approaches, and asymptotically slower variants of the constant-time algorithms perform even better for today's common problem sizes.

1 Introduction

The problem of finding the *lowest common ancestor* (LCA) of a pair of nodes in a tree has attracted much attention in the past three decades, starting with Aho et al. [1]. It is not only algorithmically beautiful, but also has numerous applications, most importantly in the area of string processing and computational biology, where LCA is often used in conjunction with suffix trees. There are several variants of the problem (see [2]), the most prominent being the one where the tree is static and known in advance, and there are several queries to be answered on-line. In this case it makes sense to spend some time on *preprocessing* the tree in order to answer future queries faster. In their seminal paper [2], Harel and Tarjan showed that an intrinsic preprocessing in time *linear* in the size of the tree is sufficient to answer LCA-queries in *constant* time. Their algorithm was later simplified by Schieber and Vishkin [3], but remained rather complicated.

A major breakthrough in practicable constant-time LCA-computation was made by Berkman and Vishkin [4], and again, in a simplified presentation, by Bender et al. [5, 6]. The key idea for this algorithm is the connection between LCA-queries on trees and *range minimum queries* on arrays (RMQs). Basically, an RMQ asks for the position of the minimum element between two specified indices, and this problem was shown to be linearly equivalent to the LCA-problem by Gabow et al. [7], in the sense that both problems can be transformed into each other in time linear in the size of the input. The reduction from LCA to RMQ is in fact a reduction to a *restricted* version of RMQ, where consecutive array elements differ by exactly 1. The authors give an algorithm for this restricted version of RMQ, which is then used to answer LCA-queries.

However, RMQs are not only of interest because they can be used to answer LCA-queries, but have their own right to exist. A recent trend in text indexing tries to substitute the powerful but rather space-consuming *suffix tree* by alternative array-based data structures, most prominently the *suffix array*, discovered independently by Gonnet et al. [8] and by Manber and Myers [9]. While this data structure supports string searches in time almost as good as suffix trees, Kasai et al. [10] and Abouelhoda et al. [11] went one step further and showed that the addition of another array to the suffix array, namely the LCP-array, is sufficient to simulate full tree traversals of the suffix tree. It is thus possible to change many (but not all) algorithms based on suffix trees such that they operate on arrays only. One important exception to this are algorithms that rely on constant-time LCA-retrieval, such as computing longest common extensions of strings (LCEs), and all algorithms based on constant-time LCE-computations.

It is well-known that LCA-queries on the leaves of a suffix tree correspond to RMQs on the LCP-array. So an algorithm that solves the RMQ-problem would make it possible to re-formulate many algorithms based on suffix trees *and* LCA-retrieval such that they operate on arrays only. Unfortunately, the LCP-array does not exhibit the nice property that subsequent elements differ by exactly one, so the algorithm for the restricted RMQ-problem cannot immediately be used for this purpose. Gabow et al. [7] give an algorithm to reduce the general RMQ-problem to the LCA-problem by transforming the array into a special kind of tree. Their method, explained in more detail in Sect. 2.2, has two major drawbacks: First, it doubles the size of the input, and second, even more importantly, it relies on dynamic structures (trees) during the preprocessing. This resembles the suffix-tree/suffix-array duality: It *is* possible to infer the array from the tree; nevertheless, direct construction algorithms for the array are well studied.

Our paper overcomes this very dilemma by presenting the first[1] *direct* algorithm for the general RMQ-problem with linear preprocessing time and constant query time, without making use of *any* dynamic data structure (Sect. 3). It is also less space-consuming than previous approaches, as it uses only $4n + O(\sqrt{n} \log n)$ words of extra space, a major improvement compared with the $9n + O(\sqrt{n} \log^2 n)$ words plus the space for the tree used by the currently best algorithm. (Both O-constants are small.) In Sect. 4, we stress the impact of our new method by showing that it leads to improvements in the LCA-computation for binary trees, and further to the first constant-time LCE-algorithm solely based on arrays. In Sect. 5, we show that our RMQ-method is faster in practice than previous constant-time approaches (and therefore also the methods from Sect. 4). We will also see that for today's common problem sizes it makes more sense to use methods that answer long queries in constant time, but short queries in time logarithmic in

[1] By the time of writing we were unaware of another direct algorithm for RMQ [12].

the query length. These asymptotically *slower* RMQ-algorithms are slightly less space consuming than the constant-time approaches, and also faster in practice.

2 Definitions and Previous Results

The *Range Minimum Query* (RMQ) problem is defined as follows: given an array $A[1,n]$ of elements from a totally ordered set (with order relation "$\leq$"), $\textsc{rmq}_A(i,j)$ returns the index of a smallest element in $A[i,j]$, i.e., $\textsc{rmq}_A(i,j) = \arg\min_{k\in\{i,\dots,j\}}\{A[k]\}$. (The subscript A will be omitted if the context is clear.) The most naive algorithm for this problem searches the array from i to j each time a query is presented, resulting in $O(n)$ query time. As mentioned in the introduction, we consider the variant where A is first preprocessed in order to answer future queries faster. Following the notation from [6], we say that an algorithm with preprocessing time $p(n)$ and query time $q(n)$ has complexity $\langle p(n), q(n)\rangle$. Thus, the naive method described above would be $\langle O(1), O(n)\rangle$, because it requires no preprocessing.

The following definition [13] will be central for both our algorithm and that of [4].

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 similarly 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 is unique.

In [6] an algorithm for constructing $\mathcal{C}^{\text{can}}(A)$ is given as follows. Let $\mathcal{C}_i^{\text{can}}(A)$ be the Canonical Cartesian Tree for $A[1,i]$. Then $\mathcal{C}_{i+1}^{\text{can}}(A)$ is obtained by climbing up from the rightmost leaf of $\mathcal{C}_i^{\text{can}}(A)$ to the root, thereby finding the position where $A[i+1]$ belongs. To be precise, let $v_1,\dots,v_k$ be the nodes of the rightmost path in $\mathcal{C}_i^{\text{can}}(A)$ with labels $l_1,\dots,l_k$, respectively, where v_1 is the root and v_k is the rightmost leaf. Let m be defined such that $A[l_m] \leq A[i+1]$ and $A[l_{m'}] > A[i+1]$ for all $m < m' \leq k$. To build $\mathcal{C}_{i+1}^{\text{can}}(A)$, create a new node w with label $i+1$ which becomes the right child of v_m, and the subtree rooted at v_{m+1} becomes the left child of w. This process inserts each element to the rightmost path exactly once, and each comparison removes one element from the rightmost path, resulting in a total $O(n)$ construction time to build $\mathcal{C}^{\text{can}}(A)$.

2.1 An $\langle O(n\log n), O(1)\rangle$-Algorithm for RMQ

We briefly present a simple method [6] to answer RMQs in constant time using $O(n\log n)$ space. This algorithm will be used to answer 'long' RMQs both in our algorithm and that of [4][2]. The idea is to precompute all RMQs whose length is a power of two. For every $1 \le i \le n$ and every $1 \le j \le \lfloor \log n \rfloor$ 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 $O(n \log n)$ space and can be filled in optimal 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 $\textsc{rmq}(i, j)$, 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 $\textsc{rmq}(i, j) = \arg\min_{k \in \{M[i][l], M[j-2^l+1][l]\}} \{A[k]\}$.

2.2 The $\langle O(n), O(1) \rangle$-Algorithm for RMQ by Berkman and Vishkin

This section describes the solution to the general RMQ-problem as a combination of the results obtained in [4] and [7]. We follow the presentation from [6].

$\pm$1RMQ is a special case of the RMQ-problem, where consecutive array elements differ by exactly 1. The solution to the general RMQ-problem given in [4] (from now on called Berkman-Vishkin algorithm) starts by reducing RMQ to $\pm$1RMQ as follows: given an array $A[1, n]$ to be preprocessed for RMQ, build $\mathcal{C}^{\text{can}}(A)$ as shown above. Then perform a Euler Tour[3] in this tree, storing the labels of the visited nodes in an array $E[1, 2n - 1]$, and their respective heights in $H[1, 2n - 1]$. Further, store the position of the first occurrence of $A[i]$ in the Euler Tour in a representative array $R[1, n]$. 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 $\textsc{rmq}_A(i, j) = E[\pm1\textsc{rmq}_H(R[i], R[j])]$. Note in particular the doubling of the input when going from A to H; i.e., H has size $n' := 2n - 1$. We now sketch the solution to the $\pm$1RMQ-problem.

To solve $\pm$1RMQ on H, partition H into blocks of size $\frac{\log n'}{2}$.[4] 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. Now A' is preprocessed using the algorithm from Sect. 2.1, occupying $O(\frac{2n'}{\log n'} \log \frac{2n'}{\log n'}) = O(n)$ space. This preprocessing enables out-of-block queries (i.e., queries that span over several blocks) to be answered in $O(1)$. It remains to show how in-block-queries are handled. This is done with the so-called Four-Russians-Trick [15] where one precomputes the answers to all possible queries when the number of possible instances is sufficiently small. The authors of [6] noted that due to the $\pm$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 $\pm$1-vector of length $2^{1/2 \log n' - 1} = O(\sqrt{n'})$. For each such block precompute all $\frac{1}{2} \frac{\log n'}{2} (\frac{\log n'}{2} + 1)$ possible RMQs and store them in a table P of total size $O(\sqrt{n'} \log^2 n') = O(n)$. To index table P, precompute the *type* of each block and store it in array $T[1, \frac{2n'}{\log n'}]$.

[2] The original description in [4] used a slightly more complicated algorithm, which is, however, equivalent to the one presented here.

[3] The name “Euler Tour” is derived from the Euler Tour-technique [14], and is not to be confused with a Eulerian circuit.

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

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. Table 1 summarizes the tables needed for the algorithm and their sizes (ignore the last column for now).

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

3 An Improved $\langle O(n), O(1)\rangle$-Algorithm for RMQ

Our aim is to solve the general RMQ-problem *without* constructing the Cartesian Tree first; in fact, without employing *any* dynamic data structure such as trees. We also wish to find a solution that does not double the input array, as the Berkman-Vishkin algorithm does. The key to our solution is the following theorem. (From now on, we assume that the $\prec$-relation is used for answering RMQs, such that the answers become unique.)

Theorem 1. *Let A and B be two arrays, both of size n. Then $\text{RMQ}_A(i,j) = \text{RMQ}_B(i,j)$ for all $1 \le i \le j \le n$ if and only if $\mathcal{C}^{can}(A) = \mathcal{C}^{can}(B)$.*

Proof. It is easy to see that $\text{RMQ}_A(i,j) = \text{RMQ}_B(i,j)$ for all $1 \le i \le j \le n$ iff the following three conditions are satisfied: (i) The minimum under "$\prec$" occurs at the same position m, i.e., $\arg\min A = \arg\min B = m$. (ii) $\forall 1 \le i \le j < m : \text{RMQ}_{A[1,m-1]}(i,j) = \text{RMQ}_{B[1,m-1]}(i,j)$. (iii) $\forall m < i \le j \le n : \text{RMQ}_{A[m+1,n]}(i,j) = \text{RMQ}_{B[m+1,n]}(i,j)$. Due to the definition of the Canonical Cartesian Tree, points (i)–(iii) are true if and only if the root of $\mathcal{C}^{\text{can}}(A)$ equals the root of $\mathcal{C}^{\text{can}}(B)$, and $\mathcal{C}^{\text{can}}(A[1,m-1]) = \mathcal{C}^{\text{can}}(B[1,m-1])$, and $\mathcal{C}^{\text{can}}(A[m+1,n]) = \mathcal{C}^{\text{can}}(B[m+1,n])$. This is true iff $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$. $\square$

It is well known that the number of binary trees with n nodes is C_n, where $C_n = \frac{1}{n+1}\binom{2n}{n} = 4^n/(\sqrt{\pi}n^{3/2})(1+o(1))$ is the nth *Catalan Number.*

Lemma 1. *It is possible to precompute the answers to all possible range minimum queries on arrays of size s in a table P of size $O(4^s\sqrt{s})$.*

Proof. Because the Cartesian Tree is a binary tree with s nodes, table P has $O(\frac{4^s}{s^{3/2}})$ rows for each possible type of block. For each type we need to precompute $\text{RMQ}(i,j)$ for all $1 \le i \le j \le s$, so the number of columns in P is $O(s^2)$. $\square$

We now come to the description of our $\langle O(n), O(1)\rangle$-algorithm for the general RMQ-problem. Like the ± 1RMQ-algorithm presented in Sect. 2.2 it is an application of the Four-Russians-Trick. However, Lemma 1 allows us to apply the trick

Table 1. Additional space needed by the $\langle O(n), O(1)\rangle$-algorithms for RMQ (in words)

Array/Table	Berkman-Vishkin	our algorithm
E, H, R	$2(2n-1)+n=5n-2$	(arrays not needed)
A', B, T	$3\frac{2n}{\log(2n)/2}=12n/\log(2n)$	$3\frac{n}{\log(n)/4}=12n/\log n$
M	$4n+4n/\log(2n)-4n\log\log(2n)/\log(2n)$	$4n+8n/\log n-4n\log\log n/\log n$
P	$\sqrt{n}\log^2 n/(8\sqrt{2})(1+o(1))$	$\sqrt{n}\log^{1/2} n/(4\sqrt{\pi})(1+o(1))$
total (simpl.)	$9n + O(\sqrt{n}\log^2 n)$	$4n + O(\sqrt{n}\log n)$

to *any* array (not only to those with the ± 1-property), which leads to substantial improvements. Start by partitioning the array A into blocks $B_1, \ldots, B_{n/s}$ of size $s := \frac{\log n}{4}$. Define two arrays A' and B of size $n/s = \frac{4n}{\log n}$, where $A'[i]$ stores the minimum of block B_i, and $B[i]$ stores the position of this minimum in A. Now A' is preprocessed using the algorithm from Sect. 2.1, occupying $O(\frac{4n}{\log n}\log\frac{4n}{\log n}) = O(n)$ space. Then precompute the answers to all possible queries on arrays of size s and store the results in a table P. According to Lemma 1, this table occupies $O(4^{(\log n)/4}(\frac{\log n}{4})^{1/2}) = O(n)$ space. Finally, compute the type of each block in A and store these values in array $T[1, \frac{4n}{\log n}]$. As this is not as obvious as in Sect. 2.2, it is explained in detail in the following subsection. A query $\textsc{rmq}(i, j)$ is now answered exactly as explained in the last paragraph of Sect. 2.2, namely by comparing at most three minima, depending on the blocks where i and j occur. Again, the time for answering a query is constant, leading to the $\langle O(n), O(1)\rangle$ time bounds stated before. See Table 1 for a comparison of the two methods (space for $\mathcal{C}(A)$ not included).

3.1 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

$$\textit{type}: \mathcal{A}_s \rightarrow \{0, \ldots, C_s - 1\}, \text{ and } \textit{type}(B_i)=\textit{type}(B_j) \text{ iff } \mathcal{C}^{\text{can}}(B_i)=\mathcal{C}^{\text{can}}(B_j), \quad (1)$$

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 most naive way to calculate the type would be to actually *construct* the Cartesian Tree of each block, and then use an inverse enumeration of binary trees [16] to compute its type. This, however, would counteract our aim to avoid dynamic data structures. The algorithm in Fig. 1 shows how to compute the block type directly. It makes use of the so-called *ballot numbers* C_{pq} [16], defined by

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

It can be proved that a closed formula for C_{pq} is given by $\frac{q-p+1}{q+1}\binom{p+q}{p}$ [16], which immediately implies that C_{ss} equals the s'th Catalan number C_s. If we look at

```
    **Input**: block $B_j$ of size $s$
    **Output**: $type(B_j)$
 1  let $rp$ be an array of size $s+1$
 2  $rp[1] \leftarrow -\infty$
 3  $q \leftarrow s, N \leftarrow 0$
 4  **for** $i \leftarrow 1, \ldots, s$ **do**
 5      **while** $rp[q+i-s] > B_j[i]$ **do**
 6          $N \leftarrow N + C_{(s-i)q}$
 7          $q \leftarrow q-1$
 8      **end**
 9      $rp[q+i+1-s] \leftarrow B_j[i]$
10  **end**
11  **return** $N$
```

Fig. 1. An algorithm to compute the type of a block

$$\begin{tikzcd}
0\,0 & & & & & \\
0\,1 \arrow[u] & 1\,1 \arrow[l] & & & & \\
0\,2 \arrow[u] & 1\,2 \arrow[u] \arrow[l] & 2\,2 \arrow[l] & & & \\
0\,3 \arrow[u] & 1\,3 \arrow[u] \arrow[l] & 2\,3 \arrow[u] \arrow[l] & 3\,3 \arrow[l] & & \\
0\,4 \arrow[u] & 1\,4 \arrow[u] \arrow[l] & 2\,4 \arrow[u] \arrow[l] & 3\,4 \arrow[u] \arrow[l] & 4\,4 \arrow[l] & \\
0\,5 \arrow[u] & 1\,5 \arrow[u] \arrow[l] & 2\,5 \arrow[u] \arrow[l] & 3\,5 \arrow[u] \arrow[l] & 4\,5 \arrow[u] \arrow[l] & 5\,5 \arrow[l] \\
{} \arrow[u] & {} \arrow[u] & {} \arrow[u] & {} \arrow[u] & {} \arrow[u] & {} \arrow[u]
\end{tikzcd}$$

Fig. 2. The infinite graph arising from the definition of the ballot numbers. Its vertices are $\boxed{p\ q}$ for all $0 \leq p \leq q$. There is an edge from $\boxed{p\ q}$ to $\boxed{(p-1)\ q}$ if $p > 0$ and to $\boxed{p\ (q-1)}$ if $q > p$.

the infinite directed graph shown in Fig. 2 then C_{pq} is clearly the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$, because of (2). This interpretation will be important for the proof of the following

Theorem 2. *The algorithm in Fig. 1 correctly computes the type of a block B_j of size s in $O(s)$ time, i.e., it computes a function satisfying the conditions given in* (1)*.*

Proof. (Sketch.) Intuitively, the algorithm simulates the algorithm for constructing $\mathcal{C}^{\text{can}}(B_j)$ given in Sect. 2. First note that array $rp[1, s+1]$ simulates the stack containing the labels of the nodes on the rightmost path of the partial Canonical Cartesian Tree $\mathcal{C}_i^{\text{can}}(B_j)$, with $q+i-s$ pointing to the top of the stack (i.e., the rightmost leaf), and $rp[1]$ acting as a 'stopper'. Now let l_i be the number of times the while-loop (lines 5–8) is executed during the ith iteration of the outer for-loop. Note that l_i equals the number of elements that are removed from the rightmost path when going from $\mathcal{C}_{i-1}^{\text{can}}(B_j)$ to $\mathcal{C}_i^{\text{can}}(B_j)$. Because one cannot remove more elements from the rightmost path than one has inserted, and each element is removed at most once, we have $\sum_{k=1}^{i} l_k \leq i$ for all $1 \leq i \leq s$. Thus, the sequence $l_1, \ldots, l_s$ corresponds to a path from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in Fig. 2 (and vice versa): in step i, go l_i steps upwards and one step to the left, and after step s go upwards until reaching $\boxed{0\ 0}$. The current position in the graph is $\boxed{(s-i+1)\ q}$, so every time one makes an upward step, N is incremented by the number of paths that have been 'skipped' by going upwards (line 6). This is exactly $C_{(s-i)q}$, the value of the cell to the left of the current one. The effect of this incrementation is that paths going from the current position to the left are assigned lower numbers than paths going upwards.

The proof is completed by noting that a Canonical Cartesian Tree can be uniquely described by $l_1, \dots, l_s$ satisfying $\sum_{k=1}^{i} l_k \leq i$ for all $1 \leq i \leq s$. □

4 Applications

This section sketches two easy (but non-trivial) new results on LCA and LCE that can be obtained with our RMQ-algorithm. Apart from yielding simpler and less space-consuming methods, we will see in Sect. 5 that one can also expect improvements in running times.

4.1 A Space Saving Algorithm for LCA on Binary Trees

The LCA-problem [1] is formally defined as follows: given a rooted tree T with n nodes and two vertices v and w, find the deepest node $\textsc{lca}_T(v, w)$ which is an ancestor of both v and w. Again, we consider the variant where T is static and the queries are posed on-line. As mentioned in the introduction, the RMQ- and the LCA-problem are closely related. In [7], it has been shown that an LCA-query on T basically corresponds to a ± 1RMQ-query on the heights of the nodes visited during an Euler-Tour in T. Because the size of an Euler-Tour is exactly $2n - 1$, this leads to an input doubling. We show in this section that using the algorithm presented in Sect. 3 overcomes this problem for binary trees.

Let T be a rooted binary tree with n nodes. First perform an *inorder tree walk* in T and store it in an array $I[1, n]$. Further, store the heights of each node in $H[1, n]$, i.e., $H[i]$ is the height of node $I[i]$ in T. Finally, let R be the inverse array of I, i.e., $I[R[i]] = i$. It is then easy to see that $\textsc{lca}_T(v, w) = I[\textsc{rmq}_H(R[v], R[w])]$: the elements in I between $R[v]$ and $R[w]$ are exactly the nodes encountered between v to w during an inorder tree walk in T, so the range minimum query returns the position k in H of the shallowest such nodes. As the LCA of v and w must be encountered between v and w during the inorder tree walk, $\textsc{lca}(v, w)$ is given by $I[k]$.

The extra space needed is $7n + O(\sqrt{n \log n})$ words: $4n + O(\sqrt{n \log n})$ words from Table 1 for the RMQ-preprocessing, plus $3n$ words for the arrays I, H and R. This *is* an improvement compared with the $9n + O(\sqrt{n} \log^2 n)$ words needed if one were to use the LCA-algorithm presented in [4]. We note that our result could also be generalized to arbitrary trees; the space reduction, however, is only relevant if the number of internal nodes is relatively close to the number of leaves.

4.2 An Improved Algorithm for Longest Common Extensions

The problem of *longest common extensions* is defined for a static string t of size n: given two indices i and j, $\textsc{lce}_t(i, j)$ returns in $O(1)$ the length of the longest common prefix of t's suffixes starting at position i and j; i.e., $\textsc{lce}_s(i, j) = \max\{k : t_{i,\dots,k} = t_{j,\dots,k}\}$.[5] The problem has numerous applications in string

[5] LCE is often defined for *two* strings t' and t'' s.th. i is an index in t' and j in t''. This can be transformed to our definition by setting $t = t'\#t''$, where $\#$ is a new symbol.

matching, e.g., for tandem repeats [17, 18], approximate tandem repeats [19], and inexact pattern matching [20, 21]. The easiest solution [22] to LCE combines suffix trees with constant-time LCA-retrieval: build a suffix tree T for t and preprocess it for LCA-queries. Then $\textsc{lce}(i,j)$ is given by the height of node $\textsc{lca}(v_i, v_j)$, where v_i and v_j are the leaves corresponding to suffix i and j, respectively.

The crucial point to observe is that the LCA-queries are only posed on the *leaves* of the suffix tree T for t. It is well-known [22,11] that there is a one-to-one correspondence between the leaves of T and the elements of the corresponding *suffix array* [8,9] SA, and also between the heights of T's internal nodes and the *LCP-array* LCP for SA. Basically, SA describes the order of the suffixes of t, and LCP stores the lengths of the longest common prefixes of t that are consecutive in SA. This gives us all the ingredients we need for our new LCE-algorithm: compute SA and its inverse SA^{-1} for t.[6] Further, compute the LCP-array for t in linear time [10, 24] and store it in LCP. (SA is not further needed at this point and can thus be deleted.) Then prepare LCP for RMQs as presented in Sect. 3. It is now easy to see that $\textsc{lce}(i,j) = \textsc{rmq}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i]+1, \mathsf{SA}^{-1}[j])$.

Note that this is the first algorithm that solves the LCE-problem without using trees of any form.[7] Apart from SA^{-1} and LCP, the space needed is $4n + O(\sqrt{n \log n})$ words. Compare this with $9n + O(\sqrt{n}\log^2 n)$ words plus the space for the Cartesian Tree that would be needed if one were to preprocess LCP for RMQ using the Berkman-Vishkin algorithm (not to talk about the solution based on suffix trees).

5 Practical Considerations

We now wish to evaluate the practical performance of our new algorithm by comparing it with the Berkman-Vishkin algorithm. We further include three non-$\langle O(n), O(1)\rangle$-algorithms in our evaluation:

1. An algorithm that divides the array into blocks of size $\frac{\log n}{2}$ and preprocesses the block-minima for the out-of-block queries (i.e., it creates table A', B, T and M). The in-block-queries are handled naively (i.e., table P is *not* created). Call this method $\langle O(n), O(\log n)\rangle_2$.
2. The same as above with block size $\frac{\log n}{4}$. Call this method $\langle O(n), O(\log n)\rangle_4$.
3. The naive $\langle O(1), O(n)\rangle$-algorithm that requires no preprocessing.

We performed all tests on an Athlon XP3000 with 2GB of RAM under Linux. All programs were written in C++ and compiled using the same compiler options. All our figures are averages over 5 repetitions of each experiment.

[6] There are fast algorithms that construct SA and its inverse with only $o(n)$ extra space, e.g., [23].

[7] While this has the consequence that the algorithms [17,19,20,21] can be implemented without trees, it is not immediately obvious how to do this for [18] because it uses the tree structure also for representing the tandem repeats.

Fig. 3 shows the time spent on preprocessing by all methods except the naive one, because the latter does no preprocessing. As expected, the Berkman-Vishkin method is the slowest, which is due to the explicit construction of the Cartesian Tree. The preprocessing times for the other three methods are within the same order of magnitude, where our method is slightly slower than the two $\langle O(n), O(\log n)\rangle$-algorithms, as expected.

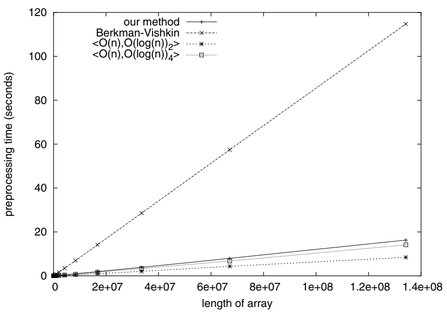


Fig. 3. Preprocessing times for varying array lengths

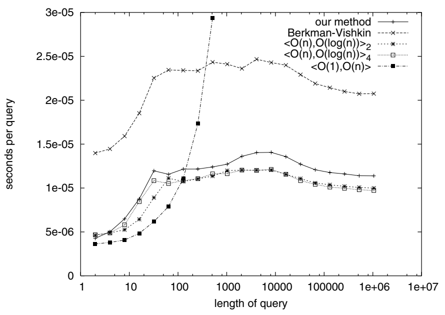
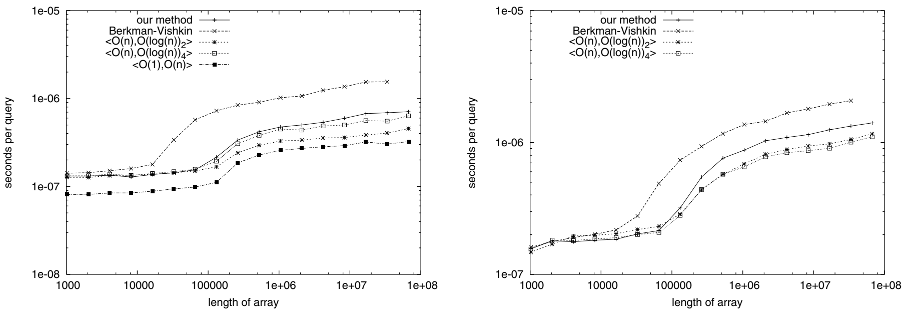


Fig. 4. The influence of different query lengths on the query time (w/o preproc.)

The next test was to evaluate the influence of the query length on the query time. We took a random array of length $n = 10^7$ and posed 10^6 random RMQs on this array. Fig. 4 shows the average query time for all five methods, and there are several points to note.

- As expected, the $\langle O(1), O(n)\rangle$-algorithm behaves linearly in the query length (note the logarithmic x-axis). It is very fast for short queries (up to length 100), but out of the questions for longer queries.
- Our $\langle O(n), O(1)\rangle$-algorithm is about twice as fast as the one by Berkman and Vishkin.
- The two methods with $O(\log n)$ query time are even slightly faster than our constant-time method. This is because quite some arithmetic is necessary to answer the in-block-queries in constant time. With block size $\frac{\log 10^7}{2} \approx 11$ the overhead for this is much too big.
- For all methods except the naive one the query time levels off for very long queries. We can only speculate that this is due to caching phenomena.

In a last test we checked up on the influence of the array length n on the query time. We performed separate tests for short and long random queries, where *short* means to be of length $\log n/2$ such that only in-block-queries are to be handled. *Long* queries were of length $n/100$. The largest arrays that we were able to handle on our computer were of length $\approx 6 \times 10^7$ for both tests. (Because of the input-doubling, the largest array length for the Berkman-Vishkin method was $\approx 3 \times 10^7$ for both tests.) See Fig. 5(a)–(b) for the results. In (a), the naive method is the best, for the same reasons as given before. The other four methods show the same performance as in Fig. 4. For the long queries in (b), the naive method was excluded for obvious reasons. Again, the two $\langle O(n), O(\log n)\rangle$-algorithms perform better than the $\langle O(n), O(1)\rangle$-methods, but our method is about twice as fast as the Berkman-Vishkin algorithm. It is interesting to see that both in (a) and (b) all methods exhibit a significant increase in running time at some point. This happens at roughly $n = 10^5$, whereas the Berkman-Vishkin method has this increase earlier. The effect can most likely be explained by the second-level-cache of the processor. Because of the input-doubling in the Berkman-Vishkin algorithm the cache size is reached earlier for this method. In summary, all our tests show that for practical applications with arrays up to length 10^8 or so it is advisable to use the $\langle O(n), O(\log n)\rangle_2$-algorithm. Unfortunately, our computer is not large enough to test when our algorithm becomes faster than the $\langle O(n), O(\log n)\rangle$-algorithms.



(a) Short queries of length $0.5 \log n$. (b) Long queries of length $n/100$.

Fig. 5. The influence of different array lengths on the query time (w/o preprocessing)

6 Summary and Discussion

We have seen a new method to answer range minimum queries in constant time after a linear preprocessing step. The key to our algorithm was the strong connection between Cartesian Trees and RMQs, reflected in the employment of the Catalan- and ballot numbers. This led to substantial improvements over previous RMQ-algorithms, namely a space reduction of more than 50%, the complete absence of dynamic data structures, and a boost in query time. We have also seen how our method leads to space reductions in the computation of lowest common ancestors in binary trees, and to an improved algorithm for the computation of longest common extensions in strings. On the practical side, we have seen that it is sometimes wiser to spend a little bit less effort in preprocessing, because even for large problem sizes (arrays up to length 10^8) asymptotically slower algorithms may perform faster in practice.

We finally note that our approach can be combined with the ideas from [25] to give the first *succinct* data structure for constant time RMQ, in the sense that the extra space needed is only $O(n)$ bits. We will elaborate on this in future work.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. SIAM J. Comput. **5** (1976) 115–132
2. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13** (1984) 338–355
3. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. **17** (1988) 1253–1262
4. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22** (1993)
5. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proc. LATIN, LNCS 1776, Springer (2000) 88–94
6. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57** (2005) 75–94
7. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. of the ACM STOC, ACM Press (1984) 135–143
8. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In Frakes, W.B., Baeza-Yates, R.A., eds.: Information Retrieval: Data Structures and Algorithms. Prentice-Hall (1992) 66–82
9. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22** (1993) 935–948
10. 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, LNCS 2089, Springer (2001) 181–192
11. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2** (2004) 53–86
12. 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
13. Vuillemin, J.: A unifying look at data structures. Comm. ACM **23** (1980) 229–239
14. Tarjan, R.E., Vishkin, U.: An efficient parallel biconnectivity algorithm. SIAM J. Comput. **14** (1985) 862–874
15. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph. Dokl. Acad. Nauk. SSSR **194** (1970, in Russian) 487–488, Engl. transl. in *Soviet Math. Dokl.*, **11** 1209–1210, 1975
16. Knuth, D.E.: The Art of Computer Programming Vol. 4, Fasc. 4: Generating All Trees; History of Combinatorial Generation. Addison-Wesley (2006)
17. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms **5** (1984) 422–432
18. Gusfield, D., Stoye, J.: Linear time algorithm for finding and representing all tandem repeats in a string. J. Comput. Syst. Sci. **69** (2004) 525–546
19. Landau, G., Schmidt, J.P., Sokol, D.: An algorithm for approximate tandem repeats. J. Comput. Biol. **8** (2001) 1–18
20. Myers, E.W.: An $O(nd)$ difference algorithm and its variations. Algorithmica **1** (1986) 251–266

21. Landau, G., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proc. STOC, ACM Press (1986) 220–230
22. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
23. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. In: Proc, ALENEX/ANALCO, SIAM Press (2005) 77–85.
24. Manzini, G.: Two space saving tricks for linear time lcp array computation. In: Proc. SWAT. LNCS 3111, Springer (2004) 372–383
25. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proc. SODA, ACM/SIAM (2002) 225–237