Two-Dimensional Range Minimum Queries[⋆]

Amihood Amir[1], Johannes Fischer[2], and Moshe Lewenstein[1]

[1] Computer Science Department,
Bar Ilan University,
Ramat Gan 52900, Israel
{moshe,amir}@cs.biu.ac.il
[2] Ludwig-Maximilians-Universität München,
Institut für Informatik,
Amalienstr. 17, D-80333 München
Johannes.Fischer@bio.ifi.lmu.de

Abstract. We consider the two-dimensional Range Minimum Query problem: for a static $(m \times n)$-matrix of size $N = mn$ which may be preprocessed, answer on-line queries of the form "where is the position of a minimum element in an axis-parallel rectangle?". Unlike the one-dimensional version of this problem which can be solved in provably optimal time and space, the higher-dimensional case has received much less attention. The only result we are aware of is due to Gabow, Bentley and Tarjan [1], who solve the problem in $O(N \log N)$ preprocessing time and space and $O(\log N)$ query time. We present a class of algorithms which can solve the 2-dimensional RMQ-problem with $O(kN)$ additional space, $O(N \log^{[k+1]} N)$ preprocessing time and $O(1)$ query time for any $k > 1$, where $\log^{[k+1]}$ denotes the iterated application of $k + 1$ logarithms. The solution converges towards an algorithm with $O(N \log^* N)$ preprocessing time and space and $O(1)$ query time. All these algorithms are significant improvements over the previous results: query time is optimal, preprocessing time is quasi-linear in the input size, and space is linear. While this paper is of theoretical nature, we believe that our algorithms will turn out to have applications in different fields of computer science, e.g., in computational biology.

1 Introduction

One of the most basic problems in computer science is finding the minimum (or maximum) of a list of numbers. An elegant and interesting dynamic version of this problem is the *Range Minimum Query (RMQ) problem*. The popular one dimensional version of this problem is defined as follows:

INPUT: An array $A[1..n]$ of natural numbers.
We seek to preprocess the array A in a manner that yields efficient solutions to the following queries:

[⋆] The second author was partially funded by the German Research Foundation (DFG, Bioinformatics Initiative).

QUERY: Given $1 \leq i \leq j \leq n$, output an index k, $i \leq k \leq j$, such that $A[k] \leq A[\ell]$, $i \leq \ell \leq j$.

Clearly, with $O(n^2)$ preprocessing time, one can answer such queries in constant time. Answering queries in time $O(j - i)$ needs no preprocessing. The surprising news is that linear time preprocessing can still yield constant time query solutions. The trek to this result was long. Harel and Tarjan [2] showed how to solve the *Lowest Common Ancestor (LCA)* problem with a linear time preprocessing and constant time queries. The LCA problem has as its input a tree. The query gives two nodes in the tree and requests the lowest common ancestor of the two nodes. It turns out that constructing a Cartesian tree of the array A and seeking the LCA of two indices, gives the minimum in the range between them [1].

The Harel-Tarjan algorithm was simplified by Schieber and Vishkin [3] and then by Berkman et al. [4] who presented optimal work parallel algorithms for the LCA problem. The parallelism mechanism was eliminated and a simple serial algorithm was presented by Bender and Farach-Colton [5]. In all above papers, there was an interplay between the LCA and the RMQ problems. Fischer and Heun [6] presented the first algorithm for the RMQ problem with linear preprocessing time, optimal $2n + o(n)$ bits of additional space, and constant query time that makes no use of the LCA algorithm. In fact, LCA can then be solved by doing RMQ on the array of levels of the tree's inorder tree traversal. This last result gave another beautiful motivation to the naturally elegant RMQ problem.

The problem of finding the minimum number in a given range is by no means restricted to one dimension. In this paper, we investigate the two-dimensional case. Consider an $(m \times n)$-matrix of $N = mn$ numbers. One may be interested in preprocessing it so that queries seeking the minimum in a given rectangle can be answered efficiently. Gabow, Bentley and Tarjan [1] solve the problem in $O(N \log N)$ preprocessing time and space and $O(\log N)$ query time.

We present a class of algorithms which can solve the 2-dimensional RMQ-problem with $O(kN)$ additional space, $O(N \log^{[k+1]} N)$ preprocessing time and $O(1)$ query time for any $k > 1$. The solution converges towards an algorithm with $O(N \log^* N)$ preprocessing time and space and $O(1)$ query time.

2 Preliminaries

Let us first give some general definitions. By $\log n$ we mean the binary logarithm of n, and $\log^{[k]} n$ denotes the k-th iterated logarithm of n, i.e. $\log^{[k]} n = \log\log \ldots \log n$, where there are k log's. Further, $\log^* n$ is the usual *iterated logarithm* of n, i.e., $\log^* n = \min\{k : \log^{[k]} n \leq 1\}$. For natural numbers $l \leq r$, the notation $[l : r]$ stands for the set $\{l, l + 1, \ldots, r\}$.

Now let us formally define the problem which is the issue of this paper. We are given a 2-dimensional array $A[0 : m - 1][0 : n - 1]$ of size $m \times n$. We wish to preprocess A such that queries asking for the position of the minimal element in an axis-parallel rectangle (denoted by $\text{RMQ}(y_1, x_1, y_2, x_2)$ for *range* *minimum query*) can be answered efficiently. More formally, $\text{RMQ}(y_1, x_1, y_2, x_2) = \arg\min_{(y,x)\in[y_1:y_2]\times[x_1:x_2]}\{A[y][x]\}$. Throughout this paper, let $N = mn$ denote the size of the input.

3 Methods

For simplicity, assume that the input array is a square, i.e., we have $m = n$ and $N = n^2$. The reader can verify that this assumption is not necessary for the validity of our algorithm. Further, because the query time will be constant throughout this section, we do not always explicitly mention this fact.

We first give a high-level overview of the algorithm (see also Fig. 1). The idea is to cover the input array with grids of decreasing widths $s_1, s_2, \dots$, thus dividing the array into blocks of decreasing size. For each grid of a certain width, we preprocess the array such that queries which *cross* the grid of a certain width s_k, but no grid of width $s_{k'}$ for $k' < k$, can be answered in constant time. Each such preprocessing will use $O(N)$ space and $O(N)$ time to construct. E.g., query q_1 in Fig. 1 will be answered on level 1 because it crosses the grid with width s_1, whereas q_2 will be answered on level 3. If the query rectangle does not cross any of the grids (e.g., q_3 in Fig. 1), we solve it by having precomputed *all* queries inside such small blocks which we call *microblocks*. If the size of these microblocks is constant, this constitutes no extra (asymptotic) time for preprocessing (leading to the $\log^*$-solution); otherwise we have to employ sorting of the blocks for a constant time preprocessing, leading to the $O(N \log^{[k+1]} N)$ preprocessing time. The details are as follows.

3.1 A General Trick for Query Precomputation

Assume we want to answer in $O(1)$ time all queries $\text{RMQ}(y_1, x_1, y_1 + l_y, x_1 + l_x)$ for y_1 taken from a certain subset of indices $Y \subseteq [0 : n-1]$ (and likewise x_1), and certain query lengths $l_y \in L_y = [1 : |L_y|]$ ($l_x \in L_x$). It suffices to precompute the answers for query rectangles whose side lengths are a power of 2; i.e., precompute $\text{RMQ}(y_1, x_1, y_1 + l_y, x_1 + l_x)$ for all $y_1 \in Y, x_1 \in X$, $l_y \in \{2^1, 2^2, 2^3, \dots, |L_y|\}$, and $l_x \in \{2^1, 2^2, 2^3, \dots, |L_x|\}$ and store the results in a table. These precomputations can be done in optimal time using dynamic programming, similar to the one-dimensional case [6]. The reason why precomputing these queries is enough is given by the simple fact that *all* queries can be answered by decomposing them into 4 different rectangles whose side lengths is a power of 2; see Fig. 2. Note the similarity to the *sparse table* algorithm for the 1-dimensional solution [5]. We denote by $g(|Y|, |X|, |L_y|, |L_x|) := |Y| \cdot |X| \cdot \log|L_y| \cdot \log|L_x|$ the space occupied by the resulting table for this kind of preprocessing.

Note how this idea already yields an RMQ-algorithm with $O(N \log^2 n)$ preprocessing time and space and $O(1)$ query time: simply perform the above preprocessing for $X, Y, L_x, L_y = [1 : n]$; the space needed is then $g(n, n, n, n) = N \log^2 n$.

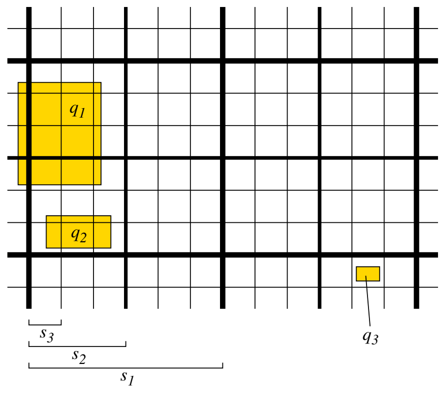

Fig. 1. Covering the input array with grids of different width. q_1, q_2, q_3 denote queries.

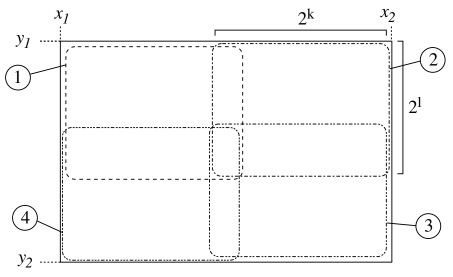

Fig. 2. Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into four equal-sized overlapping rectangles whose side lengths are a power of two. Taking the position of where the overall minimum occurs is the answer to the query.

3.2 $O(N)$ Preprocessing of the First Level

We now present a preprocessing to answer all queries which cross the grid for width $s := s_1 := \log n$. The array is partitioned into blocks of size $s \times s$. Then a query can be decomposed into at most 9 different sub-queries, as seen in Fig. 3. Query number 1 exactly spans over several blocks in both x- and y-direction. Queries 2–5 span over several blocks in one direction, but not in the other direction. Queries 6–9 lie completely inside one block (but meet at least one of the four block "boundaries").

Next, we show how to preprocess A such that all queries 1–9 can be answered in constant time. Taking the position where the overall minimum occurs is the final result.

Queries of type 1. We apply the idea from Sect. 3.1 on the set $Y = X = L_y = L_x = \{0, s, 2s, \dots, n/s\}$; i.e., we precompute $\text{RMQ}(ys, xs, (y + 2^k)s, (x + 2^l)s)$ for all $x, y \in \{0, \dots, n/s\}$ and all $k, l \in \{0, \dots, \log(n/s)\}$. The results are stored in a table of size $g(n/s, n/s, n/s, n/s) \leq n/s \cdot n/s \cdot \log n \cdot \log n = O(N)$. As usual, queries of this type are then answered by selecting the minimum of at most 4 overlapping precomputed queries.

Queries of type 2–5. We just show how to handle queries 2 and 4; the ideas for 3 and 5 are similar. Note that unlike Fig. 3 suggests, such queries are not guaranteed to share an upper or lower edge with the grid; the general appearance of these queries can be seen in Fig. 4. So the task is to answer all queries $\text{RMQ}(y_1, x_1 s, y_1 + l_y, x_1 s + l_x)$ for all $y_1 \in \{0, \dots, n-1\}$, $x_1 \in \{0, \dots, n/s\}$, $l_y \in \{1, \dots, s\}$ and $l_x \in \{s, 2s, \dots, n/s\}$. It is easy to verify that simply applying the trick from Sect. 3.1 would result in super-linear space; we therefore have to introduce another "preprocessing layer" by bundling $s' := s^2$ cells into one superblock. Then divide the type 2 (or 4)-query into a query that spans over

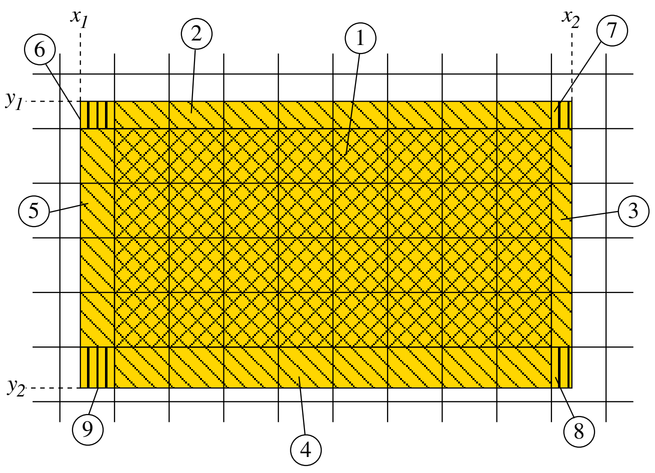

Fig. 3. Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into at most 9 sub-queries

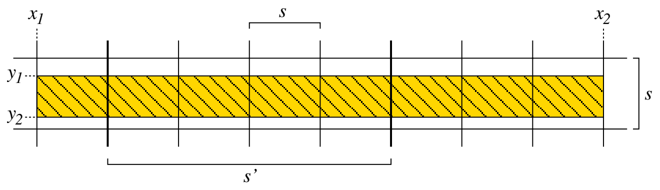

Fig. 4. Queries spanning over more than one block in x-direction, but not in the y-direction. Sub-queries 2 and 4 from Fig. 3 are special cases of these.

several superblocks, and at most two queries that span over several blocks, but *not* over a superblock. All three such queries are handled with the usual idea; this means that the space needed for the superblock-queries is $g(n, n/s', s, n/s') = n \cdot n/\log^2 n \cdot \log\log n \cdot \log(n/\log^2 n) \leq n^2 \log\log n/\log n = O(N)$. The space for the block-queries is $g(n, n/s, s, s'/s) = n \cdot n/\log n \cdot \log\log n \cdot \log\log n = O(N)$.

Queries of type 6–9. Again, unlike Fig. 3 suggests, it is *not* sufficient to precompute queries that have a common border with *two* edges of a block; we also have to precompute queries that share an edge with just *one* block-edge. (E.g., imagine the query in Fig. 4 were shifted slightly to the left. Then there would be a part of the query in the block to the very left which only touches the right border of the block.) We just show how to solve queries that share a border with the upper edge of a block (see Fig. 5); these structures have to be duplicated for the other three edges. This means that we want to answer $\text{RMQ}(y_1 s, x_1, y_1 s + l_y, x_1 + l_x)$ for all $y_1 \in \{0, \ldots, n/s\}$, $x_1 \in \{0, \ldots, n-1\}$, and $l_y, l_x \in \{1, \ldots, s\}$. In this case, the idea from Sect. 3.1 can be applied directly, leading to a space consumption of $g(n/s, n, s, s) = n/\log n \cdot n \cdot \log\log n \cdot \log\log n = O(N)$.

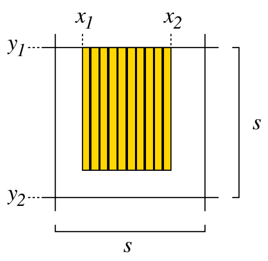

Fig. 5. Queries lying completely within a block, but sharing the upper edge with it. Sub-queries 8 and 9 from Fig. 3 are special cases of these.

3.3 Recursive Partitioning

We are left with the task to answer RMQs which lie completely inside one of the blocks with side length $s = \log n$. We now offer two recursion strategies which yield the $\log^{[k]}$- and $\log^*$-algorithms that have been promised before.

The first idea is to recurse at least one more time into the blocks, and thereafter precompute all queries which lie completely in one of the microblocks. To be precise, we take each of the $(n/s)^2$ resulting blocks from Sect. 3.2 and prepare them with the same method. Then the resulting blocks have side length $s_2 := \log^{[2]} n$. Continue this process until the resulting blocks have side length $s_k = \log^{[k]} n$ for some fixed $k > 1$. As each level needs $O(N)$ space, the resulting space is $O(Nk)$. We now show that already for $k = 2$ we can precompute all queries inside of microblocks in $O(N)$ space and $O(N \log^{[k+1]} N)$ time. We denote by $S := s_k^2$ the size of the microblocks (i.e., the number of elements one microblock contains).

The idea is to precompute all RMQs for all permutations of $[1 : S]$ and look up the result for a certain query block in the right place in this precomputed table. To do so, assign a *type* to each microblock $[y_1 : y_1 + s_k - 1] \times [x_1 : x_1 + s_k - 1]$ in A as follows: (conceptually) write the elements from the microblock row-wise into an array B; i.e., $B[1..S] = A[y_1][x_1..x_1 + s_k - 1] \ldots A[y_1 + s_k - 1][x_1..x_1 + s_k - 1]$. Then stably-sort B to obtain a permutation π of $\{1, \ldots, S\}$ s.th. $B[\pi_1] \leq B[\pi_2] \leq \cdots \leq B[\pi_S]$. The index of π in an enumeration of all permutations of $\{1, \ldots, S\}$ is the microblock-type. As there are N/S blocks of size $S = s_k^2$ to be sorted, this takes a total of $O(N/S \times S \log S) = O(N \log^{[k+1]} n)$ time.[1]

The reason for assigning the same type to microblocks whose elements are in the same order can be seen by the following (obvious) lemma:

Lemma 1. *Let A_1 and A_2 two arrays that have the same relative order π. Then $\text{RMQ}_{A_1}(y_1, x_1, x_2, y_2) = \text{RMQ}_{A_2}(y_1, x_1, x_2, y_2)$ for all values of y_1, x_1, y_2, x_2.* □

This implies that the following is enough to answer RMQs inside of microblocks: For all permutations π of $\{1, \ldots, S\}$, precompute all possible RMQs inside the block

[1] In the special case where the elements from the original array are in the range from 1 to N, we can bucket-sort all blocks simultaneously in $O(N)$ time.

$$\begin{pmatrix} \pi_1 & \cdots & \pi_{s_k} \\ \pi_{s_k+1} & \cdots & \pi_{2s_k} \\ \vdots & \ddots & \vdots \\ \pi_{(s_k-1)s_k+1} & \cdots & \pi_S \end{pmatrix}$$

and store them in a table P (for "precomputed") of size

$$\begin{aligned} S^2(S)! &= s_k^4\sqrt{2\pi s_k^2}\cdot\left(\frac{s_k^2}{e}\right)^{s_k^2}\cdot(1+O(s_k^{-2}))\text{ (by Stirling)} \\ &\le \log^5\log n\cdot\left(\log^2\log n\right)^{\log^2\log n}\cdot(1+O(s_k^{-2}))\text{ (because } k>1) \\ &= (\log\log n)^{2\log^2\log n+5}\cdot(1+O(s_k^{-2})) \\ &= O(N)\ . \end{aligned}$$

The last equation is true because $b^2 = O(2^b)$, so $(2b^2+5)/\log_b 2 \le 2^{b+1}$ for large enough b; exponentiating with 2 yields $b^{2b^2+5} \le 2^{\left(2^{b+1}\right)} = \left(2^{\left(2^b\right)}\right)^2$, which yields the result with $b = \log\log n$. Now to answer a query, simply look up the result in this table.

The second idea is to recurse further into the blocks until the resulting microblocks have constant size; this happens after $O(\log^* n)$ recursive steps. If the resulting micro-blocks have constant size they can be sorted in $O(1)$ time each; and because there are $(n/\log^* n)^2$ microblocks this takes a total of $O(N)$ time. The space consumed by this kind of preprocessing is clearly bounded by $O(N\log^* N)$ due to the number of recursive steps performed.

If we now apply the same recursive steps also for *answering* queries, this yields, of course, $O(\log^{[k+1]} N)$ and $O(\log^* N)$ query time, respectively. The next section shows how to reduce query time to $O(1)$.

3.4 What's Left: How to Find the Right Grid

For both the $\log^{[k]}$- and the $\log^*$-algorithm it remains to show how to determine in $O(1)$ time the grid with the largest width $s_i = \log^{[i]} n$ such that the query block crosses this grid. In other words, we wish to find the *smallest* $1 \le i \le k$ such that the query crosses the grid with width s_i, because at this level the answers have been precomputed and can hence be looked up. We will just show how to do this for the x-direction; the ideas for the y-direction are similar. For simplicity, assume that s_j is a multiple of s_{j+1} for all $1 \le j < k$.

Let $\text{RMQ}(y_1, x_1, y_2, x_2)$ be the given query, and let $l_x := x_2 - x_1 + 1$ denote the side length of the query rectangle in x-direction. Let i be defined such that $s_{i-1} > l_x \ge s_i$. This i can be found in $O(1)$ time by precomputing an array $I[1:n]$ with $I[l] = \min\{k : l \ge s_k\}$; then $i = I[l_x]$. Then the query crosses *at least* one column from the s_i-grid, and *at most* one column from the s_{i-1}-grid. As an example, consider query q_1 in Fig. 6. We have $s_2 > l_x \ge s_3$, and indeed, q_3 crosses a column from the s_3-grid, but no column from the s_2-grid. Likewise, for query q_2 we have $s_2 > l_x \ge s_1$, and it crosses an s_1- and an s_2-column. Let

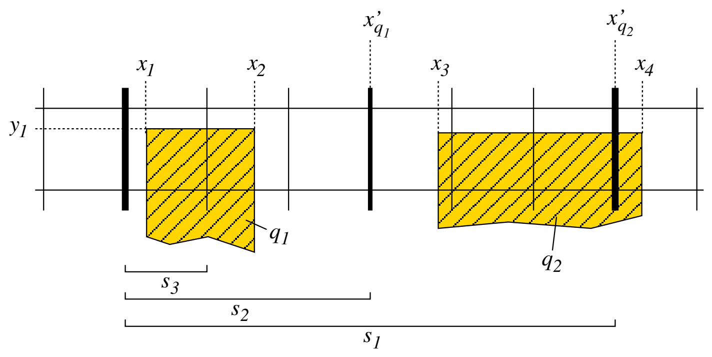

Fig. 6. How to determine the level on which a specific query has been precomputed

$x' := \lfloor \frac{x_2}{s_{i-1}} \rfloor \cdot s_{i-1}$ be the x-coordinate of where this crossing with a column from the s_{i-1}-grid can occur.

Assume first that $x' \notin [x_1 : x_2]$ (as for q_1 in Fig. 6). This means that the s_{i-1}-grid does *not* cross the query rectangle in x-direction; and the same is true for all $i' < i$. So we know for sure that i is the smallest value such that the s_i-grid passes through the query rectangle in x-direction. In this case we are done.

Now assume that $x' \in [x_1 : x_2]$ (as for q_2 in Fig. 6). In other words, the s_{i-1}-grid crosses the query rectangle in x-direction at position x'. In this case we are not yet done, because it could still be that an $s_{i'}$-column with $i' < i - 1$ also passes through x'. To find the smallest such i', define an array $I'[0 : n - 1]$ such that $I'[x'] = j$ iff j is the smallest value such that there is a column from the s_j-grid passing through x'. This array can certainly be precomputed during the k rounds of the preprocessing algorithm from the previous sections. As an example, in Fig. 6 it could still be that a column from a different grid (say from a hypothetical s_0-grid) passes through the query rectangle. But as this *must* happen at x'_{q_2}, we find this value at $I'[x'_{q_2}]$.

In total, we do the above for both the x- and y-direction, and look up the query result at the minimum level i from both steps. If, on the other hand, we find that the query rectangle does not cross a grid in any direction, the result can be looked up in table P of precomputed queries.

4 Conclusion

We have seen a class of algorithms which solve the two-dimensional RMQ-problem. While some ideas of our algorithm were similar to the one-dimensional counterpart of the problem, others were completely new, e.g., the idea of iterating the algorithm for k levels, while still handling all queries as if they were on the first level. Note that preprocessing time of our algorithm is not yet linear in the size of the input, as it is the case with the 1D-RMQ (though being *very* close to linear!). We conjecture that achieving linear time is impossible. In particular, we believe that it should be possible to show that there is no such nice relation as the one between the number of different RMQs and the number of different Cartesian Trees in the one-dimensional case [7]. This could mean that ideas such as sorting or recursing become un-avoidable, thereby hinting at a super-linear lower bound.

Although our results are currently more of theoretical nature, we believe that our solution will turn out to have interesting applications in the future. One conceivable application comes from computational biology, where one often wishes to identify minimal (or maximal) numbers in a given region of an alignment tableau [8].

Acknowledgments

J.F. wishes to thank Simon Ginzinger for fruitful discussions on this subject.

References

1. 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, pp. 135–143. ACM Press, New York (1984)
2. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
3. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (1988)
4. Berkman, O., Breslauer, D., Galil, Z., Schieber, B., Vishkin, U.: Highly parallelizable problems. In: Proc. of the ACM Symp. on Theory of Computing, pp. 309–319. ACM Press, New York (1989)
5. 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), 75–94 (2005)
6. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Proc. ESCAPE. LNCS (to appear)
7. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)