On Space Efficient Two Dimensional Range Minimum Data Structures

Gerth Stølting Brodal[1], Pooya Davoodi[1], and S. Srinivasa Rao[2]

[1] MADALGO*, Department of Computer Science, Aarhus University, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: {gerth,pdavoodi}@cs.au.dk
[2] School of Computer Science and Engineering, Seoul National University, S. Korea. E-mail: ssrao@cse.snu.ac.kr

Abstract. The two dimensional range minimum query problem is to preprocess a static two dimensional m by n array A of size $N = m \cdot n$, such that subsequent queries, asking for the position of the minimum element in a rectangular range within A, can be answered efficiently. We study the trade-off between the space and query time of the problem. We show that every algorithm enabled to access A during the query and using $O(N/c)$ bits additional space requires $\Omega(c)$ query time, for any c where $1 \leq c \leq N$. This lower bound holds for any dimension. In particular, for the one dimensional version of the problem, the lower bound is tight up to a constant factor. In two dimensions, we complement the lower bound with an indexing data structure of size $O(N/c)$ bits additional space which can be preprocessed in $O(N)$ time and achieves $O(c \log^2 c)$ query time. For $c = O(1)$, this is the first $O(1)$ query time algorithm using optimal $O(N)$ bits additional space. For the case where queries can not probe A, we give a data structure of size $O(N \cdot \min\{m, \log n\})$ bits with $O(1)$ query time, assuming $m \leq n$. This leaves a gap to the lower bound of $\Omega(N \log m)$ bits for this version of the problem.

1 Introduction

In this paper, we study time-space trade-offs for the two dimensional range minimum query problem (2D-RMQ). This problem has applications in computer graphics, image processing, computational Biology, and databases. The input is a two dimensional m by n array A of $N = m \cdot n$ elements from a totally ordered set. A query asks for the position of the minimum element in a query range $q = [i_1 \cdots i_2] \times [j_1 \cdots j_2]$, where $1 \leq i_1 \leq i_2 \leq m$ and $1 \leq j_1 \leq j_2 \leq n$, i.e., $\text{RMQ}(A, q) = \text{argmin}_{(i,j) \in q} A[i, j]$. We assume w.l.o.g. that $m \leq n$ and that all the entries of A are distinct (identical entries of A are ordered lexicographically by their index).

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

Table 1. Results for the 1D-RMQ problem. The term $|A|$ denotes the size of the input array A in bits.

Reference	Using Cartesian tree	Using LCA	Access to A	Space (bits)	Query Time
[13, 14, 17, 8, 6]	Yes	Yes	Yes	$O(n \log n) + \|A\|$	$O(1)$
[2]	No	No	Yes	$O(n \log n) + \|A\|$	$O(1)$
[16]	Yes	Yes	No	$4n + o(n)$	$O(1)$
[12]	Yes	No	Yes	$2n + o(n) + \|A\|$	$O(1)$
[11]	No	Yes	No	$2n + o(n)$	$O(1)$
Theorem 1	-	-	Yes	$O(n/c) + \|A\|$	$\Omega(c)$
Theorem 2	Yes	Yes	Yes	$O(n/c) + \|A\|$	$O(c)$

1.1 Previous Work

One Dimensional. The 1D-RMQ problem is the special case of the two dimensional version where $N = n$. It has been studied intensively and has numerous applications (Fischer [11] mentions some of them). Several solutions achieve $O(1)$ query time using additional space $O(n \log n)$ bits, by transforming RMQ queries into lowest common ancestor (LCA) queries [1] on the *Cartesian tree* [18] of A [13, 14, 17, 8, 6]. Alstrup et al. [2] solved the problem with the same bounds but without using Cartesian trees. Fischer and Heun [12] presented an $O(1)$ query time solution using $2n + o(n)$ additional bits which uses a Cartesian tree but makes no use of the LCA structure, and gives a simple solution for the static LCA problem[3]. Sadakane [16] gave an $O(1)$ query time algorithm using $4n + o(n)$ bits space which does not access A during the query. Fischer [11] decreased the space to $2n + o(n)$ bits by introducing a new data structure named 2d-Min-Heap instead of using the Cartesian tree. Table 1 summarizes these results along with the results of this paper.

Two Dimensional. A naïve solution for the 2D-RMQ problem is to perform a brute force search through all the entries of the query q in worst case $\Theta(N)$ time. Preprocessing A can reduce the query time. A naïve preprocessing is to store the answer to all the $O(N^2)$ possible queries in a lookup table of size $O(N^2 \log N)$ bits. The query time becomes $O(1)$ with no probe into A. All the published algorithms, on the $d > 1$ dimensional RMQ problem, perform probes into A during the query. The d-dimensional RMQ problem was first studied by Gabow et al. [13]. They apply the range trees introduced by Bentley [7] to achieve $O(\log^{d-1} N)$ query time using additional space $O(N \log^d N)$ bits and $O(N \log^{d-1} N)$ preprocessing time. Chazelle and Rosenberg [9] gave an algorithm to compute the range sum in the semigroup model, which can be applied to solve the RMQ problem. Their construction achieves $O(1)$ query time using additional space $O(N \cdot \alpha_k(n)^2 \cdot \log N)$ bits with $O(N \cdot \alpha_k(n)^2)$ preprocessing

[3] Fischer and Heun [12] claim $2n - o(n)$ bits lower bound for the additional space, however their proof is incorrect which, e.g., follows by Theorem 2.

Table 2. Results for the 2D-RMQ problem. The contributions of [13, 9, 4] and Theorem 1 can be generalized to the multidimensional version of the problem.

Reference	Query time	Space (bits)	Preprocessing time
[13]	$O(\log N)$	$O(N \log^2 N) + \|A\|$	$O(N \log N)$
[9]	$O(1)$	$O(N\alpha_k(n)^2 \log N) + \|A\|$	$O(N\alpha_k(n)^2)$
[3]	$O(1)$	$O(kN \log N) + \|A\|$	$O(N \log^{[k+1]} N)$
[10]	-	$\Omega(N \log m)$	-
[4]	$O(1)$	$O(N \log N) + \|A\|$	$O(N)$
Theorem 1	$\Omega(c)$	$N/c + \|A\|$	-
Theorem 3	$O(1)$	$O(N) + \|A\|$	$O(N)$
Theorem 4	$O(c \log^2 c)$	$O(N/c) + \|A\|$	$O(N)$
Theorem 5	-	$\Omega(N \log m)$	-
Section 3	$O(1)$	$O(N \cdot \min\{m, \log n\})$	$O(N)$

time for any fixed value of k, where $\alpha_k(n)$ is the k^{th} function of the inverse Ackermann hierarchy. The two dimensional version of the problem was considered by Amir et al. [3]. They presented a class of algorithms using $O(N \log^{(k+1)} N)$ preprocessing time, $O(kN \log N)$ bits additional space and $O(1)$ query time for any constant $k > 1$, where $\log^{(k+1)} N$ is the result of applying the log function $k+1$ times on N. Recently Atallah and Yuan [4] gave the first linear time preprocessing algorithm for d-dimensional arrays. Their algorithm answers any query in constant time using $O(N \log N)$ bits additional space. Demaine et al. [10] proved that the number of different 2D-RMQ n by n matrices is $\Omega((\frac{n}{4}!)^{n/4})$, where two 2D-RMQ matrices are considered different only if their range minima are in different locations for some rectangular range. This implies a lower bound $\Omega(n^2 \log n)$ for both the number of preprocessing comparisons and the number of bits required for a data structure capturing the answer to all the queries. This proves the impossibility of achieving a linear upper bound for the 2D-RMQ problem conjectured by Amir et al. [3]. Table 2 summarizes these results along with the results of this paper.

1.2 Our Results

We consider the 2D-RMQ problem in the following two models: 1) *indexing model* in which the query algorithm has access to the input array A in addition to the data structure constructed by preprocessing A, called an *index*; and 2) *encoding model* in which the query algorithm has no access to A and can only access the data structure constructed by preprocessing A, called an *encoding*.

In the indexing model, we initiate the study of the trade-off between the query time and the additional space for the 2D-RMQ problem. We prove the lower bound trade-off that $\Omega(c)$ query time is required if the additional space is N/c bits, for any c where $1 \leq c \leq N$. The proof is in a non-uniform cell probe model [15] which is more powerful than the indexing model. We complement the lower bound with an upper bound trade-off: using an index of size $O(N/c)$

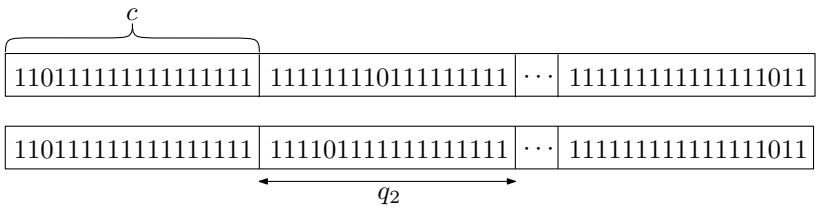

Fig. 1. Two arrays from $\mathcal{C}_{n,c}$, each one has n/c blocks. In this example $c = 18$. The query q_2 has different answers for these arrays.

bits we can achieve $O(c \log^2 c)$ query time. For the indexing model, this is the first $O(N)$-bit index which answers queries in $O(1)$ time.

In the encoding model, the only earlier result on the 2D-RMQ problem is the information-theoretic lower bound of Demaine et al. [10] who showed a lower bound of $\Omega(N \log n)$ bits for n by n matrices. We generalize their result to m by n (rectangular) matrices to show a lower bound of $\Omega(N \log m)$ bits. We also present an encoding structure of size $O(N \cdot \min\{m, \log n\})$ bits with $O(1)$ query time. Note that the upper and lower bounds are not tight for non-constant $m = n^{o(1)}$: the lower bound states that the space requirement per element is $\Omega(\log m)$ bits, whereas the upper bound requires $O(\min\{m, \log n\})$ bits per element.

2 Indexing Model

2.1 Lower Bound

In the indexing model, we prove a lower bound for the query time of the 1D-RMQ problem where the input is a one dimensional array of n elements, and then we show that the bound also holds for the RMQ problem in any dimension. The proof is in the non-uniform cell probe model [15]. In this model, computation is free, and time is counted as the number of cells accessed (probed) by the query algorithm. The algorithm is also allowed to be non-uniform, i.e., for different values of input parameter n, we can have different algorithms.

For n and any value of c, where $1 \leq c \leq n$, we define a set of arrays $\mathcal{C}_{n,c}$ and a set of queries $\mathcal{Q}$. We w.l.o.g. assume that c divides n. We will argue that for any 1D-RMQ algorithm which has access to an index of size n/c bits (in addition to the input array A), there exists an array in $\mathcal{C}_{n,c}$ and a query in $\mathcal{Q}$ for which the algorithm is required to perform $\Omega(c)$ probes into A.

Definition 1. *Let n and c be two integers, where $1 \leq c \leq n$. The set $\mathcal{C}_{n,c}$ contains the arrays $A[1 \cdots n]$ such that the elements of A are from the set $\{0, 1\}$, and in each block $A[(i-1)c + 1 \cdots ic]$ for all $1 \leq i \leq n/c$, there is exactly a single zero element (see Figure 1).*

The number of possible data structures of size n/c bits is $2^{n/c}$, and the number of arrays in $\mathcal{C}_{n,c}$ is $c^{n/c}$. By the pigeonhole principle, for any algorithm $\mathcal{G}$

there exists a data structure $D_{\mathcal{G}}$ which is shared by at least $(\frac{c}{2})^{n/c}$ input arrays in $\mathcal{C}_{n,c}$. Let $\mathcal{C}_{n,c}^{D_{\mathcal{G}}} \subseteq \mathcal{C}_{n,c}$ be the set of these inputs.

Definition 2. *Let $q_i = [(i-1)c + 1 \cdots ic]$. The set $\mathcal{Q} = \{q_i \mid 1 \leq i \leq n/c\}$ contains n/c queries, each covering a distinct block of A.*

For algorithm $\mathcal{G}$ and data structure $D_{\mathcal{G}}$, we define a binary decision tree capturing the behavior of $\mathcal{G}$ on the inputs from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ to answer a query $q \in \mathcal{Q}$.

Definition 3. *Let $\mathcal{G}$ be a deterministic algorithm. For each query $q \in \mathcal{Q}$, we define a binary decision tree $T_q(D_{\mathcal{G}})$. Each internal node of $T_q(D_{\mathcal{G}})$ represents a probe into a cell of the input arrays from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$. The left and right edges correspond to the output of the probe: left for zero and right for one. Each leaf is labeled with the answer to q.*

For each algorithm $\mathcal{G}$, we have defined n/c binary trees depicting the probes of the algorithm into the inputs from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ to answer the n/c queries in $\mathcal{Q}$. Note that the answers to all these n/c queries uniquely determine the input. We compose all the n/c binary trees into a single binary tree $T_{\mathcal{Q}}(D_{\mathcal{G}})$ in which every leaf determines a particular input. We first replace each leaf of $T_{q_1}(D_{\mathcal{G}})$ with the whole $T_{q_2}(D_{\mathcal{G}})$, and then replace each leaf of the obtained tree with $T_{q_3}(D_{\mathcal{G}})$, and so on. Every leaf of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ is labeled with the answers to all the n/c queries in $\mathcal{Q}$ which were replaced on the path from the root to the leaf. Every two input arrays in $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ correspond to different leaves of $T_{\mathcal{Q}}(D_{\mathcal{G}})$. Otherwise the answers to all the queries in $\mathcal{Q}$ are the same for both the inputs which is a contradiction. Therefore, the number of leaves of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ is at least $(\frac{c}{2})^{n/c}$, the minimum number of inputs in $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$.

We next prune $T_{\mathcal{Q}}(D_{\mathcal{G}})$ as follows: First we remove all nodes not reachable by any input from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$. Then we repeatedly replace all nodes of degree one with their single child. Since the inputs from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ correspond to only reachable leaves, in the pruned tree, the number of leaves becomes equal to the number of inputs from $\mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ which is at least $(\frac{c}{2})^{n/c}$. In the unpruned tree, the result of a repeated probe is known already and one child of the node corresponding to the probe is unreachable. Therefore, on a root to leaf path in the pruned tree, there is no repeated probe. Every path from the root to a leaf has at most n/c left edges (zero probes), since the number of zero elements in each input from $\mathcal{C}_{n,c}$ is n/c. The branches along each of these paths represents a binary sequence of length at most d containing at most n/c zeros where d is the depth of the pruned tree. By padding each of these sequences with further 0s and 1s, we can ensure that each sequence has length exactly $d + n/c$ and contains exactly n/c zeros. The number of these binary sequences is at most $\binom{d+n/c}{n/c}$, which becomes an upper bound for the number of leaves in the pruned tree.

Lemma 1. *For all n and c, where $1 \leq c \leq n$, the worst case number of probes required to answer a query in $\mathcal{Q}$ over the inputs from $\mathcal{C}_{n,c}$ using a data structure of size n/c bits is $\Omega(c)$.*

Proof. Comparing the lower and upper bounds from the above discussion for the number of leaves of $T_{\mathcal{Q}}(D_{\mathcal{G}})$, we have

$$\left(\frac{c}{2}\right)^{n/c} \leq \binom{d+\frac{n}{c}}{\frac{n}{c}}.$$

By Stirling's formula, the following is obtained

$$\log\left(\frac{c}{2}\right)^{n/c} \leq \frac{n}{c}\log\frac{c}{2} \leq \log\binom{d+\frac{n}{c}}{\frac{n}{c}} \leq \frac{n}{c}\log\left[\frac{(d+\frac{n}{c})e}{\frac{n}{c}}\right],$$

which implies $c/2 \leq \frac{(d+n/c)e}{n/c}$, and therefore $d \geq n(\frac{1}{2e} - \frac{1}{c})$. For any arbitrary algorithm $\mathcal{G}$, the depth d of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ equals the sum of the depths of the n/c binary trees composed into $T_{\mathcal{Q}}(D_{\mathcal{G}})$. By the pigeonhole principle, there exists an input $x \in \mathcal{C}_{n,c}^{D_{\mathcal{G}}}$ and an i, where $1 \leq i \leq n/c$, such that the query q_i on x requires at least $d/(n/c) = \Omega(c)$ probes into the array A maintaining the input. □

Theorem 1. *Any algorithm solving the RMQ problem for an input array of size N (in any dimension), which uses N/c bits additional space, requires $\Omega(c)$ query time, for any c, where $1 \leq c \leq N$.*

Proof. Lemma 1 gives the lower bound for the 1D-RMQ problem. The proof for the 2D-RMQ is a simple extension of the proof of Lemma 1. Instead of $\mathcal{C}_{n,c}$, a set $\mathcal{C}_{m,n,c_1,c_2}$ of matrices is utilized. Each matrix is composed of mn/c submatrices $[ic_1+1\cdots(i+1)c_1] \times [jc_2+1\cdots(j+1)c_2]$ of size c_1 by c_2, for $1 \leq i < m/c_1$ and $1 \leq j < n/c_2$, where $c = c_1 \cdot c_2$ (assuming w.l.o.g. that c_1 divides m, and c_2 divides n). Each submatrix has exactly one zero element, and all the others are one. There are N/c queries in $\mathcal{Q}$, each one asks for the minimum of each submatrix. As in the proof of Lemma 1, we can argue that there exists a query requiring $\Omega(c)$ probes by utilizing the methods of decision trees, composing and pruning them, and bounding the number of leaves. The proof can be generalized straightforwardly to higher dimensional version of the RMQ problem. □

Theorem 2. *The 1D-RMQ problem for an input array of size n is solved in $O(n)$ preprocessing time and optimal $O(c)$ query time using $O(n/c)$ additional bits.*

Proof. Partition the input array into n/c blocks of size c. Construct an 1D-RMQ encoding structure for the list of n/c block minimums (minimum elements of the blocks) in $O(n/c)$ bits [16]. The query is decomposed into three subqueries. All the blocks spanned by the query form the middle subquery, which can be answered by querying the $O(n/c)$-bit data structure in $O(1)$ time and then scanning the block containing the answer in $O(c)$ time. The remaining part of the query which includes two subqueries contained in two blocks is answered in $O(c)$ time by scanning the blocks. □

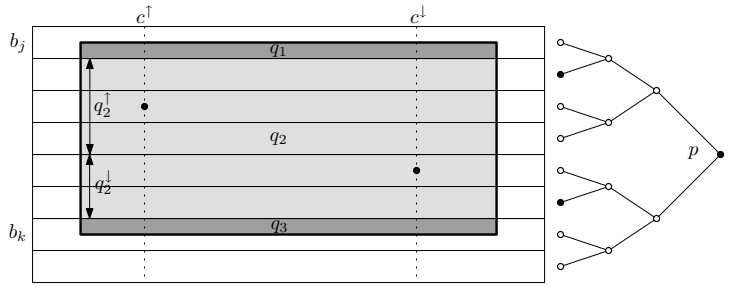

Fig. 2. Partitioning the input and building the binary tree structure. The node p is the LCA of the leaves corresponding to b_{j+1} and b_{k-1}. The columns $c^{\uparrow}$ and $c^{\downarrow}$, which contain the answers to $q_2^{\uparrow}$ and $q_2^{\downarrow}$ respectively, are found using the Cartesian trees stored in p. The minimum element in each of the columns $c^{\uparrow}$ and $c^{\downarrow}$ is found using the Cartesian tree constructed for that column.

2.2 Linear Space Optimal Data Structure

Preliminaries A *block* is a rectangular range in a matrix. Let B be a block of size m' by n'. For the block B, the list MinColList$[1 \cdots n']$ contains the minimum element of each column and MinRowList$[1 \cdots m']$ contains the minimum element of each row. Let TopPrefix(B, ℓ) be the set of blocks $B[m'/2 - i\ell + 1 \cdots m'/2] \times [1 \cdots n']$ and BottomPrefix(B, ℓ) be the set of blocks $B[m' - i\ell + 1 \cdots m'] \times [1 \cdots n']$, for $1 \leq i \leq m'/(2\ell)$ (assuming w.l.o.g. that m' is even and ℓ divides $m'/2$). If the rows of B (instead of its columns as the above) are divided by the blocks, then top and bottom denote left and right.

Data Structure and Querying We present an indexing data structure of size $O(N)$ bits achieving $O(1)$ query time to solve the 2D-RMQ problem. The input matrix of size m by n is partitioned into blocks $\mathcal{B} = \{b_1, \ldots, b_{m/\log m}\}$ of size $\log m$ by n. According to these blocks, the query q is divided into subqueries q_1, q_2 and q_3 such that w.l.o.g. q_1 is contained in b_j and q_3 is contained in b_k, and q_2 spans over $b_{j+1}, \ldots, b_{k-1}$ vertically, where $1 \leq j, k \leq m/\log m$ (see Figure 2). A binary tree structure is utilized to answer q_2. Since q_1 and q_3 are range minimum queries in the submatrices b_j and b_k respectively, they are answered recursively. Lastly, the answers to q_1, q_2 and q_3, which are indices into three matrix elements, are used to find the index of the smallest element in q.

The binary tree structure has $m/\log m$ leaves, one for each block in $\mathcal{B}$, assuming $m/\log m$ is a power of 2. Each leaf maintains a Cartesian tree for MinColList of its corresponding block. Each internal node having $2k$ leaf descendants matches with a submatrix M composed of $2k$ consecutive blocks of $\mathcal{B}$ corresponding to the leaf descendants, for $1 \leq k \leq m/(2\log m)$. Note that each of the sets TopPrefix$(M, \log m)$ and BottomPrefix$(M, \log m)$ contains k blocks, and each block corresponds with a MinColList. The internal node maintains $2k$ Cartesian trees constructed for these $2k$ MinColLists.

Let M be the submatrix matched with the lowest common ancestor p of the two leaves corresponding to b_{j+1} and b_{k-1}. The subquery q_2 is composed of the top part $q_2^{\uparrow}$ and the bottom part $q_2^{\downarrow}$, where $q_2^{\uparrow}$ and $q_2^{\downarrow}$ are two blocks in TopPrefix$(M, \log m)$ and BottomPrefix$(M, \log m)$ respectively. Two of the Cartesian trees, maintained in p, are constructed for MinColLists of $q_2^{\uparrow}$ and $q_2^{\downarrow}$. These two Cartesian trees are utilized to find two columns containing the answer to $q_2^{\uparrow}$ and $q_2^{\downarrow}$. The Cartesian trees constructed for these two columns are utilized to find the answer to $q_2^{\uparrow}$ and $q_2^{\downarrow}$. Then the answer to q_2 is determined by comparing the smallest element in $q_2^{\uparrow}$ and $q_2^{\downarrow}$.

In the second level of the recursion, each block of $\mathcal{B}$ is partitioned into blocks of size $\log m$ by $\log n$. The recursion continues until the size of each block is $\log \log m$ by $\log \log n$ (i.e. four levels). In the binary tree structures built for all the four recursion levels, we construct the Cartesian trees for the appropriate MinColLists and MinRowLists respectively. In the second and fourth levels of recursion, where the binary tree structure gives two rows containing the minimum elements of $q_2^{\uparrow}$ and $q_2^{\downarrow}$, the Cartesian trees constructed for the rows of the matrix are used to answer $q_2^{\uparrow}$ and $q_2^{\downarrow}$.

We solve the 2D-RMQ problem for a block of size $\log \log m$ by $\log \log n$ using the table lookup method given by Atallah and Yuan [4]. Their method preprocesses the block by making at most $c'G$ comparisons, for a constant c', where $G = \log \log m \cdot \log \log n$, such that any 2D-RMQ can be answered by performing four probes into the block. Each block is represented by a *block type* which is a binary sequence of length $c'G$, using the results of the comparisons. The lookup table has $2^{c'G}$ rows, one for each possible block type, and G^2 columns, one for each possible query within a block. Each cell of the table contains four indices to address the four probes into the block. The block types of all the blocks of size G in the matrix are stored in another table T. The query within a block is answered by first recognizing the block type using T, and then checking the lookup table to obtain the four indices. Comparing the results of these four probes gives the answer to the query [4].

Theorem 3. *The 2D-RMQ problem for an m by n matrix of size $N = m \cdot n$ is solved in $O(N)$ preprocessing time and $O(1)$ query time using $O(N)$ bits additional space.*

Proof. The subquery q_2 is answered in $O(1)$ time by using a constant query time LCA structure [5], querying the Cartesian trees in constant time [16], and performing $O(1)$ probes into the matrix. The number of recursion levels is four. In the last level, the subqueries contained in blocks of size G are also answered in $O(1)$ time by using the lookup table and performing $O(1)$ probes into the matrix. Therefore the query is answered in $O(1)$ time.

The depth of the binary tree, in the first recursion level, is $O(\log(m/\log m))$. Each level of the tree has $O(m/\log m)$ Cartesian trees for MinColLists of size n elements. Since a Cartesian tree of a list of n elements is stored in $O(n)$ bits [16], the binary tree can be stored in $O(n \cdot m/\log m \cdot \log(m/\log m)) = O(N)$ bits. Since the number of recursion levels is $O(1)$, the binary trees in all the recursion

levels are stored in $O(N)$ bits. The space used by the $m+n$ Cartesian trees constructed for the columns and rows is $O(N)$ bits. Since $G \leq c'' \log N$ for a constant c'', the size of the lookup table is $O(2^{c'c'' \log N} G^2 \log G) = o(N)$ bits when $c'' < 1/c'$. The size of table T is $O(N/G \cdot \log(2^{c'G})) = O(N)$ bits. Hence the total additional space is $O(N)$ bits.

In the binary tree, in the first level of the recursion, each leaf maintains a Cartesian tree constructed for a MinColList of size n elements. These $m/\log m$ lists are constructed in $O(N)$ time by scanning the whole matrix. Each MinColList in the internal nodes is constructed by comparing the elements of two MinColLists built in the lower level in $O(n)$ time. Therefore constructing these lists, for the whole tree, takes $O(n \cdot m/\log m \cdot \log(m/\log m)) = O(N)$ time. Since a Cartesian tree can be constructed in linear time [16], the Cartesian trees in all the nodes of the binary tree are constructed in $O(N)$ time. The LCA structure is also constructed in linear time [5]. Therefore the binary tree is built in $O(N)$ time. Since the number of recursion levels is $O(1)$, all the binary trees are built in $O(N)$ time. The lookup table and table T are also constructed in $O(N)$ time [4]. □

2.3 Space Time Trade-off Data Structure

We present an indexing data structure of size $O(N/c \cdot \log c)$ bits additional space solving the 2D-RMQ problem in $O(c \log c)$ query time and $O(N)$ preprocessing time, where $1 \leq c \leq N$. The input matrix is divided into N/c blocks of size 2^i by $c/2^i$, for $0 \leq i \leq \log c$; assuming w.l.o.g. that c is a power of 2. Let M_i be the matrix of size N/c containing the minimum elements of the blocks of size 2^i by $c/2^i$. Let D_i be the linear space data structure of Section 2.2 applied to the matrix M_i in $O(N/c)$ bits. Each D_i handles a different ratio between the number of rows and the number of columns of the blocks. Note that the matrices M_i are constructed temporarily during the preprocessing and not maintained in the data structure.

A query q is resolved by answering $\log c+1$ subqueries. Let q_i be the subquery of q spanning the blocks of size 2^i by $c/2^i$ for $0 \leq i \leq \log c$. The minimum elements of the blocks spanned by q_i assemble a query over M_i which has the same answer as q_i. Therefore, q_i is answered by using D_i. Note that whenever the algorithm wants to perform a probe into a cell of M_i, a corresponding block of size c of the input is searched for the minimum (since M_i is not maintained in the data structure). The subqueries q_i overlap each other. Altogether, they compose q except for at most $c \log c$ elements in each of the four corners of q. We search these corners for the minimum element. Eventually, we compare the minimum elements of all the subqueries to find the answer to q (see Figure 3).

Theorem 4. *The 2D-RMQ problem for a matrix of size N is solved in $O(N)$ preprocessing time and $O(c \log^2 c)$ query time using $O(N/c)$ bits additional space.*

Proof. The number of linear space data structures D_i is $\log c + 1$. Each D_i requires $O(N/c)$ bits. Therefore, the total additional space is $O(\log c \cdot N/c)$ bits.

The number of subqueries q_i is $\log c + 1$. Each q_i is answered by using D_i in $O(1)$ query time in addition to the $O(1)$ probes into M_i. Since instead of each

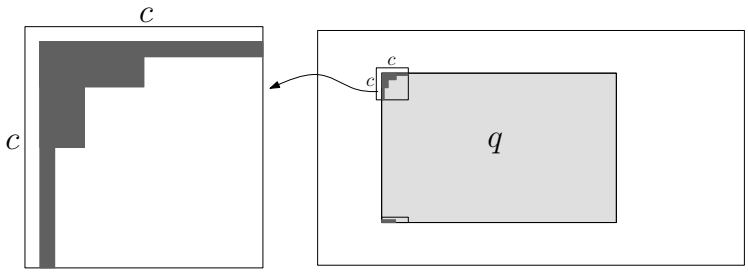

Fig. 3. Right: The grey area depicts the subqueries of q spanning the blocks of size 2^i by $c/2^i$. Left: The dark area depicts a corner of q which is contained in a block of size c by c and includes at most $c \log c$ elements.

probe into M_i, we perform $O(c)$ probes into the input, the query time to answer q_i is $O(c)$. The four corners are searched in $O(c \log c)$ time for the minimum elemenm. In the end, the minimum elements of the subqueries are compared in $O(\log c)$ time to answer q. Consequently, the total query time is $O(c \log c)$.

Each D_i is constructed in $O(N/c)$ time (Section 2.2) after building the matrix M_i. To be able to make all M_i efficiently, we first construct an $O(N)$-bit space data structure of Section 2.2 for the input matrix in $O(N)$ time. Then, M_i is built in $O(N/c)$ time by querying a block of the input matrix in $O(1)$ time for each element of M_i. Therefore, the total preprocessing time is $O(\log c \cdot N/c + N) = O(N)$. Substituting the parameter c by $c \log c$ gives the claimed bounds. □

3 Encoding Model

Upper Bound The algorithm described in Section 2.2 can preprocess the m by n input array A of size $N = m \cdot n$ into a data structure of size $O(N)$ bits in $O(N)$ time. But the query algorithm in Section 2.2 is required to perform some probes into the input. Since A is not accessible in the encoding model, we store another 2D array maintaining the rank of all the N elements using $O(N \log n)$ bits. Whenever the algorithm wants to perform a probe into A, it does it into the rank matrix. Therefore the problem can be solved in the encoding model using $O(N \log n)$ preprocessing time (to sort A) and $O(1)$ query time using $O(N \log n)$ bits space.

Another solution in the encoding model is the following. For each of the n columns of A, we build a 1D-RMQ structure using $O(m)$ bits space [16], in total using $O(mn) = O(N)$ bits space. Furthermore, for each possible pair of rows (i_1, i_2), $i_1 \leq i_2$, we construct an 1D-RMQ structure for MinColList of $A[i_1 \cdots i_2] \times [1 \cdots n]$ using $O(n)$ bits space; in total using $O(m^2 n) = O(Nm)$ bits. The column j containing the answer to a query $q = [i_1 \cdots i_2] \times [j_1 \cdots j_2]$ is found by querying for the range $[j_1 \cdots j_2]$ in the 1D-RMQ structure for the rows given by the pair (i_1, i_2). The query q is answered by querying for the range $[i_1 \cdots i_2]$ in the 1D-RMQ structure for column j. Since both 1D-RMQ queries take $O(1)$ time, the total query time is $O(1)$.

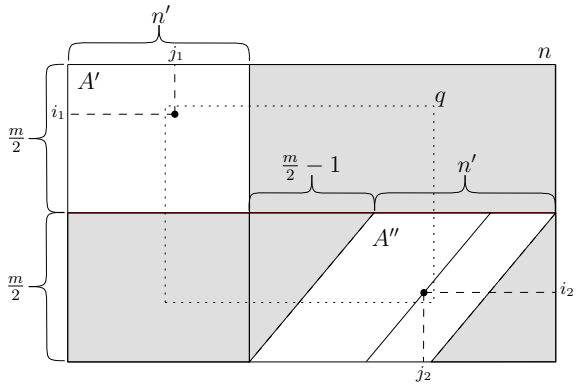

Fig. 4. The elements in the gray area are greater than the elements in the white area. The dotted rectangle denotes the query q which has different answers for A_1 and A_2.

Selecting the most space efficient solution of the above two solutions gives an encoding structure of size $O(N \cdot \min\{m, \log n\})$ bits with $O(1)$ query time.

Lower Bound We present a set of $\Omega((m!)^n)$ different 2D arrays, i.e., for every pair of the arrays there exists a 2D-RMQ with different answers. The elements of the arrays are from the set $\{1, \ldots, mn\}$. Every array of the set has two parts $A' = A[1 \cdots m/2] \times [1 \cdots n']$ and A'' containing all the anti-diagonals of length $m/2$ within the block $A[m/2+1 \cdots m] \times [n'+1 \cdots n]$ where $n' = \lfloor (n - m/2 + 1)/2 \rfloor$, assuming w.l.o.g. that m is even (see Figure 4). These two parts contain the smallest elements of the array, i.e., $\{1, \ldots, mn'\}$. From this set, the odd numbers are placed in A' in increasing order from left to right and then top to bottom, i.e. $A'[i, j] = 2((i-1)n' + j) - 1$. The even numbers are placed in A'' such that the elements of each anti-diagonal are not sorted but are larger than the elements of the anti-diagonals to the right. The total number of arrays constructed by permuting the elements of each anti-diagonal of A'' is $(\frac{m}{2}!)^{n'}$.

For any two matrices A_1 and A_2 in the set, there exists an index $[i_2, j_2]$ in the anti-diagonals of A'' such that $A_1[i_2, j_2] \neq A_2[i_2, j_2]$. Let $[i_1, j_1]$ be the index of an arbitrary odd number between $A_1[i_2, j_2]$ and $A_2[i_2, j_2]$. Since the query $q = [i_1 \cdots i_2] \times [j_1 \cdots j_2]$ has different answers for A_1 and A_2, it follows that any two matrices in the set are different (see Figure 4).

Theorem 5. *The minimum space to store an encoding data structure for the 2D-RMQ problem is $\Omega(mn \log m)$ bits, assuming that $m \leq n$.*

Proof. Since the number of different arrays in the set is $(\frac{m}{2}!)^{n'}$, the space for a data structure encoding these arrays is $\Omega(\log(\frac{m}{2}!)^{n'}) = \Omega(mn \log m)$ bits. □

References

1. A. Aho, J. Hopcroft, and J. Ullman. On finding lowest common ancestors in trees. In *Proc. 5th Annual ACM Symposium on Theory of Computing*, pages 253–265. ACM, 1973.
2. S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proc. 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 258–264. ACM, 2002.
3. A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching*, volume 4580 of *LNCS*, pages 286–294. Springer-Verlag, 2007.
4. M. J. Atallah and H. Yuan. Data structures for range minimum queries in multidimensional arrays. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 150–160. SIAM, 2010.
5. M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Theoretical Informatics Symposium*, volume 1776 of *LNCS*, pages 88–94. Springer-Verlag, 2000.
6. M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
7. J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
8. O. Berkman, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proc. 21st Ann. ACM Symposium on Theory of Computing*, pages 309–319. ACM, 1989.
9. B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proc. 5th Annual Symposium on Computational Geometry*, pages 131–139. ACM, 1989.
10. E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. In *Proc. 36th International Colloquium on Automata, Languages and Programming*, volume 5555 of *LNCS*, pages 341–353. Springer-Verlag, 2009.
11. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *LNCS*, pages 158–169. Springer-Verlag, 2010.
12. J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *LNCS*, pages 459–470. Springer-Verlag, 2007.
13. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM, 1984.
14. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
15. P. B. Miltersen. Cell probe complexity - a survey. In *Advances in Data Structures Workshop (FSTTCS)*, 1999.
16. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
17. B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
18. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.