The Encoding Complexity of Two Dimensional Range Minimum Data Structures

Gerth Stølting Brodal[1], Andrej Brodnik[2,3⋆], and Pooya Davoodi[4⋆⋆]

[1] MADALGO⋆⋆⋆, Department of Computer Science, Aarhus University, Denmark. gerth@cs.au.dk
[2] University of Primorska, Department of Information Science and Technology, Slovenia. andrej.brodnik@upr.si
[3] University of Ljubljana, Faculty of Computer and Information Science, Slovenia.
[4] Polytechnic Institute of New York University, USA. pooyadavoodi@gmail.com

Abstract. In the two-dimensional range minimum query problem an input matrix A of dimension $m \times n$, $m \leq n$, has to be preprocessed into a data structure such that given a query rectangle within the matrix, the position of a minimum element within the query range can be reported. We consider the space complexity of the encoding variant of the problem where queries have access to the constructed data structure but can not access the input matrix A, i.e. all information must be encoded in the data structure. Previously it was known how to solve the problem with space $O(mn \min\{m, \log n\})$ bits (and with constant query time), but the best lower bound was $\Omega(mn \log m)$ bits, i.e. leaving a gap between the upper and lower bounds for non-quadratic matrices. We show that this space lower bound is optimal by presenting an encoding scheme using $O(mn \log m)$ bits. We do not consider query time.

1 Introduction

We study the problem of preprocessing a two dimensional array (matrix) of elements from a totally ordered set into a data structure that supports range minimum queries (RMQs) asking for the *position* of the minimum element within a range in the array. More formally, we design a data structure for a matrix $A[1..m] \times [1..n]$ with $N = mn$ elements, and each RMQ asks for the position of the minimum element within a range $[i_1..i_2] \times [j_1..j_2]$. We refer to this problem as the 2D-RMQ problem, in contrast with the 1D-RMQ problem, where the input array is one-dimensional.

Study of the 1D-RMQ and 2D-RMQ problems dates back to three decades ago, when data structures of linear size were proposed for both of the problems [8, 3]. Both problems have known applications including information retrieval,

⋆ Research supported by grant BI-DK/11-12-011: Algorithms on Massive Geographical LiDAR Data-sets - AMAGELDA.
⋆⋆ Research supported by NSF grant CCF-1018370 and BSF grant 2010437.
⋆⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

computational biology, and databases [4]. In this paper, we study the space-efficiency of the 2D-RMQ data structures.

1.1 Previous work

There exists a long list of articles on the 1D-RMQ problem which resolve various aspects of the problem including the space-efficiency of the data structures. However, the literature of the 2D-RMQ problem is not so rich in space-efficient data structures. In the following, we review the previous results on both of the problems that deal with the space-efficiency of data structures.

1D-RMQs. Let A be an input array with n elements which we preprocess into a data structure that supports 1D-RMQs on A. A standard approach to solve this problem is to make the *Cartesian tree* of A, which is a binary tree with n nodes defined as follows: the root of the Cartesian tree is labeled by i, where $A[i]$ is the minimum element in A; the left and right subtrees of the root are recursively the Cartesian trees of $A[1..i-1]$ and $A[i+1..n]$ respectively.

The property of the Cartesian tree is that the answer to a query with range $[i..j]$ is the label of the lowest common ancestor (LCA) of the nodes with labels i and j. Indeed, the Cartesian tree stores a partial ordering between the elements that is appropriate to answer 1D-RMQs. This property has been utilized to design data structures that support 1D-RMQs in constant time [8, 9].

The space usage of the 1D-RMQ data structures that rely on Cartesian trees is the amount of space required to represent a Cartesian tree plus the size of an LCA data structure. There exists a one-to-one correspondence between the set of Cartesian trees and the set of *different* arrays, where two arrays are different iff there exists a 1D-RMQ with different answers on these two arrays. The total number of binary trees with n nodes is $\binom{2n}{n}/(n+1)$, and thus the logarithm of this number yields an information-theoretic lower bound on the number of bits required to store a 1D-RMQ data structure, which is $2n - \Theta(\log n)$ bits.

Storing the Cartesian tree using the standard pointer-based representation and a linear-space LCA data structure takes $O(n \log n)$ bits. There have been a number of attempts to design 1D-RMQ data structures of size close to the lower bound. Sadakane [10] and then Fischer and Heun [7] improved the space to $4n + o(n)$ and $2n + o(n)$ bits respectively by representing the Cartesian tree using succinct encoding of the topology of the tree and utilizing the encoding to support LCA queries (in fact, Fischer and Heun [7] proposed a representation of another tree which is a transformed Cartesian tree [5]). Both of these data structures support 1D-RMQs in $O(1)$ time.

2D-RMQs. A standard two-level range tree along with a 1D-RMQ data structure can be used to design a 2D-RMQ data structure. This method was used to give a data structure of size $O(N \log N)$ that supports 2D-RMQs in $O(\log N)$ time [8]. The state of the art 1D-RMQ data structures can improve the space to $O(N)$.

The literature contains a number of results that have advanced the performance of 2D-RMQ data structures in terms of preprocessing and query times

[3, 1], which ended in a brilliant 2D-RMQ data structure of size $O(N)$ (that is, $O(N \log N)$ bits) which can be constructed in $O(N)$ time and supports queries in $O(1)$ time [11].

Although the complexity of the preprocessing and query times of 2D-RMQ data structures have been settled, the space complexity of 2D-RMQ data structures has been elusive.

In contrast with 1D-RMQs where the partial ordering of elements can be encoded in linear number of bits using Cartesian trees, it has been shown that encoding partial ordering of elements for 2D-RMQs would require super-linear number of bits (there is no "Cartesian tree-like" data structure for 2D-RMQs) [6]. The number of different $n \times n$ matrices is $\Omega((\frac{n}{4}!)^n/4)$, where two matrices are different if there exists a 2D-RMQ with different answers on the two matrices [6]. This counting argument implies the lower bound $\Omega(N \log N)$ on the number of bits required to store a 2D-RMQ data structure on an $n \times n$ matrix with $N = n^2$ elements.

The above counting argument was later extended to rectangular $m \times n$ matrices where $m \leq n$, yielding the information-theoretic lower bound $\Omega(N \log m)$ bits on the space of 2D-RMQ data structures [2]. This lower bound raised the question of encoding partial ordering of elements for 2D-RMQs using space close to the lower bound. In other words, the new question shifted the focus from designing 2D-RMQ data structures with efficient queries to designing encoding data structures, which we simply call encodings, that support 2D-RMQs disregarding the efficiency of queries. In fact, the fundamental problem here is to discover whether the partial ordering of elements for 2D-RMQs can be encoded in $O(N \log m)$ bits.

There exist simple answers to the above question which do not really satisfy an enthusiastic mind. On the one hand, we can ensure that each element in A takes $O(\log N)$ bits in the encoding data structure by sorting the elements and replacing each element with its rank (recall that a query looks for the position of minimum rather than its value). This provides a simple encoding of size $O(N \log N)$ bits, while this upper bound was already achieved by all the linear space 2D-RMQ data structures [3, 1, 11]. On the other hand, we can make a data structure of size $O(Nm)$ bits which improves the size of the first encoding for $m = o(\log n)$. This latter encoding is achieved by taking the multi-row between every two rows i and j; making an array $R[1..n]$ out of each multi-row by assigning the minimum of the k-th column of the multi-row to $R[k]$; encoding R using a 1D-RMQ data structure of size $O(n)$ bits; and making a 1D-RMQ data structure of size $O(m)$ bits for each column of A. A 2D-RMQ can be answered by finding the appropriate multi-row, computing the column of the multi-row with minimum element, and computing the minimum element in the column within the query range [2].

1.2 Our results

The 2D-RMQ encodings mentioned above leave a gap between the lower bound $\Omega(N \log m)$ and the upper bound $O(N \cdot \min\{\log N, m\})$ on the number of bits

stored in a 2D-RMQ encoding. A quick comparison between the encoding complexity of 1D-RMQ and 2D-RMQ data structures can convince the reader that the optimal number of bits per element stored in an encoding should be a function of m, at least for small enough m. For example, we should not look for functions such as $O(N \log\log N)$ as the encoding complexity of the problem. While the upper bound $O(N \cdot \min\{\log N, m\})$ would provide such a characteristic, it is often hard to believe the minimum of two functions as the complexity of a problem. In this work, we prove that the encoding complexity of 2D-RMQ data structures is $\Theta(N \log m)$ bits. As previously mentioned, we only consider the encoding complexity of the problem, and a problem that remains open from this paper is how to actually support queries efficiently, ideally in $O(1)$ time.

We describe our solution in three steps, incrementally improving the space. First we present a solution using space $O(N(\log m + \log\log n))$ bits, introducing the notion of components. We then improve this to an $O(N \log m \log^* n)$ bit solution by bootstrapping our first solution and building an $O(\log^* n)$ depth hierarchical partitioning of our components. Finally, we arrive at an $O(N \log m)$ bit solution using a more elaborate representation of a hierarchical decomposition of components of depth $O(\log n)$.

1.3 Preliminaries

We formally define the problem and then we introduce the notation used in the paper. The input is assumed to be an $m \times n$ matrix A, where $m \leq n$. We let $N = m \cdot n$. The j-th entry in row i of A is denoted by $A[i, j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. For a query range $q = [i_1, i_2] \times [j_1, j_2]$, the query $\mathrm{RMQ}(q)$ reports the index (i, j) of A containing the minimum value in rows $i_1..i_2$ and columns $j_1..j_2$, i.e.

$$\mathrm{RMQ}(q) = \mathrm{argmin}_{(i,j)\in q} A[i, j] \ .$$

For two cells c_1 and c_2 we define $\mathrm{Rect}[c_1, c_2]$ to be the minimal rectangle containing c_1 and c_2, i.e. c_1 and c_2 are at the opposite corners of $\mathrm{Rect}[c_1, c_2]$.

We assume all values of a matrix to be distinct, such that $\mathrm{RMQ}(q)$ is always a unique index into A. This can e.g. be achieved by ordering identical matrix values by the lexicographical ordering of the indexes of the corresponding cells.

2 Tree representation

The basic approach in all of our solutions is that we convert our problem on the matrix A into a tree representation problem, where the goal is to find a tree representation that can be represented within small space.

For a given input matrix A, we build a left-to-right ordered tree T with N leaves, where each leaf of T corresponds to a unique cell of A and each internal node of T has at least two children. Furthermore, the tree should satisfy the following crucial property:

Requirement 1 *For any rectangular query q, consider the leaves of T corresponding to the cells contained in q. The answer to the query q must be the rightmost of these leaves.*

Trivial solution. The most trivial solution is to build a tree of depth one, where all cells/leaves are the children of the root and sorted left-to-right in decreasing value order. A query q will consist of a subset of the leaves, and the rightmost of these leaves obviously stores the minimal value within q, since the leaves are sorted in decreasing order with respect to values. To represent such a tree we store for each leaf the index (i, j) of the corresponding cell, i.e. such a tree can be represented by a list of N pairs, each requiring $\lceil \log m \rceil + \lceil \log n \rceil$ bits. Total space usage is $O(N \log n)$ bits.

Note that at each leaf we explicitly store the index of the corresponding cell in A, which becomes the space bottleneck of the representation: The left to right ordering of the leaves always stores the permutation of all matrix values, i.e. such a representation will require $\Omega(N \cdot \log N)$ bits in the worst-case. In all the following representations we circumvent this bottleneck by not storing the complete total order of all matrix values and by only storing relative positions to other (already laid out) cells of A.

The structure (topology) of any tree T with N leaves can be described using $O(N)$ bits (since all internal nodes are assumed to be at least binary), e.g. using a parenthesis notation. From the known lower bound of $\Omega(N \log m)$ bits for the problem, it follows that the main task will be to find a tree T satisfying Requirement 1, and where the mapping between leaves of T and the cells of A can be stored succinctly.

3 Space $O(N(\log m + \log \log n))$ solution

We present a solution using space $O(N(\log m + \log \log n))$ bits, i.e. it achieves optimal space $O(N \log m)$ for $\log n = m^{O(1)}$. The idea is to build a tree T of depth two in which the nodes $C_1, C_2, \ldots, C_k$ with depth one, from left-to-right, form a partitioning of A. Let C_i denote both a node in T and the subset of cells in A corresponding to the leaves/cells in the subtree of T rooted at C_i. We call such a C_i a *component*. We construct $C_1, \ldots, C_k$ incrementally from left-to-right such that Requirement 1 is satisfied. The children of C_i (i.e. leaves/cells) are sorted in decreasing value ordered left-to-right. We first describe how to construct C_1, and then describe how to construct C_i given $C_1, \ldots, C_{i-1}$, generalizing the notation and algorithm used for constructing C_1. In the following we let α denote a parameter of our construction. We will later set $\alpha = \lceil \log N / \log \log N \rceil$.

Constructing C_1. To construct C_1 we need the following notation: Two cells in A are *adjacent* if they share a side, i.e. (i, j) is adjacent to the (up to) four cells $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. Given a set of cells $S \subseteq A$, we define the undirected graph $G_S = (S, E)$, where $E \subseteq S \times S$ and $(c_1, c_2) \in E$ if and only if c_1 and c_2 are adjacent cells in A.

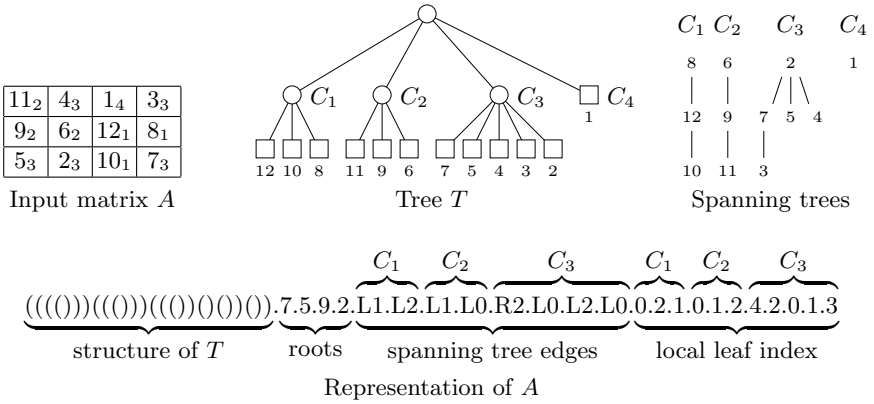

Fig. 1. Construction of $C_1, \ldots, C_4$ for a 3×4 matrix and $\alpha = 3$: In the input matrix (left) subscripts indicate the component numbers. In the tree T (middle) the leaves are labeled with the corresponding values of the cells in A. The spanning trees for $C_1, \ldots, C_4$ are shown (right). In the final representation of A (bottom) "." is not part of the stored representation.

Let S be an initially empty set. We now incrementally include the cells of A in S in decreasing order of the value of the cells. While all the connected components of G_S contain less than α cells, we add one more cell to S. Otherwise the largest component C contains at least α cells, and we terminate and let $C_1 = C$. In the example in Fig. 1, where $\alpha = 3$, we terminate when $S = \{12, 11, 10, 9, 8\}$ and G_S contains the two components $\{11, 9\}$ and $\{12, 10, 8\}$, and let C_1 be the largest of these two components.

We first prove the size of the constructed component C_1.

Lemma 1. $|C_1| \leq 4\alpha - 3$.

Proof. Let S be the final set in the construction process of C_1, such that S contains a connected component of size at least α. Before inserting the last cell c into S, all components in $G_{S \setminus \{c\}}$ have size at most $\alpha - 1$. Since c is adjacent to at most four cells, the degree of c in G_S is at most four, i.e. including c in S can at most join four existing components in $G_{S \setminus \{c\}}$. It follows that the component C_1 in G_S containing c contains at most $1 + 4(\alpha - 1)$ cells. □

Next we will argue that the partition of A into C_1 and $R = A \setminus C_1$ (R will be the leaves to the right of C_1 in T) supports Requirement 1, i.e. for any query q that overlaps both with C_1 and R there is an element $c \in q \cap R$ that is smaller than all elements in $q \cap C_1$, implying that the answer is not in C_1 and will be to the right of C_1 in T. Since by construction all values in $R \setminus S$ are smaller than all values in $S \supseteq C_1$, it is sufficient to prove that there exists a $c \in (q \cap R) \setminus S$.

Lemma 2. *For a query q with $q \cap C_1 \neq \emptyset$ and $q \cap R \neq \emptyset$, there exists a cell $c \in (q \cap R) \setminus S$.*

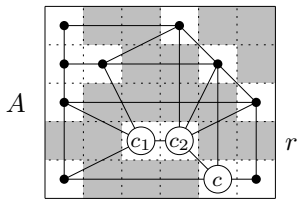

Fig. 2. The edges of $G^{(L)}_{A \setminus L}$. The cells of L are shown as grey.

Proof. Assume by contradiction that $q \subseteq S$. Then each cell in q is either in C_1 or in $S \setminus C_1$, and by the requirements of the lemma and the assumption $q \subseteq S$, there exists at least one cell in both $q \cap C_1$ and $(q \cap S) \setminus C_1$. Since each cell in q belongs to one of these sets, there must exist two adjacent cells c_1 and c_2 in q such that $c_1 \in q \cap C_1$ and $c_2 \in (q \cap S) \setminus C_1$. But since c_1 and c_2 are adjacent, both in S, and $c_1 \in C_1$, then c_2 must also be in C_1 which is a contradiction. □

Constructing C_i. Given how to construct C_1, we now give a description of how to construct C_i when $C_1, \ldots, C_{i-1}$ have already been constructed. To construct C_i we let $L = C_1 \cup \cdots \cup C_{i-1}$. We generalize the notion of adjacency of cells to *L-adjacency*: Given a set L, two cells c_1 and c_2 in $A \setminus L$ are *L-adjacent* if and only if $\text{Rect}[c_1, c_2] \setminus \{c_1, c_2\} \subseteq L$. Note that the adjacency definition we used to construct C_1 is equivalent to $\emptyset$-adjacency. Finally given a set $S \subseteq A \setminus L$, we define the undirected graph $G^{(L)}_S = (S, E)$, where $(c_1, c_2) \in E$ if and only if c_1 and c_2 are L-adjacent. See Fig. 2 for an example of edges defined by L-adjacency.

The algorithm for constructing C_1 now generalizes as follows to C_i: Start with an empty set S. Consider the cells in $A \setminus L$ in decreasing value order and add the cells to S until the largest connected component C of $G^{(L)}_S$ has size at least α (or until all elements in $A \setminus L$ have been added to S). We let $C_i = C$. Note that with $L = \emptyset$ this algorithm is exactly the algorithm for computing C_1. Fig. 1 illustrates an example of such a partitioning of a 3×4 matrix for $\alpha = 3$.

Lemma 3. *All nodes in a graph $G^{(L)}_S$ have degree at most $2m$.*

Proof. Let $G^{(L)}_S = (S, E)$. Consider a cell $c \in S$. For any row r, there exists at most one cell $c_1 \in S$ in row r to the left of c (or immediately above c), such that $(c, c_1) \in E$. Otherwise there would exist two cells c_1 and c_2 in S to the left of c in row r, where c_1 is to the left of c_2 (see Fig. 2). But then $c_2 \in \text{Rect}[c, c_1]$, i.e. c and c_1 would not be L-adjacent and $(c, c_1) \notin E$. It follows that for any row, at most one cell to the left and (symmetrically) one cell to the right of c can have an edge to c in $G^{(L)}_S$. □

The following two lemmas generalize Lemmas 1 and 2 for C_1 to C_i.

Lemma 4. $|C_i| \leq m(\alpha - 1) + \alpha \leq 2m\alpha$.

Proof. Since the last cell c added to S is in C_i and by Lemma 3 has degree at most $2m$ in $G_S^{(L)}$, and before adding c to S all components had size at most $\alpha - 1$, it follows that $|C_i| \leq 1 + 2m(\alpha - 1) \leq 2m\alpha$. This bound can be improved to $1 + (m+1)(\alpha - 1)$ by observing that adding c can at most join $m+1$ existing components: if c is adjacent to two cells c_1 and c_2 in the same row, one in a column to the left and one in a column to the right of c, then c_1 and c_2 must be L-adjacent, provided if c is not in the same row as c_1 and c_2, i.e. c_1 and c_2 were already in the same component before including c. □

We next prove the generalization of Lemma 2 to C_i. Let $L = C_1 \cup \cdots \cup C_{i-1}$ be the union of the already constructed components (to the left of C_i) and let $R = A \setminus \{C_1 \cup \cdots \cup C_i\}$ be the set of cells that eventually will be to the right of C_i in T. The following lemma states that if a query q contains one element from each of C_i and R, then there exists a cell $c \in q \cap R$ with smaller value than all values in C_i. It is sufficient to show that there exists a cell $c \in q \cap R \setminus S$ since by construction all values in $R \setminus S$ are smaller than $\min(S) \leq \min(C_i)$.

Lemma 5. *For a query q with $q \cap C_i \neq \emptyset$ and $q \cap R \neq \emptyset$, there exists a $c \in q \cap R \setminus S$.*

Proof. Assume by contradiction that $q \cap R \setminus S = \emptyset$. Then $q \subseteq S \cup L = C_i \cup (S \setminus C_i) \cup L$. By the requirement of the lemma and the assumption $q \cap R \setminus S = \emptyset$, there exist $c_1 \in q \cap C_i$ and $c_2 \in q \cap S \setminus C_i$. We will now show that c_1 and c_2 are connected in $G_S^{(L)}$ (not necessarily by a direct edge), i.e. c_1 and c_2 are both in C_i, which is a contradiction.

We construct a path from c_1 to c_2 in $G_S^{(L)} = (S, E)$ as follows. If $\text{Rect}[c_1, c_2] \setminus \{c_1, c_2\} \subseteq L$, then c_1 and c_2 are L-adjacent, and we are done. Otherwise let c_3 be the cell, where $\text{Rect}[c_1, c_3] \setminus \{c_1, c_3\}$ contains no cell from S (c_3 is the closest cell to c_1 in $S \cap \text{Rect}[c_1, c_2] \setminus \{c_1, c_2\}$). Therefore, c_1 and c_3 are L-adjacent, i.e. $\text{Rect}[c_1, c_3] \setminus \{c_1, c_3\} \subseteq L$ and $(c_1, c_3) \in E$. By applying this construction recursively between c_3 and c_2 we construct a path from c_1 to c_2 in $G_S^{(L)}$. □

Representation. We now describe how to store sufficient information about the tree T such that we can decode T and answer a query. For each connected component C_i we store a *spanning tree* $\mathcal{S}_i$, such that $\mathcal{S}_i$ consists exactly of the edges that were joining different smaller connected components while including new cells into S. We root $\mathcal{S}_i$ at the cell with smallest value in C_i (but any node of C_i could have been chosen as the root/anchor of the component).

In the following representation we assume n and m are already known (can be coded using $O(\log N)$ bits). Furthermore it is crucial that when we layout C_i we have already described (and can decode) what are the cells of $C_1, \ldots, C_{i-1}$, i.e. L in the construction of C_i.

Crucial to our representation of C_i is that we can efficiently represent edges in $G_S^{(L)}$: If there is an edge from c_1 to c_2 in $G_S^{(L)}$, then c_1 and c_2 are L-adjacent, i.e. to navigate from c_1 to c_2 it is sufficient to store what row c_2 is stored in, and if c_2 is to the left (L) or right (R) of c_1 (if they are in the same column, then c_2 is to the left of c_1), since we just move vertically from c_1 to the row of

c_2 and then move in the left or right direction until we find the first non-L slot (here it is crucial for achieving our space bounds that we do not need to store the number of columns moved over). It follows that a directed edge from c_1 to c_2 identifying the cell c_2 can be stored using $1 + \lceil \log m \rceil$ bits, provided we know where the cell c_1 (and L) is.

The details of the representation are the following (See Fig. 1 for a detailed example of the representation):

- We first store the structure of the spanning trees for $C_1, \ldots, C_k$ as one big tree in parenthesis notation (putting a super-root above the roots of the spanning trees); requires $2(N+1)$ bits. This encodes the number of components, the size of each component, and for each component the tree structure of the spanning tree.
- The *global index* of the root cell for each C_i, where we enumerate the cells of A by $0..N-1$ row-by-row, top-down and left-to-right; requires $k\lceil \log N \rceil$ bits.
- For each of the components $C_1, \ldots, C_k$, we traverse the spanning tree in a DFS traversal, and for each edge (c_1, c_2) of the spanning tree for a component C_i, where c_1 is the parent of c_2, we store a bit L/R and the row of c_2 (note that our representation ensures that the cells of $L = C_1 \cup \cdots \cup C_{i-1}$ are previously identified, which is required). Since there are $N-k$ edges in total in the k spanning trees, we have that the edges in total require $(N-k)(1+\lceil \log m \rceil)$ bits.
- For each component C_i, we store the leaves/cells of C_i in decreasing value order (only if $|C_i| \geq 2$): For each leaf, we store its local index in C_i with respect to the following enumeration of the cells. Since the spanning tree identifies exactly the cells in C_i and $|C_i| \leq 2m\alpha$, we can enumerate these cells by $0..2m\alpha - 1$, say row-by-row, top-down and left-to-right; in total at most $N\lceil \log(2m\alpha) \rceil$ bits.

Lemma 6. *The representation of T requires $O(N(\log m + \frac{\log N}{\alpha} + \log \alpha))$ bits.*

Proof. From the above discussion we have that the total space usage in bits is

$$2(N+1) + k\lceil \log N \rceil + (N-k)(1 + \lceil \log m \rceil) + N\lceil \log(2m\alpha) \rceil \ .$$

For all C_i we have $\alpha \leq |C_i| \leq 2m\alpha$, except for the last component C_k, where we have $1 \leq |C_k| \leq 2m\alpha$. It follows that $k \leq \lceil N/\alpha \rceil$, and the bound stated in the lemma follows. □

Setting $\alpha = \lceil \log N / \log \log N \rceil$ and using $\log N \leq 2 \log n$, we get the following space bound.

Theorem 1. *There exists a 2D-RMQ encoding of size $O(N(\log m + \log \log n))$ bits.*

We can reconstruct T by reading the encoding from left-to-right, and thereby answer queries. We reconstruct the tree structure of the spanning trees of the

components by reading the parenthesis representation from left-to-right. This gives us the structure of T including the size of all the components from left to right. Given the structure of T and the spanning trees, the positions of the remaining fields in the encoding are uniquely given. Since we reconstruct the components from left-to-right and we traverse the spanning trees in depth first order, we can decode each spanning tree edge since we know the relevant set L and the parent cell of the edge. Finally we decode the indexes of the leaves in each C_i, by reading their local indexes in C_i in decreasing order.

4 Space $O(N \log m \log^* n)$ solution

In the following we describe how to recursively apply our $O(N(\log m + \log\log n))$ bit space solution to arrive at an $O(N \log m \log^* n)$ bit solution, where $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$, $\log^{(0)} n = n$, and $\log^{(i+1)} n = \log\log^{(i)} n$.

Let K be an integer determining the depth of the recursion. At the root we have a problem of size $N_0 = N$ and apply the partitioning algorithm of Section 3 with $\alpha = \lceil \log N_0 \rceil$, resulting in sub-components $C_1, C_2, C_3, \ldots$ each of size at most $N_1 = 2m\lceil \log N_0 \rceil$. Each C_i is now recursively partitioned. A component C at level i in the recursion has size at most N_i. When constructing the partitioning of C, we initially let $L = A \setminus C$, i.e. all cells outside C act as neutral "$+\infty$" elements for the partitioning inside C_i. The component C is now partitioned as in Section 3 with $\alpha = \lceil \log N_i \rceil$ into components of size at most $N_{i+1} = 2m\lceil \log N_i \rceil$. A special case is when $|C| \leq N_{i+1}$, where we skip the partitioning of C at this level of the recursion (to avoid creating degree one nodes in T).

Let N_i denote an upper bound on the problem size at depth i of the recursion, for $0 \leq i \leq K$. We know $N_0 = N$ and $N_{i+1} \leq 2m \log N_i$. By induction it can be seen that $N_i \leq 2m \log^{(i)} N + 12m \log m$: For $i = 0$ we have $N_0 = N \leq 2m \log^{(0)} N$ and the bound is true. Otherwise we have

$$\begin{aligned}
N_{i+1} &\leq 2m\lceil \log N_i \rceil \\
&\leq 2m\lceil \log(2m \log^{(i)} N + 12m \log m) \rceil \\
&\leq 2m\lceil \log((2m + 12m \log m) \log^{(i)} N) \rceil \quad (\text{for } \log^{(i)} N \geq 1) \\
&\leq 2m \log^{(i+1)} N + 2m \log(2m + 12m \log m) + 2m \\
&\leq 2m \log^{(i+1)} N + 12m \log m \quad (\text{for } m \geq 2)\ .
\end{aligned}$$

Representation. Recall that T is an ordered tree with depth K, in which each leaf corresponds to a cell in A, and each internal node corresponds to a component and the children of the node is a partitioning of the component.

- First we have a parenthesis representation of T; requiring $2N$ bits.
- For each internal node C of T in depth first order, we store a parenthesis representation of the structure of the spanning tree for the component represented by C. Each cell c of A corresponds to a leaf ℓ of T and is a node in

the spanning tree of each internal node on the path from ℓ to the root of T, except for the root, i.e. c is in at most K spanning trees. It follows that the parenthesis representation of the spanning trees takes $2KN$ bits.
- For each internal node C of T in depth first order, we store the local index of the root r of the spanning tree for C as follows. Let C' be the parent of C in T, where C' is partitioned at depth i of the recursion. We enumerate all cells in C' row-by-row, top-down and left-to-right by $0..|C'|-1$ and store the index of r among these cells. Since $|C'| \leq N_{i-1}$, this index requires $\lceil \log N_{i-1} \rceil$ bits. The total number of internal nodes in T resulting from partitionings at depth i in the recursion is at most $2N/\lceil \log N_{i-1} \rceil$ (at least half of the components in the partitionings are large enough containing at least $\lceil \log N_{i-1} \rceil$ cells), from which it follows that to store the roots we need at most $2N$ bits at each depth of the recursive partitioning, i.e. in total at most $2KN$ bits.
- For each internal node C of T in depth first order, we store the edges of the spanning tree of C in a depth first traversal of the spanning tree edges. Each edge requires $1 + \lceil \log m \rceil$ bits. Since there are less than N spanning tree edges for each level of T, the total number of spanning tree edges is bounded by NK, i.e. total space $NK(1 + \lceil \log m \rceil)$ bits.
- For each leaf ℓ of T with parent C we store the index of ℓ among the up to N_K cells in C, requiring $\lceil \log N_K \rceil$ bits. Total space $N\lceil \log N_K \rceil$ bits.

The total number of bits required becomes $2N + 2KN + 2KN + NK(1 + \lceil \log m \rceil) + N\lceil \log N_K \rceil = O(NK \log m + N \log^{(K+1)} N)$. Setting $K = \log^* N = O(\log^* n)$ we get the following theorem.

Theorem 2. *There exists a 2D-RMQ encoding using $O(N \log m \log^* n)$ bits.*

5 Space $O(N \log m)$ solution

The $O(N \log m \log^* n)$ bits solution of the previous section can be viewed as a top-down representation, where we have a representation identifying all the cells of a component C before representing the recursive partitioning of C, and where each C_i is identified using a spanning tree local to C. To improve the space to $O(N \log m)$ bits we adopt a bottom-up representation, such that the representation of C is the concatenation of the representations of $C_1, \ldots, C_k$, prefixed by information about the sizes and the relative placements of the C_i components.

We construct T top-down as follows. Let C be the current node of T, where we want to identify its children $C_1, \ldots, C_k$. Initially C is the root of T covering all cells of A. If $|C| < m^8$, we just make all cells of (the component) C leaves below (the node) C in T in decreasing value order from left-to-right. Otherwise we create a partition $C_1, \ldots, C_k$ of C, make $C_1, \ldots, C_k$ the children of C from left-to-right, and recurse for each of the generated C_i components.

This solution which partitions a component C into a set of smaller components $C_1, \ldots, C_k$, takes a parameter $\alpha = \lfloor |C|^{1/4}/2 \rfloor$ and the set of cells L to the left of C in the final tree T, i.e. given C and L, we for C_i have the corresponding

$L_i = L \cup C_1 \cup \cdots \cup C_{i-1}$. To be able to construct the partitioning, the graph $G_C^{(L)}$ must be connected and $|C| \geq \alpha$. Given such a C we can find a partition satisfying that each of the constructed C_i components has size at least α and at most $4\alpha^2 m^2$, and $G_{C_i}^{(L_i)}$ is a connected graph. It follows that $k \leq 4 \cdot |C|^{3/4}$ (since $\alpha = \lfloor |C|^{1/4}/2 \rfloor \geq |C|^{1/4}/4$) and $|C_i| \leq 4\alpha^2 m^2 \leq 4 \cdot \lfloor |C|^{1/4}/2 \rfloor^2 \cdot |C|^{1/4} \leq |C|^{3/4}$ (since $m^2 = (m^8)^{1/4} \leq |C|^{1/4}$ for $|C| \geq m^8$).

The details of how to construct the partitions and how to use it to derive a space-efficient representation is described in the full version of the paper.

Theorem 3. *There exists a 2D-RMQ encoding using $O(N \log m)$ bits.*

6 Conclusion

We have settled the encoding complexity of the two dimensional range minimum problem as being $\Theta(N \log m)$. In the worst case, a query algorithm requires to decode the representation to be able the answer queries. It remains an open problem to build a data structure with this space bound and with efficient queries.

References

1. A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In B. Ma and K. Zhang, editors, *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 286–294. Springer, 2007.
2. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.
3. B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In K. Mehlhorn, editor, *SoCG*, pages 131–139. ACM, 1989.
4. P. Davoodi. *Data Structures: Range Queries and Space Efficiency.* PhD thesis, Aarhus University, Aarhus, Denmark, 2011.
5. P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In J. Gudmundsson, J. Mestre, and T. Viglas, editors, *COCOON*, volume 7434 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2012.
6. E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, editors, *ICALP (1)*, volume 5555 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 2009.
7. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
8. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In R. A. DeMillo, editor, *STOC*, pages 135–143. ACM, 1984.
9. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
10. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
11. H. Yuan and M. J. Atallah. Data structures for range minimum queries in multidimensional arrays. In M. Charikar, editor, *SODA*, pages 150–160. SIAM, 2010.