Two Dimensional Range Minimum Queries and Fibonacci Lattices*

Gerth Stølting Brodal[1], Pooya Davoodi[2]**, Moshe Lewenstein[3]***, Rajeev Raman[4], and S. Srinivasa Rao[5]†

[1] MADALGO‡, Aarhus University, Denmark. E-mail: gerth@cs.au.dk
[2] Polytechnic Institute of New York University, United States. E-mail: pooyadavoodi@gmail.com
[3] Bar-Ilan University, Israel. E-mail: moshe@cs.biu.ac.il
[4] University of Leicester, UK. E-mail: r.raman@leicester.ac.uk
[5] Seoul National University, S. Korea. E-mail: ssrao@cse.snu.ac.kr

Abstract. Given a matrix of size N, two dimensional range minimum queries (2D-RMQs) ask for the position of the minimum element in a rectangular range within the matrix. We study trade-offs between the query time and the additional space used by indexing data structures that support 2D-RMQs. Using a novel technique—the discrepancy properties of Fibonacci lattices—we give an indexing data structure for 2D-RMQs that uses $O(N/c)$ bits additional space with $O(c \log c (\log \log c)^2)$ query time, for any parameter c, $4 \leq c \leq N$. Also, when the entries of the input matrix are from $\{0, 1\}$, we show that the query time can be improved to $O(c \log c)$ with the same space usage.

1 Introduction

The problem we consider is to preprocess a matrix (two dimensional array) of values into a data structure that supports *range minimum queries (2D-RMQs)*, asking for the position of the minimum in a rectangular range within the matrix. More formally, an input is an $m \times n$ matrix A of $N = m \cdot n$ distinct totally ordered values, and a range minimum query asks for the position of the minimum value in a range $[i_1 \cdots i_2] \times [j_1 \cdots j_2]$, where $0 \leq i_1 \leq i_2 \leq m-1$ and $0 \leq j_1 \leq j_2 \leq n-1$ (the case when $m = 1$ is referred to hereafter as 1D-RMQ). Both 1D-

* Part of this work appeared in *ESA 2012*.
** Research supported by NSF grant CCF-1018370 and BSF grant 2010437. Research partly done while the author was a PhD student at MADALGO, Aarhus University, Denmark.
*** Research supported by BSF grant 2010437 and GIF grant 1147/11. Research partly done while the author was visiting MADALGO, Aarhus University, Denmark.
† Research partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant number 2012-0008241).
‡ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

Table 1. Results for 2D-RMQs. Encoding results marked with $*$.

Reference	Query time	Space (bits)
[13]	$O(\log N)$	$O(N \log^2 N)$
[1, 2]	$O(1)$	$O(N \log N)$
[8]	Any	$\Omega(N \log N)^*$
[4]	$O(1)$	$O(N \cdot \min\{m, \log n\})^*$
[4]	Any	$\Omega(N \log m)^*$
[4]	$O(c \log^2 c)$	$O(N/c)$
[4]	$\Omega(c)$	$O(N/c)$
New	$O(c \log c (\log \log c)^2)$	$O(N/c)$

and 2D-RMQ problems are fundamental problems with a long history dating back nearly 25 years [13, 5] that find applications in computer graphics, image processing, computational biology, databases, etc.

We consider this problem in two models [4]: the *encoding* and *indexing* models. In the encoding model, we preprocess A to create a data structure *enc* and queries have to be answered just using *enc*, *without* access to A. In the indexing model, we create an index *idx* and are able to refer to A when answering queries. The main measures we are interested in are (i) the time to answer queries (ii) the space (in bits) taken by *enc* and *idx* respectively and (iii) the preprocessing time (in general there may be a trade-off between (i) and (ii)). Note that an encoding is only interesting if it uses $o(N \log N)$ bits—otherwise we could store the sorted permutation of A in *enc* and trivially avoid access to A when answering queries. We defer to [4] for a further discussion of these models.

The 1D-RMQ problem seems to be well-understood. In the encoding model, it is known that *enc* must be of size at least $2N - O(\log N)$ bits [17, 12]. Furthermore, 1D-RMQs can be answered in $O(1)$ time[6] using an encoding of $2N + o(N)$ bits after $O(N)$-time preprocessing [12, 7]. In the indexing model, Brodal et al. [4] showed that an index of size $O(N/c)$ bits suffices to answer queries in $O(c)$ time, and that this trade-off is optimal.

Following Chazelle and Rozenberg's work [5], the study of 2D-RMQs was restarted by Amir et al. [1], who explicitly asked in what ways the 2D- and 1D-RMQ problems differ; this question turned out to have a complex answer:

1. In the indexing model, Atallah and Yuan [2] showed that, just as for 1D-RMQs, $O(N)$ preprocessing time suffices to get an index that answers 2D-RMQs in $O(1)$ time, improving upon the preprocessing time of [1]. Their index size of $O(N)$ words, or $O(N \log N)$ bits, was further reduced to $O(N)$ bits by Brodal et al. [4], while maintaining query and preprocessing times.
2. Demaine et al. [8] showed that, unlike 1D-RMQ, non-trivial (size $o(N \log N)$ bits) encodings for 2D-RMQs do not exist in general. Non-trivial encodings may only exist when $m \ll n$ [4], or in expectation for random A [14].

[6] This result uses, as do all the results in this paper, the word RAM model of computation with word size logarithmic in the size of inputs.

3. In the indexing model, Brodal et al. [4] showed that with an index of size $O(N/c)$ bits, $\Omega(c)$ query time is needed to answer queries, and gave an index of size $O(N/c)$ bits that answered queries in $O(c \log^2 c)$ time.

Thus, the question of whether 1D-RMQs and 2D-RMQs differ in the time-space trade-off for the indexing model remains unresolved, and we make progress towards this question in this paper.

Our Contributions. We show the following results:

- We give an index of size $O(N/c)$ bits that for any $4 \leq c \leq N$, takes $O(c \log c (\log \log c)^2)$ time to answer 2D-RMQs, improving upon Brodal et al.'s result by nearly a $\log c$ factor.
- For the case where A is a 0-1 matrix, and the problem is simplified to finding the location of *any* 0 in the query rectangle (or else report that there are no 0s in the query rectangle) we give an index of size $O(N/c)$ bits that supports 2D-RMQs in $O(c \log c)$ time, improving upon the Brodal et al.'s result by a $\log c$ factor. Note that in the 1D-RMQ case, the 0-1 case and the general case have the same time-space trade-off [4].

We recursively decompose the input matrix, which in turn partitions a 2D-RMQ into several disjoint sub-queries, some of which are *3-sided* or *2-sided* queries, which have one or two sides of the query rectangle ending at the edge of a recursive sub-matrix. To deal with 3-sided queries in the general case, we recursively partition the 3-sided query into a number of disjoint sub-queries including 2-sided queries (also called dominance queries), whereas for 0-1 matrices we make a simple non-recursive data structure. The techniques used to answer the 2-sided queries in the general case are novel. In the following we explain *Fibonacci lattices* that are used in our data structure.

Fibonacci lattices. The sequence of Fibonacci numbers is defined as follows:

$$f_1 = f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}; \text{ for } n \geq 3 \ .$$

Fibonacci lattices are 2D point sets with applications in graphics and image processing [6, 11] and parallel computing [10], defined as follows:

Definition 1. [15] *Let f_k be the k-th number in the Fibonacci sequence, where $f_1 = f_2 = 1$. In an $n \times n$ grid, the Fibonacci lattice of size $n = f_k$ is the two dimensional point set*

$$\{(i, i \cdot f_{k-1} \bmod n) \mid i = 0, \ldots, n-1\} \ .$$

Figure 1 depicts an example of Fibonacci lattices. A Fibonacci lattice has several "low-discrepancy" properties including:

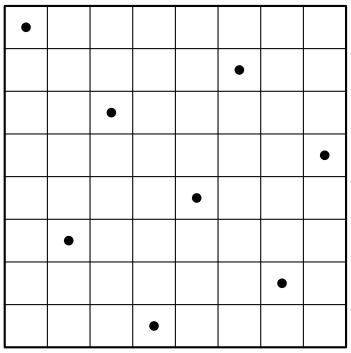

Fig. 1. The Fibonacci lattice of size 8.

Lemma 1. [11] *In an $n \times n$ grid, there exists a constant α such that any axis-aligned rectangle whose area is larger than $\alpha \cdot n$, contains at least one point from the Fibonacci lattice of size n.*

We solve 2-sided queries via *weighted* Fibonacci lattices, in which every point is assigned a value from A. Such queries are reduced, via new properties of Fibonacci lattices (which may be useful in other contexts as well), to 2-sided queries on the weighted Fibonacci lattice. The weighted Fibonacci lattice is space-efficiently encoded using a succinct index of Farzan et al. [9].

In our data structure, we make use of the property of Fibonacci numbers that f_x divides f_y iff x divides y, for $x, y \geq 3$. Let the function $\gcd(x, y)$ return the greatest common divisor of the numbers x and y. Therefore, $\gcd(f_k, f_{k-1}) = 1$.

Outline. The paper is structured as follows. In Section 2 we explain how to support 2-sided queries. In Section 3 we show how to use it to support 4-sided queries, giving our main result, the trade-off of $O(c \log c (\log \log c)^2)$ query time and $O(N/c)$ bits additional space for 2D-RMQs. In Section 4, we consider 0-1 matrices. Section 4 is devoted to 2D-RMQs in 0-1 matrices.

2 2-Sided 2D-RMQs and Fibonacci Lattices

We present data structures that support 2-sided 2D-RMQs within an $n \times n$ matrix. Without loss of generality, assume that the query range is of the form $[0 \cdots i] \times [0 \cdots j]$, i.e. that 2-sided queries have a fixed corner on the top-left corner of the matrix. First we present a data structure of size $O(n \log n)$ bits additional space that supports 2-sided queries in $O(n \log n)$ time. Although this result is weaker than that of [4], it gives the main ideas. We then reduce the query time to $O(n)$ with the same space usage.

A First Solution. Given n, we create a modified Fibonacci lattice as follows. Choose the Fibonacci lattice of smallest size $n' \geq n$, clearly $n' = O(n)$. Observe

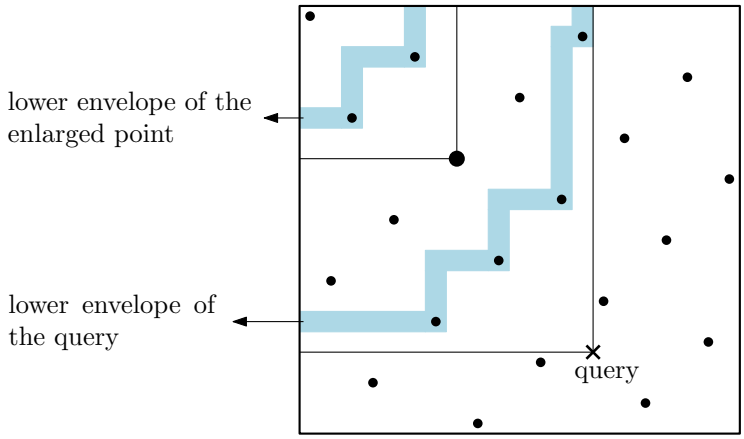

Fig. 2. The 2-sided query is divided into two sub-queries by its lower envelope. The enlarged point above the lower envelope is the FP with minimum priority within the query range, and its priority comes from an element below its lower envelope.

that no points in the Fibonacci lattice have the same x or y coordinate (this follows from the fact that $\gcd(f_k, f_{k-1}) = 1$). Eliminate all points that do not lie in $[0 \cdots n-1] \times [0 \cdots n-1]$ and call the resulting set F; note that F satisfies Lemma 1, albeit with a different constant in the $O(\cdot)$ notation. The points of F, which specify entries of the matrix, are called the *FPs*. For any point p in the matrix, its *2-sided region* is the region above and to the left of p. As shown in Figure 2, a 2-sided query is divided into two disjoint sub-queries by a polygonal chain (called the *lower envelope*) derived from all the FPs within the query range that are not in the 2-sided region of any other FP within the query range. We find the answer to each of the two sub-queries and return the smaller. The lower sub-query (the area below the lower envelope) is answered by a brute-force (scan of all entries in the area), which will be described later. In the following, we explain how to answer the upper sub-query (area above and on the lower envelope).

Upper Sub-query. Each FP $p \in F$ is assigned a "priority" which equals the value of the minimum element within its 2-sided region (i.e. p's priority is the value of the element returned by a 2-sided query at p). As the upper sub-query region is entirely covered by the 2-sided regions of all FPs within the query range, the minimum element in the upper sub-query is the smallest priority of all the FPs within the upper sub-query. The data structure for the upper sub-query comprises three parts: the first maps each FP p to the row and column of the matrix element containing the minimum value in the 2-sided query at p, using say a balanced search tree.

The next part is an *orthogonal range reporting* data structure that returns a list of all FPs that lie in a rectangle $[i_1 \cdots i_2] \times [j_1 \cdots j_2]$.

Lemma 2. *Given the Fibonacci lattice of size n, there exists a data structure of size $O(n)$ bits that supports 4-sided orthogonal range reporting queries in $O(k)$ time, where k is the number of reported points.*

Proof. We report all the FPs within the query rectangle by sweeping a horizontal line from the top of the rectangle towards the bottom in k steps, reporting one FP at each step in $O(1)$ time. The line initially jumps down from above the rectangle to the row in which the top-most FP within the rectangle is located. In subsequent steps, we move the sweep line from the row i containing the just-reported FP down to a row $i + x$ containing the next FP within the query rectangle. Notice that the function $L(i) = i \cdot f_{k-1} \bmod n$ is invertible, and thus row and column indexes in the lattice can be converted to each other in constant time.

Let the columns j_ℓ and j_r be the left and right sides of the query. The point in row $i + x$ is within the rectangle if $j_\ell \leq L(i + x) = (L(i) + L(x)) \bmod n \leq j_r$; thus $L(x) \geq j_\ell - L(i) + n$ or $L(x) \leq j_r - L(i)$. Let $Z_1(i) = j_\ell - L(i) + n$ and $Z_2(i) = j_r - L(i)$. Since there is no point between the rows i and $i + x$ within the rectangle, therefore x must be the minimum positive number such that $L(x) \geq Z_1(i)$ or $L(x) \leq Z_2(i)$. We preprocess $L(i)$ for $i \in [1 \cdots n-1]$ into a data structure to find x in $O(1)$ time.

Construct an array $L_{inv}[0 \cdots n-1]$, where $L_{inv}[L(x)] = x$. We find the minimum in each of the ranges $[Z_1(i) \cdots n-1]$ and $[0 \cdots Z_2(i)]$, and we return the smaller one as the right value for x. To do this, we encode L_{inv} with a 1D-RMQ structure of size $O(n)$ bits that answers queries in $O(1)$ time [16, 12, 7]. □

The final part solves the following problem: given a point q, find the point $p \in F$ in the 2-sided region specified by q with the smallest priority. For this, we use a *succinct index* by Farzan et al. [9, Corollary 1], which takes F and the priorities as input, preprocesses it and outputs a data structure D of size $O(n)$ bits. D encodes priority information about points in F, but *not* the coordinates of the points in F. Given a query point p, D returns the desired output q, but while executing the query, it calls the instance of Lemma 2 a number of times to retrieve coordinates of relevant points. The index guarantees that each call will report $O(\log n)$ points. The result is stated as follows:

Lemma 3. [9, Corollary 1] *Given n points in 2D rank space, where each point is associated with a priority, there exists a succinct index of size $O(n)$ bits that supports queries asking for the point with minimum priority within a 2-sided range in $O(\log \log n \cdot (\log n + T))$ time. Here T is the time taken to perform orthogonal range reporting on the n input points, given that the index guarantees that no query it asks will ever return more than $O(\log n)$ points. The space bound does not include the space for the data structure for orthogonal range reporting.*

We now put things together. To solve the upper sub-query, we use Lemmas 2 and 3[7] to find the FP with lowest priority in $O(\log n \log \log n)$ time (by Lemma 2

[7] The points in F are technically not in 2D rank space, as some rows may have no points from F; this can be circumvented as in [3] (details omitted).

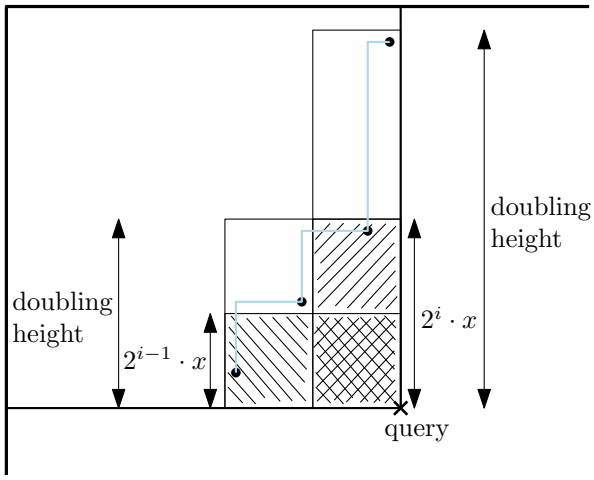

Fig. 3. The cross sign is the bottom-right corner of a 2-sided query.

we can use $T = O(\log n)$ in Lemma 3). Since each FP is mapped to the location of the minimum-valued element that lies in its 2-sided region, we can return the minimum-valued element in the upper query region in $O(\log n)$ further time. The space usage is $O(n)$ bits for Lemmas 2 and 3, and $O(n \log n)$ bits for maintaining the location of minimum-valued elements (the mapping).

Corollary 1. *The upper sub-query can be solved in $O(\log n \log\log n)$ time using $O(n \log n)$ bits.*

Lower Sub-Query. The lower sub-query is answered by scanning the whole region in time proportional to its size. The following lemma states that its size is $O(n \log n)$, so the lower sub-query also takes $O(n \log n)$ time (the region can be determined in $O(n)$ time by a sweep line similar to Lemma 2).

Lemma 4. *The area of the lower sub-query region is $O(n \log n)$.*

Proof. Consider $O(\log n)$ axis-aligned rectangles each with area $\alpha \cdot n$, with different aspect ratios, where the bottom-right corner of each rectangle is on the query point, where α is the constant of Lemma 1. By Lemma 1, each of the rectangles has at least one FP (see Figure 3).

Let $x \times (\alpha n/x)$ be the size of the first rectangle such that it contains the topmost FP on the lower envelope. Let $(2^i x) \times ((\alpha n)/(2^i x))$ be the size of the other rectangles, for $i = 1 \cdots \lceil \log(\alpha n/x) \rceil$. Doubling the height of each rectangle, i.e. taking rectangle $(2^{i+1} x) \times ((\alpha n)/(2^i x))$ instead of $(2^i x) \times ((\alpha n)/(2^i x))$, ensures that all the rectangles together cover the area below the lower envelope, and this does not increase the size of the rectangles asymptotically, thus the total area is $O(\alpha \cdot n \cdot \log n) = O(n \log n)$. □

From Corollary 1 and Lemma 4 we obtain:

Lemma 5. *There exists an index of size $O(n \log n)$ bits that can answer 2-sided 2D-RMQs within an $n \times n$ matrix in $O(n \log n)$ query time.*

Improving Query Time to $O(n)$. The bottleneck in the query time of Lemma 5 is the brute force algorithm, which reads the entire region below the lower envelope. To reduce the size of this region to $O(n)$, we increase the number of FPs in the grid from n to at most $n \log n$.

Lower Sub-Query. We create a *dense* Fibonacci lattice of at most $n \log n$ FPs in the $n \times n$ grid A as follows. In the preprocessing of A, we expand A to an $n \log n \times n \log n$ grid A' by expanding each cell to a $\log n \times \log n$ *block*. We make a (modified) Fibonacci lattice of size $n \log n$ in A'. In the next step, we shrink the Fibonacci lattice as follows, such that it fits in A. Put a FP in a cell of A iff the block of A' corresponding to the cell has at least one FP. Notice that the number of FPs in A is at most $n \log n$. The following lemma states that the area below the lower envelope, after growing the number of FPs to $n \log n$, is $O(n)$.

Lemma 6. *In the dense FP set, the area of the lower sub-query is $O(n)$.*

Proof. Recall that A and A' are respectively the $n \times n$ and $n \log n \times n \log n$ grids. We can deduce from Lemma 4 that the area below the lower envelope in the Fibonacci lattice of size $n \log n$ in A' is $O(n \log^2 n)$. We prove that the area below the lower envelope in A is $O(n)$ by contradiction. Suppose that the area is $\omega(n)$. Since each element in A corresponds to a $\log n \times \log n$ block in A', the same region in A' has area $\omega(n \log^2 n)$, and it can be easily shown that this region is also below the lower envelope in A'. This is a contradiction. □

Upper Sub-Query. We proceed as in the previous subsection, but with a space budget of $O(n \log n)$ bits. We begin by modifying Lemma 2 to work with the dense Fibonacci lattices:

Lemma 7. *Given the dense Fibonacci lattice of size at most $n \log n$ within an $n \times n$ grid A, there exists a data structure of size $O(n \log n)$ bits that supports orthogonal range reporting queries in $O(k \log^2 n)$ time, where k is the number of reported points.*

Proof. Make the data structure D of Lemma 2 for the $n \log n \times n \log n$ grid A'. Recall that D supports reporting queries in $O(k)$ time using $O(n \log n)$ bits. Now given a query range within A, extend it to A'. While D reports each point in the query within A', change the coordinates of the point to the corresponding coordinates in A, and do not report it if the point has been already reported. Since each of the k points in the query within A can correspond to at most $\log^2 n$ points in A', the query time is $O(k \log^2 n)$. □

From Lemmas 7 and 3, we can find the FP with lowest priority in $O(n \log n)$ bits and $O(\log^3 n \log \log n)$ time[8]. Unfortunately, the mapping from FPs to minimum

[8] Again, when using Lemma 3, the fact that the dense FP points are not in 2D rank space is circumvented as in [3].

values now requires $O(n(\log n)^2)$ bits, which is over budget. For this, recall that the priority of each FP p was previously the minimum value in the 2-sided region of p. Instead of mapping p to the position of this minimum, we change the procedure of assigning priorities as follows. Suppose the minimum value in p's 2-sided region is x. If there is another FP q contained in p's 2-sided region, such that x is also the minimum in q's 2-sided region, then we assign p a priority of $+\infty$; otherwise we assign p a priority equal to x (ties between priorities of FPs are broken arbitrarily). This ensures that entry containing the value equal to the priority of p always comes from below the lower envelope of p: once we obtain a FP with minimum priority within the sub-query, we read the entire region of the lower envelope of the FP to find the position that its priority comes from (see Figure 2). This can be done in $O(n)$ time, by Lemma 6. We have thus shown:

Lemma 8. *There exists a data structure of size $O(n \log n)$ bits additional space that can answer 2-sided 2D-RMQs within an $n \times n$ matrix in $O(n)$ query time, excluding the space to perform range reporting on the dense FP set.*

3 Improved Trade-off for 2D-RMQs

We present a data structure of size $O(N/c \cdot \log c \log\log c)$ bits additional space that supports 2D-RMQs in $O(c \log\log c)$ time in a matrix of size N, for any $c \leq N$. Substituting c with $O(c \log c \log\log c)$ gives a data structure of size $O(N/c)$ bits additional space with $O(c \log c (\log\log c)^2)$ query time.

We first reduce the problem to answering queries that are within small $c \times c$ matrices, and then we explain how we can answer such queries. Answering these queries is in fact the bottleneck of our solution. The reduction algorithm spends $O(N/c)$ bits additional space and $O(c)$ query time (later in Lemma 10). We show how to answer each query within a $c \times c$ matrix in $\mathrm{O}(c \log\log c)$ time using $O(c \log c \log\log c)$ bits additional space. The reduction algorithm makes this sub-structure for $O(N/c^2)$ disjoint sub-matrices of size $c \times c$. This space and query time dominate the bounds of the reduction algorithm, since we need to make this data structure for $O(N/c^2)$ disjoint sub-matrices of size $c \times c$. We will make use of the following result in our solution.

Lemma 9. [4] *Given a matrix of size N, there exists a data structure supporting 2D-RMQs in the matrix in $O(1)$ time using $O(N)$ bits additional space.*

3.1 Reduction to Queries within Matrices of size $c \times c$

We show how to reduce general 2D-RMQs in a matrix of size N to 2D-RMQs in small matrices of size $c \times c$. We partition the matrix into blocks of size $1 \times c$, we build a data structure D_1 of Lemma 9 for the matrix M of size $m \times n/c$ containing the minimum element of each block, and then we delete M. The size of D_1 is $O(N/c)$ bits, and whenever its query algorithm wants to read an element from M, we read the corresponding block and find its minimum in $O(c)$ time. Similarly, we make another partitioning of the original matrix into blocks of size

$c \times 1$, and build another data structure D_2 of Lemma 9 on the minimum of the blocks. Now, we explain how to make the reduction using D_1 and D_2.

The two partitionings together divide the matrix into $O(N/c^2)$ square blocks each of size $c \times c$. If a query is contained within one of these square blocks, the reduction is done. If a query is large, it spans over several square blocks; the first partitioning divides the query into three vertical sub-queries: the middle part that consists of full $1 \times c$ blocks, and the left and right sub-queries that contain partial $1 \times c$ blocks. We find the minimum element of the middle part in $O(c)$ query time using D_1. Each of the other two parts is similarly divided into three parts by the second partitioning: the middle part which consists of full $c \times 1$ blocks, and the other two parts that contain partial $c \times 1$ blocks. The middle part can be answered using D_2 in $O(c)$ query time, and the other two parts are contained in square blocks of size $c \times c$. We sum up in the following lemma:

Lemma 10. *If, for a matrix of size $c \times c$, there exists a data structure of size S bits additional space that answers 2D-RMQs in time T, then, for a rectangular matrix of size N, we can build a data structure of size $O(N/c + N/c^2 \cdot S)$ bits additional space that supports 2D-RMQs in time $O(c + T)$.*

3.2 2D-RMQs within Matrices of size $c \times c$

We present a recursive data structure of size $O(c \log c \log\log c)$ bits additional space that supports 2D-RMQs within a $c \times c$ matrix in $O(c \log\log c)$ time. Let r denote the size of the recursive problem with an input matrix of size $r \times r$ (at the top level $r = c$). We assume that c is a power of 2. We divide the matrix into r/k mutually disjoint horizontal slabs of size $k \times r$, and r/k mutually disjoint vertical slabs of size $r \times k$. A horizontal and a vertical slab intersect in a $k \times k$ square. We choose k the power of 2 such that $k/2 < \lceil \sqrt{r} \rceil \leq k$; observe that $r/k \leq k$, $k = \Theta(\sqrt{r})$, and $k^2 = \Theta(r)$.

We recurse on each horizontal or vertical slab. A horizontal slab of size $k \times r$ is *compressed* to a $k \times k$ matrix by dividing each row into k groups of k consecutive elements, and representing each group by its minimum element. A vertical slab is also compressed to a $k \times k$ matrix similarly. The compressed matrix is not represented explicitly. Instead, if a solution to a recursive $k \times k$ problem wants to read some entry x from its input (which is a compressed $k \times k$ matrix), we scan the entire sub-array that x represents to find the location of x in the $r \times r$ matrix; the size of the sub-array will be called the *weight* of x. Note that all values in a recursive sub-problem will have the same weight denoted by w. The recursion terminates when $r = O(1)$.

A given query rectangle is decomposed into a number of disjoint 2-sided, 3-sided, and 4-sided queries (see Figure 4). In addition to the (trivial) base case that $r = O(1)$, there are three kinds of *terminal* queries which do not generate further recursive problems: small queries contained within the $k \times k$ squares that are the intersection of slabs, also called *micro queries*; 4-sided *square-aligned* queries whose horizontal and vertical boundaries are aligned with slab boundaries; and 2-sided queries. To answer micro queries, we simply scan the entire

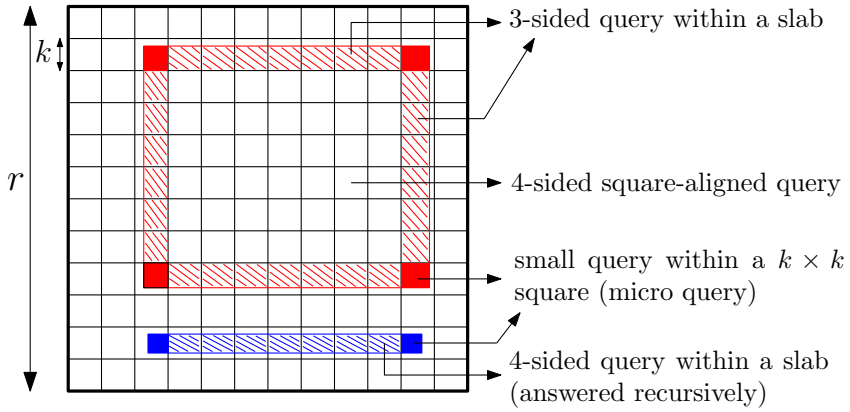

Fig. 4. The recursive decomposition. Two different queries which are decomposed in two different ways.

query range in $O(k^2 w) = O(rw)$ time without using any additional space. For each of the recursive problems, we store a data structure of Lemma 8 that answers 2-sided queries in $O(rw)$ time using $O(r \log r)$ bits additional space.

4-Sided Square-Aligned Queries. To answer 4-sided square-aligned queries in an $r \times r$ recursive problem, we divide the $r \times r$ matrix into k^2 disjoint blocks of size $k \times k$ in the preprocessing. We make a $k \times k$ *rank-matrix* out of the minimum element of the blocks, and then replace each element in the rank-matrix by its rank among all the elements in the rank-matrix. The rank-matrix is stored using $O(k^2 \log k) = O(r \log r)$ bits. A square-aligned query is first converted into a 4-sided query within the rank-matrix, which can be answered by brute force in $O(k^2) = O(r)$ time. This determines the block containing the minimum within the query range. Then we find the position of the minimum within the block in $O(k^2 w) = O(rw)$ time by scanning the block.

Query Algorithm. The recursion terminates when $r = O(1)$, or after $\log \log c - O(1)$ levels (recall that $k = \Theta(\sqrt{r})$); no data is stored with these terminal problems at the base of the recursion, and they are solved by brute-force.

A query, depending on its size and position, is decomposed in one of the following two ways (Figure 4): (1) decomposed into at most four micro queries, at most one 4-sided square-aligned query, and at most four 3-sided queries each within a slab; (2) decomposed into at most two micro queries, and at most one 4-sided query within a slab.

As previously described, micro queries and 4-sided square-aligned queries are terminals. A 4-sided query within a slab is answered recursively on the compressed matrix of the slab. A 3-sided query within a slab is converted to a 3-sided query within the compressed matrix of the slab, and that is answered using our decomposition as follows (see Figure 5). A 3-sided query within an $r \times r$ matrix is

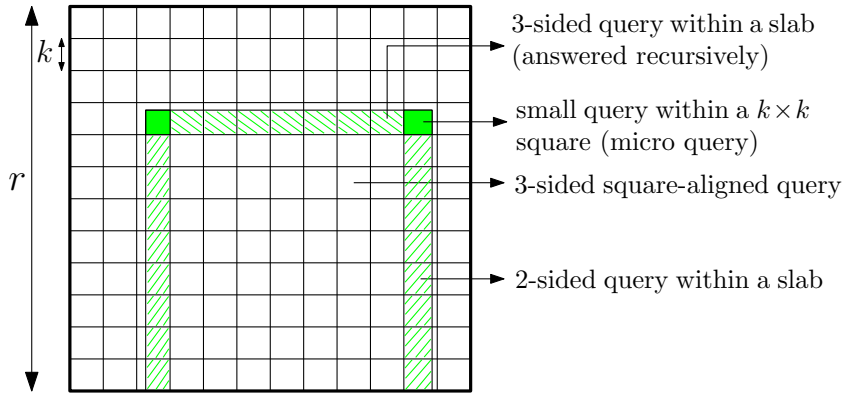

Fig. 5. The recursive decomposition of 3-sided queries.

decomposed into at most two micro queries, at most one 3-sided square-aligned query, at most two 2-sided queries each within a slab, and at most one 3-sided query within a slab. As previously described, micro queries, and 4-sided (including 3-sided) square-aligned queries are terminals. Also a 2-sided query within a slab is converted into a 2-sided query within the compressed matrix of the slab, and thus becomes a terminal query. A 3-sided query within a slab is answered recursively on the compressed matrix of the slab. In Section 2, we describe how we answer 2-sided queries in $O(rw)$ time using $O(r \log r)$ bits.

Lemma 11. *Given a $c \times c$ matrix, there exists a data structure supporting 2D-RMQs in $O(c \log\log c)$ time using $O(c \log c \log\log c)$ bits additional space.*

Proof. The query time of the algorithm for an $r \times r$ problem is given by

$$T(r, w) = O(rw) + T(k, kw) \ ,$$

where $k = \Theta(\sqrt{r})$. The recurrence is terminated with $O(w)$ cost when $r = O(1)$. Since the product rw is maintained across the recursive calls, and the depth of the recursion is $O(\log\log c)$, it follows that $T(c, 1) = O(c \log\log c)$.

The space usage of the data structure for an $r \times r$ problem is given by

$$S(r) = O(r \log r) + (2r/k) \cdot S(k) \ ,$$

where $k = \Theta(\sqrt{r})$, which solves to $S(r) = O(r \log r \log\log r)$. □

Theorem 1. *There exists a data structure of size $O(N/c)$ bits additional space that supports 2D-RMQs in a matrix of size N in $O(c \log c (\log\log c)^2)$ query time, for a parameter $c \leq N$.*

Proof. Lemmas 10 and 11 together imply a data structure of size $O(N/c' \cdot \log c' \log\log c')$ bits additional space with $O(c' \log\log c')$ query time, for a parameter $c' \leq N$. Substituting c' with $O(c \log c \log\log c))$ proves the claim. □

4 2D-RMQs in 0-1 Matrices

In this section we consider the binary case, where the RMQ query is essentially an "emptiness" query: is there a 0 value in the query rectangle? We present results for both the 4-sided and 2-sided cases.

4.1 The 4-sided case

We first present a data structure of size $O(N/c)$ bits additional space that supports 2D-RMQs in $O(c \log c)$ time within a 0-1 matrix of size N. Analogous to Section 3, the problem is reduced to 2D-RMQs in $c \times c$ matrices using Lemma 10. We show that a 2D-RMQ in a $c \times c$ matrix can be answered in $O(c)$ query time using $O(c \log c)$ bits additional space. This implies a data structure of size $O(N/c \cdot \log c)$ bits with $O(c)$ query time, which then leads to our claim after substituting c with $c \log c$.

Lemma 12. *There exists a data structure of size $O(c \log c)$ bits additional space that can answer 2D-RMQs within a $c \times c$ matrix in $O(c)$ time.*

Proof. The data structure is recursive and similar to the one in Section 3.2, with a few differences. Let $r \times r$ denote the size of the matrix for the recursive sub-problem, and $k = \Theta(\sqrt{r})$. Similar to Section 3.2, a 4-sided query, depending on its size and position, is decomposed into one of the following two ways. (1) decomposed into at most four micro queries, at most one 4-sided square-aligned query, and at most four 3-sided queries each within a slab; (2) decomposed into at most two micro queries, and at most one 4-sided query within a slab. We solve (1) in a non-recursive way, whereas we approach (2) recursively, the recursion continues until (2) changes to (1) at some level of the recursion.

(1) Similar to Section 3.2, we answer each micro sub-query by brute force (scanning the whole sub-query region). For 4-sided square-aligned queries, as opposed to using the rank-matrix (which takes $O(r \cdot \log r)$ bits), we instead make the data structure of Lemma 9 for the rank-matrix using $O(r)$ bits. This and the following 3-sided sub-structure together can solve (1) using $O(r)$ bits with $O(rw)$ query time.

We present a non-recursive structure for 3-sided queries, and thus yield a non-recursive solution for (1). The sub-structure supports 3-sided queries in $O(r)$ query time using $O(r)$ bits additional space. This sub-structure is much simpler than the recursive structure we use in Section 3.2 for 3-sided queries (the latter provides $O(r \log \log r)$ query time and $O(r \log r)$ bits space). Assume without loss of generality that the 3-sided query is open to the left. We always return the position of the left-most 0 within the query. If such an element does not exist, we return an arbitrary position within the query. In the preprocessing of a matrix, we make an array A of size r such that $A[i]$ stores the column-number of the left most 0 within the i-th row of the matrix. We build an encoding data structure of size $O(r)$ bits that supports 1D-RMQs in A in constant time, and then we delete A. To answer the 3-sided query, we first find the row j that contains the

left most 0 within the query, using the 1D-RMQ structure of A in $O(1)$ time. Now that we have the row j, we scan the row j in $O(r)$ time to find the position of the left-most zero within the query, if a 0 exists within the query; otherwise (containing 1) we return an arbitrary position within the query.

(2) A 4-sided query within a slab is reduced to a 4-sided query within the compressed slab (a $k \times k$ square) and is answered recursively. We answer each micro sub-query by brute force (scanning the whole sub-query region). We show that the total area consisting all the micro sub-queries in all the recursive levels altogether is $O(c)$ in the worst case. This implies that the query time of our data structure is $O(c)$, and the space given by $S(r) = O(r) + (2r/k) \cdot S(k)$ is $O(c \log c)$ bits.

Recall that (from Section 3.2) at each recursive level of the problem, all entries in the matrix have a weight. In the first level, the weight is $w(1) = 1$; in the second level, the weight increases to $w(2) = c^{1/2}$, and in the i-th level $w(i) = c^{1/2^{i-1}} \cdot \prod_{j=1}^{i-1} w(j)$. Notice that the weight is increasing exponentially along the levels. At each level, the query can occur within either a horizontal or a vertical slab. A query that occurs in a horizontal slab of some level is called a horizontal query and that level is called a horizontal level. Let $w_h(i)$ denote the *horizontal-weight* in the i-th horizontal level which is the weight that only increases in horizontal levels, that is, $w_h(i) = c^{1/2^{l-1}} \cdot \prod_{j=1}^{i-1} w_h(j)$, where the i-th horizontal level occurs in level l. Vertical levels, vertical queries, and vertical-weights $w_v(i)$ are defined analogously. Let ℓ_h and ℓ_v be the number of horizontal and vertical levels respectively.

We only consider the horizontal micro sub-queries, as the vertical ones are analogous. Also the horizontal micro sub-queries are either in the left or right side of the original query; we compute the total area of the horizontal micro sub-queries on the left side, as the other side is analogous. We bound the size of this area on the left side by the area of the smallest rectangle R that encloses all the horizontal micro sub-queries on the left side. We bound the width and height of R separately, and then multiply them. It is easy to see that the width of R equals the sum of the width of all the horizontal micro sub-queries, and the height of R equals the sum of the height of all the *vertical* micro sub-queries plus the height of the sub-query of type (1) that is left after the last level (see Fig.). Regarding the fact that the horizontal/vertical-weights are increasing exponentially along the levels, we show that each of these sums is dominated by the width or height in the last corresponding level.

Observe that the width of a micro sub-query in the i-th horizontal level is at most $w_h(i)$. Since the weights increase exponentially, the width of R is dominated by the width of the micro sub-query in the ℓ_h-th horizontal level, that is $O(w_h(\ell_h))$. The height of R is dominated by the height of the sub-query of type (1) arising after the last recursive level, using the same argument. If the last recursive level ℓ is a horizontal level, then the height of R is dominated by $O(c^{1/2^{\ell-1}} \cdot w_v(\ell_v))$. Otherwise, the height of R is dominated by $O(c^{1/2^{l_h-1}} \cdot w_v(l_v))$, where the last horizontal level occurs in level l_h and the l_v-th vertical level is the last vertical level before l_h. Regarding the fact that $c^{1/2^{i-1}} \cdot w(i) = O(c)$

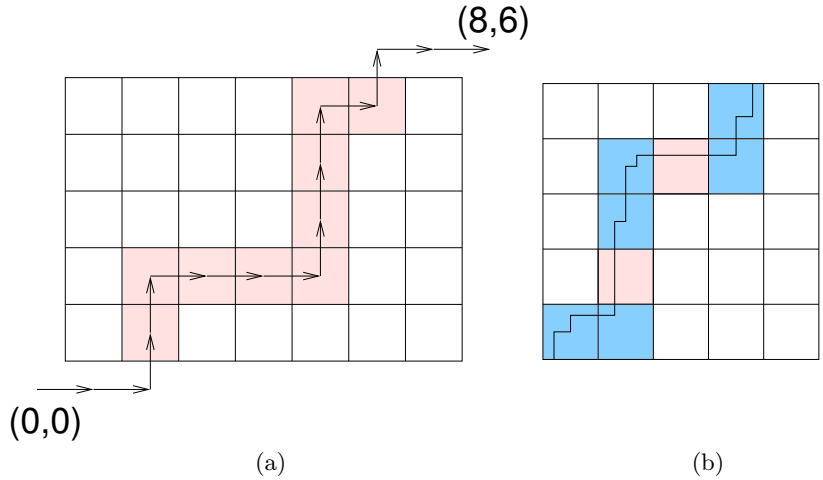

Fig. 6. (a) An example $(5, 7)$ staircase. The bit-string B corresponding to this staircase is 00110001110100. (b) The approximate version of the (n, n) staircase is shaded in: marked squares are shown as blue.

and $w_h(\ell_h) \cdot w_v(\ell_v) = w_\ell$, in both cases the product of width and height of R is dominated by $O(c)$. □

Theorem 2. *For a matrix of size N with elements from $\{0, 1\}$, there exists a data structure of size $O(N/c)$ bits additional space that supports 2D-RMQs in $O(c \log c)$ time, for a parameter $c \leq N$.*

4.2 The 2-sided case

We now consider the 2-sided case. It is helpful to describe an auxiliary data structure first. For integers $r, s > 0$, let an (r, s)-*staircase* S be a sequence of integers $(0, 0) = (x_0, y_0), (x_1, y_1), \ldots, (x_{r+s+2}, y_{r+s+2}) = (r+1, s+1)$, such that for all $i = 0, \ldots, r + s + 1$, exactly one of $x_{i+1} = x_i + 1$ or $y_{i+1} = y_i + 1$ holds. As illustrated in Fig. 6(a), S can represent an arbitrary monotone boundary dividing an $r \times s$ grid into two parts. Given a query point $(x, y) \in [1..r] \times [1..s]$, we wish to determine if (x, y) is on, above, or below S. We observe:

Proposition 1. *An (r, s)-staircase can be represented in $O(r + s)$ bits so that given $(x, y) \in [1..r] \times [1..s]$, we can determine whether (x, y) is on, below or above S in $O(1)$ time.*

Proof. We encode S as a bit-string $B = b_1 b_2 \ldots b_{r+s+2}$ where $b_i = 1$ if $x_i = x_{i-1} + 1$ and 0 otherwise. It is easy to see that if $y' = \texttt{select}_1(B, x) - x + 1$ and $y'' = \texttt{select}_1(B, x + 1) - 1$ then (x, y') and (x, y'') are the point in S with the lowest and highest second coordinate among points in S with first coordinate

equal to x. Thus (x, y) is below S if $y < y'$, on S iff $y' \leq y \leq y''$ and above S otherwise. □

We now show the following:

Theorem 3. *For an $n \times n$ matrix with elements from $\{0, 1\}$, there exists a data structure of size $O(n/c)$ bits additional space that supports 2D-RMQs in $O(c^2)$ time, for a parameter $1 \leq c \leq n$.*

Proof. We first determine the maximal 0s in the input: these determine a *boundary* such that query points above the boundary have a null answer, while points below or on the boundary can be answered by giving the location of one of the maximal 0s on the boundary.

Assuming without loss of generality that c divides n, we again divide the $n \times n$ input into $c \times c$ blocks. Shrinking each $c \times c$ block to a single entry, we call each block that contains a maximal 0 a *marked* block. The marked blocks determine an approximate boundary, which is an $(n/c, n/c)$-staircase S (see Fig 6). We represent the approximate boundary using Proposition 1. In addition to the bit-vector B that is used to represent the staircase in Proposition 1 we store a bitvector B', also of length $|S|$, whose i-th bit is 1 iff the i-th entry of S represents a marked block.

A query (x, y) is answered as follows. If (x, y) is above S, the null answer is returned. If (x, y) is on S, then let b be the $c \times c$ block containing (x, y) and let b_1 and b_2 represent the first (if any) marked blocks strictly before and after b in S; b_1 and b_2 can be found in $O(1)$ time by `rank` operations on B'. It is easy to see that by scanning b, b_1 and b_2 in $O(c^2)$ time, we can determine the intersection of the boundary with b. This allows us to either return a maximal 0 in b as an answer, determine that the query point (x, y) is above the boundary and return a null answer, or that a maximal 0 in either b_1 or b_2 is the answer. The case where (x, y) is below S is similar. □

Remark 1. As we only need to access blocks on the boundary, we can choose $c = 1$ in Theorem 3. In this case, the bit-vector B' effectively encodes the contents of the blocks on the boundary, and we can dispense with accesses to the input matrix.

References

1. 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.
2. 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.
3. P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. 11th International Symposium on Algorithms and Data Structures*, pages 98–109, 2009.

4. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.
5. B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proc. 5th Annual Symposium on Computational Geometry*, pages 131–139. ACM, 1989.
6. B. Chor, C. E. Leiserson, R. L. Rivest, and J. B. Shearer. An application of number theory to the organization of raster-graphics memory. *Journal of ACM*, 33(1):86–104, 1986.
7. P. Davoodi, R. Raman, and S. S. Rao. Succinct representations of binary trees for range minimum queries. In *Proc. 18th Annual International Conference on Computing and Combinatorics*, 2012. To appear.
8. 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.
9. A. Farzan, J. I. Munro, and R. Raman. Succinct indices for range queries with applications to orthogonal range maxima. In *Proc. 39th International Colloquium on Automata, Languages and Programming*, 2012. To appear, available as arXiv:1204.4835v1 [cs.DS].
10. A. Fiat and A. Shamir. Polymorphic arrays: A novel VLSI layout for systolic computers. *Journal of Computer and System Sciences*, 33(1):47–65, 1986.
11. A. Fiat and A. Shamir. How to find a battleship. *NETWORKS*, 19:361–371, 1989.
12. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
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. M. J. Golin, J. Iacono, D. Krizanc, R. Raman, and S. S. Rao. Encoding 2D range maximum queries. In *In Proc. 22nd International Symposium on Algorithms and Computation*, volume 7074 of *LNCS*, pages 180–189. Springer Verlag, 2011.
15. J. Matousek. *Geometric Discrepancy*. Algorithms and Combinatorics. Springer Verlag, 1999.
16. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
17. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.