Optimal Encodings and Indexes for Nearest Larger Value Problems

Seungbum Jo¹, Rajeev Raman², and Srinivasa Rao Satti¹

¹ Seoul National University, sbcho@tcs.snu.ac.kr, ssrao@cse.snu.ac.kr

² University of Leicester, r.raman@leicester.ac.uk

Abstract. Given a d -dimensional array, for any integer $d > 0$, the *nearest larger value* (NLV) query returns the position of the element which is closest, in L_1 distance, to the query position, and is larger than the element at the query position. We consider the problem of preprocessing a given array, to construct a data structure that can answer NLV queries efficiently. In the 2-D case, given an $n \times n$ array A , we give an asymptotically optimal $O(n^2)$ -bit encoding that answers NLV queries in $O(1)$ time. When A is a binary array, we describe a simpler $O(n^2)$ -bit encoding that also supports NLV queries in $O(1)$ time. Using this, we obtain an index of size $O(n^2/c)$ bits that supports NLV queries in $O(c)$ time, for any parameter c , where $1 \leq c \leq n$, matching the lower bound. For the 1-D case we consider the *nearest larger right value* (NLRV) problem where the nearest larger value to the right is sought. For an array of length n , we obtain an index that takes $O((n/c) \log c)$ bits, and supports NLRV queries in $O(c)$ time, for any parameter c , where $1 \leq c \leq n$, improving the earlier results of Fischer et al. and Jayapaul et al.

1 Introduction and Motivation

We consider cases of the following general problem. We are given a d -dimensional array A consisting of (possibly not all distinct) items from an ordered universe. After preprocessing A we are given a series of queries, each of which specifies an element of A , and our objective is to return the element of A nearest to the query element that is strictly larger than the query element. One may also restrict that the answer to the query comes from some particular sub-array (e.g. a quadrant) of A . Specifically, we consider the two queries below:

NLRV: Given a 1-D array A and an index i , returns the first larger element to i 's right, i.e., returns $\min\{j > i \mid A[j] > A[i]\}$ (and is undefined if this set is empty). The query NLLV is defined analogously to i 's left.

NLV: Given a d -dimensional array A and an index $p = (i_1, i_2, \dots, i_d)$, returns an index $q = (i'_1, i'_2, \dots, i'_d)$ such that $A[q] > A[p]$ and the distance between p and q , $\text{dist}(p, q) = |i_1 - i'_1| + |i_2 - i'_2| + \dots + |i_d - i'_d|$, is minimized (note that we use the L_1 metric). In case of many equidistant larger values, ties can be broken arbitrarily. If there is no larger value, then it is undefined.

Encoding and indexing models. We consider these problems in two different models that have been studied in the succinct data structures literature, namely the *indexing* and *encoding* models. In both these models, the data structure is created after preprocessing A . In the indexing model, the queries can be answered by probing the data structure as well as the input data, whereas in the encoding model, the query algorithm cannot access the input data.

Previous Work and Motivation. The off-line version of this problem: given A , to compute nearest larger values for *all* entries of A (the ANLV problem), has been studied previously³. In the 1-D case, Berkman et al. [3] noted that the best highly-parallel solutions to a number of tasks including answering range minimum queries, triangulating monotone polygons and matching parentheses, are obtained by reducing to the ANLV problem, and efficient parallel solutions to the ANLV problem were also given by the same authors. A number of plausible applications, and algorithms, for the ANLV problem in 2 and higher dimensions were given by Asano et al. [1], and time-space tradeoffs for the 1-D case were given by Asano and Kirkpatrick [2].

Fischer et al. [10] considered the problem of supporting NLRV and NLLV in the 1-D case, and showed how a data structure supporting these two queries is essential to a space-efficient compressed suffix tree. They also considered the problem of supporting NLRV and NLLV in the indexing model, and gave a space-time tradeoff (the precise result is given later). Fischer [9] gave a structure in the encoding model that uses $2.54n + o(n)$ bits and supports NLRV and NLLV queries in $O(1)$ time.

Jayapaul et al. [12] considered the problem of encoding and indexing NLV in the 2-D case. Below, we describe the directly relevant results from their work.

Our results. We obtain new results for encoding and/or indexing 1-D and 2-D nearest larger value queries. In all the 2-D results we assume L_1 distances.

- We show that 2-D NLV can be encoded in the asymptotically optimal $O(n^2)$ bits in the *general* case. Jayapaul et al. showed this only for the case where all elements of A are distinct. Distinctness is a strong assumption in these kinds of problems. For example, in the 1-D case with distinct values, NLRV and NLLV can both be trivially encoded by the Cartesian tree (giving a $2n - O(\log n)$ bit encoding). By contrast, if we do not assume distinctness, the optimal space is about $2.54n$ bits, and the data structure achieving this bound is also more complex [9]. Also, Asano et al. [1] remark that the ANLV problem for any dimension is “simplified considerably” if one assumes distinctness. In fact, for the general case, Jayapaul et al. were only able to give an encoding with size $\Theta(n^2 \log \log n)$ bits and $O(1)$ query time.

³ The terminology varies considerably. Berkman et al. studied the all nearest *smaller* values (ANSV) problem, which is symmetric to the ANLV problem. The previous/next smaller value (PSV/NSV) problems of Fischer et al. are symmetric to the NLLV/NLRV problems. Asano et al. and Jayapaul et al. call the NLV problem the *nearest larger neighbour* (NLN) problem: we consider the term “neighbour” to be mildly misleading, as the answer may not be a neighbour of the query element in A .

We also remark that in 1-D case, the NLRV and NLLV problems are closely connected to the *range minimum query (RMQ)* problem (see e.g. [9]), another problem of wide interest. In the 1-D case, there is no asymptotic difference between the encoding complexity of RMQ and NLRV/NLLV. The 2-D RMQ problem has received a great deal of attention lately [7, 6, 5, 4]. It is known that any 2-D RMQ encoding takes $\Omega(n^2 \log n)$ bits [7, 6]; thus, this result shows that NLV is different from RMQ in the 2-D encoding scenario.

- For the special case where A comprises 0-1 values, we provide an optimal trade-off. Specifically, given an $n \times n$ array A and any $1 \leq c \leq n$, we describe an index of size $O(n^2/c)$ bits that can answer NLV queries in $O(c)$ time.
- For indexing 1-D NLRV and NLLV, we give an index that takes $O((n/c) \log c)$ bits and answers these queries in $O(c)$ time. This improves the two previous trade-offs for this problem:
 - Fischer et al. [10] showed that, for any $1 \leq c, \ell \leq n$, one can use:

$$O\left(\frac{n}{c} \log c + \ell \frac{n \log \log n}{\log n} + \frac{n \log n}{c^\ell}\right)$$

bits of space and answer queries in $O(c\ell)$ time. As given, they are unable to go below $O(n \log \log n / \log n)$ space, and use more space than we do whenever $c = \omega(\log n)$. To attain $O((n/c) \log c)$ space for $c = O(\log n)$, observe that for $c = (\log n)^{\Omega(1)}$, one can choose $\ell = O(1)$ and obtain $O(c)$ time. For smaller values of c , the middle term in the space usage will never dominate for reasonable values of ℓ (clearly, we must always choose $c \geq 2$ and $\ell = O(\log \log n)$ in this range) and it suffices (and is optimal) to choose $\ell = O(\log_c \log n) = O(\log \log n - \log \log c)$. Thus, for any $c = O(\log n)$ their running time for space $O((n/c) \log c)$ is $O(c(\log \log n - \log \log c))$, and our solution is better for small enough c .

- Jayapaul et al. [12] gave a solution that uses $O((n/c) \log c + (n \log n)/c^2)$ bits and $O(c)$ time; this space usage equals ours for $c = \Omega(\log n / \log \log n)$ but is worse otherwise.

Our solution is a minor modification of the approach of Fischer et al.: we replace a data structure they use by a *slower* one to obtain the result.

We assume a standard word RAM model with word size $\Theta(\lg n)$, and we count space in terms of the number of bits used.

2 Indexing NLRV on 1-dimensional arrays

In this section, we give a new time-space trade-off for the indexing model for supporting NLRV queries in 1-D arrays. The approach follows closely the proof of Fischer et al. [10], which in turn adapts ideas from Jacobson's representation of balanced parentheses sequences [11], and is given in full for completeness.

We begin with some definitions. Given a string X over an alphabet Σ , define the following operations:

- $\text{rank}_\alpha(X, i)$ returns the number of occurrences of α in the first i positions of X , for any $\alpha \in \Sigma$.
- $\text{select}_\alpha(X, i)$ returns the position of the i th α in X , for any $\alpha \in \Sigma$.

Lemma 1 ([13]). *Given a string X over $\{0,1\}$ with length n , containing m 1s, it can be represented in $O(m \log(n/m))$ bits such that rank_1 and select_1 can be supported in $O(n/m)$ time.*

Lemma 2 ([10]). *There exists a data structure in the encoding model that solves NLRV queries using $2n + o(n)$ bits in $O(1)$ time.*

Theorem 1. *Given a 1-D array A of size n , there exists a data structure which supports NLRV queries in the indexing model in $O(c)$ time using $O((n/c) \lg c)$ bits for any parameter $2 \leq c \leq n$.*

Proof. Divide A into n/c blocks of size c . For any value $1 \leq i \leq n$, if i and $\text{NLRV}(i)$ are in the same block, say that i is a *near* value, otherwise say that i is a *far* value. Consider a block B and suppose that one or more of its far values have an NLRV in a block B' . Then the leftmost far value in B whose NLRV is in B' is called a *pioneer*, and its NLRV is called its *match*. It is known that there are $O(n/c)$ pioneers in A [11].

We maintain a bit-vector V in which the i -th bit is a 1 if $A[i]$ is a pioneer or a match of one, and 0 otherwise. This bit-vector has length n and weight $O(n/c)$ so by Lemma 1, we can store it in $O((n/c) \lg c)$ bits and perform rank/select queries on it in $O(c)$ time. Next, we take the sub-sequence S_P consisting of all pioneers and their matches. This subsequence is of length $O(n/c)$. We represent this sequence using Lemma 2 using $O(n/c)$ bits, to support NLRV queries in $O(1)$ time. We claim that for any pioneer in the list, its NLRV in the sequence of pioneers/matches is the same as its NLRV in the original sequence. Suppose that this claim is not true. This means there is a pioneer i_p such that $\text{NLRV}(i_p)$ is the value between i_p and the match of i_p . It cannot be the case that i_p and $\text{NLRV}(i_p)$ are in the same block, since i_p is a far value. If i_p and $\text{NLRV}(i_p)$ are in different blocks, then $\text{NLRV}(i_p)$ is the match of i_p . So the claim is true.

To answer the query $\text{NLRV}(i)$, we first check to see if the answer is in the same block as i taking $O(c)$ time. If so, we are done. Else, (assuming wlog that $A[i]$ is not a pioneer value) we find the first pioneer p_i before position i by doing rank/select on V . As $A[i] < A[p_i]$, $\text{NLRV}(i)$ is less than or equal to the match of p_i . Since i is the far value in this case, $\text{NLRV}(i)$ and $\text{NLRV}(p_i)$ are in the same block. We find $\text{NLRV}(p_i)$ using the NLRV encoding of S_P and find the corresponding position i_{ap} in A using rank/select on V . Finally we scan left from i_{ap} to find $\text{NLRV}(i)$. The overall time taken to answer the query is $O(c)$. $\square$

3 NLV on 2-D binary arrays

In this section, we first give an optimal encoding for NLV, and using this obtain an almost optimal trade-off for an NLV index for a 2-D binary array. We use the following lemma:

Lemma 3 ([8]). *Given a string X of length n over an alphabet Σ , $|\Sigma| = O(1)$, there is an encoding of X using $O(n)$ bits, that supports rank_α and select_α in $O(1)$ time, for any $\alpha \in \Sigma$.*

Theorem 2. *There is an encoding for an $n \times n$ binary array A which takes $O(n^2)$ bits and supports NLV queries in $O(1)$ time.*

Proof. We compute the NLV by computing the nearest larger value in all four quadrants induced by a vertical and a horizontal line that pass through the query position, and then returning the closest of these four positions to the query. Thus, it is enough to describe a structure that supports NLV in the upper-right quadrant. Given a query position $p = (i, j)$, let $q = (i', j')$ be its NLV in the upper-right quadrant. There are four possibilities: (1) $A[p] = 1$ (q is not defined); (2) $i = i'$; (3) $j = j'$; or (4) $i < i'$ and $j < j'$. The encoding will simply store, for each position, which case it belongs to. Then, in Case (2), we can find its answer by following the positions $(i, j + k)$, for $k = 1, 2, \dots$ (i.e., elements in the same row) till we reach a position that belongs to Case (1). Also, one can easily show that all the intermediate elements also belong to Case (2). Analogously, in Case (3), we follow the positions in the same column until we reach a position that belongs to Case (1). Finally, in Case (4), we first follow the positions $(i + k, j + k)$, for $k = 1, 2, \dots$ till we reach the first position $(i + \ell, j + \ell)$ that does not belong to Case (4), and then find the answer using the algorithm for Case (2) or (3), or return the position $(i + \ell, j + \ell)$ if it belongs to Case (1). To support the queries faster, we build rank/select structures (Lemma 3) for the encoding of each row, each column and each diagonal. The total space usage is clearly $O(n^2)$ bits. Now, queries can be supported in constant time by using rank/select to jump to the appropriate positions as described in the above procedures. $\square$

Now we describe an index for a given 2-D binary array. We begin by introducing some notation that will be used later. Suppose we divide an $n \times n$ array A into blocks of size $c \times c$, for $0 < c \leq n$, and divide each block into c sub-blocks of size $\sqrt{c} \times \sqrt{c}$. We define an (i, j) -block as the sub-array $A[(i-1)c + 1 \dots ic][(j-1)c + 1 \dots jc]$ and an (i, j, k, l) -sub-block as the sub-array $A[(i-1)c + (k-1)\sqrt{c} \dots (i-1)c + k\sqrt{c}][(j-1)c + (l-1)\sqrt{c} \dots (j-1)c + l\sqrt{c}]$. For each (i, j) -block, we define eight regions, consisting of sets of blocks (some of which can be empty) as follows: the region

$N(i, j)$ consists of all (i, l) -blocks with $l > j$;

$S(i, j)$ consists of all (i, l) -blocks with $l < j$;

$E(i, j)$ consists of all (k, j) -blocks with $k > i$;

$W(i, j)$ consists of all (k, j) -blocks with $k < i$;

$NE(i, j)$ consists of all (k, l) -blocks with $k > i$ and $l > j$;

$SE(i, j)$ consists of all (k, l) -blocks with $k > i$ and $l < j$;

$NW(i, j)$ consists of all (k, l) -blocks with $k < i$ and $l > j$; and

$SW(i, j)$ consists of all (k, l) -blocks with $k < i$ and $l < j$.

Similarly, for each (i, j, k, l) -sub-block, we also define the regions $N_{i,j}(k, l)$, $S_{i,j}(k, l)$, $E_{i,j}(k, l)$, $W_{i,j}(k, l)$, $NE_{i,j}(k, l)$, $NW_{i,j}(k, l)$, $SE_{i,j}(k, l)$ and $SW_{i,j}(k, l)$.

We construct an $n/c \times n/c$ array $A'[1 \dots n/c][1 \dots n/c]$ such that $A'[i][j] = 1$ if there exists at least a single **1** in the (i, j) -block, and **0** otherwise. We also construct another $n/\sqrt{c} \times n/\sqrt{c}$ array $A''[1 \dots n/\sqrt{c}][1 \dots n/\sqrt{c}]$ such that $A''[i][j] = 1$ if there exists at least a single **1** in the $(\lfloor i/\sqrt{c} \rfloor, \lfloor j/\sqrt{c} \rfloor, i \bmod \sqrt{c}, j \bmod \sqrt{c})$ -sub-block, and **0** otherwise.

Theorem 3. *Given an $n \times n$ binary array $A[1 \dots n][1 \dots n]$ one can construct an index of size $O(n^2/c)$ bits to support NLV queries in $O(c)$ time for $0 < c \leq n$.*

Proof. We divide the array A into blocks and sub-blocks as mentioned earlier. Suppose the query q is in the (i, j, k, l) -sub-block. If $A''[ic+k, jc+l] = 1$, scanning $O(1)$ sub-blocks is enough to find the NLV of q , and this takes $O(c)$ time.

Now, consider the case when $A''[ic+k, jc+l] = 0$ but $A'[i, j] = 1$. In this case, it is clear that we can identify $O(c)$ sub-blocks in which the answer may lie – namely all the sub-blocks in its block, and in the eight neighbouring blocks. We find the potential answer in each of the eight directions (E, W, N, S, NE, NW, SE, and SW), and then compare their positions to find the actual answer. To find the the answer in E direction, we scan the bits in A'' that are to the right of the current sub-block, till we find a **1**. We then scan this sub-block, and the sub-block to its right to find the potential answer in this direction. Similarly, we can find the potential answers in the W, S, and N directions. Next, we find the nearest **1** to the query in the $NE_{i,j}(k, l)$ region. This element is the nearest **1** from the bottom-left conner of $(i, j, k+1, l+1)$ -sub-block. The nearest **1** from the bottom-left conner of (a, b, c, d) -sub-block in the $NE_{a,b}(c, d)$ region is same as either the nearest **1** in the same block, or is the nearest **1** from the bottom-left corners of one of these three blocks: (1) $(a, b, c+1, d)$ -sub-block, (2) $(a, b, c, d+1)$ -sub-block, or (3) $(a, b, c+1, d+1)$ -sub-block. Therefore we encode each sub-blocks using 2 bits indicating the case it belongs to ((1), (2) or (3)), which takes a total of $O(n^2/c)$ bits. Now, to find the answer in the NE direction, we scan $O(c)$ sub-blocks to find the sub-block which contains the nearest **1** from q in $NE(i, j, k, l)$. Once we find the corresponding sub-block, finding the nearest **1** from the bottom-left conner in the sub-block takes $O(c)$ time. We can find the nearest **1** in the $NW_{ij}(k, l)$, $SE_{ij}(k, l)$ and $SW_{ij}(k, l)$ regions in the same way. Then NLV of q is the closest one among these eight candidates.

Finally, consider the case when $A'[i, j] = 0$. By storing the encoding of Theorem 2 for the array A' using $O(n^2/c^2)$ bits, we can find the nearest block to the query position which contains a **1**, in $O(1)$ time. Let this block be the (i', j') -block, let ℓ be the L_1 distance from (i, j) to (i', j') in A' . The value ℓc is an estimate (within an additive factor of $2c$) for the L_1 distance from q to its NLV. Assume, wlog, that (i', j') is in the $NE(i, j)$ region. We first describe how to find the nearest **1** in $NE(i, j)$ region. Define $d(i, j)$ as the set of blocks in the top-left to the bottom-right diagonal that contains the (i, j) -block and define the array $D_{(i,j)}$ of size at most n/c such that $D_{(i,j)}[m]$ is the distance from the bottom-left element to the nearest **1** in the m -th block in $d(i, j)$. Now we construct a linear-bit RMQ (range minimum query) data structure for each $D_{(i,j)}$ (using a total of $O(n^2/c)$ bits), so that RMQ queries can be supported in $O(1)$

Fig. 1. Suppose the nearest block that contains **1** from (2,2)-block is (4,3)-block, Then $d(4, 3)$ are the blocks colored by green and we can find the nearest **1** in $NE(2,2)$ using RMQ for $D_{4,3}[2, 3]$ and $D_{4,4}[1 \dots 3]$

time. Now, we find the two potential blocks in $NE(i, j)$ region that may have the nearest **1** from q by performing RMQs on $D_{(i', j')}$ and $D_{(i', j'+1)}$ among all the blocks that are contained in the $NE(i, j)$ region (it is easy to see that they form a consecutive range). We then choose the closer one between these two from the q . (Figure 1 shows the example). Note that if (i', j') is in a different region from $NE(i, j)$, then we may not find any potential answer in $NE(i, j)$, as all the ‘relevant’ blocks in $D_{(i', j')}$ and $D_{(i', j'+1)}$ may be empty. We can find the nearest **1** in $NW(i, j)$, $SE(i, j)$ and $SW(i, j)$ in a similar way.

Next, we describe how to find the nearest **1** in the $N(i, j)$ region (analogous for $S(i, j)$, $E(i, j)$ and $W(i, j)$ regions). For each position in the bottom row of an (a, b) -block with $A'[a, b] = \mathbf{1}$, we store two bits indicating whether its answer within the block is in (1) the same column (H), or (2) some column to the left (L), or (3) some column to the right (R). (The query algorithm simply “follows” the L or R “pointers” till it reaches a H , and then scans the column upwards till it finds a **1** in that column. Note that L and R cannot be in two adjacent columns.) This takes $O(c \times n^2/c^2) = O(n^2/c)$ bits over all the blocks. This encoding enables us to find the closest **1** within the block from any column in the bottom row of that block in $O(c)$ time. Since ℓ is the L_1 distance between (i, j) and (i', j') in A' , we know that all the blocks $A[i, j - r]$, for $1 \leq r < \ell$ are empty (otherwise, we have a closer non-empty block than (i', j')). Let k be the column corresponding to the query position q . We claim that the closest **1** to q in the $N(i, j)$ region is closest **1** to the bottom row and column k of the either $(i, j + \ell)$ -block or $(i, j + \ell + 1)$ -block. These can be computed in $O(c)$ time using the above encoding, and then compared to find the required answer. Finally we can find NLV of q by comparing these eight candidate answers. $\square$

4 Encoding of NLV on 2-D arrays

In this section, we give an encoding which supports NLV queries in 2-D array with $O(n^2)$ bits. We consider the 1-D array case first. Jayapaul et al. [12] showed how to encode an array A with n distinct items using $O(n)$ bits to answer NLV queries. We give an alternate proof of this, based on ideas from [2, 12]:

Lemma 4. *There exists an encoding of an array $A[1 \dots n]$ that uses $O(n)$ bits while supporting NLV queries, provided all elements are distinct.*

Proof. We write down the sequence $d(1), d(2), \dots, d(n)$ explicitly, where $d(i) = |i - \text{NLV}(i)|$, for $1 \leq i \leq n$, together with a sequence of n bits that indicate if $i < \text{NLV}(i)$ or $i > \text{NLV}(i)$. Because for $k > 0$, the elements in A are distinct, there are $\geq n/2^k$ elements for which $d(i) \leq 2^k$, and $d(i)$ for these elements can be encoded in $O(k)$ bits. In all, $\sum_{k=1}^{\lg n} (n/2^k \cdot O(k)) = O(n)$ bits are used. $\square$

If the elements in A are not distinct, the above argument does not hold. So instead of encoding the NLV of a position i explicitly as in Lemma 4, we encode the distance between i and the nearest value which is $\geq A[i]$ in the same direction as $\text{NLV}(i)$. Formally, we define $d_l(i) = i - (\max_{j < i, A[j] \geq A[i]} j)$ and $d_r(i) = (\min_{j > i, A[j] \geq A[i]} j) - i$ and $d(i) = d_l(i)$ if $\text{NLV}(i) < i$ and $d(i) = d_r(i)$ otherwise. For each i , we encode $d(i)$ and store a bit stating whether $d(i) = d_r(i)$ or $d(i) = d_l(i)$, and view this as a “pointer” to $j = i + d_r(i)$ or $j = i - d_l(i)$ respectively. Finally, we also store a bit indicating whether or not $A[i] = A[j]$. With this encoding, $\text{NLV}(i)$ can be easily found by following the $d(\cdot)$ “pointers” from i until we reach a position that is greater than $A[i]$.

The following lemma says that this encoding still uses $O(n)$ bits:

Lemma 5. *For any array $A[1 \dots n]$, $\sum_{i=1}^n \lg d(i) = O(n)$.*

Proof. Consider the array $A'[1 \dots n]$ of size n , where $A'[i] = A[i] + \epsilon i$ if $\text{NLV}(i) > i$ and $A'[i] = A[i] - \epsilon i$ if $\text{NLV}(i) < i$ for some $\epsilon > 0$. If we set ϵ small enough then if $A[i] > A[j]$ for some i, j then $A'[i] > A'[j]$ as well, but all elements in A' are distinct. So if we define $d'(i)$ and NLV' on A' analogously to $d(i)$ and NLV on A , $D' = \sum_{i=1}^n \lg d'(i) = O(n)$ by Lemma 4. We now show that $D = \sum_{i=1}^n \lg d(i) \leq 2D'$.

To prove this claim, let $0 \leq i_0 < \dots < i_r \leq n$ with $r > 0$ be a maximal sequence of indices such that $A[i_0] > A[i_1]$, $A[i_{r-1}] < A[i_r]$, $A[i_1] = A[i_2] = \dots = A[i_{r-1}]$, and $A[j] < A[i_1]$ for all $i_0 < j < i_r$ and $j \notin \{i_k | 1 \leq k \leq r-1\}$. For $0 \leq k \leq r$, let i_k be the index such that $\text{NLV}(i_l) = i_0$ for all $0 < l \leq k$ and $\text{NLV}(i_l) = i_r$ for all $k < l \leq r-1$. Then by the definition of A' , for all $k < l \leq r-1$, $\text{NLV}'(i_l) = i_{l+1}$ so $d(i_l) = d'(i_l)$. For the elements to the left of i_k , we can consider the case that there exist $0 < m \leq k$ such that $\text{NLV}'(i_l) = i_{l-1}$ for all $0 < l \leq m-1$ and $\text{NLV}'(i_l) = i_{l+1}$ for $m \leq l \leq k$. Then:

$$\begin{aligned} D - D' &= \sum_{i=1}^n \lg d(i) - \sum_{i=1}^n \lg d'(i) \\ &= \left(\sum_{i=1}^{m-1} \lg d(i) + \sum_{i=k+1}^{r-1} \lg d(i) + \sum_{j=m}^k \lg(i_j - i_{j-1}) \right) \\ &\quad - \left(\sum_{i=1}^{m-1} \lg d'(i) + \sum_{i=k+1}^{r-1} \lg d'(i) + \sum_{j=m}^k \lg(i_{k+1} - i_j) \right) \\ &= \sum_{j=m}^k \lg(i_j - i_{j-1}) - \sum_{j=m}^k \lg(i_{k+1} - i_j) \end{aligned}$$

$$\begin{aligned}
&\leq \lg(i_m - i_{m-1}) - \lg(i_{k+1} - i_k) \quad (\because i_j - j_{j-1} \leq i_{k+1} - i_{j-1} \text{ for all } 0 \leq j \leq k) \\
&\leq \lg(i_m - i_{m-1}) \leq \lg(i_m - i_0) \leq \lg(i_r - i_m) \quad (\because \text{NLV}(i_m) = i_0) \\
&\leq \lg(i_{k+1} - i_m) + \sum_{j=k+1}^{r-1} \lg(i_{j+1} - i_j) \quad (\text{by the concavity of } \lg \text{ function}) \\
&\leq \sum_{i=1}^n \lg d'(i) = D' \quad \square
\end{aligned}$$

We now extend this encoding to encode NLVs for a 2-D array $A[1 \dots n][1 \dots n]$ of size n^2 . In our encoding, each (i, j) “points to” another location (i', j') , such that $A[i', j'] \geq A[i, j]$, as follows: $|i - i'|$ is encoded using $O(1 + \lg |i' - i|)$ (the *row cost* of the pointer) and $|j - j'|$ is encoded using $O(1 + \lg |j' - j|)$ bits (the *column cost* of the pointer), the direction from (i, j) to (i', j') is given using two bits, and finally one extra bit indicates whether or not $A[i', j'] > A[i, j]$. Now we explain how to specify the pointers. Pick an element $A[i, j]$ and wlog assume that $\text{NLV}(i, j) = (i^*, j^*)$ with $i^* \geq i, j^* \geq j$. We choose pointers as follows:

Case (1) Let $i' > i$ be the smallest value such that $i' \leq i^*$ and $A[i, j] = A[i', j]$. If i' exists, we store a pointer to (i', j) and set the extra bit to 0.

Case (2) If not, let $j' > j$ be the smallest value such that $j' \leq j^*$ and $A[i, j] = A[i, j']$. If j' exists, we store a pointer to (i, j') and set the extra bit to 0.

Case (3) Otherwise we store a pointer to (i^*, j^*) and set the extra bit to 1.

We call this encoding scheme encoding_{2D} . To obtain $\text{NLV}(i, j)$, we follow pointers starting from (i, j) until we follow one with the extra bit set to 1, and return the position pointed to by this pointer. The correctness of this procedure can be proved by induction on k ; we omit the details due to lack of space.

Theorem 4. *There exists an encoding of 2-D array $A[1 \dots n][1 \dots n]$ that uses $O(n^2)$ bits while supporting NLV queries.*

Proof. We describe an encoding, called encoding_{grid} as follows. We first encode each column and row of A using Lemma 5, using $O(n^2)$ bits. These pointers are called *grid pointers*. However, the maximal values in each row and column do not have pointers by Lemma 5, as their NLV is not defined. So, in addition, for each row r which has (locally) maximum values in columns $i_1 < \dots < i_k$, we store *extra* pointers in both directions from (r, i_j) to (r, i_{j+1}) for $j = 0, \dots, k$, taking $i_0 = 0$ and $i_{k+1} = n + 1$. The space taken by these extra pointers is $O(\lg i_1 + \sum_{j=2}^{k-1} \lg(i_j - i_{j-1}) + \lg(n + 1 - i_k)) = O(n)$ bits for row r . We do this for all rows and columns, at a cost of $O(n^2)$ bits overall.

Although encoding_{grid} does not encode NLV, we use it to upper bound the space used by encoding_{2D} . Let a *grid pointer* and a *2D pointer* refer to a pointer in encoding_{grid} and encoding_{2D} respectively. For any 2D pointer, the cost of encoding it can be upper-bounded by the cost of encoding (one or more) grid pointers. Each grid pointer will be used $O(1)$ times this way. Below, we show how to upper bound all Case 2) 2D pointers and the row cost of all Case 3) 2D pointers by grid pointers in rows, using each grid pointer at most thrice. The costs of Case (1) 2D pointers and the column cost of Case (3) 2D pointers can similarly be bounded by the costs of grid pointers in the columns. This will prove the theorem.

We consider a fixed location (i, j) , and assume wlog that $\text{NLV}(i, j) = (i^*, j^*)$ with $i^* \geq i$ and $j^* > j$ (if $j^* = j$ then the pointer from (i, j) will have row distance 0 and there is nothing to bound). There are four cases to consider.

Fig. 2. Pointers in encoding_{2D} and encoding_{grid}

Case (a) Let $j' > j$ be the minimum index such that $A[i, j'] \geq A[i, j]$. Suppose that j' exists and there is a grid pointer from (i, j) to (i, j') or vice versa. There are two sub-cases:

- (a.1) The 2D pointer from (i, j) points to (i, j') . We use the cost of the grid pointer to upper bound the cost of this 2D pointer. Observe that if there is a 2D pointer from (i, j) to (i, j') , there cannot be a 2D pointer from (i, j') to (i, j) , so the grid pointer is used for upper-bounding only once in this case (Case (a,1) in Figure 2).
- (a.2) The 2D pointer from (i, j) points to (i^*, j^*) . Observe that $j' \geq j^*$, since otherwise either (i, j') is a larger value that is closer than (i^*, j^*) , a contradiction, or we would have a Case (2) 2D pointer from (i, j) to (i, j') . The grid pointer between (i, j) and (i, j') will only be used twice for upper-bounding in this case (Case (a,2) in Figure 2).

Case (b) There is either no value $A[i, j'] \geq A[i, j]$ for $j' > j$, or if there is, then there are no grid pointers either from (i, j) to (i, j') or vice versa. As before, we consider two sub-cases.

- (b.1) First suppose that the 2D pointer from (i, j) points to (i, j') , where $j' > j$ is the smallest index such that $A[i, j'] \geq A[i, j]$, and there is no grid pointers between (i, j) and (i, j') in either direction.

If $A[i, j]$ is a maximal value in row i , the cost of the pointer is upper-bounded by the extra pointer between (i, j) and (i, j') . If not, the absence of grid pointers between (i, j) and (i, j') implies that the NLV of (i, j) in the i -th row is (i, j_0) for some $j_0 < j$. Note that $|j_0 - j| \geq \text{dist}((i, j), (i^*, j^*))$, otherwise $\text{NLV}(i, j)$ would be (i, j_0) . As $\text{dist}((i, j), (i^*, j^*)) = |j - j'| + |j' - j^*| + |i^* - i|$, $|j_0 - j| \geq |j' - j|$. The path p between (i, j) and (i, j_0) in encoding_{grid} may

comprise a number of grid edges. We can bound the cost of the 2D edge from (i, j) to (i, j') by the total cost of the grid edges on the path p (since the log function is concave, the sum of the costs of the path p is no less than the cost of a single edge from (i, j) to (i, j_0)). Let p comprise the elements $j = j_l, j_{l-1}, \dots, j_1, j_0$ (omitting the row number for brevity). Note that for any $0 < k < l$, no 2D pointer from (i, j_k) can end up in Case (b), so this path can only be used twice to upper-bound the cost of a 2D edge: once from (i, j) and once (possibly) from (i, j_0) (Case (b,1) in Figure 2).

- (b.2)** The 2D pointer from (i, j) points to (i^*, j^*) , and either there is no value $\geq A[i, j]$ in locations $A[i, j']$ for $j' > j$, or if there is, and j' is the minimum such value, then there is no grid pointer between (i, j) and (i, j') . If $A[i, j]$ is a maximal value in row i , then if j' exists, then it must be that $j' > j^*$, and the row cost of the 2D pointer is bounded by the extra pointer between (i, j) and (i, j') . On the other hand, if j' does not exist, then the row cost of the 2D pointer is bounded by the extra pointer from (i, j) to $(i, n+1)$. If $A[i, j]$ is not maximal, then arguing as above, we see that the NLV of (i, j) in the i -th row is (i, j_0) for some $j_0 < j$, that $|j_0 - j| \geq |j - j^*|$, and so we can upper-bound the row cost of this 2D pointer by the total cost of all the grid pointers between j and j_0 , and each of these grid pointers is used at most twice (once each for the pointers out of (i, j) and (i, j_0) in Case (b) to upper bound a 2D pointer (Case (b,2) in Figure 2). $\square$

Now we describe the $O(n^2)$ -space data structure that supports NLV query in constant time on 2-D array $A[1 \dots n][1 \dots n]$. To support this, first divide A into blocks of size $b \times b$ and divide each block into sub-blocks of size $s \times s$. By following lemma, we can bound the number of distinct NLVs for all the maximal elements in the block.

Lemma 6 ([12]). *Given a block of size $k \times k$, the maximal elements in the block have at most $O(k)$ distinct NLVs.*

We divide the area outside each blocks (sub-blocks) into 8 regions (N, S, W, E, NW, NE, SE, SW) as defined in Section 3. Now we prove the main theorem.

Theorem 5. *There exists an encoding of 2-D array $A[1 \dots n][1 \dots n]$ that uses $O(n^2)$ bits while supporting NLV queries in $O(1)$ time.*

Proof. For each element $A[i, j]$, we assign a color from the set $\{C_1, C_2, \dots, C_9\}$, as follows. If $\text{NLV}(i, j)$ is in one of the 8 regions, we give one of the colors from C_1 to C_8 ; and if the answer is within the block containing (i, j) , then we give the color C_9 . Also for block B , let $B_{\max}$ be the set of NLV of the maximal elements in B . Then for each boundary element e_B in block B , we store a pointer to an element in $B_{\max}$ which is closest to e_B . These structures take $O(b^2 + b \lg n)$ bits, for each block by Lemma 6. For each sub-block, we maintain similar structure as above using $O(s^2 + s \lg b)$ bits. (Note that a maximal element in a sub-block which is not a maximal element in its block can have its answer outside the block, but its distance to NLV is bounded by $O(b)$. So, we can store the answer explicitly.)

Also for each element, we assign the color c_1 to c_9 by the position of its NLV analogous to C_1 to C_9 . To support NLV queries for non-maximal elements in a sub-block, we encode each sub-block together with its 8 neighboring sub-blocks, using the encoding of Theorem 4, using $O(s^2)$ bits. In addition, we construct a precomputed table that is indexed by the above $O(s^2)$ -bit encoding of a sub-block and a position within it, and returns the NLV for that position.

We now describe the query algorithm. Consider the query $q = A[i, j]$ and let B_q (b_q) be the block (sub-block) that contains q . We first check whether q is a maximal element in b_q . If q is not a maximal element in b_q , we use the precomputed table to find the answer, in $O(1)$ time. Otherwise, if q is not a maximal element in B_q , then $\text{NLV}(q)$ can be answered in $O(1)$ time by finding the nearest boundary in the direction corresponding to q 's assigned color. If q is a maximal element in B_q , we can find its NLV in $O(1)$ time by a similar procedure using colors C_1 to C_9 . So total query time is $O(1)$, and total space is $O(n^2/b^2 \times (b^2 + b \lg n) + n^2/s^2 \times (s^2 + s \lg b) + 2^{O(s^2)})$ bits. If we set $b = \lg n$ and $s = c\sqrt{\lg n}$, we can encode A with supporting NLV queries in $O(n^2)$ bits. $\square$

References

1. Asano, T., Bereg, S., Kirkpatrick, D.G.: Finding nearest larger neighbors. In: Efficient Algorithms. pp. 249–260 (2009)
2. Asano, T., Kirkpatrick, D.G.: Time-space tradeoffs for all-nearest-larger-neighbors problems. In: WADS. pp. 61–72 (2013)
3. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms 14(3), 344–370 (1993)
4. Brodal, G.S., Brodnik, A., Davoodi, P.: The encoding complexity of two dimensional range minimum data structures. In: ESA. pp. 229–240 (2013)
5. Brodal, G.S., Davoodi, P., Lewenstein, M., Raman, R., Rao, S.S.: Two dimensional range minimum queries and Fibonacci lattices. In: ESA. pp. 217–228 (2012)
6. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica 63(4), 815–830 (2012)
7. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica 68(3), 610–625 (2014)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
9. Fischer, J.: Combined data structure for previous- and next-smaller-values. Theor. Comput. Sci. 412(22), 2451–2456 (2011)
10. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comput. Sci. 410(51), 5354–5364 (2009)
11. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. pp. 549–554. IEEE Computer Society (1989)
12. Jayapaul, V., Jo, S., Raman, V., Satti, S.R.: Space efficient data structures for nearest larger neighbor. In: Proc. IWOCA 2014 (2014), to appear.
13. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: ALENEX (2007)