Available online at www.sciencedirect.com
ScienceDirect
Journal of Discrete Algorithms 00 (2015) 1–14
Journal Logo

Space Efficient Data Structures for Nearest Larger Neighbor☆

Varunkumar Jayapaul[a], Seungbum Jo[b], Venkatesh Raman[c], Rajeev Raman[d], Srinivasa Rao Satti[b]

[a]*Chennai Mathematical Institute, India*
[b]*Seoul National University, South Korea*
[c]*The Institute of Mathematical Sciences, India*
[d]*University of Leicester, United Kingdom*

Abstract

Given a sequence of n elements from a totally ordered set, and a position in the sequence, the nearest larger neighbor (NLN) query returns the position of the element which is closest to the query position, and is larger than the element at the query position. The problem of finding all nearest larger neighbors has attracted interest due to its applications for parenthesis matching and in computational geometry [1, 2, 3]. We consider a data structure version of this problem, which is to preprocess a given sequence of elements to construct a data structure that can answer NLN queries efficiently. We consider time-space tradeoffs for the problem in both the encoding (where the input is not accessible after the data structure has been created) and indexing model, and consider cases when the input is in a one dimensional array, and also initiate the study of this problem on two-dimensional arrays.

Keywords: Nearest larger neighbor; succinct data structures; indexing model; encoding model; Cartesian tree; range minimum query.

1. Introduction and Motivation

Given a sequence of n elements from a totally ordered set, and a position in the sequence, the *nearest largest neighbor* (NLN) query asks for the position of an element which is closest to the query position, and is larger than the element at the query position. More formally, given an array $A[1 \ldots n]$ of length n containing elements from a totally ordered set, and a position i in A, we define the query:
$NLN(i)$: return the index j such that $A[j] > A[i]$ and $|i - j| = \min\{k : A[i+k] > A[i]$ *or* $A[i-k] > A[i]$ for $k > 0\}$. Ties are broken to the left, and if there is no element greater than the query element, the query returns the answer ∞.

In a similar way, we define NLRN (*right* nearest larger neighbor), and NLLN (*left* nearest larger neighbor) queries, which return the position of the nearest larger neighbor to the right and left, respectively, of the query position. In a symmetric way, one can also define nearest *smaller* neighbor problems. In this paper, we will stick to the version that seeks the larger neighbors.

☆Preliminary version of these results have appeared in the proceedings of the 25th International Workshop on Combinatorial Algorithms (IWOCA-2014) [4] and in the proceedings of International Workshop on Algorithms and Computation (WALCOM-2015) [5]

Email addresses: varunkumarj@cmi.ac.in (Varunkumar Jayapaul), sbcho@tcs.snu.ac.kr (Seungbum Jo), vraman@imsc.res.in (Venkatesh Raman), r.raman@leicester.ac.uk (Rajeev Raman), ssrao@cse.snu.ac.kr (Srinivasa Rao Satti)

2-dimensional NLN. We also consider a natural extension of the NLN problem to two-dimensional arrays. Here, we define the NLN of a query position as the closest position in the array, in terms of the L_1 distance, that contains an element larger than the element at the query position. More formally, given a position (i, j) in $A[1 \ldots n][1 \ldots n]$, $NLN((i, j)) = (i', j')$ such that $A[i', j'] > A[i, j]$, and $|i - i'| + |j - j'| = \min\{(|x| + |y| : A[i + x, j + y] > A[i, j]\}$. If there is no element greater than the query element, the query returns the answer (∞, ∞).

Encoding and indexing models. We consider the data structure versions of 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 the input data. 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. The size of the data structure in the encoding model is also referred to as the *effective entropy* [6] (of the input data, with respect to the problem).

Previous Work and Motivation. The problem of computing all (right) nearest larger neighbors has attracted much attention due to its importance as a preprocessing routine for answering range minimum queries, triangulation algorithms, reconstructing a binary tree from its traversal orders and matching a sequence of balanced parentheses [3]. While Berkman et al. [3] have given efficient parallel algorithms for the problem, recently Asano et al. [1] have given sequential time-space tradeoff results.

Fischer et al. [7] considered the problem of supporting NLRN and NLLN, and showed how a data structure supporting these two queries can be used in obtaining *entropy-bounded compressed suffix tree* representation. (They considered the min version of the problem instead of max, and named the operations NSV and PSV, for the next and previous smaller values, respectively.) They considered the problem of supporting NLRN and NLLN in the indexing model, and obtained the following time-space tradeoff result. For any $1 \leq c, \ell \leq n$, one can use:

$$O\left(\frac{n}{c}\lg c + \ell\frac{n\lg\lg n}{\lg n} + \frac{n\lg n}{c^\ell}\right)^1$$

bits of space and answer queries in $O(c\ell)$ time. As given, they are unable to go below $O(n \lg\lg n/\lg n)$ space, and use more space than we do whenever $c = \omega(\lg n)$. As mentioned later, we improved the trade-off to $O((n/c)\lg c)$ bits with $O(c)$ time. To attain $O((n/c)\lg c)$ space for $c = (\lg n)^{\Omega(1)}$, 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 $\ell = O(\lg\lg n)$ in this range) and it suffices (and is optimal) to choose $\ell = O(\lg_c \lg n) = O(\lg\lg n - \lg\lg c)$. Thus, for any $c = O(\lg n)$, their running time for space $O((n/c)\lg c)$ is $O(c(\lg\lg n - \lg\lg c))$, and our solution is better for small enough c.

Fischer et al. [7] also gave an encoding that supports the PSV and NSV queries in constant time, using $4n + o(n)$ bits. The encoding size was later reduced to the optimal $2.54n + o(n)$ bits by Fischer [8]. We observe that this can be further improved to $2n + o(n)$ bits if all the elements are distinct.

For the case of binary sequences, the data structure version of the NLN problem can be solved by building an auxiliary structure to support *rank* and *select* queries on the bit vector [9]. This uses $o(n)$ bits of extra space, in addition to the input array, and answers NLN (and also NLRN and NLLN) queries in $O(1)$ time.

Given a two-dimensional array, Asano et al. [1] considered the *All Nearest Larger Neighbours* problem which asks for computing the NLN values for all the elements in the input array. They showed that this problem can be solved in $O(n^2 \lg n)$ time (and more generally, for any d-dimensional array in $O(n^d \lg n)$ time). To the best of our knowledge, the data structure version of the two-dimensional NLN problem, in which we are interested in constructing a data structure that answers online queries efficiently, has not been considered earlier.

Our results. For the case of one dimension, we first observe in Section 2.1, a $2n - o(n)$-bit lower bound for NLRN in the encoding model, by relating the problems to the well studied range minimum queries (RMQ). We also give an independent lower bound proof by directly counting the number of distinct configurations, which maybe of independent interest. In terms of upper bounds in the encoding model, a $2n + o(n)$ bits to answer NLRN queries and a $2.54n + o(n)$

[1] We use $\lg n$ to denote $\log_2 n$

bit structure to answer NLN queries exist through a structure of Fischer et al. [7]. We develop a $2n + o(n)$ bits structure for NLN if all elements are distinct.

Then in Section 2.2, we look at the problems in indexing model. We give a lower bound for time-space tradeoff for NLN and NLRN queries adapting the lower bound tradeoff proof of Brodal et al. [10] for RMQ. We also provide an algorithm that matches the tradeoff for NLN. For NLRN, our algorithm achieves the time-space product of $O((n/c) \lg c)$ (where the query takes $O(c)$ time) while the lower bound is $\Omega(n)$.

For the 2-dimensional NLN problem in the encoding model, we first show that $\Omega(n^2)$ bits are necessary to encode the array to support NLN queries, even when all the elements are distinct. We then describe an asymptotically optimal $\Theta(n^2)$-bit encoding that answers queries in $O(1)$ time. One can achieve this result easily when all the elements in the array are distinct. But, distinctness is a strong assumption in these kinds of problems. For example, in the 1-D case with distinct values, NLRN and NLLN can both be trivially encoded by the Cartesian tree (giving a $2n - O(\lg n)$ 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 [8]. Also, Asano et al. [1] remark that the ANLN problem for any dimension is "simplified considerably" if one assumes distinctness.

We also remark that in 1-D case, the NLRN and NLLN problems are closely connected to the RMQ problem (see e.g. [8]), another problem of wide interest. In the 1-D case, there is no asymptotic difference between the encoding complexity of RMQ and NLRN/NLLN. The 2-D RMQ problem has received a great deal of attention lately [11, 12, 13, 14]. It is known that any 2-D RMQ encoding takes $\Omega(n^2 \lg n)$ bits [11, 12]; thus, this result shows that NLN is different from RMQ in the 2-D encoding scenario.

We assume a standard word RAM model [15] with $O(1)$-time arithmetic and bitwise boolean operations, and we count space in terms of the number of bits used.

2. Data structures for one dimensional arrays

In this section, we give upper and lower bounds for answering the NLN and NLRN queries on a one dimensional (1-D) array, in the encoding and indexing models.

2.1. Data structures for 1-D in the encoding model

NLRN. We first show tight space bounds for NLRN encodings.

Given an array A, the query $RMQ(i, j)$ (range maximum query) returns a position k between i and j such that $A[k]$ is a maximal element among $A[i, \ldots, j]$. Suppose we have a data structure to solve NLRN query, then one can answer the query $RMQ(i, j)$ by finding the leftmost position $k \in \{i, \ldots, j\}$ such that $NLRN > j$. From the $2n - \Theta(\lg n)$-bit lower bound for RMQ in encoding model [16], it follows that any encoding of NLRN requires the same space. We now give an alternative proof, which may be independent interest, for the space lower bound for NLRN.

Theorem 2.1. *Any data structure answering NLRN and NLN queries in the encoding model requires* $2n - \Theta(\lg n)$ *bits.*

Proof. We first define what are called *answer arrays*. An answer array for array A is defined as an array B such that $B[i] = NLRN(A[i])) - i$ if $NLRN(A[i] \neq \infty$ and is ∞ otherwise. We calculate the total number of possible answer arrays for all permutations of n distinct elements.

Let $E(n)$ be the set of all answer arrays for all inputs of size n and $L(n)$ be the set of all answer arrays for all inputs of size n where the element at n^{th} position is the largest element (i.e. n). First we show that $|L(n)| = |E(n-1)|$.
To prove the above relation, we show that there is a bijection between $L(n)$ and $E(n-1)$.

For an answer array B, of an array A corresponding to $E(n-1)$, we simply consider the answer array B' corresponding to the array A' which is A appended with the element n (in the last position). Conversely for an answer array B corresponding to an array in $L(n)$, we simply map it to the answer array of the array with the last element removed.

Adding a new element larger than each of the elements at the right end of the input doesn't change the answer of those indices which already have a NLRN, and the indices that have ∞ as NLRN, change the answer of their indices to direct it to the last position.

Similarly removing the largest element from the end doesn't change the answer of those indices which didn't point to this position, and the indices which pointed to this position will have their answer changed to ∞. This establishes the bijection and hence $|L(n)| = |E(n-1)|$.

Consider all the inputs which have their largest element at index i. In those permutations, all indices $j < i$ do not have their NLRN beyond index i. The last $n - i$ indices also do not need information regarding the first i indices, since each of these elements need to locate the NLRN on the right of their index. So the first i and last $n - i$ indices of the input become disjoint subproblems. Thus, the total number of solutions for these input cases is $|L(i)| * |E(n - i)|$, since the first i indices would have $|L(i)|$ number of solutions, while the last $n - i$ indices have $|E(n - i)|$ solutions. The total number of answer sets for all permutations on n elements can be found by summing up the total number of answer sets, when the largest element of the input is at i^{th} index for all possible values of i.

$|E(n)| = \sum_{i=1}^{n} |L(i)| * |E(n - i)|$ which can be re-written as $|E(n)| = \sum_{i=1}^{n} |E(i - 1)| * |E(n - i)|$

This recurrence solves to be the Catalan number C_n, which is $\Theta(4^n/n^{3/2})$.

Thus, we need at least $2n - O(\lg n)$ bits to represent any NLRN structure to answer all the NLRN queries without access to input at query time (i.e. in encoding model).

To extend the proof for NLN problem, we notice that number of answer sets $F(n)$ when the largest element is in the first position and the number of answer sets $L(n)$ when the largest element is at the last position are the same. This is because if we reverse the array, then the answer array for NLN also gets reversed.

Also the recursion $|E(n)| = \sum_{i=1}^{n} |L(i)| * |E(n-i)|$ for NLRN changes to $\sum_{i=1}^{n} |L(i)| * |F(n-i+1)| = \sum_{i=1}^{n} |L(i)| * |L(n-i+1)|$ which also solves to the Catalan number asymptotically. □

The converse of the simulation in the proof of Theorem 2.1 – i.e., supporting NLRN queries using RMQ data structure also works, giving a $2n+o(n)$-bit encoding that supports NLRN queries in $O(\lg n)$ time. In fact, the query time can be improved to $O(1)$ using a 2d-Max-Heap as observed by Fischer [16]. We give the description for completeness.

Lemma 2.1 ([7, 16]). *There exists a data structure in the encoding model that solves NLRN queries using $2n + o(n)$ bits in $O(1)$ time.*

Proof. A 2d-Max-Heap analogous to 2d-Min-Heap as described by Fischer [16] is constructed on array $A[1, n + 1]$. The original structure in that paper solves the nearest smaller left neighbor problem.

The 2d-Max-Heap M_A of A is a labeled ordered tree with vertices $v_1 \dots v_{n+1}$ and each vertex corresponds to an index i. The root is labelled with $n + 1$, and we treat $A[n + 1]$ as ∞. The parent node of v_i is v_j if and only if $i < j$, $A[i] < A[j]$, and $A[k] \leq A[i]$ for all $i < k \leq j$. I.e. as we read the array from right to left, we make v_i the child of v_j if v_j is the nearest larger element (in the right) to v_i. It is clear that the sibling values are in increasing order from left to right, and that the parent of the node labelled v_i corresponds to the NLRN of $A[i]$.

As explained by Fischer [16], the 2d-Min-Heap can be represented using any of the succinct ordinal tree representations, taking $2n + o(n)$ bitsto support NLRN in constant time. □

NLN. Clearly an NLN query can be answered by constructing two data structures that support NLLN and NLRN queries. This results in a space of $4n + o(n)$ bits. Fischer [8] has given a data structure that takes $2.54n + o(n)$ bits to answer both NLLN and NLRN queries.

It is not clear whether NLN can be answered without explicitly answering NLLN and NLRN, and hence whether a representation using $(2.54 - \epsilon)n$ bits suffice to answer NLN queries. We conjecture that this is not possible, i.e., $2.54n + o(n)$ bits are necessary to answer NLN queries. However, when all elements are distinct, we show that one can do better.

Theorem 2.2. *There exists a data structure that uses $2n + o(n)$ bits and can answer NLLN and NLRN queries in $O(1)$ time, when all elements are distinct.*

Proof. We show how to support NLLN and NLRN using a succinct representation of a Cartesian tree [17]. Given a node v in the Cartesian tree corresponding to a position i in the input array, the position j corresponding to the parent of v is either $NLLN(i)$ or $NLRN(i)$, depending on whether $j < i$ or $j > i$ [3]. Suppose that $j < i$ (the other case is symmetric). Then the node corresponding to the position $NLRN(i)$ is the lowest ancestor of v which corresponds to a position greater than i, which is in fact the next node in preorder after the rightmost leaf of v. This can be computed in $O(1)$ time using any succinct binary tree representation that supports rank and select of nodes in preorder and subtree size. The binary tree representation of [17] supports all these operations in $O(1)$ time. □

2.2. Data structures for 1-D in the indexing model

NLN. Recall that in the indexing model, the input can be accessed at query time, and hence significant space saving should be possible. We first observe that this can be extended to a time-space tradeoff for NLN.

Theorem 2.3. *For any parameter c, where $1 \leq c \leq n$, there exists a data structure which solves NLN queries in the indexing model using $O(n/c)$ bits in $O(c)$ time.*

Proof. Break the input array A of n elements into chunks of size c. Extract the maximal of each block and construct a conceptual array B of size n/c. Build an encoding data structure for B that answers NLLN and NLRN queries, using $2.54 \cdot n/c + o(n/c)$ bits.

To solve an NLN query for index i, check in $O(c)$ time by scanning the original input array, whether $A[i]$ is a maximal element of its block. If $A[i]$ is a maximal element in its block, use the encoding structure for B to find the blocks containing its NLLN and NLRN. Now, sequentially scan these two blocks to find the NLN of the query element (in fact, if the two blocks are not at the same distance from the block containing the query element, it is enough to scan the nearest one). If $A[i]$ is not the maximum of the block then in $O(c)$ time, check the block containing the query index, as well as the block before the query index and the block after the query index, by sequentially scanning the elements and find the NLN. The space used in addition to the input is $2.54 \cdot n/c + o(n/c)$ bits, and the time taken to solve a query is $O(c)$. □

In the indexing model, Brodal et al. [10] proved a lower bound for space-time tradeoff of the RMQ problem. The proof is in the non-uniform cell probe model [15]. In this model, computation is free, and time is counted as the number of cells accessed (probed) by the query algorithm. The algorithm is also allowed to be non-uniform, i.e., for different values of input parameter n, we can have different algorithms.

For n and any value of c, where $1 \leq c \leq n$, they define a set of arrays $C_{n,c}$ and a set of queries Q. They argue that for any RMQ algorithm which has access to an index of size n/c bits (in addition to the input array A), there exists an array in $C_{n,c}$ and a query in Q for which the algorithm is required to perform $\Omega(c)$ probes into A. The following lower bound for NLRN and NLN follows along those lines.

Theorem 2.4. *Any data structure which stores $O(n/c)$ bits and answers NLRN (or NLN) queries in the indexing model, requires at least $\Omega(c)$ query time.*

Proof. The proof follows the argument of Brodal et al. [10] who proved a lower bound for space-time tradeoff of the RMQ problem. For completeness, here, we reproduce some of their analysis.

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

The number of possible data structures of size n/c bits is $2^{n/c}$, and the number of arrays in $C_{n,c}$ is $c^{n/c}$. By the pigeonhole principle, for any algorithm G there exists a data structure D_G which is shared by at least $(c/2)^{n/c}$ input arrays in $C_{n,c}$. Let $C_{n,c}^{D_G} \subseteq C_{n,c}$ be the set of these inputs. Let $q_i = NLRN(ic + 1)$. The set $Q = \{q_i \mid 0 \leq i \leq n/c - 1\}$ contains n/c queries, where $NLRN(x)$ returns the index of NLRN of query index x.

For algorithm G and data structure D_G , we define a binary decision tree capturing the behavior of G on the inputs from $C_{n,c}$ to answer a query $q \in Q$. Let G be a deterministic algorithm. For each query $q \in Q$, we define a binary decision tree $T_q(D_G)$. Each internal node of $T_q(D_G)$ represents a probe into a cell of the input arrays from $C_{n,c}$. The left and right edges correspond to the output of the probe: left for zero and right for one. Each leaf is labeled with the answer to q.

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

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

Comparing the lower and upper bounds from the above discussion for the number of leaves of $T_Q(D_G)$, we have

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

Using Stirling's formula, we obtain the following

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

which implies $c/2 \leq (d + n/c)e$, and therefore $d \geq n(\frac{1}{2e} - \frac{1}{c})$. For any arbitrary algorithm G, the depth d of $T_Q(D_G)$ equals the sum of the depths of the n/c binary trees composed into $T_Q(D_G)$. By the pigeonhole principle, there exists an input $x \in C_{n,c}^{D_G}$ and an i, where $1 \leq i \leq n/c$, such that the query q_i on x requires at least $d/(n/c) = \Omega(c)$ probes into the array A. This shows the lower bound for NLRN.

For NLN, we change $q_i = NLRN(ic+1)$ to $q_i = NLN(ic+c/2)$, and the rest of the proof remains the same, proving the same query time and space tradeoff for the NLN problem. □

NLRN. Now we give a result for the NLRN problem in the indexing model analogous to Theorem 2.3. The approach follows closely the proof of Fischer et al. [7], which in turn adapts ideas from Jacobson's representation of balanced parentheses sequences [18], and is given in full for completeness.

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

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

Lemma 2.2 ([19]). *Given a string X over {0,1} with length n, containing m 1s, it can be represented in $O(m\lg(n/m))$ bits such that rank_1 and select_1 can be supported in $O(n/m)$ time.*

Theorem 2.5. *Given a 1-D array A of size n, there exists a data structure which supports NLRN 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 $NLRN(i)$ 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 NLRN in a block B'. Then the leftmost far value in B whose NLRN is in B' is called a *pioneer*, and its NLRN is called its *match*. It is known that there are $O(n/c)$ pioneers in A [18].

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 contains $O(n/c)$ 1's, so by Lemma 2.2, 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 2.1 using $O(n/c)$ bits, to support NLRN queries in $O(1)$ time. We claim that for any pioneer in the list, its NLRN in the sequence of pioneers/matches is the same as its NLRN in the original sequence. Suppose that this claim is not true. This means there is a pioneer i_p such that NLRN(i_p) is the value between i_p and the match of i_p. It cannot be the case that i_p and NLRN(i_p) are in the same block, since i_p is a far value. If i_p and NLRN(i_p) are in different blocks, then NLRN(i_p) is the match of i_p. So the claim is true.

To answer the query $NLRN(i)$, 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 $A[i] < A[p_i]$, NLRN(i) is less than or equal to the match of p_i. Since i is the far value in this case, $NLRN(i)$ and $NLRN(p_i)$ are in the same block. We find NLRN(p_i) using the NLRN 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 NLRN(i). The overall time taken to answer the query is $O(c)$. □

3. NLN on two dimensional binary arrays

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

Lemma 3.1 ([9]). *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 3.1. *There is a data structure that takes $O(n^2)$ bits for a binary array $A[1 \dots n][1 \dots n]$ which supports* NLN *queries in $O(1)$ time.*

Proof. Given a query position p, we compute the NLN(p) by computing the positions of the nearest larger values in all four *quadrants* induced by a vertical and a horizontal lines passing through p, and then returning the closest of these four positions as the answer. Thus, it is enough to describe a structure that supports finding the position of the nearest larger value in (say) the upper-right quadrant; in the rest of this proof, we use NLN_{NE} to denote this.

Given a position $p = (i, j)$, let $q = (i', j')$ be its NLN_{NE} if there is a 1 in the upper-right quadrant of p. We give a label from the alphabet $\{R, C, D, O, Z\}$ as follows. The position (i, j) is labeled with: (1) O if $A[p] = 1$; (2) R if $i = i'$; (3) C if $j = j'$; (4) D if $i < i'$ and $j < j'$; and (5) Z if $A[p] = 0$ and there is no 1 in the upper-right quadrant of p.

Now, given a query position p, if the position p is labeled with O or Z, then we conclude that NLN_{NE} does not exist (in this quadrant). Otherwise, if the label is R, 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 with label O, and return that position as the answer. Also, one can easily show that all the intermediate positions have label O. Analogously, if the label is C, then we follow the positions in the same column until we reach position with label O and return that position. Finally, if the label is D, then we first follow the positions $(i + k, j + k)$, for $k = 1, 2, \dots$ till we reach the first position $(i + \ell, j + \ell)$ with a label different from D. The label of position $(i + \ell, j + \ell)$ can be O, R or C. If it is O, then we return that position as the answer. In the other two cases, we can find the answer by following the row or column as described above. The data structure simply stores the labels of all positions in the array (for each quadrant). In addition, to support the queries faster, we build rank/select structures (over constant alphabet strings) for the encoding of each row, each column and each diagonal. By Lemma 3.1,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. □

Now we describe an index for a given 2-D binary array, in the bit-probe model. 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 $1 \le c \le n$, 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 $A[(i-1)c + 1 \dots ic][(j-1)c \dots jc]$ 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 $l < j$;
$E(i, j)$ contains all (k, j)-blocks with $k > i$;
$W(i, j)$ contains all (k, j)-blocks with $k < i$;
$NE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l > j$;
$NW(i, j)$ contains all (k, l)-blocks with $k < i$ and $l > j$;
$SE(i, j)$ contains all (k, l)-blocks with $k > i$ and $l < j$; and
$SW(i, j)$ contains 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 $SW_{i,j}(k, l)$ in the same way.

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 $(\lfloor i/c \rfloor, \lfloor j/c \rfloor, i - \lfloor i/c \rfloor, j - \lfloor j/c \rfloor)$-sub-block, and 0 otherwise.

Theorem 3.2. *Given a binary array $A[1 \dots n][1 \dots n]$ one can construct an index of size $O(n^2/c)$ bits to support NLN queries in $O(c)$ time for $1 \le c \le n$.*

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 NLN 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 one to the query in the $NE_{i,j}(k, l)$ region. This element is the nearest one from the bottom-left conner of $(i, j, k + 1, l + 1)$-sub-block. The nearest one 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 one in the same block, or is the nearest one 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 one from q in $NE(i, j, k, l)$. Once we find the corresponding sub-block, finding the nearest one from the bottom-left conner in the sub-block takes $O(c)$ time. We can find the nearest one in the $NW_{ij}(k, l)$, $SE_{ij}(k, l)$ and $SW_{ij}(k, l)$ regions in the same way. Then NLN of q is the closest one among these eight candidates.

Finally, consider the case when $A'[i, j] = 0$. By storing the data structure of Theorem 3.1 for the array A' using $O(n^2/c^2)$ bits, we can find the nearest block to the query position which contains a one, 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 NLN. Assume, wlog, that (i', j') is in the $NE(i, j)$ region. We first describe

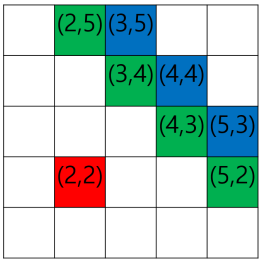

Figure 1. Suppose the nearest block that contains one from (2,2)-block is (4,3)-block, Then $d(4, 3)$ are the blocks colored by green and we can find the nearest one in NE(2,2) using RMQ for $D_{4,3}[2, 3]$ and $D_{4,4}[1 \dots 3]$

how to find the nearest one 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 one 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 $O(1)$ time. Now, we find the two potential blocks in $NE(i, j)$ region that may have the nearest one 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 one in $NW(i, j)$, $SE(i, j)$ and $SW(i, j)$ in a similar way.

Next, we describe how to find the nearest one in the $N(i, j)$ region (finding the nearest one in the $S(i, j)$, $E(i, j)$ and $W(i, j)$ regions is analogous). For each position in the bottom row of an (a, b)-block with $A'[a, b] = 1$, 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 one 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 one 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 one to q in the $N(i, j)$ region is closest one 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 NLN of q by comparing these eight candidate answers. □

4. NLN on two dimensional arrays

Consider an $n \times n$ 2-dimensional (2-D) array $A[1 \ldots n][1 \ldots n]$. Given two positions (i, j) and (i', j') in A, we define $dist((i, j), (i' j')) = |i - i'| + |j - j'|$. A trivial solution to the NLN problem in 2-D array is to store $NLN((i, j))$, for $1 \leq i, j \leq n$. This requires $O(n^2 \lg n)$ bits, and supports queries in $O(1)$ time. In the following, we obtain improved results for the 2-D NLN in the encoding and indexing models, and also describe some trade-off results.

4.1. 2-D NLN in the encoding model – distinct case

When there is no restriction on the elements of the array, one can show an n^2-bit lower bound for NLN encoding (follows easily for the case of a bit array). Using a simple encoding method, one can prove that the same asymptotic lower bound applies even when all the elements of the array are distinct to obtain the following.

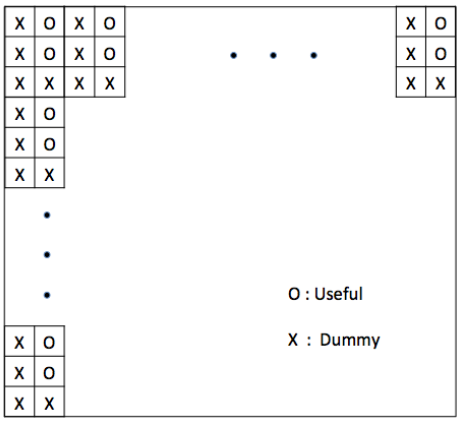

Figure 2. *useful* and *dummy* elements in A_2

Theorem 4.1. *Any data structure which supports NLN queries on an $n \times n$ array $A[1 \ldots n][1 \ldots n]$ in encoding model requires at least $n^2/6$ bits, even when all the elements in A are distinct.*

Proof. We define a set $\mathcal{A}$ of $2^{n^2/6}$ matrices, such that the answers to the NLN queries in any array $A \in \mathcal{A}$ can be used to distinguish A from $\mathcal{A} \setminus \{A\}$. This proves that encoding for an arbitrary array A in $\mathcal{A}$ requires at least $n^2/6$ bits in the worst case.

Each array $A \in \mathcal{A}$ contains elements from the set $\{1, 2, \ldots, n^2\}$, where each element appears exactly once. To describe the arrays in A, we classify the elements of each array into *useful* and *dummy* elements. The elements from 1 to $2n^2/3$ constitute the dummy elements, and the rest (from $2n^2/3 + 1$ to n^2) constitute the useful elements. As shown in Figure 2, the positions $(3i + 1, 2j)$ and $(3i + 2, 2j)$, for $0 \leq i < n/3$ and $1 \leq j \leq n/2$, contain useful elements, and the remaining $2n^2/3$ positions contain the dummy elements. For all the positions containing dummy elements, we assign the values of dummy elements in any fixed order; this assignment is same for every array $A \in \mathcal{A}$. Also, given two adjacent positions containing useful elements, $(3i + 1, 2j)$ and $(3i + 2, 2j)$, for $0 \leq i < n/3$ and $1 \leq j \leq n/2$, the values of elements for these two positions respectively are fixed, arbitrarily, to be two elements x and y, or y and x. Since there are $n^2/6$ such adjacent pairs, and for each pair, we have two choices for the order of its elements, there are, in total, $2^{n^2/6}$ matrices in $\mathcal{A}$.

Now, it is easy to see that given the answers to the NLN queries of all adjacent pairs of useful elements, we can distinguish the array A from $\mathcal{A} \setminus \{A\}$. □

We now obtain an asymptotically optimal upper bound for 2-D NLN encoding for the distinct case.

Lemma 4.1. *A 2-D array $A[1 \ldots n][1 \ldots n]$ can be encoded using $O(n^2)$ bits to support NLN queries, provided all elements are distinct.*

Proof. The main idea is to divide the array recursively into blocks of geometrically increasing size, and store the NLN values of all elements, except the largest element and the elements whose answers are stored at a previous level, in each block explicitly. The following argument shows that this requires $O(n^2)$ bits overall.

In the first level, we divide A into $n^2/4$ blocks of size 2×2 each. Except for the largest element in each 2×2 block, the distance of NLN answer for the other three elements are bounded by 2. In general, at level k, we divide A into $n^2/4^k$ blocks of size $2^k \times 2^k$ each. In each of these $2^k \times 2^k$-sized blocks, there are four elements left for which we need to store the answer to their NLN queries. For three of these four elements, which do not correspond to the maximum value in the block, we store their answers at level k. Since the distance to the NLN answer for these three elements is bounded by 2^{k+2}, we can store these answers using $O(k)$ bits. Thus the total space usage is bounded by $\sum_{k=1}^{\lg n}(3n^2/4^k) * O(k) = O(n^2)$ bits. □

We now describe another $O(n^2)$-bit encoding for the 2-D NLN problem that supports queries in constant time.

Theorem 4.2. *A 2-D array $A[1 \ldots n][1 \ldots n]$ can be encoded using $O(n^2)$ bits to support NLN queries in $O(1)$ time, provided all elements are distinct.*

Proof. The encoding is a small variant of the encoding described in the proof of Lemma 4.1. For each position in A, in some canonical order (say, row-major order), we write down the relative position (i.e., the distance in from the position to its answer in horizontal and vertical directions) of its NLN answer. We use a variable-length encoding, such as γ-code or δ-code [20], to write these answers. The proof of Lemma 4.1 implies that the sum of the lengths of all these answers is $O(n^2)$. We also store an indexable bit vector [9] indicating the starting positions of each code. This enables us to find the position where the answer to a given query starts and ends, in constant time. □

4.2. 2-D NLN in the encoding model – general case

In this section, we give an encoding which supports NLN queries in a 2-D array with $O(n^2)$ bits in the general case. Before starting the 2-D case, we consider the 1-D case first. Theorem 2.2 shows how to encode an array A with n distinct items using $O(n)$ bits to answer NLN queries. We give an alternate proof of this, based on ideas from [2]:

Lemma 4.2. *There exists an encoding of an array $A[1 \ldots n]$ that uses $O(n)$ bits while supporting NLN queries, provided all elements are distinct.*

Proof. We write down the sequence $d(1), d(2), \ldots, d(n)$ explicitly, where $d(i) = n$ if $A[i]$ is the maximum elements of A, and $d(i) = |i - NLN(i)|$ otherwise, for $1 \leq i \leq n$, together with a sequence of n bits that indicate if $i < \mathsf{NLN}(i)$ or $i > \mathsf{NLN}(i)$. Because 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. □

If the elements in A are not distinct, the above argument does not hold. So instead of encoding the NLN of a position i explicitly as in Lemma 4.2, we encode the distance between i and the nearest value which is $\geq A[i]$ in the same direction as $NLN(i)$. 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 $NLN(i) < i$ 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, $NLN(i)$ can be easily found by following the $d(\cdot)$ "pointers" from i until we reach a position that is greater than $A[i]$. We now show that this encoding uses $O(n)$ bits[2]:

Lemma 4.3. *For any array $A[1 \dots n]$, $D = \sum_{i=1}^{n} \lg d(i) = O(n)$.*

Proof. Consider the array $A'[1 \dots n]$ of size n, where $A'[i] = A[i] + \epsilon i$ if $NLN(i) > i$ and $A'[i] = A[i] - \epsilon i$ if $NLN(i) < i$ 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 NLN' on A' analogously to $d(i)$ and NLN on A, $D' = \sum_{i=1}^{n} \lg d'(i) = O(n)$ by Lemma 4.2. We now show that $D \leq 2D'$.

To prove this claim, let $0 \leq i_0 < \cdots < i_r \leq n$ with $r > 0$ be a maximal sequence of indices such that $A[i_0] > A[i_1]$, $A[i_{r-1}] < A[i_r]$, $A[i_i] = A[i_2] = \cdots = A[i_{r-1}]$ and for all $i_0 < j < i_r$, $A[j] < A[i_1]$. For $0 \leq k \leq r$, let i_k be the index such that $NLN(i_l) = i_0$ for all $0 < l \leq k$ and $NLN(i_l) = i_r$ for all $k < l \leq r - 1$. Then by the definition of A', for all $k < l \leq r - 1$, $NLN'(i_l) = i_{l+1}$ 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 $NLN'(i_l) = i_{l-1}$ for all $0 < l \leq m - 1$ and $NLN'(i_l) = i_{l+1}$ for $m \leq l \leq k$. Then:

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

□

We now extend this encoding to encode NLNs for a 2-D array $A[1 \dots n][1 \dots n]$. 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 coded 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 without loss of generality assume that $NLN(i, j) = (i^*, j^*)$ 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.

[2]Note that this encoding cannot be obtained by simply breaking ties among equal elements in some arbitrary fashion and applying Lemma 4.2. For example, if $A[i] = A[i + 1]$ and $A[i - t]$ and $A[i + 1 + t]$ for some $t > 1$ are the nearest larger values, then in the current encoding, neither $A[i]$ nor $A[i + 1]$ would point to one another. If we break ties then either $A[i]$ points to $A[i + 1]$ or $A[i + 1]$ points to $A[i]$.

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

We call this encoding scheme $encoding_{2-D}$. To obtain $\mathsf{NLN}(i, j)$, 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 is shown in the following lemma.

Lemma 4.4. *Given a 2-D array $A[1 \dots n][1 \dots n]$, for all $1 \le i, j \le n$, $NLN(i, j)$ is obtained by the above procedure.*

Proof. The proof is by induction on k, the distance between (i, j) and $\mathsf{NLN}(i, j) = (i^*, j^*)$. The base case $k = 1$ follows directly from case 3) above.

Assume the induction hypothesis holds for all NLNs at distance $\le k$, and choose an (i, j) such that $\mathsf{NLN}(i, j) = (i^*, j^*)$ and $dist((i, j), (i^*, j^*)) = k + 1$. Assume wlog that the pointer from (i, j) has its extra bit set to 0 (otherwise, the induction step is trivial) and it points to (i', j) with $i' > i$. Assume that $NLN(i', j) = (x, y) \ne (i^*, j^*)$, and $dist((x, y), (i, j)) > dist((x, y), (i^*, j^*))$. Since $A[i', j] = A[i, j]$, $dist((i', j), (x, y)) \le dist((i', j), (i^*, j^*)) < dist((i, j), (i^*, j^*)) = k + 1$. By the induction hypothesis, following pointers from (i', j) leads to (x, y). Now:

$$\begin{aligned} dist((i, j), (x, y)) &\le dist((i, j), (i', j)) + dist((i', j), (x, y)) \\ &\le dist((i, j), (i', j)) + dist((i', j), (i^*, j^*)) \qquad (\because NLN(i', j) = (x, y)) \\ &\le dist((i, j), (i^*, j^*)), \end{aligned}$$

contradicting the assumption that $dist((x, y), (i, j)) > dist((x, y), (i^*, j^*))$. □

Theorem 4.3. *There exists an encoding of a 2-D array $A[1 \dots n][1 \dots n]$ that supports NLN queries, using $O(n^2)$ bits.*

Proof. We describe an encoding, called $encoding_{grid}$ as follows. We first encode each column and row of A using Lemma 4.3, 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 4.3, as their NLN 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 (i_j, r) to (i_{j+1}, r) 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 NLN, we use it to upper bound the space used by $encoding_{2-D}$. Let a *grid pointer* and a *2-D pointer* refer to a pointer in $encoding_{grid}$ and $encoding_{2-D}$ respectively. For any 2-D 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) 2-D pointers and the row cost of all case 3) 2-D pointers by grid pointers in rows, using each grid pointer at most thrice. The costs of Case (1) 2-D pointers and the column cost of Case (3) 2-D 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 $\mathsf{NLN}(i, j) = (i^*, j^*)$ with $i^* \ge i$ 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 (see Figure 3).

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

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

Case (b) There is either no value $A[i, j'] \ge A[i, j]$ 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.

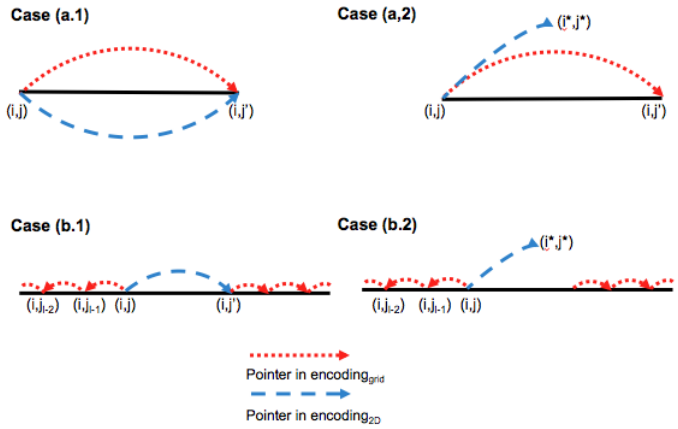

Figure 3. Pointers in $encoding_{2-D}$ and $encoding_{grid}$

(b.1) First suppose that the 2-D 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 NLN of (i, j) in the i-th row is (i, j_0) for some $j_0 < j$. Note that $|j_0 - j| \geq dist((i, j), (i^*, j^*))$, otherwise NLN$(i, j)$ would be (i, j_0). As $dist((i, j), (i^*, j^*)) = |j - j'| + |j' - j^*| + |i^* - i|$, $|j_0 - j| \geq |j' - j|$. 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 2-D 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 $j = j_l, j_{l-1}, \ldots, j_1, j_0$ (omitting the row number for brevity). Note that for any $0 < k < l$, no 2-D 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 2-D edge: once from (i, j) and once (possibly) from (i, j_0).

(b.2) The 2-D 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 2-D 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 2-D 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 NLN 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 2-D 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 2-D pointer. □

Now we describe the $O(n^2)$-space data structure that supports NLN query in constant time on 2-D array $A[1 \ldots n][1 \ldots n]$. To support this, first divide A into blocks of size $b \times b$ and divide each block into sub-blocks of size $s \times s$.

Also we divide the outside of blocks and sub-blocks into 8 areas (N, S, W, E, NW, NE, SE, SW) as defined in Section 3. Now we prove the main theorem.

Theorem 4.4. *There exists an encoding of 2-D array $A[1 \ldots n][1 \ldots n]$ that uses $O(n^2)$ bits while supporting NLN queries in $O(1)$ time.*

Proof. For each element $A[i, j]$, we assign a color from the set $\{C_1, C_2, \ldots, C_9\}$, as follows. If $NLN(i, j)$ 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 NLN 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. 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 NLN 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 NLN analogous to C_1 to C_9. To support NLN quries for non-maximal elements in a sub-block, we encode each sub-block together with its 24 neighboring sub-blocks, using the encoding of Theorem 4.3, 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 NLN 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 $NLN(q)$ 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 NLN 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 NLN queries in $O(n^2)$ bits. □

5. Conclusions

Our main contribution is a systematic study of data structures for NLN and NLRN in one dimensional arrays, and NLN in two dimensional arrays, in the encoding and indexing models. We end with specific open problems that can trigger further work.

- Is there a data structure that takes less than $2.54n + o(n)$ bits and can answer NLN queries in a one dimensional array in constant time in the general case (when elements may repeat) in the encoding model?
- For a 1-D array, is there an index for NLRN that uses $O(n/c)$ bits and supports queries in $O(c)$ query time?
- For a 2-D array, is there an index for NLN that uses $O(n^2/c)$ bits and supports queries in $O(c)$ query time?

References

[1] T. Asano, S. Bereg, D. G. Kirkpatrick, Finding nearest larger neighbors, in: Efficient Algorithms, 2009, pp. 249–260.
[2] T. Asano, D. G. Kirkpatrick, Time-space tradeoffs for all-nearest-larger-neighbors problems, in: WADS, 2013, pp. 61–72.
[3] O. Berkman, B. Schieber, U. Vishkin, Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values, J. Algorithms 14 (3) (1993) 344–370.
[4] V. Jayapaul, S. Jo, V. Raman, S. R. Satti, Space efficient data structures for nearest larger neighbor, in: Proc. IWOCA 2014, 2014, to appear.
[5] S. Jo, R. Raman, S. R. Satti, Compact encodings and indexes for the nearest larger neighbor problem, in: WALCOM, 2015, pp. 53–64.
[6] M. J. Golin, J. Iacono, D. Krizanc, R. Raman, S. S. Rao, Encoding 2d range maximum queries, in: ISAAC, 2011, pp. 180–189.
[7] J. Fischer, V. Mäkinen, G. Navarro, Faster entropy-bounded compressed suffix trees, Theor. Comput. Sci. 410 (51) (2009) 5354–5364.
[8] J. Fischer, Combined data structure for previous- and next-smaller-values, Theor. Comput. Sci. 412 (22) (2011) 2451–2456.
[9] R. Raman, V. Raman, S. R. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Transactions on Algorithms 3 (4) (2007) Article 43.
[10] G. S. Brodal, P. Davoodi, S. S. Rao, On space efficient two dimensional range minimum data structures, Algorithmica 63 (4) (2012) 815–830.
[11] E. D. Demaine, G. M. Landau, O. Weimann, On cartesian trees and range minimum queries, Algorithmica 68 (3) (2014) 610–625. doi:10.1007/s00453-012-9683-x.
[12] G. S. Brodal, P. Davoodi, S. S. Rao, On space efficient two dimensional range minimum data structures, Algorithmica 63 (4) (2012) 815–830.
[13] G. S. Brodal, P. Davoodi, M. Lewenstein, R. Raman, S. S. Rao, Two dimensional range minimum queries and Fibonacci lattices, in: ESA, 2012, pp. 217–228.
[14] G. S. Brodal, A. Brodnik, P. Davoodi, The encoding complexity of two dimensional range minimum data structures, in: ESA, 2013, pp. 229–240.
[15] P. B. Miltersen, Cell probe complexity - a survey, FSTTCS: Advances in Data Structures Workshop.
[16] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM Journal on Computing 40 (2) (2011) 465–492.
[17] P. Davoodi, R. Raman, S. R. Satti, Succinct representations of binary trees for range minimum queries, in: J. Gudmundsson, J. Mestre, T. Viglas (Eds.), COCOON, Vol. 7434 of Lecture Notes in Computer Science, Springer, 2012, pp. 396–407.
[18] G. Jacobson, Space-efficient static trees and graphs, in: FOCS, IEEE Computer Society, 1989, pp. 549–554.
[19] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007, 2007. doi:10.1137/1.9781611972870.6.
[20] P. Elias, Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory 21 (2) (1975) 194–203.