Encoding Nearest Larger Values

Patrick K. Nicholson[1] and Rajeev Raman[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] University of Leicester, Leicester, United Kingdom

Abstract. In *nearest larger value (NLV)* problems, we are given an array $A[1..n]$ of numbers, and need to preprocess A to answer queries of the following form: given any index $i \in [1, n]$, return a "nearest" index j such that $A[j] > A[i]$. We consider the variant where the values in A are distinct, and we wish to return an index j such that $A[j] > A[i]$ and $|j - i|$ is minimized, the *nondirectional NLV (NNLV)* problem. We consider NNLV in the *encoding* model, where the array A is deleted after preprocessing, and note that NNLV encoding problem has an unexpectedly rich structure: the *effective entropy* (optimal space usage) of the problem depends crucially on details in the definition of the problem. Using a new *path-compressed* representation of binary trees, that may have other applications, we encode NNLV in $1.9n + o(n)$ bits, and answer queries in $O(1)$ time.

1 Introduction

Nearest Larger Value (NLV) problems have had a long and storied history. Given an array $A[1..n]$ of values, the objective is to preprocess A to answer queries of the general form: given an index i, report the index or indices nearest to i that contain values strictly larger that $A[i]$. Berkman et al. [3] studied the parallel pre-processing for this problem and noted a number of applications, such as parenthesis matching and triangulating monotone polygons. The connection to string algorithms for both the data structuring and the pre-processing variants of this problem is since well-established.

Since the definition of "nearest" is a bit ambiguous, we propose replacing it by one of the following options in order to fully specify the problem:

- *Unidirectionally nearest*: the solution is the index $j \in [1, i-1]$ such that $A[j] > A[i]$ and $i - j$ is minimized.
- *Bidirectionally nearest*: the solution consists of indices $j_1 \in [1, i-1]$ and $j_2 \in [i+1, n]$ such that $A[j_k] > A[i]$ and $|i - j_k|$ is minimized for $k \in \{1, 2\}$.
- *Nondirectionally nearest*: the solution is the index j such that $A[j] > A[i]$ and $|i - j|$ is minimized. As far as we are aware, this formulation has not been considered before.

Furthermore, the data structuring problem has different characteristics depending on whether we consider the elements of A to be distinct (Berkman et al. considered the undirectional variant when all elements in A are distinct).

We consider the problem in the *encoding* model, where once the data structure to answer queries has been created, the array A is deleted. Since it is not possible to reconstruct A from NLV queries on A, the *effective entropy* of NLV queries [9], the log of the number of distinguishable NLV configurations, is very low and an NLV encoding of A can be much smaller than A itself. The encoding variant has several applications in space-efficient data structures for string processing, in situations where the values in A are intrinsically uninteresting:

- The bidirectional NLV when A contains distinct values boils down essentially to encoding a Cartesian tree, through which route $2n + o(n)$-bit and $O(1)$-time data structures exist [7,4].
- The unidirectional NLV when A contains non-distinct values can be encoded in $2n + o(n)$ bits and queries answered in $O(1)$ time [8,10].
- The bidirectional NLV for the case where elements in A need not be distinct was first studied by Fischer [6]. His data structure occupies $\log_2(3+2\sqrt{2})n + o(n) \approx 2.544n + o(n)$ bits of space, and supports queries in $O(1)$ query time.

All of the above space bounds are tight to within lower-order terms.[3]

In this paper, we consider the nondirectionally nearest larger value (NNLV) problem, in the case that all elements in A are distinct. The above results already hint at the combinatorial complexity of NLV problems. However, the NNLV problem appears to be even richer, and the space bound appears not only to depend upon whether A is distinct or not, but also upon the specific tie-breaking rule to use if there are two equidistant nearest values to the query index i.

For instance, given a location i where there is a tie, we might always select the larger value to the right of location i to be its nearest larger value. We call this *rule I*. We give an illustration in the middle panel of Figure 1 (on page 4). Alternative tie breaking rules might be: to select the smallest of the two larger values (*rule II*), or to select the larger of the two larger values (*rule III*). Interestingly, it turns out that the tie breaking rule is important for the space bound. That is, if we count the number of distinguishable configurations of the NNLV problem for the various tie breaking rules, then we get significantly different answers. We counted the number of distinguishable configurations, for problem instances of size $n \in [1, 12]$, and got the sequences presented in Table 1.

Table 1. Number of distinguishable configurations of nearest larger value problems with the three tiebreaking rules discussed.

n	1	2	3	4	5	6	7	8	9	10	11	12
rule I	1	2	5	14	40	116	341	1010	3009	9012	27087	81658
rule II	1	2	5	14	42	126	383	1178	3640	11316	35263	110376
rule III	1	2	5	12	32	88	248	702	1998	5696	16304	46718

[3] For the unidirectional NLV the bound is tight even when all values are distinct.

Unfortunately, none of the above sequences appears in the Online Encyclopedia of Integer Sequences[4]. Consider the sequence generated by some arbitrary tie breaking rule. If z_i is the i-th term in this sequence, then $\lim_{n\to\infty} \lg(z_n)/n$ is the constant factor in the asymptotic space bound required to store all the answers to the NNLV problem subject to that tiebreaking rule.

Our Contributions. Our main result is the following:

Theorem 1. *Let $A[1..n]$ be an array containing distinct numbers. The array A can be processed to obtain an encoding data structure that occupies $1.9n+o(n)$ bits of space, that can answer the query $\texttt{NNLV}(A, i)$ in $O(1)$ time for any $i \in [1, n]$. Ties are resolved using rule I. At no point after preprocessing does the data structure require access to the array A.*

As mentioned before, the Cartesian tree (defined later) occupies $2n+o(n)$ bits and can solve NNLV queries. In Section 3 we describe a novel *path-compressed* representation of a binary tree that uses $2n + O(\lg n)$ bits (but supports no operations). To get the improved space bound of Theorem 1 we prove combinatorial properties of the NNLV problem relating to long chains in the Cartesian tree. These properties allow us to compress the Cartesian tree using the representation of Section 3, losing some information, but still retaining the ability to answer NNLV queries. The constant factor (1.9) comes from a numeric calculation bounding the worst case structure of chains in the Cartesian tree for our compression scheme (Section 4). In Section 4.1 we show how to support operations on the "lossy" Cartesian tree, thereby proving Theorem 1.

Finally, in Section 5, we prove a lower bound, via exhaustive search:

Theorem 2. *Any encoding data structure that can answer the query $\texttt{NNLV}(A, i)$ for any $i \in [1, n]$ (breaking ties according to rule I) must occupy at least $1.3173n - \Theta(1)$ bits, for sufficiently large values of n.*

Other Related Work: Asano et al. [1] studied the time complexity of computing all nearest larger values in an array as well as higher dimensions, and mention applications to communication protocols. Asano and Kirkpatrick [2] considered sequential time-space tradeoffs for computing the nearest larger values of all elements in the array. Finally, Jo et al. [11] and Jayapaul et al. recently studied the nearest larger value problem in two dimensional arrays.

2 Cartesian Tree Review

Given a binary tree T, let $d(v)$ denote the degree (i.e., number of children) of node v, and $p(v)$ denote the parent of v. We define the rank $r(v)$ to be the inorder rank of the node v in the binary tree T. Define the *range* of a node v to be the range $[e_1(v), e_2(v)]$, where $e_1(v)$ (resp. $e_2(v)$) is the inorder rank of the leftmost (resp. rightmost) descendant of v.

[4] https://oeis.org/

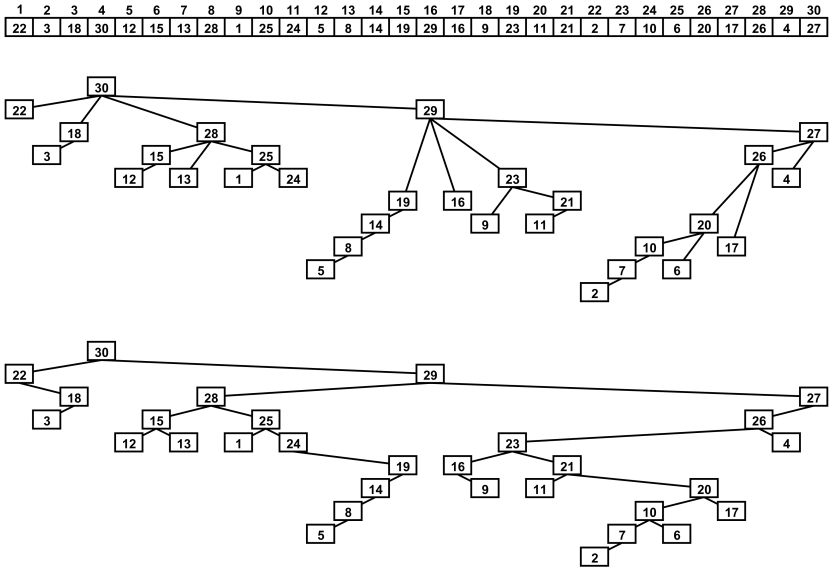

Fig. 1. Top: an array containing a permutation of $\{1, \ldots, 30\}$. Middle: The tree structure of the NNLV problem. Here the parent of a node represents its NNLV, breaking ties by selecting the element on the right (rule I). Bottom: The Cartesian tree.

Suppose we are given an array $A[1..n]$ which stores an n element permutation π, i.e., $A[i] = \pi(i)$. The Cartesian tree of $A[1..n]$ is the n node binary tree T such that the root v of T has rank $r(v) = \arg\max_i A[i]$. If $r(v) > 1$, then the left child of v is the Cartesian tree of $A[1..r(v) - 1]$, otherwise it has no left child. If $r(v) < n$ then the right child of v is the Cartesian tree of $A[r(v)+1..n]$, otherwise it has no right child. We give an example of these definitions in Figure 1.

We require the following technical lemma about Cartesian trees:

Lemma 1. *Consider a node v in a Cartesian tree having range $[e_1(v), e_2(v)]$. If $e_1(v) - 1 \geq 1$ then $A[e_1(v) - 1] > A[r(v)]$. Similarly, if $e_2(v) + 1 \leq n$ then $A[e_2(v) + 1] > A[r(v)]$.*

3 A Path Compressed Tree Representation

Consider an arbitrary binary tree T with n nodes. All binary trees we discuss are rooted. We next describe a path compressed encoding of such a tree that occupies no more than $2n + \Theta(\lg n)$ bits.

We identify all maximal chains $v_1, ..., v_\ell, v_{\ell+1}$ such that:

1. Either v_1 is the root of T, or $d(p(v_1)) = 2$;
2. $d(v_i) = 1$ for $i \in [1, \ell]$, and;
3. $d(v_{\ell+1}) \in \{0, 2\}$.

We refer to $v_{\ell+1}$ as the *terminal* of the chain. Iteratively, we remove each such maximal chain: i.e., the nodes $v_1, ..., v_\ell$ are removed from the tree. If v_1 was the root, then $v_{\ell+1}$ is set to be the new root. Otherwise, $v_{\ell+1}$ is set to be the left (resp. right) child of $p(v_1)$ iff v_1 was the left (resp. right) child of $p(v_1)$. We call the chain *left hanging* if $p(v_1)$ had v_1 as a left child, and *right hanging* otherwise. After removing all such maximal chains, the tree T' that remains is a full binary tree and has $n' \leq n$ nodes. Suppose that we have removed k nodes, for some $k \in [0, n-1]$, and so $n = n' + k$.

Suppose there are m maximal chains removed during the process just described. We now describe the representation of the original tree T.

- We store the tree T', which is a full binary tree and requires $n' + O(1)$ bits to represent.
- We store a bitvector B of length n'. Bit $B[i] = 1$ iff the node v, corresponding to the i-th node in an inorder traversal of T', is the terminal of a removed chain. This requires $\lceil \lg \binom{n'}{m} \rceil$ bits.
- Suppose we order the subset of nodes that are terminals by their inorder rank, and that v is the terminal ordered i-th. We refer to the chain having v as its terminal as $\mathcal{C}_i$, and its length as c_i. We store a bitvector L of length k, which represents the lengths of each removed chain; i.e., the values $c_1, ..., c_m$. Let $p_i = \sum_{j=1}^{i} c_i$ for $i \in [1, m]$. Then $L[p_i] = 1$ for $i \in [1, m]$, and all other entries of L are 0. As L is a bit sequence of length k with m one bits, it can be stored using $\lceil \lg \binom{k}{m} \rceil$ bits.
- For each chain $\mathcal{C}_i = \{v_1, ..., v_{c_i}\}$ having terminal node v_{c_i+1}, we store a bitvector Z_i of length c_i, in which $Z_i[j] = 0$ if v_{j+1} is the left child of v_j, and $Z_i[j] = 1$ otherwise. Let Z be the concatenation of each Z_i, $i \in [1, m]$ and is of length k. We store Z naively using k bits.

We call the above data structures, bitvectors B, L, Z and the tree T' the *path compressed* representation of T. Note that to decode this and recover the tree T, we require the value of n and n'. These can be stored using an additional $\Theta(\lg n)$ bits. By summing the above space costs, we get the following lemma.

Lemma 2. *The path compressed representation of T completely describes the combinatorial structure of T, and can be stored using $n' + \lg \binom{n'}{m} + \lg \binom{k}{m} + k + \Theta(\lg n) \leq 2n' + 2k + \Theta(\lg n) = 2n + \Theta(\lg n)$ bits.*

4 Encoding Nearest Larger Values

In this section we show how to use the path compressed tree representation to compress Cartesian trees—losing some information in the process—but still retaining the ability to answer NNLV queries. Our key observation is that chains in the Cartesian tree can be compressed to save space, as illustrated by the following lemma:

Lemma 3. *Consider the set of all possible chains with c_i deleted nodes in a path compressed representation of a Cartesian tree, excluding chains having nodes representing array elements $A[1]$ or $A[n]$. There are exactly c_i+1 combinatorially distinct chains with respect to answering nearest larger value queries, breaking ties according to rule I.*

Proof. Consider a chain with c_i deleted nodes, $\{v_1, ..., v_{c_i}\}$, where v_{c_i+1} is the terminal. Clearly, v_1 represents the maximum element in the chain, and either $r(v_j) = e_1(v_j)$ or $r(v_j) = e_2(v_j)$ for each $j \in [1, c_i]$. This follows because since v_j is in a chain it is either the left or right endpoint of the range $[e_1(v_j), e_2(v_j)]$. In turn, this implies that the range $[e_1(v_1), e_2(v_1)]$ has a *deleted prefix* and *deleted suffix* which in total contain the inorder ranks of the c_i deleted nodes.

The deleted nodes corresponding to this prefix (resp. suffix) appear contiguously in the array A, and form a decreasing (resp. increasing) run of values in A. Furthermore, by Lemma 1, and since $1, n \notin [e_1(v_1), e_2(v_1)]$ (by the assertion in the statement of the lemma), we can assert that both $A[e_1(v_1) - 1] > A[e_1(v_1)]$ and $A[e_2(v_1) + 1] > A[e_2(v_1)]$. Thus, for each k such that v_k is in the prefix we have that $A[e_1(v_k) - 1] > A[e_1(v_k)]$, and we can return the nearest larger value of $r(v_k) = e_1(v_k)$ to be $e_1(v_k) - 1$. Similarly, for each k such that v_k is in the suffix we have that $A[e_2(v_k)+1] > A[e_2(v_k)]$, and return the nearest larger value of $r(v_k) = e_2(v_k)$ to be $e_2(v_k) + 1$.

This implies that, if we know the value c_i, then we additionally need only know how many nodes are in the prefix in order to determine the answer to a nearest larger value query for any index represented by a deleted node. There are at most $c_i + 1$ possible options: $\{0, ..., c_i\}$. Moreover, for an arbitrary index $i \in [1, n] \setminus [e_1(v_1), e_2(v_1)]$ the answer to a nearest larger value query cannot be in $[e_1(v_1), e_2(v_2)]$, since this range is sandwiched between larger values by Lemma 1. Finally, consider indices in the range $[e_1(v_{c_i+1}), e_2(v_{c_i+1})]$. Using the fact that $A[e_1(v_{c_i+1}) - 1]$ and $A[e_2(v_{c_i+1}) + 1]$ by are larger than all elements in $A[e_1(v_{c_i+1}), e_2(v_{c_i+1})]$ by Lemma 1, we can correctly answer queries for a position i in the subtree. First, we find the solution j within the subtree, and then return the nearest position to i of either j, $e_1(v_{c_i+1}) - 1$, or $e_2(v_{c_i+1}) + 1$, breaking ties according to rule I. □

Recall that to recover a chain of c_i deleted nodes exactly required c_i bits in the path compressed tree representation. The previous lemma allows us to get away with $\lg(c_i + 1)$ bits: an exponential improvement. Using the above lemma, we get the following upper bound for the NNLV problem (note that it does not allow queries to be performed efficiently).

Lemma 4. *The solutions to all nearest larger value queries can be encoded using $n' + \lg \binom{n'}{m} + \lg \binom{k}{m} + m \lg(\frac{k}{m} + 1) + \Theta(\lg n) \leq 1.9198n + \Theta(\lg n)$ bits.*

Proof (Sketch). We store the path compressed version of T, the Cartesian tree of A. However, we replace index Z, by an index Z' consisting of $\lceil \lg \prod_{i=1}^{m} (c_i+1) \rceil$ bits. Z' represents, for each deleted chain—including those that contain nodes

representing $A[1]$ and $A[n]$—the length of its deleted prefix. We explicitly store the answers to nearest larger value queries for $A[1]$ and $A[n]$.

The space bound for storing the data structures described is $n' + \lg \binom{n'}{m} + \lg \binom{k}{m} + \lg \prod_{i=1}^{m}(c_i + 1) + O(\lg n)$ bits. This is bounded by $n' + \lg \binom{n'}{m} + \lg \binom{k}{m} + m \lg(\frac{k}{m} + 1) + O(\lg n)$ bits using Jensen's inequality. Finally, a numerical calculation reveals that this expression is upper bounded by $1.9198n + \Theta(\lg n)$ bits. □

Using a slightly more complicated analysis that bounds the space required to store L in terms of the zeroth-order empirical entropy of the sequence of chain lengths, we can improve the space bound (slightly) to $1.9n + o(n)$, resulting in Theorem 1. We defer details to the full version.

4.1 Supporting Queries

Until now we have only discussed space bounds for encoding NNLV queries, and have made no effort to actually answer them efficiently. In this section we discuss how to support NNLV queries in $O(1)$ time.

In the previous section we showed how to encode a Cartesian tree in a lossy way (losing information about the structure of chains in the tree). Thus, we can view the encoding algorithm, given an input Cartesian tree T, as mapping it to a new tree T_0, in which chains follow a path through a descending run in the prefix, then an ascending run in the suffix, and finally end at a terminal. We call T_0 the *lossy Cartesian tree* in this section. We wish to support the following tree operations on the lossy Cartesian tree T_0:

1. `is_chain_prefix(i)` (resp. `is_chain_suffix(i)`): given i, return whether the node with inorder number i in T_0 is within the prefix (resp. suffix) of a chain. To clarify what we mean by prefix or suffix, refer to Lemma 3.
2. `select_inorder`(i): return the node u in T_0 having inorder number i.
3. `subtree_size(u)`: Return the size of the subtree rooted at node u in T_0.
4. `left`(u) (resp. `right`(u)): return the left (resp. right) child of node u in T_0.

Given the above operations on T_0, we can answer NNLV queries as in Algorithm 1. Correctness of the algorithm follows from the fact that the root of a subtree in T_0 is the largest value in a Cartesian tree, and Lemma 1.

Mini-micro Decomposition All that remains is to show that we can support the operations listed above on the tree T_0. The problem is that we only have space available to store a path compressed version of T_0. Thus, we require a technical modification of the mini-micro tree decomposition presented by Farzan and Munro [5] which can be stated as follows:

Lemma 5 (Theorem 1 [5]). *For any parameter $\alpha > 1$, a tree with n nodes can be decomposed into $\Theta(\frac{n}{\alpha})$ subtrees of size at most 2α, which are pairwise disjoint aside from their roots. With the exception of edges branching from the root of a subtree, there is at most one edge from a non-root node in a subtree to a node outside the subtree.*

Algorithm 1 Computing `NNLV`(A, i).

```
1: if i = 1 or i = n then
2:     return explicitly stored answer for A[1] or A[n].
3: else if is_chain_prefix(i) then
4:     return i − 1
5: else if is_chain_suffix(i) then
6:     return i + 1
7: else
8:     ℓ ← subtree_size(left(select_inorder(i)))
9:     r ← subtree_size(right(select_inorder(i)))
10:    if ℓ < r and i − ℓ − 1 ≥ 1 then
11:        return i − ℓ − 1
12:    else if i + r + 1 ≤ n then
13:        return i + r + 1
14:    else
15:        return A[i] is the maximum (it has no NNLV)
16:    end if
17: end if
```

The binary tree structure of Davoodi et al. [4] essentially applies Lemma 5 twice to the input tree, getting a set of $O(\frac{n}{\lg^2 n})$ mini-trees of size $O(\lg^2 n)$ and $O(\frac{n}{\lg n})$ micro-trees of size $\lceil \frac{\lg n}{\gamma} \rceil$, for some $\gamma \geq 8$. Since a rooted binary tree with g nodes can be represented using $2g$ bits, we can store a *fingerprint* of size at most $\lceil \frac{2 \lg n}{\gamma} \rceil$ bits for each micro-tree. We can then perform tree operations by using these fingerprints to index into using a universal table of size $o(n)$. Overall, the space is bounded by the sum of the sizes of the fingerprints, and totals $2n + o(n)$. Their representation supports a large number of operations, which includes `select_inorder`, `subtree_size`, `left`, `right`.

The main idea of our approach is to take the lossy Cartesian tree T_0, and to decompose it using Lemma 5. We then adjust the decomposition to, roughly speaking, ensure that chains do not cross subtree boundaries. The following technical lemma captures this intution:

Lemma 6. *For any parameter $\alpha > 1$, a tree with n nodes can be decomposed into $\Theta(\frac{n}{\alpha})$ subtrees which are pairwise disjoint aside from their roots. Furthermore, we have the following properties for the subtrees:*

1. *All nodes in a chain, except possibly the terminal, are contained in the same subtree.*
2. *If a subtree contains a node of degree two, then it has size at most 2α.*
3. *Excepting edges branching from the root of a subtree, there is at most one edge from a non-root node in a subtree to a node outside the subtree.*

We apply the Lemma 6 twice to T_0. The first application has parameter $\alpha = \lceil \lg^2 n \rceil$, which gives us a set of subtrees. We change and extend the definitions of mini-trees and micro-trees slightly from the previous papers. Subtrees which have at least one degree two node are referred to as mini-trees, and are

otherwise referred to as *mini-chains*. The second application of the lemma is done to each mini-tree separately with $\alpha = \lceil \frac{\lg n}{\beta} \rceil$, for $\beta \geq 16$. Similarly, the resultant subtrees are called micro-trees if they contain a degree two node, and *micro-chains* otherwise.

Next, we apply path compression to each micro-tree, micro-chain, and mini-chain. We note that micro-chains and mini-chains end up as a single node after path compression, and have degree 1. Furthermore, prior to path compression, micro-chains were chains of length at least $\lceil \frac{\lg n}{\beta} \rceil$ and at most $\Theta(\lg^2 n)$, and mini-chains were chains of length at least $\Theta(\lg^2 n)$. For micro-trees, each node (after path compression) has either degree two or zero. Each micro-tree which contains g degree two nodes can therefore be represented using g bits, rather than $2g$ bits. Recall that we used n' to represent the number of nodes in the path compressed lossy Cartesian tree. If we sum over all the micro-trees there are $\frac{n'-1}{2}$ degree two nodes in total after path compression. This means the sum of the sizes of the fingerprints for all of the micro-trees can be stored in $n' + o(n)$ bits. One technical issue is that we must mark the branching edge of each micro-tree, which can be done using an additional $\Theta(\lg \lg n)$ bits per micro-tree. Thus, this additional cost is $O(\frac{n \lg \lg n}{\lg n})$ when summed over all micro-trees. Note that the number of micro- and mini-chains is bounded by $\Theta(\frac{n}{\lg n})$, so we can also afford to mark these using a bit vector, indicating whether they are a micro-chain or a mini-chain. Recalling the encoding from the previous section, this path compressed tree we have constructed here is almost (but not quite) the path compressed version of the lossy Cartesian tree T_0: it has $\Theta(\frac{n}{\lg n})$ additional degree one nodes, but nonetheless occupies $n' + o(n)$ bits.

We call the fingerprints of the micro-trees the *path compressed fingerprints*. In the full version, we show that for an arbitrary micro-tree M we can use the path compressed fingerprint to recover the fingerprint corresponding to M the original (not path compressed) tree T_0. We have the following lemma:

Lemma 7. *We can recover the fingerprint of any micro-tree in T_0 in $O(1)$ time, using space:*

$$n' + \lg \binom{n'}{m} + \sum_{i=1}^{\sigma} \left(m_i \lg \frac{m(i+1)}{m_i} \right) + O\left(\frac{n \lg \lg n}{\lg n} \right) \quad \textit{bits.}$$

Using the previous lemma, it is not hard to prove Theorem 1. The main idea is to construct the data structure of Davoodi et al. [4] using Lemma 7 as an oracle to access the fingerprints of micro-trees. This allows us to support nearly all the required query operations, except `is_chain_prefix` and `is_chain_suffix`. These two operations can be supported by considering the cases of micro-trees, micro-chains, and mini-chains separately.

5 Lower Bound

The main idea of the lower bound is to show that for a given n, there are many configurations of A that can be distinguished by NNLV queries. To do this, we

define a *restricted* NNLV problem (RNNLV). The restricted problem is like the original NNLV problem on an array $A[1..n]$, except we pretend that the array has entries $A[0] = \infty$ and $A[n+1] = \infty$. Thus, an answer to the restricted NNLV query ($\texttt{RNNLV}(A, i)$) is either $\texttt{NNLV}(A, i)$, 0, or $n + 1$: we choose the nearest of these three possibilities, breaking ties using rule I. This restricts the solution space, but will allow us to lower bound the unrestricted problem.

For an n element array, we use R_n to denote the number of different solutions to RNNLV, and S_n to denote the number of solutions to NNLV, both subject to tie breaking rule I. We computed the following sequences:

Table 2. Number of solutions to RNNLV problem (rule I).

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
R_n	1	2	4	9	22	55	142	378	1015	2768	7662	21340	59962	169961
S_n	1	2	5	14	40	116	341	1010	3009	9012	27087	81658	246841	747728

Next we discuss how to use Table 2 to derive a lower bound. Consider an array of length n, for n sufficiently large. Without loss of generality, we assume that a parameter $\beta \geq 1$ divides $n - 2$ and that $\frac{n-2}{\beta}$ is odd. Let D_i denote the i-th odd block, and E_i denote the i-th even block. Locations $A[1]$ and $A[n]$ are assigned values $n - 1$ and n, respectively. Odd block D_i is assigned values $[(i-1)\beta + 1, i\beta]$, and can be arranged in one of R_β configurations, to form an instance of the RNNLV problem. Suppose there are Δ odd blocks. Even block E_i will be assigned values from $[(\Delta + i - 1)\beta + 1, (\Delta + i)\beta]$, and arranged in one of the S_β configurations of the NNLV problem.

Our claim is that each even (resp. odd) block can be assigned any of the S_β (resp. R_β) possible configurations, without interference from other blocks. To see this, consider that for each even block we have assigned values so that—with the exception of the maximum element—the nearest larger value to all elements must be within the same block. This follows since the adjacent odd blocks contain strictly smaller values than those in any even block. Moreover, for odd blocks, the values immediately to the left and right of the block are strictly larger than any values in the block. Thus, we can force the global solution to the NNLV problem on the entire array into at least $(S_\beta R_\beta)^{\frac{n-2}{2\beta}}$ distinct structures. This implies that $\lg S_n$ is at least $\frac{(n-2)}{2\beta} \lg(S_\beta R_\beta)$: selecting $\beta = 14$ yields the lower bound of Theorem 2.

6 Conclusions

We have introduced the encoding NNLV problem, and have noted its combinatorial richness. Using a novel path-compressed representation of Cartesian trees, we gave a space-efficient NNLV encoding that supports queries in $O(1)$ time. Determining the effective entropy of NNLV, and to consider the other NNLV variants, is an open problem, as is extending the path-compressed Cartesian

tree representation of Section 4.1 to general binary trees. Finding ways to apply NNLV encodings to compressed suffix trees, as Fischer [6] did for his bidirectional NLV encoding, would also be interesting.

References

1. Asano, T., Bereg, S., Kirkpatrick, D.G.: Finding nearest larger neighbors. In: Albers, S., Alt, H., Näher, S. (eds.) Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 5760, pp. 249–260. Springer (2009), http://dx.doi.org/10.1007/978-3-642-03456-5_17
2. Asano, T., Kirkpatrick, D.G.: Time-space tradeoffs for all-nearest-larger-neighbors problems. In: Dehne, F., Solis-Oba, R., Sack, J. (eds.) Algorithms and Data Structures - 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8037, pp. 61–72. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40104-6_6
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), http://dx.doi.org/10.1006/jagm.1993.1018
4. Davoodi, P., Navarro, G., Raman, R., Rao, S.: Encoding range minima and top-2 queries. Phil. Trans. R. Soc. A 372(2016), 1471–2962 (2014)
5. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica 68(1), 16–40 (2014), http://dx.doi.org/10.1007/s00453-012-9664-0
6. Fischer, J.: Combined data structure for previous- and next-smaller-values. Theor. Comput. Sci. 412(22), 2451–2456 (2011), http://dx.doi.org/10.1016/j.tcs.2011.01.036
7. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
8. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comput. Sci. 410(51), 5354–5364 (2009)
9. Golin, M.J., Iacono, J., Krizanc, D., Raman, R., Rao, S.S.: Encoding 2d range maximum queries. In: ISAAC. pp. 180–189 (2011)
10. Jayapaul, V., Jo, S., Raman, V., Satti, S.R.: Space efficient data structures for nearest larger neighbor. In: Proc. IWOCA 2014 (2014), to appear.
11. Jo, S., Raman, R., Satti, S.R.: Compact encodings and indexes for the nearest larger neighbor problem. In: Rahman, M.S., Tomita, E. (eds.) WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8973, pp. 53–64. Springer (2015), http://dx.doi.org/10.1007/978-3-319-15612-5_6