On Cartesian Trees and Range Minimum Queries

Erik D. Demaine[⋆1] Gad M. Landau[⋆⋆2,3] and Oren Weimann[∗1]

[1] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA.
{edemaine,oweimann}@mit.edu
[2] Department of Computer Science, University of Haifa, Haifa, Israeal
[3] Department of Computer and Information Science, Polytechnic Institute of NYU.
landau@cs.haifa.ac.il

Abstract.
We present new results on Cartesian trees with applications in range minimum queries and bottleneck edge queries. We introduce a cache-oblivious Cartesian tree for solving the range minimum query problem, a Cartesian tree of a tree for the bottleneck edge query problem on trees and undirected graphs, and a proof that no Cartesian tree exists for the two-dimensional version of the range minimum query problem.

Key words: Range minimum query, lowest common ancestor, Cartesian tree, cache-oblivious, bottleneck paths, minimum spanning tree verification, partial sums.

1 Introduction

In the *Range Minimum Query* (RMQ) problem, we wish to preprocess an array A of n numbers for subsequent queries asking for $\min\{A[i], \ldots, A[j]\}$. In the two-dimensional version of RMQ, an $n \times n$ matrix is preprocessed and the queries ask for the minimum element in a given rectangle. The *Bottleneck Edge Query* (BEQ) problem further generalizes RMQ to graphs. In this problem, we preprocess a graph for subsequent queries asking for the maximum amount of flow that can be routed between some vertices u and v along any single path. The capacity of a path is captured by its edge with minimal capacity (weight). Thus, RMQ can be seen as a special case of BEQ on line-like graphs.

In all solutions to the RMQ problem the *Cartesian tree* plays a central role. Given an array A of n numbers its Cartesian tree is defined as follows: The root of the Cartesian tree is $A[i] = \min\{A[1], \ldots, A[n]\}$, its left subtree is computed recursively on $A[1], \ldots, A[i-1]$ and its right subtree on $A[i+1], \ldots, A[n]$. In this paper we present new results on Cartesian trees with applications in RMQ and BEQ. We introduce a cache-oblivious version of the Cartesian tree that leads to an optimal cache-oblivious RMQ solution. We then give a natural generalization of the Cartesian tree from arrays to trees and show how it can be used for solving BEQ on undirected graphs and on trees. Finally, we show that there is no two-dimensional generalization of a Cartesian tree.

Range Minimum Queries and Lowest Common Ancestors. The traditional RMQ solutions rely on the tight connection between RMQ and the *Lowest Common Ancestor* (LCA) problem. In LCA, we wish to preprocess a rooted tree T for subsequent queries asking for the common ancestor of two nodes that is located farthest from the root. RMQ and LCA were shown by Gabow*et al.* [18] to be equivalent in the sense that either one can be reduced in linear time to the other. An LCA instance can be obtained from an RMQ instance on an array A by letting T be the Cartesian tree of A that can be constructed in linear time [18]. It

⋆ Supported in part by the Center for Massive Data Algorithmics (MADALGO), a center of the Danish National Research Foundation.
⋆⋆ Supported in part by the Israel Science Foundation grant 35/05, the Israel-Korea Scientific Research Cooperation and Yahoo.

is easy to see that RMQ(i, j) in A translates to LCA$(A[i], A[j])$ in A's Cartesian tree. In the other direction, an RMQ instance A can be obtained from an LCA instance on a tree T by writing down the *depths* of the nodes visited during an *Euler tour* of T. That is, A is obtained by listing all first and last node-visitations in a DFS traversal of T that starts from the root. The LCA of two nodes translates to a range minimum query between the first occurrences of these nodes in A. An important property of the array A is that the difference between any two adjacent cells is ± 1.

As the classical RMQ and LCA problems are equally hard, it is tempting to think that this is also the case in the cache-oblivious model, where the complexity is measured in terms of the number of memory-blocks transferred between cache and disk (see Section 2 for a description of the cache-oblivious model). Indeed, to solve RMQ on array A, one could solve LCA on the Cartesian tree of A that can be constructed optimally using $O(\frac{n}{B})$ memory transfers (where B is the block-transfer size). An optimal $O(\frac{n}{B})$ cache-oblivious LCA solution follows directly from [9] after obtaining the Euler tour of the Cartesian tree. However, the best cache-oblivious algorithm for obtaining the Euler tour requires $\Theta(\frac{n}{B}\log_{M/B}\frac{n}{B})$ memory transfers [6] where M is the cache-size. The first result of our paper is an optimal cache-oblivious RMQ data structure that requires $O(\frac{n}{B})$ memory transfers and gives constant-time queries. This makes RMQ cache-obliviously easier than LCA (unless the LCA instance is given in Euler tour form).

Bottleneck Edge Queries on Graphs and Trees. Given an edge weighted graph, the *bottleneck edge e* between a pair of vertices s, t is defined as follows: If P is the set of all simple paths from s to t then e's weight is given by $\max_{p \in P}$(lightest edge in p). In the BEQ problem, we wish to preprocess a graph for subsequent bottleneck edge queries. Hu [20] proved that in undirected graphs bottleneck edges can be obtained by considering only the unique paths in a maximum spanning tree. This means that BEQ on undirected graphs can be solved by BEQ on trees. Another reason for the importance of BEQ on trees is the equivalent *online minimum spanning tree verification* problem. Given a spanning tree T of some edge weighted graph G, the problem is to preprocess T for queries verifying if an edge $e \in G - T$ can replace some edge in T and decrease T's weight. It is easy to see that this is equivalent to a bottleneck edge query between e's endpoints.

The second result in our paper is a natural generalization of the Cartesian tree from arrays to trees. We show that a Cartesian tree of a tree can be constructed in linear time plus the time required to sort the edges-weights. It can then be used to answer bottleneck edge queries in constant time for trees and undirected graphs. We further show how to maintain this Cartesian tree and constant-time bottleneck edge queries while leaf insertions and deletions are preformed on the input tree. Insertions and deletions require $O(\log n)$ amortized time and $O(\log\log u)$ amortized time for integral edge weights bounded by u.

Two-dimensional RMQ. In the two-dimensional version of RMQ, we wish to preprocess an $n \times n$ matrix for subsequent queries asking for the minimum element in a given rectangle. Amir, Fischer, and Lewenstein [5] conjectured that it should be possible to show that in two dimensions there is no such nice relation as the one between RMQs and Cartesian Trees in the one-dimensional case. Our third result proves this conjecture to be true by proving that the number of *different* RMQ matrices is roughly $(n^2)!$, where two RMQ matrices are different if their range minimum is in different locations for some rectangular range.

1.1 Relation to Previous Work

Practically all known solutions to the range minimum query (RMQ) problem, its two-dimensional version (2D-RMQ), and its bottleneck edge version (BEQ) on trees share the same high-level description. They all partition the problem into some n/s smaller subproblems of size s each. From each of the small subproblems, one representative (the minimal element of the subproblem) is chosen and the problem is solved recursively on the n/s representatives. A similar recursion is applied on each one of the small subproblems. Besides different choices of s, the main difference between these solutions is the recursion's halting conditions that can take one of the following forms.

(I) *keep recursing* - the recursive procedure is applied until the subproblem is of constant size.
(II) *handle at query* - for small enough s do nothing and handle during query-time.

(III) *sort* - for small enough s use a linear-space $O(s \log s)$-time solution with constant query time.
(IV) *table-lookup* - for roughly logarithmic size s construct a lookup-table for all possible subproblems.
(V) *table-lookup* & *handle at query* - for roughly logarithmic size s construct a lookup-table. A query to this table will return a fixed number of candidates to be compared during query-time.

Table 1.1 describes how our solutions (in bold) and the existing solutions relate to the above four options. We next describe the existing solutions in detail.

	RMQ	BEQ on trees	2D-RMQ
keep recursing		$\langle n\alpha_k(n), n\alpha_k(n), k\rangle$ [3]	$\langle n^2\alpha_k(n)^2, n^2\alpha_k(n)^2, k\rangle$ [13]
handle at query		$\langle n, n, \alpha(n)\rangle$ [3]	$\langle n^2, n^2, \alpha^2(n)\rangle$ [13]
sort		$\boldsymbol{\langle n \log^{[k]} n, n, k\rangle}$	$\langle n^2 \log^{[k]} n, n^2, k\rangle$ [5]
table-lookup	$\langle n, n, 1\rangle$ [19] $\boldsymbol{\langle n/B, 1\rangle}$	impossible [24]	**impossible**
table-lookup & *handle at query*		impossible [24]	$\langle n^2, n^2, 1\rangle$ [7]

Table 1. Our results (in bold) and their relation to the existing solutions. We denote a solution by ⟨*preprocessing time, space, query time*⟩ and by ⟨*memory transfers in preprocessing, memory transfers in query*⟩ for a cache-oblivious solution.

RMQ and LCA. As we mentioned before, RMQ can be solved by solving LCA. Harel and Tarjan [19] were the first to show that the LCA problem can be optimally solved with linear-time preprocessing and constant-time queries by relying on word-level parallelism. Their data structure was later simplified by Schieber and Vishkin [28] but remained rather complicated and impractical. Berkman and Vishkin [10], and then Bender *et al.* [9] presented further simplifications and removed the need for word-level parallelism by actually reducing the LCA problem back to an RMQ problem. This time, the RMQ array has the property that any two adjacent cells differ by ± 1. This property was used in [9, 10, 16] to enable table lookups as we now explain.

It is not hard to see that two arrays that admit the same ± 1 vector have the same location of minimum element for every possible range. Therefore, it is possible to compute a lookup-table P storing the answers to all range minimum queries of all possible ± 1 vectors of length $s = \frac{1}{2}\log n$. Since there are $O(s^2)$ possible queries, the size of P is $O(s^2 \cdot 2^s) = o(n)$. Fischer and Heun [16] recently presented the first optimal RMQ solution that makes no use of LCA algorithms or the ± 1 property. Their solution uses the Cartesian tree but in a different manner. It uses the fact that the number of *different*[4] RMQ arrays is equal to the number of possible Cartesian trees and thus to the Catalan number which is $\frac{1}{s+1}\binom{2s}{s} = O(\frac{4^s}{s^{1.5}})$. This means that if we pick $s = \frac{1}{4}\log n$ then again the lookup-table P requires only $O(s^2 \cdot \frac{4^s}{s^{1.5}}) = O(n)$ space.

BEQ. On directed edge weighted graphs, the BEQ problem has been studied in its offline version, where we need to determine the bottleneck edge for every pair of vertices. Pollack [27] introduced the problem and showed how to solve it in $O(n^3)$ time. Vassilevska, Williams, and Yuster [31] gave an $O(n^{2+\omega/3})$-time algorithm, where ω is the exponent of matrix multiplication over a ring. This was recently improved by Duan and Pettie [14] to $O(n^{(3+\omega)/2})$. For the case of vertex weighted graphs, Shapira *et al.* [30] gave an $O(n^{2.575})$-time algorithm.

On trees, the BEQ problem was studied when the tree T is a spanning tree of some graph and a bottleneck edge query verifies if an edge $e \notin T$ can replace some edge in T and decrease T's weight. A celebrated result

[4] Two RMQ arrays are different if their range minimum is in different locations for some range.

of Komlós [23] is a linear-time algorithm that verifies all edges in G. The idea of progressively improving an approximately minimum solution T is the basis of all recent minimum spanning tree algorithms [12, 21, 25, 26]. Alon and Schieber [3] show that after an almost linear $O(n \cdot \alpha_k(n))$ preprocessing time and space bottleneck edge queries can be done in constant time for any fixed k. Here, $\alpha_k(n)$ is the inverse of the k^{th} row of Ackerman's function[5]: $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$ so that $\alpha_1(n) = n/2$, $\alpha_2(n) = \log n$, $\alpha_3(n) = \log^* n$, $\alpha_4(n) = \log^{**} n$ and so on[6]. Pettie [24] gave a tight lower bound of $\Omega(n \cdot \alpha_k(n))$ preprocessing time required for $O(k)$ query time. Alon and Schieber further prove that if optimal $O(n)$ preprocessing space is required, it can be done with $\alpha(n)$ query time.

If the edge-weights are already sorted, our solution is better than the one of [3]. For arbitrary unsorted edge-weights, we can improve [3] in terms of space complexity. Namely, we give a linear space and constant query time solution that requires $O(n \log^{[k]} n)$ preprocessing time for any fixed k, where $\log^{[k]} n$ denotes the iterated application of k logarithms.

2D-RMQ. For the two-dimensional version of RMQ on an $n \times n$ matrix, Gabow, Bentley and Tarjan [18] suggested an $O(n^2 \log n)$ preprocessing time and space and $O(\log n)$ query time solution. This was improved by Chazelle and Rosenberg [13] to $O(n^2 \cdot \alpha_k(n)^2)$ preprocessing time and space and $O(1)$ query time for any fixed k. Chazelle and Rosenberg further prove that if optimal $O(n^2)$ preprocessing space is required, it can be done with $\alpha^2(n)$ query time. Amir, Fischer, and Lewenstein [5] showed that $O(n^2)$ space and constant query time can be obtained by allowing $O(n^2 \log^{[k]} n)$ preprocessing time for any fixed k.

Amir, Fischer, and Lewenstein also conjectured that in two dimensions there is no such nice relation as the one between the number of different RMQs and the number of different Cartesian Trees in the one-dimensional case. We prove this conjecture to be true thereby showing that $O(n^2)$ preprocessing time and constant query time can not be achieved using the existing methods for one-dimensional RMQ. Indeed, shortly after we proved this, Atallah and Yuan (in a yet unpublished result [7]) discovered a new optimal RMQ solution that does not use Cartesian trees and extends to two dimensions.

2 A Cache-Oblivious Cartesian Tree

While modern memory systems consist of several levels of cache, main memory, and disk, the traditional RAM model of computation assumes a flat memory with uniform access time. The I/O-model, developed by Aggarwal and Vitter [2], is a two-level memory model designed to account for the large difference in the access times of cache and disks. In this model, the disk is partitioned into blocks of B elements each, and accessing one element on disk copies its entire block to cache. The cache can store up to M/B blocks, for a total size of M. The efficiency of an algorithm is captured by the number of block transfers it makes between the disk and cache.

The cache-oblivious model, introduced by Frigo *et al.* [17], extends the I/O-model to a multi-level memory model by a simple measure: the algorithm is not allowed to know the value of B and M. More precisely, a cache-oblivious algorithm is an algorithm formulated in the standard RAM model, but analyzed in the I/O-model, with an analysis valid for any value of B and M and between any two adjacent memory-levels. When the cache is full, a cache-oblivious algorithm can assume that the *ideal* block in cache is selected for replacement based on the future characteristics of the algorithm, that is, an *optimal* offline paging strategy is assumed. This assumption is fair as most memory systems (such as LRU and FIFO) approximate the omniscient strategy within a constant factor. See [17] for full details on the cache-oblivious model.

It is easy to see that the number of memory transfers needed to read or write n contiguous elements from disk is $scan(n) = \Theta(\frac{n}{B})$, even if B is unknown. A *stack* that is implemented using a doubling array can

[5] We follow Seidel [29]. The function $\alpha(\cdot)$ is usually defined slightly differently, but all variants are equivalent up to an additive constant.

[6] $\log^{**} n$ is the number of times $\log^*$ function is applied to n to produce a constant, $\alpha_k(n) = \log^{\overbrace{**\cdots*}^{\text{k times}}} n$ and the inverse Ackerman function $\alpha(n)$ is the smallest k such that $\alpha_k(n)$ is a constant.

support n push\pop operations in $scan(n)$ memory transfers. This is because the optimal paging strategy can always keep the last block of the array (accessed by both push and pop) in cache. Other optimal cache-oblivious data structures have been recently proposed, including priority queues [6], B-trees [8], string dictionaries [11], kd-trees and Range trees [1]. An important result of Frigo *et al.* shows that the number of memory transfers needed to sort n elements is $sort(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

For the RMQ problem on array A, the Cartesian tree of A can be constructed optimally using $scan(n)$ memory transfers by implementing the following construction of [18]. Let C_i be the Cartesian tree of $A[1, \ldots, i]$. To build C_{i+1}, we notice that node $A[i+1]$ will belong to the rightmost path of C_{i+1}, so we climb up the rightmost path of C_i until we find the position where $A[i+1]$ belongs. It is easy to see that every "climbed" node will be removed from the rightmost path and so the total time complexity is $O(n)$. A cache-oblivious stack can therefore maintain the current rightmost path and the construction outputs the nodes of the Cartesian tree in postorder. However, in order to use an LCA data structure on the Cartesian tree we need an Euler tour order and not a postorder. The most efficient way to obtain an Euler tour [6] requires $sort(n)$ memory transfers. Therefore $sort(n)$ was until now the upper bound for cache-oblivious RMQ. In this section we prove the following result.

Theorem 1. *An optimal RMQ data structure with constant query-time can be constructed using $scan(n)$ memory transfers.*

We start by showing a simple constant-time RMQ data structure [9] that can be constructed using $scan(n) \log n$ memory transfers. The idea is to precompute the answers to all range minimum queries whose length is a power of two. Then, to answer $RMQ(i,j)$ we can find (in constant time) two such overlapping ranges that exactly cover the interval $[i,j]$, and return the minimum between them. We therefore wish to construct arrays $M_0, M_1, \ldots, M_{\log n}$ where $M_j[i] = \min\{A[i], \ldots, A[i+2^j-1]\}$ for every $i = 1, 2, \ldots, n$. M_0 is simply A. For $j > 0$, we construct M_j by doing two parallel scans of M_{j-1} using $scan(n)$ memory transfers (we assume $M \geq 2B$). The first scan starts at $M_{j-1}[1]$ and the second at $M_{j-1}[1+2^{j-1}]$. During the parallel scan we set $M_j[i] = \min\{M_{j-1}[i], M_{j-1}[i+2^{j-1}]\}$ for every $i = 1, 2, \ldots, n$.

After describing this $scan(n) \log n$ solution, we can now describe the $scan(n)$ solution. Consider the partition of A into disjoint intervals (blocks) of $s = \frac{1}{4} \log n$ consecutive elements. The *representative* of every block is the minimal element in this block. Clearly, using $scan(n)$ memory transfers we can compute an array of the n/s representatives. We use the RMQ data structure above on the representatives array. This data structure is constructed with $scan(n/s) \log(n/s) = scan(n)$ memory transfers and is used to handle queries whose range spans more than one block. The additional in-block prefix and suffix of such queries can be accounted for by pre-computing $RMQ(i,j)$ of every block prefix or suffix. This again can easily be done using $scan(n)$ memory transfers.

We are therefore left only with the problem of answering queries whose range is entirely inside one block. Recall that two RMQ arrays are *different* if their range minima are in different locations for some range. Fischer and Heun [16] observed that the number of different blocks is equal to the number of possible Cartesian trees of s elements and thus to the s'th Catalan number which is $o(4^s)$. For each such unique block type, the number of possible in-block ranges $[i,j]$ is $O(s^2)$. We can therefore construct an $s^2 \times 4^s$ lookup-table P of size $O(s^2 \cdot 4^s) = o(n)$ that stores the locations of all range minimum queries for all possible blocks.

It remains to show how to index table P (i.e. how to identify the type of every block in the partition of A) and how to construct P using $scan(n)$ memory transfers. We begin with the former. The most naive way to calculate the block types would be to actually construct the Cartesian tree of each block in A, and then use an inverse enumeration of binary trees [22] to compute its type. This approach however can not be implemented via scans. Instead, consider the Cartesian tree *signature* of a block as the sequence $\ell_1 \ell_2 \cdots \ell_s$ where $0 \leq \ell_i < s$ is the number of nodes removed from the rightmost path of the block's Cartesian tree when inserting the i'th element. For example, the block "3421" has signature "0021".

Notice that for every signature $\ell_1 \ell_2 \cdots \ell_s$ we have $\sum_{k=1}^{i} \ell_k < i$ for every $1 \leq i \leq s$. This is because one cannot remove more elements from the rightmost path than one has inserted before. Fischer and Heun used this property to identify each signature by a special sum of the so-called *Ballot Numbers* [22]. We suggest a simpler and cache-oblivious way of computing a unique number $f(\ell_1 \ell_2 \cdots \ell_s) \in \{0, 1, \ldots, 4^s - 1\}$ for every signature $\ell_1\ell_2\cdots\ell_s$. The binary representation of this number is simply

$$\overbrace{11\cdots1}^{\ell_1}0\overbrace{11\cdots1}^{\ell_2}0\cdots0\overbrace{11\cdots1}^{\ell_s}0.$$

Clearly, each signature is assigned a different number and since $\sum_{i=1}^{s}\ell_i < s$ this number is between 0 and $2^{2s}-1$ as its binary representation is of length at most $2s$. Notice that some binary strings of length at most $2s$ (for example, strings starting with 1 or with 011) are not really an $f(\ell_1\ell_2\cdots\ell_s)$ of a valid signature $\ell_1\ell_2\cdots\ell_s$ (i.e. f is not surjective). Using a stack, in one scan of A we can compute the signatures of all blocks in the partition of A in the order they appear. We refer to the sequence of signatures as $S(A)$, this sequence has n/s signatures each of length s. In a single scan of $S(A)$ we can compute $f(\ell_1\ell_2\cdots\ell_s)$ for all signatures in $S(A)$ thus solving our problem of indexing P.

We are left only with showing how to construct P using $scan(n)$ memory transfers. In the non cache-oblivious world (that Fischer and Heun consider) this is easy. We really only need to compute the entries in P that correspond to blocks that actually appear in the partition of A. During a scan of $S(A)$, for each signature $\ell_1\ell_2\cdots\ell_s$ we check if column $f(\ell_1\ell_2\cdots\ell_s)$ in P was already computed (this requires n/s such checks). If not, we can compute all $O(s^2)$ range minima of the block trivially in $O(s)$ time per range. In the cache-oblivious model however, each of the n/s checks might bring a new block to cache. If $s < B$ then this incurs more than $scan(n)$ memory transfers.

Therefore, for an optimal cache-oblivious performance, we must construct the entire table P and not only the columns that correspond to signatures in $S(A)$. Instead of computing P's entries for all possible RMQ arrays of length s we compute P's entries for all possible binary strings of length $2s$. Consider an RMQ block A' of length s with signature $\ell_1\ell_2\cdots\ell_s$. In Fig. 1 we give a simple procedure that computes $RMQ(i,j)$ in A' for any $1 \le i \le j \le s$ by a single scan of $f' = f(\ell_1\ell_2\cdots\ell_s)$. Again, we note that for binary strings f' that are not really an $f(\ell_1\ell_2\cdots\ell_s)$ of a valid signature $\ell_1\ell_2\cdots\ell_s$ this procedure computes “garbage” that will never be queried.

initialize $min \leftarrow i$ and $x \leftarrow 0$
scan f' until the ith 0
for $j' = i+1,\ldots,j$
    continue scanning f' until the next 0
    set $x \leftarrow x + 1 -$ the number of 1's read between the last two 0's
    if $x \le 0$ set $min \leftarrow j'$ and $x \leftarrow 0$
return min as the location of $RMQ(i,j)$

Fig. 1. Pseudocode for computing $RMQ(i,j)$ for some $1 \le i \le j \le s$ using one scan of the binary string $f' = f(\ell_1\ell_2\cdots\ell_s)$ of some signature $\ell_1\ell_2\cdots\ell_s$.

In order to use the procedure of Fig. 1 on all possible signatures, we construct a sequence S of all binary strings of length $2s$ in lexicographic order. S is the concatenation of 4^s substrings each of length $2s$, and S can be written using $scan(2s\cdot4^s) = o(scan(n))$ memory transfers. For correctness of the above procedure, a parallel scan of S can be used to apply the procedure only on those substrings that have exactly s 0's. We can thus compute each row of P using $scan(2s\cdot4^s)$ memory transfers and the entire table P using $s^2\cdot scan(2s\cdot4^s) = o(scan(n))$ memory transfers[7]. This concludes the description of our cache-oblivious RMQ data structure that can be constructed using a constant number of scans.

[7] Since the (unknown) B can be greater than the length of S we don't really scan S for s^2 times. Instead, we scan once a sequence of s copies of S.

3 A Cartesian Tree of a Tree

In this section we address bottleneck edge queries on trees. We introduce the Cartesian tree of a tree and show how to construct it in $O(n)$ time plus the time required to sort the edge weights. Recall that an LCA data structure on the standard Cartesian tree can be constructed in linear time to answer range minimum queries in constant time. Similarly, an LCA data structure on our Cartesian tree can be constructed in linear time to answer bottleneck edge queries in constant time for trees.

Given an edge weighted input tree T, we define its Cartesian tree C as follows. The root r of C represents the edge $e = (u, v)$ of T with minimum weight (ties are resolved arbitrarily). The two children of r correspond to the two connected component of $T - e$: the left child is the recursively constructed Cartesian tree of the connected component containing u, and the right child is the recursive construction for v. Notice that C's internal nodes correspond to T's edges and C's leaves correspond to T's vertices.

Theorem 2. *The Cartesian tree of a weighted input tree with n edges can be constructed in $O(n)$ time plus the time required to sort the weights.*

Decremental connectivity in trees. The proof of Theorem 2 uses a data structure by Alstrup and Spork [4] for decremental connectivity in trees. This data structure maintains a forest subject to two operations: deleting an edge in $O(1)$ amortized time, and testing whether two vertices u and v are in the same connected component in $O(1)$ worst-case time.

The data structure is based on a *micro-macro decomposition* of the input tree T. The set of nodes of T is partitioned into disjoint subsets where each subset induces a connected component of T called the *micro tree*. The division is constructed such that each micro tree is of size $\Theta(\lg n)$ and at most two nodes in a micro tree (the *boundary nodes*) are incident with nodes in other micro trees. The nodes of the *macro tree* are exactly the boundary nodes and it contains an edge between two nodes iff T has a path between the two nodes which does not contain any other boundary nodes.

It is easy to see that deletions and connectivity queries can be preformed by a constant number of deletions and connectivity queries on the micro and macro trees. The macro tree can afford to use a standard $O(\lg n)$ amortized solution [15], which explicitly relabels all nodes in the smaller of the two components resulting from a deletion. The micro trees use simple word-level parallelism to manipulate the logarithmic-size subtrees.

Although not explicitly stated in [4], the data structure can in fact maintain a canonical name (record) for each connected component, and can support finding the canonical name of the connected component containing a given vertex in $O(1)$ worst-case time. The macro structure explicitly maintains such names, and the existing tools in the micro structure can find the highest node (common ancestor) of the connected component of a vertex, which serves as a name. This slight modification enables us to store a constant amount of additional information with each connected component, and find that information given just a vertex in the component.

Proof of Theorem 2. We next describe the algorithm for constructing the Cartesian tree C of an input tree T. The algorithm essentially bounces around T, considering the edges in increasing weight order, and uses the decremental connectivity data structure on T to pick up where it left off in each component. Precisely:

1. initialize the decremental connectivity structure on T. Each connected component has two fields: "parent" and "side".
2. set the "parent" of the single connected component to null.
3. sort the edge weights.
4. for each edge $e = (u, v)$ in increasing order by weight:
   (a) make a vertex w in C corresponding to e, whose parent is the "parent" of e's connected component in the forest, and who is the left or right child of that parent according to the "side" of that component.
   (b) delete edge e from the forest.
   (c) find the connected component containing u and set its "parent" to w and its "side" to "left".
   (d) find the connected component containing v and set its "parent" to w and its "side" to "right".

After sorting the edge weights, this algorithm does $O(n)$ work plus the work spent for $O(n)$ operations in the decremental connectivity data structure, for a total of $O(n)$ time.

Optimality. It is not hard to see that sorting the edge weights is unavoidable when computing a Cartesian tree of a tree. Consider a tree T with a root and n children, where the ith child edge has weight $A[i]$. Then the Cartesian tree consists of a path, with weights equal to the array A in increasing order. Thus we obtain a linear-time reduction from sorting to computing a Cartesian tree. If you prefer to compute Cartesian trees only of bounded-degree trees, you can expand the root vertex into a path of n vertices, and put on every edge on the path a weight larger than $\max\{A[1], \ldots, A[n]\}$.

BEQ on Trees. For the BEQ problem on a tree T, if the edge weights are integers or are already sorted, our solution is optimal. We now show that for arbitrary unsorted edge-weights we can use our Cartesian tree to get an $O(n)$-space $O(1)$-query BEQ solution for trees that requires $O(n \lg^{[k]} n)$ preprocessing time for any fixed k (recall $\lg^{[k]} n$ denotes the iterated application of k logarithms). We present an $O(n \lg\lg n)$ preprocessing time algorithm, $O(n \lg^{[k]} n)$ is achieved by recursively applying our solution for k times.

Consider the micro-macro decomposition of T described above. Recall that each micro tree is of size $\Theta(\lg n)$. We can therefore sort the edges in each micro tree in $\Theta(\lg n \lg\lg n)$ time and construct the $\Theta(\frac{n}{\lg n})$ Cartesian trees of all micro trees in a total of $\Theta(n \lg\lg n)$ time. This allows us to solve BEQ within a micro tree in constant time. To handle BEQ between vertices in different micro trees, we construct the Cartesian tree of the macro tree. Recall that the macro tree contains $\Theta(\frac{n}{\lg n})$ nodes (all boundary nodes). The edges of the macro tree are of two types: edges between boundary nodes of different micro trees, and edges between boundary nodes of the same micro tree. For the former, we set their weights according to their weights in T. For the latter, we set the weight of an edge between two boundary nodes u, v of the same micro tree to be equal to BEQ(u, v) in this micro tree. BEQ(u, v) is computed in constant time from the Cartesian tree of the appropriate micro tree. Thus, we can compute all edge weights of the macro tree and then sort them in $O(\frac{n}{\lg n} \lg \frac{n}{\lg n}) = O(n)$ time and construct the Cartesian tree of the macro tree.

To conclude, we notice that any bottleneck edge query on T can be solved by a constant number of bottleneck edge queries on the Cartesian tree of the macro tree and on two more Cartesian trees of micro trees.

3.1 A Dynamic Cartesian Tree of a Tree

In this section we show how to maintain the Cartesian tree along with its LCA data structure while leaf insertions/deletions are preformed on the input tree. Our solution gives $O(\lg n)$ amortized time for updates and $O(1)$ worst-case time for bottleneck edge queries (via LCA queries). This is optimal in the comparison model since we can sort an array by inserting its elements as leafs in a star-like tree. In the case of integral edge weights taken from a universe $\{1, 2, \ldots, u\}$ we show that leaf insertions/deletions can be preformed in $O(\lg\lg u)$ amortized time.

Theorem 3. *We can maintain constant-time bottleneck edge queries on a tree while leaf insertions/deletions are preformed in $O(\lg n)$ amortized time and in $O(\lg\lg u)$ amortized time when the edge-weights are integers bounded by u.*

Proof. If we allow $O(\log n)$ time for bottleneck edge queries then we can use Sleator and Tarjan's *link-cut trees* [**?**]. This data structure can maintain a forest of edge-weighted rooted trees under the following operations (among others) in amortized $O(\log n)$ time per operation. *maketree()* creates a single-node tree, *link(v,w)* makes v (the root of a tree not containing w) a new child of w by adding an edge (v, w), *cut(v)* deletes the edge between vertex v and its parent, *LCA(v,w)* returns the LCA of v and w, and *mincost(v)* returns an ancestor edge of v with minimum weight. Thus, using link-cut trees, we can maintain a dynamic forest under link, cut, and bottleneck edge queries in $O(\log n)$ amortized time per operation without using our Cartesian tree. A bottleneck edge query between v and w does *cut(LCA(v,w))* then returns $\min\{mincost(v), mincost(w)\}$ and then links *LCA(v,w)* back as it was.

For constant-time bottleneck edge queries, we want to maintain an LCA structure on the Cartesian tree. Consider the desired change to the Cartesian tree when a new leaf v is inserted to the input tree as a child of u via an edge (u, v) of weight X. Notice that u currently appears as a leaf in the Cartesian tree and its ancestors represent edges in the input tree. Let w be u's lowest ancestor that represents an edge in the input tree of weight at most X. We need to insert a new vertex between w and its child w' (w' is the child that is also an ancestor of u). This new vertex will correspond to the new input tree edge (u, v) and will have w' as one child and a new leaf v as the other child.

Cole and Hariharan's dynamic LCA structure [?] can maintain a forest of rooted trees under LCA queries, insertion\deletion of leaves, subdivision (replacing edge (u, v) with edges (u, w) and (w, v) for a new vertex w) of edges, and merge (deletion of an internal node with one child) of edges. All operations are done in constant-time. It is easy to see that our required updates to the Cartesian tree can all be done by these dynamic LCA operations in constant time. The only problem remaining is how to locate the vertex w. Recently, Kopelowitz and Lewenstein [?] referred to this problem as a *weighted ancestor query* and presented a data structure that answers such queries in $O(\log \log u)$ amortized time when the weights are taken from a fixed universe $\{1, 2, \ldots, u\}$. Their data structure also supports all of the dynamic LCA operations of Cole and Hariharan in $O(\log \log u)$ amortized time per operation. Combining these results and assuming a universe bounded by u, we can maintain a Cartesian tree that supports bottleneck edge queries in $O(1)$ worst-case time and insert\delete leaf to the input tree in $O(\log \log u)$ amortized time.

For the case of unbounded weights, in order to locate w in $O(\log n)$ amortized time we also maintain the Cartesian tree as a link-cut tree. Link-cut trees partition the trees into disjoint paths connected via parent pointers. Each path is stored as an auxiliary splay tree [?] keyed by vertex depth on the path. Before any operation to node u, *expose(u)* makes sure that the root-to-u path is one of the paths in the partitioning. In order to locate w, we *expose(u)* and then search for w by the key X in the splay tree of u. Notice that the splay tree is keyed by vertex depth but since the Cartesian tree satisfies the heap property the order in the splay tree is also valid for vertex weight. After locating w, the modifications to the link-cut tree of the Cartesian tree can be done by $O(1)$ link-cut operations. □

4 Two-Dimensional RMQ

Apart from the new result of [7], the known solutions [9, 10, 16, 19, 28] to the standard one dimensional RMQ problem make use of the Cartesian tree. Whether as a tool for reducing the problem to LCA or in order to enable table-lookups for all different Cartesian trees. Amir, Fischer, and Lewenstein [5] conjectured that no Cartesian tree equivalent exists in the two-dimensional version of RMQ denoted 2D-RMQ. In this section we prove this conjecture to be true by showing that the number of different 2D-RMQ matrices is roughly $(n^2)!$. Two matrices are different if their range minima are in different locations for some rectangular range.

Recall that in 2D-RMQ, we wish to preprocess an $n \times n$ matrix for subsequent queries seeking the minimum in a given rectangle. All known solutions [5, 7, 13] to 2D-RMQ divide the $n \times n$ matrix into smaller $x \times x$ blocks. The smallest element in each block is chosen as the block's representative. Then, 2D-RMQ is recursively solved on the representatives and also recursively solved on each of the n^2/x^2 blocks. This is similar to the approach in 1D-RMQ, however, in 1D-RMQ small enough blocks are solved by table-lookups. This allows linear preprocessing since the type of a block of length x can be computed in $O(x)$ time. A corollary of Theorem 4 is that in 2D-RMQ linear preprocessing can not be achieved by standard table-lookups since computing the type of an $x \times x$ block requires at least $\log \left(\frac{x}{4}!\right)^{x/4} = \Omega(x^2 \log x)$ time.

This obstacle was recently overcome by Atallah and Yuan [7] who showed that by changing the definition of a block "type" and allowing four comparisons to be made during query-time we can solve 2D-RMQ optimally with $O(n^2)$ preprocessing and $O(1)$ query.

Theorem 4. *The number of different*[8] *2D-RMQ $n \times n$ matrices is $\Omega\left(\left(\frac{n}{4}!\right)^{n/4}\right)$*

Proof. We under-count the number of different matrices by counting only matrices that belong to the family $\mathcal{F}$ that is generated by a special matrix M of n^2 elements where every $M[i, j]$ ($0 \le i \le n-1$ and $0 \le j \le n-1$) is unique. We number M's main diagonal as 0, the diagonal above it 1, the one above that 2 and so on. Similarly, the diagonal below the main diagonal is numbered -1, the one below it -2 and so on. The matrix M is constructed such that the following properties hold. It is easy to verify that such a matrix M exists.

[8] Two matrices are different if their range minima are in different locations for some rectangular range.

(1) Elements in diagonal d are all smaller than elements in diagonal $d-1$ for every $1 \leq d \leq n-1$. Similarly, diagonal d's elements are all smaller than the ones of diagonal $d+1$ for every $-(n-1) \leq d \leq -1$.
(2) Let the elements of diagonal $d > 0$ be $a_0 < a_1 < \ldots < a_{n-d-1}$ and the elements of diagonal $-d$ be $b_0 < b_1 < \ldots < b_{n-d-1}$ then $a_0 < b_0 < a_1 < b_1 < \ldots < a_{n-d-1} < b_{n-d-1}$.

We focus only on range minima in rectangular ranges that contain elements in both sides of M's main diagonal. In such rectangles, by property (1), the minimum element is either the upper-right or lower-left corner of the rectangle. The family $\mathcal{F}$ contains all matrices that can be obtained from M by (possibly multiple) invocations of the following procedure: "pick some diagonal d where $\frac{n}{2} < d \leq \frac{3}{4}n$ and arbitrarily permute its elements". The number of matrices in $\mathcal{F}$ is thus $\Omega\big(\big(\frac{n}{4}!\big)^{n/4}\big)$. Furthermore, for any matrix in $\mathcal{F}$, and any rectangular range that spans both sides of its main diagonal, the minimum element is either the upper-right or lower-left corner of the rectangle.

We show that all matrices in $\mathcal{F}$ are different. That is, for every $M_1, M_2 \in \mathcal{F}$ there exists a rectangular range in which M_1 and M_2 have the minimum element in a different location. Let M_1, M_2 be two matrices in $\mathcal{F}$. There must exist a diagonal $\frac{n}{2} < d \leq \frac{3}{4}n$ in which M_1 and M_2 admit a different permutation of the (same) elements. This means that there exists $0 \leq i \leq n-d-1$ such that $M_1[i, d+i] \neq M_2[i, d+i]$. Assume without loss of generality that $M_1[i, d+i] < M_2[i, d+i]$. By property (2) there must be an element x on diagonal $-d$ such that $M_1[i, d+i] < x < M_2[i, d+i]$. Notice that x is in the same location in M_1 and in M_2 since all matrices in $\mathcal{F}$ have the exact same lower-triangle. Also notice that x's location, $[d+j', j']$ (for some $0 \leq j' \leq n-d-1$), is below and to the left of location $[i, d+i]$.

We get that in the rectangular range whose lower-left corner is $[d+j', j']$ and whose upper-right corner is $[i, d+i]$ the minimum elements is in location $[i, d+i]$ in M_1 and in a different location $[d+j', j']$ in M_2. Therefore, M_1 and M_2 are different 2D-RMQ matrices.

□

References

1. P.K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of the 19th annual ACM Symposium on Computational Geometry (SCG)*, pages 237–245, 2003.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, TR-71/87, Institute of Computer Science, Tel Aviv University, 1987.
4. S. Alstrup and M. Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.
5. A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proceedings of the 18th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 286–294, 2007.
6. L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, and J.I. Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.
7. M.J. Atallah and H. Yuan. Data structures for range minimum queries in multidimensional arrays. Manuscript, 2009.
8. M.A. Bender, E.D. Demaine, and M. Farach-colton. Cache-oblivious B-trees. In *SIAM Journal on Computing*, pages 399–409, 2000.
9. M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
10. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.
11. G.S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the 17th annual Symp. On Discrete Algorithms (SODA)*, pages 581–590, 2006.
12. B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.
13. B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the 5th annual ACM Symposium on Computational Geometry (SCG)*, pages 131–139, 1989.

14. R. Duan and S. Pettie. Fast algorithms for (max,min)-matrix multiplication and bottleneck shortest paths. In *Proceedings of the 20th annual Symposium On Discrete Algorithms (SODA)*, 2009.
15. S. Even and Y. Shiloach. An on-line edge deletion problem. *Journal of the ACM*, 28:1–4, 1981.
16. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th symposium on Combinatorial Pattern Matching (CPM)*, pages 36–48, 2006.
17. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th symposium on Foundations Of Computer Science (FOCS)*, pages 285–298, 1999.
18. H. Gabow, J.L. Bentley, and R.E Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th annual ACM Symposium on Theory Of Computing (STOC)*, pages 135–143, 1984.
19. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
20. T. C. Hu. The maximum capacity route problem. *Operations Research*, 9(6):898–900, 1961.
21. D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM*, 42:321–329, 1995.
22. D.E. Knuth. *The Art of Computer Programming Volume 4 Fascicle 4: Generating All Trees; History of Combinatorial Generation*. Addison-Wesley, 2006.
23. J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.
24. S. Pettie. An inverse-ackermann style lower bound for the online minimum spanning tree. In *Proceedings of the 43rd symposium on Foundations Of Computer Science (FOCS)*, pages 155–163, 2002.
25. S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity and set maxima algorithms. In *Proceedings of the 13th annual Symposium On Discrete Algorithms (SODA)*, pages 713–722, 2002.
26. S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.
27. M. Pollack. The maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.
28. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.
29. R. Seidel. Understanding the inverse ackermann function. PDF presenttion. Available at http://cgi.di.uoa.gr/∼ewcg06/invited/Seidel.pdf.
30. A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *Proceedings of the 18th annual Symposium On Discrete Algorithms (SODA)*, pages 978–985, 2007.
31. V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proceedings of the 39th annual ACM Symposium on Theory Of Computing (STOC)*, pages 585–589, 2007.