The LCA Problem Revisited

Michael A. Bender[*] Martín Farach-Colton[†]

SUNY Stony Brook Rutgers University

May 16, 2000

Abstract

We present a very simple algorithm for the Least Common Ancestor problem. We thus dispel the frequently held notion that an optimal LCA computation is unwieldy and unimplementable. Interestingly, this algorithm is a sequentialization of a previously known PRAM algorithm of Berkman, Breslauer, Galil, Schieber, and Vishkin [1].

Keywords: Data Structures, Least Common Ancestor (LCA), Range Minimum Query (RMQ), Cartesian Tree.

1 Introduction

One of the most fundamental algorithmic problems on trees is how to find the *Least Common Ancestor (LCA)* of a pair of nodes. The LCA of nodes u and v in a tree is the shared ancestor of u and v that is located farthest from the root. More formally, the LCA Problem is stated as follows: Given a rooted tree T, how can T be preprocessed to answer LCA queries quickly for any pair of nodes. Thus, one must optimize both the preprocessing time and the query time.

The LCA problem has been studied intensively both because it is inherently beautiful algorithmically and because fast algorithms for the LCA problem can be used to solve other algorithmic problems. In [2], Harel and Tarjan showed the surprising result that LCA queries can be answered in constant time after only linear preprocessing of the tree T. This classic paper is often cited because linear preprocessing is necessary to achieve optimal algorithms in many applications. However, it is well understood that the actual algorithm

[*]Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. Email: `bender@cs.sunysb.edu`. Supported in part by ISX Corporation and Hughes Research Laboratories.

[†]Department of Computer Science, Rutgers University, Piscataway, NJ 08855,USA. Email: `farach@cs.rutgers.edu`. Supported in part by NSF Career Development Award CCR-9501942, NATO Grant CRG 960215, NSF/NIH Grant BIR 94-12594-03-CONF.

presented is far too complicated to implement effectively. In [3], Schieber and Vishkin introduced a new LCA algorithm. Although their algorithm is vastly simpler than Harel and Tarjan's—indeed, this was the point of this new algorithm—it is far from simple and still not particularly implementable.

The folk wisdom of algorithm designers holds that the LCA problem still has no implementable optimal solution. Thus, according to hearsay, it is better to have a solution to a problem that does not rely on LCA precomputation if possible. We argue in this paper that this folk wisdom is wrong.

In this paper, we present not only a *simplified* LCA algorithm, we present a *simple* LCA algorithm! We devise this algorithm by reëngineering an existing complicated LCA algorithm: Berkman, Breslauer, Galil, Schieber, and Vishkin [1]. presented a PRAM algorithm that preprocesses and answers queries in $O(\alpha(n))$ time and preprocesses in linear work. Although at first glance, this algorithm is not a promising candidate for implementation, it turns out that almost all of the complications are PRAM induced: when the PRAM complications are excised from this algorithm so that it is lean, mean, and sequential, we are left with an extremely simple algorithm.

In this paper, we present this reëngineered algorithm. Our point is not to present a new algorithm. Indeed, we have already noted that this algorithm has appeared as a PRAM algorithm before. The point is to change the folk wisdom so that researchers are free to use the full power and elegance of LCA computation when it is appropriate.

The remainder of the paper is organized as follows. In Section 2, we provide some definitions and initial lemmas. In Section 3, we present a relatively slow algorithm for LCA preprocessing. In Section 4, we show how to speed up the algorithm so that it runs within the desired time bounds. Finally, in Section 5, we answer some algorithmic questions that arise in the paper but that are not directly related to solving the LCA problem.

2 Definitions

We begin by defining the *Least Common Ancestor (LCA) Problem* formally.

Problem 1 *The* Least Common Ancestor (LCA) *problem:*

Structure to Preprocess: *A rooted tree T having n nodes.*

Query: *For nodes u and v of tree T, query $\text{LCA}_T(u, v)$ returns the least common ancestor of u and v in T, that is, it returns the node furthest from the root that is an ancestor of both u and v. (When the context is clear, we drop the subscript T on the* LCA*.)*

The *Range Minimum Query (RMQ) Problem*, which seems quite different from the LCA problem, is, in fact, intimately linked.

Problem 2 *The* Range Minimum Query (RMQ) *problem:*

Structure to Preprocess: *A length n array A of numbers.*

Query: *For indices i and j between 1 and n, query $\text{RMQ}_A(x, y)$ returns the index of the smallest element in the subarray $A[i \ldots j]$. (When the context is clear, we drop the subscript A on the* RMQ.*)*

In order to simplify the description of algorithms that have both preprocessing and query complexity, we introduce the following notation. If an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we will say that the algorithm has complexity $\langle f(n), g(n) \rangle$.

Our solutions to the LCA problem are derived from solutions to the RMQ problem. Thus, before proceeding, we reduce the LCA problem to the RMQ problem. The following simple lemma establishes this reduction.

Lemma 3 *If there is an $\langle f(n), g(n) \rangle$-time solution for RMQ, then there is an $\langle f(2n-1) + O(n), g(2n-1) + O(1) \rangle$-time solution for LCA.*

As we will see, the $O(n)$ term in the preprocessing comes from the time needed to create the soon-to-be-presented length $2n - 1$ array, and the $O(1)$ term in the query comes from the time needed to convert the RMQ answer on this array to an LCA answer in the tree.

Proof: Let T be the input tree. The reduction relies on one key observation:

Observation 4 *The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a depth first search traversal of T.*

Therefore, the reduction proceeds as follows.

1. Let array $E[1, \ldots, 2n-1]$ store the nodes visited in an Euler Tour of the tree T. [1] That is, $E[i]$ is the label of the ith node visited in the Euler tour of T.

2. Let the *level* of a node be its distance from the root. Compute the Level Array $L[1, \ldots, 2n-1]$, where $L[i]$ is the level of node $E[i]$ of the Euler Tour.

3. Let the *representative* of a node in an Euler tour be the index of first occurrence of the node in the tour[2]; formally, the representative of i is $\text{argmin}_j\{E[j] = i\}$. Compute the Representative Array $R[1, \ldots, n]$, where $R[i]$ is the index of the representative of node i.

[1]The Euler Tour of T is the sequence of nodes we obtain if we write down the label of each node each time it is visited during a DFS. The array of the Euler tour has length $2n - 1$ because we start at the root and subsequently output a node each time we traverse an edge. We traverse each of the $n - 1$ edges twice, once in each direction.

[2]In fact, any occurrence of i will suffice to make the algorithm work, but we consider the first occurrence for the sake of concreteness.

Each of these three steps takes $O(n)$ time, yielding $O(n)$ total time. To compute $\text{LCA}_T(x, y)$, we note the following:

- The nodes in the Euler Tour between the first visits to u and to v are $E[R[u], \dots, R[v]]$ (or $E[R[v], \dots, R[u]]$).
- The shallowest node in this subtour is at index $\text{RMQ}_L(R[u], R[v])$, since $L[i]$ stores the level of the node at $E[i]$, and the RMQ will thus report the position of the node with minimum level. (Recall Observation 4.)
- The node at this position is $E[\text{RMQ}_L(R[u], R[v])]$, which is thus the output of $\text{LCA}_T(u, v)$.

Thus, we can complete our reduction by preprocessing Level Array L for RMQ. As promised, L is an array of size $2n - 1$, and building it takes time $O(n)$. Thus, the total preprocessing is $f(2n - 1) + O(n)$. To calculate the query time observe that an LCA query in this reduction uses one RMQ query in L and three array references at $O(1)$ time each. The query thus takes time $g(2n - 1) + O(1)$, and we have completed the proof of the reduction. ■

From now on, we focus only on RMQ solutions. We consider solutions to the general RMQ problem as well as to an important restricted case suggested by the array L. In array L from the above reduction adjacent elements differ by $+1$ or -1. We obtain this ± 1 restriction because, for any two adjacent elements in an Euler tour, one is always the parent of the other, and so their levels differ by exactly one. Thus, we consider the ± 1-RMQ problem as a special case.

2.1 A Naïve Solution for RMQ

We first observe that RMQ has a solution with complexity $\langle O(n^2), O(1)\rangle$: build a table storing answers to all of the n^2 possible queries. To achieve $O(n^2)$ preprocessing rather than the $O(n^3)$ naive preprocessing, we apply a trivial dynamic program. Notice that answering an RMQ query now requires just one array lookup.

3 A Faster RMQ Algorithm

We will improve the $\langle O(n^2), O(1)\rangle$-time brute-force table algorithm for (general) RMQ. The idea is to precompute each query whose length is a power of two. That is, for every i between 1 and n and every j between 1 and $\log n$, we find the minimum element in the block starting at i and having length 2^j, that is, we compute $M[i, j] = \text{argmin}_{\text{k}=\text{i}\dots\text{i}+2^{\text{j}}-1}\{\text{A[k]}\}$. Table M therefore has size $O(n \log n)$, and we fill it in time $O(n \log n)$ by using dynamic programming. Specifically, we find the minimum in a block of size 2^j by comparing the two minima of its two constituent blocks of size 2^{j-1}. More formally, $M[i,j] = M[i, j-1]$ if $A[M[i, j-1]] \leq M[i + 2^{j-1} - 1, j-1]$ and $M[i,j] = M[i + 2^{j-1} - 1, j-1]$ otherwise.

How do we use these blocks to compute an arbitrary $\text{RMQ}(i,j)$? We select two overlapping blocks that entirely cover the subrange: let 2^k be the size of the largest block that fits into the range from i to j, that is let $k = \lfloor \log(j-i) \rfloor$. Then $\text{RMQ}(i,j)$ can be computed by comparing the minima of the following two blocks: i to $i + 2^k - 1$ ($M(i,k)$) and $j - 2^k + 1$ to j ($M(j - 2^k + 1, k)$). These values have already been computed, so we can find the RMQ in constant time.

This gives the *Sparse Table (ST)* algorithm for RMQ, with complexity $\langle O(n \log n), O(1) \rangle$. Notice that the total computation to answer an RMQ query is three additions, 4 array reference and a minimum, in addition to two other operations: a log and a floor. These can be seen together as the problem of finding the most significant bit of a word. Notice that we must have one such operation in our algorithm, since Harel and Tarjan [2] showed that LCA computation has a lower bound of $\Omega(\log \log n)$ on a pointer machine. Furthermore, the most-significant-bit operation has a very fast table lookup solution.

Below, we will use the ST algorithm to build an even faster algorithm for the ± 1RMQ problem.

4 An $\langle O(n), O(1) \rangle$-Time Algorithm for ± 1RMQ

Suppose we have an array A with the ± 1 restriction. We will use a table-lookup technique to precompute answers on small subarrays, thus removing the log factor from the preprocessing. To this end, partition A into blocks of size $\frac{\log n}{2}$. Define an array $A'[1, \ldots, 2n/\log n]$, where $A'[i]$ is the minimum element in the ith block of A. Define an equal size array B, where $B[i]$ is a position in the ith block in which value $A'[i]$ occurs. Recall that RMQ queries return the position of the minimum and that the LCA to RMQ reduction uses the position of the minimum, rather than the minimum itself. Thus we will use array B to keep track of where the minima in A' came from.

The ST algorithm runs on array A' in time $\langle O(n), O(1) \rangle$. Having preprocessed A' for RMQ, consider how we answer any query $\text{RMQ}(i,j)$ in A. The indices i and j might be in the same block, so we have to preprocess each block to answer RMQ queries. If $i < j$ are in different blocks, the we can answer the query $\text{RMQ}(i,j)$ as follows. First compute the values:

1. The minimum from i forward to the end of its block.

2. The minimum of all the blocks in between between i's block and j's block.

3. The minimum from the beginning of j's block to j.

The query will return the position of the minimum of the three values computed. The second minimum is found in constant time by an RMQ on A', which has been preprocessed using the ST algorithm. But, we need to know how to answer range minimum queries inside blocks to compute the first and third minima, and thus to finish off the algorithm. Thus, the in-block queries are needed whether i and j are in the same block or not.

Therefore, we focus now only on in-block RMQs. If we simply performed RMQ preprocessing on each block, we would spend too much time in preprocessing. If two block were identical, then we could share their preprocessing. However, it is too much to hope for that blocks would be so repeated. The following observation establishes a much stronger shared-preprocessing property.

Observation 5 *If two arrays $X[1,\ldots,k]$ and $Y[1,\ldots,k]$ differ by some fixed value at each position, that is, there is a c such that $X[i] = Y[i] + c$ for every i, then all RMQ answers will be the same for X and Y. In this case, we can use the same preprocessing for both arrays.*

Thus, we can *normalize* a block by subtracting its initial offset from every element. We now use the ± 1 property to show that there are very few kinds of normalized blocks.

Lemma 6 *There are $O(\sqrt{n})$ kinds of normalized blocks.*

Proof: Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by a ± 1 vector of length $(1/2 \cdot \log n) - 1$. There are $2^{(1/2 \cdot \log n) - 1} = O(\sqrt{n})$ such vectors. ■

We are now basically done. We create $O(\sqrt{n})$ tables, one for each possible normalized block. In each table, we put all $(\frac{\log n}{2})^2 = O(\log^2 n)$ answers to all in-block queries. This gives a total of $O(\sqrt{n} \log^2 n)$ total preprocessing of normalized block tables, and $O(1)$ query time. Finally, compute, for each block in A, which normalized block table it should use for its RMQ queries. Thus, each in-block RMQ query takes a single table lookup.

Overall, the total space and preprocessing used for normalized block tables and A' tables is $O(n)$ and the total query time is $O(1)$.

4.1 Wrapping Up

We started out by showing a reduction from the LCA problem to the RMQ problem, but with the key observation that the reduction actually leads to a ± 1RMQ problem.

We gave a trivial $\langle O(n^2), O(1) \rangle$-time table-lookup algorithm for RMQ, and show how to sparsify the table to get a $\langle O(n \log n), O(1) \rangle$-time table-lookup algorithm. We used this latter algorithm on a smaller summary array A' and needed only to process small blocks to finish the algorithm. Finally, we notice that most of these blocks are the same, from the point of view of the RMQ problem, by using the ± 1 assumption given by the original reduction.

5 A Fast Algorithm for RMQ

We have a $\langle O(n), O(1)\rangle$ ± 1RMQ. Now we show that the general RMQ can be solved in the same complexity. We do this by reducing the RMQ problem to the LCA problem! Thus, to solve a general RMQ problem, one would convert it to an LCA problem and then back to a ± 1RMQ problem.

The following lemma establishes the reduction from RMQ to LCA.

Lemma 7 *If there is a $\langle O(n), O(1)\rangle$ solution for LCA, then there is a $\langle O(n), O(1)\rangle$ solution for RMQ.*

We will show that the $O(n)$ term in the preprocessing comes from the time needed to build the Cartesian Tree of A and the $O(1)$ term in the query comes from the time needed to covert the LCA answer on this tree to an RMQ answer on A.

Proof: Let $A[1, \ldots, n]$ be the input array.

The Cartesian Tree of an array is defined as follows. The root of a Cartesian Tree is the minimum element of the array, and the root is labeled with the position of this minimum. Removing the root element splits the array into two pieces. The left and right children of the root are the recursively constructed Cartesian trees of the left and right subarrays, respectively.

A Cartesian Tree can be built in linear time as follows. Suppose C_i is the Cartesian tree of $A[1, \ldots, i]$. To build C_{i+1}, we notice that node $i + 1$ will belong to the rightmost path of C_{i+1}, so we climb up the rightmost path of C_i until finding the position where $i + 1$ belongs. Each comparison either adds an element to the rightmost path or removes one, and each node can only join the rightmost path and leave it once. Thus the total time to build C_n is $O(n)$.

The reduction is as follows.

- Let C be the Cartesian Tree of A. Recall that we associate with each node in C the corresponding corresponding to $A[i]$ with the index i.

Claim 7A $\text{RMQ}_A(i, j) = \text{LCA}_C(i, j)$.

Proof: Consider the least common ancestor, k, of i and j in the Cartesian Tree C. In the recursive description of a Cartesian tree, k is the first node that separates i and j. Thus, in the array A, element $A[k]$ is between elements $A[i]$ and $A[j]$. Furthermore, $A[k]$ must be the smallest such element in the subarray $A[i, \ldots, j]$ since otherwise, there would be an smaller element k' in $A[i, \ldots, j]$ that would be an ancestor of k in C, and i and j would already have been separated by k'.

More concisely, since k is the first element to split i and j, it is between them because it splits them, and it is minimal because it is the first element to do so. Thus it is the RMQ. ◻

We see that we can complete our reduction by preprocessing the Cartesian Tree C for LCA. Tree C takes time $O(n)$ to build, and because C is an n node tree, LCA preprocessing takes $O(n)$ time, for a total of $O(n)$ time. The query then takes $O(1)$, and we have completed the proof of the reduction. ■

References

[1] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[2] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[3] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.