Lowest Common Ancestors in Trees and Directed Acyclic Graphs[1]

Michael A. Bender[2 3] Martín Farach-Colton[4] Giridhar Pemmasani[2]
Steven Skiena[2 5] Pavel Sumazin[6]

Version:

We study the problem of finding *lowest common ancestors (LCA)* in trees and *directed acyclic graphs (DAGs)*. Specifically, we extend the LCA problem to DAGs and study the LCA variants that arise in this general setting. We begin with a clear exposition of Berkman and Vishkin's simple optimal algorithm for LCA in trees. The ideas presented are not novel theoretical contributions, but they lay the foundation for our work on LCA problems in DAGs. We present an algorithm that finds all-pairs-representative LCA in DAGs in $\tilde{O}(n^{2.688})$ operations, provide a transitive-closure lower bound for the all-pairs-representative-LCA problem, and develop an LCA-existence algorithm that preprocesses the DAG in transitive-closure time. We also present a suboptimal but practical $O(n^3)$ algorithm for all-pairs-representative LCA in DAGs that uses ideas from the optimal algorithms in trees and DAGs. Our results reveal a close relationship between the LCA, all-pairs-shortest-path, and transitive-closure problems.

We conclude the paper with a short experimental study of LCA algorithms in trees and DAGs. Our experiments and source code demonstrate the elegance of the preprocessing-query algorithms for LCA in trees. We show that for most trees the suboptimal $\Theta(n \log n)$-preprocessing $\Theta(1)$-query algorithm should be preferred, and demonstrate that our proposed $O(n^3)$ algorithm for all-pairs-representative LCA in DAGs performs well in both low and high density DAGs.

Keywords: Lowest Common Ancestor (LCA), Directed Cyclic Graph (DAG), Range Minimum Query (RMQ), Shortest Path, Cartesian Tree.

1. INTRODUCTION

One of the fundamental algorithmic problems on trees is how to find the *lowest common ancestor (LCA)* of a given pair of nodes. The LCA of nodes u and v in a tree is the ancestor of u and v that is located farthest from the root. 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? The LCA problem has been studied intensively both because it is inherently beautiful and because fast algorithms for the LCA problem can be used to solve other algorithmic problems.

In this paper we introduce a natural extension of the LCA problem to directed cyclic graphs (DAGs), and we study the LCA variants that arise. Many combinatorial problems that require finding nearest common ancestors cannot be solved using the tree-LCA algorithm because the ancestor queries apply to more complicated directed structures. Nykänen and Ukkonen [23] give a linear-time preprocessing, constant-time-query algorithm for the LCA in arbitrarily directed trees. They ask whether it is possible to preprocess a DAG in $o(n^3)$ to support $\Theta(k)$-time set-LCA queries, where a set-LCA query returns all k lowest common ancestors of the given pair. In compiler design, the idea of preprocessing the object-inheritance taxonomy for fast LCA queries was introduced by Ait-Kaci, Boyer, Lincoln, and

[1] This work appeared in preliminary form in publications [2] and [3].

[2] Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794, USA. Email: {bender,giri,skiena}@cs.sunysb.edu.

[3] Supported in part by NSF Grants EIA-0112849, CCR-0208670 and HRL Laboratories, ISX Corporation, and Sandia National Laboratories.

[4] Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. Supported in part by NSF Career Development Award CCR-9501942, NATO Grant CRG 960215, NSF/NIH Grant BIR 94-12594-03-CONF, NSF grant CCR-9820879. Email: farach@cs.rutgers.edu.

[5] Supported in part by NSF Grant CCR-9625669 and ONR Award N00149710589.

[6] Department of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207, USA. Supported by NSF grant DBI-0306152. Email: ps@cs.pdx.edu.

Nasr [1]. They consider the problem of LCA on lattices and lower semi-lattices (where a node pair has a unique LCA), which are used to represent inheritance graphs. Ducournau, Habib, Huchard, and Spinrad [11, 17] consider the problem of finding modular coverings in inheritance graphs (the LCA can be used to find a maximum covering). Their objective is to decompose lower semi-lattices into modules that can be queried faster.

We develop algorithms for efficiently answering lowest-common-ancestor queries in DAGs. An LCA w of nodes u and v in a DAG is an ancestor of both u and v where w has no descendants that are also ancestors of both u and v. We present an $o(n^3)$ all-pairs algorithm for answering representative lowest-common-ancestor queries on DAGs in constant time. Our algorithm is also the first $o(n^3)$ preprocessing algorithm for constant time LCA queries in lower semi-lattices. It is an open question whether our approach can be extended to answer the more general set-LCA question posed by Nykänen and Ukkonen.

Lowest-common-ancestor queries in general DAGs appear in a variety of applications, including the following:

- *Object Inheritance in Programming Languages* – Object-oriented programming languages such as C++ and Java provide object inheritance, whose structure is analyzed in various stages of compilation and execution. Objects are instances of classes organized in a partial order, and their inheritance depends on the temporal order in which the objects are defined. The idea of formalizing object inheritance in lattice-theoretic terms has been proposed by [1, 12, 13, 16, 18, 21, 24] and others. The LCA operation is central to such object inheritance formalizations because it is the natural method to resolve object dependence.

- *Lattice Operations for Complex Systems* – Algorithms on lattices are used to model the dynamic and static behavior of complex systems arising in distributed computing [8, 20, 22]. LCA queries arise in computing the covering of maximal ideal lattices.

1.1. Results

We show how to preprocess the DAG to answer queries in constant time about whether nodes x and y have a common ancestor. We improve on the naïve $O(n^3)$ algorithm and present an $\tilde{O}(n^\omega)$ algorithm, where $\omega \approx 2.376$ is the exponent of the fastest known matrix-multiplication algorithm of Coppersmith and Winograd [10].[7]

We give an algorithm that solves the all-pair representative-LCA problem for DAGs in $\tilde{O}(n^{\frac{\omega+3}{2}}) \approx \tilde{O}(n^{2.688})$ time by establishing a relationship between the LCA and the all-pairs-shortest-path problem. The best known preprocessing algorithm for lower semi-lattices runs in $O(n^3)$ operations for $O(\log^2 n)$ queries [1]. Lower semi-lattices have a unique LCA for each node pair, and our all-pair representative-LCA algorithm is the first to find all-pairs LCAs in lower semi-lattices in $o(n^3)$. We also show a complementary lower bound for the all-pairs LCA in DAGs by giving a reduction from the transitive-closure problem.

We show a chain of reductions between the all-pairs-LCA problem, the all-pairs-shortest-path problem, the transitive-closure problem, and variants of these problems. Specifically, we reduce the all-pairs-representative-LCA problem to the all-pairs-shortest-path problem, the transitive-closure problem to the all-pairs-representative-LCA problem, and the all-pairs-common-ancestor-existence problem to the transitive-closure problem. This relationship is given in Figure 1.

We present a simple optimal LCA algorithm for trees, which is just a sequentialization of the more complicated PRAM algorithm of Berkman and Vishkin [4]. We present the algorithm to provide a clear exposition, and note that the algorithm was known to Berkman and Vishkin. In an appendix we present an experimental study, comparing the algorithm with a naïve algorithm and an efficient but asymptotically suboptimal algorithm. Through this presentation, we lay to rest the folk belief that LCA is too complicated to teach and implement.

We introduce an easily implementable LCA algorithm for DAGs. Like the tree-LCA algorithms, this algorithm answers LCA queries by answering RMQ queries. We compare this new algorithm to an intelligent straightforward algorithm and to an algorithm based on the transitive closure. The straightforward algorithm outperforms the

[7] We say that $f(n) = \tilde{O}(g(n))$ if $\exists c$ such that $f(n) = O(g(n)\log^c n)$, and $f(n) = \tilde{\Theta}(g(n))$ if $\exists c$ such that $f(n) = \Theta(g(n)\log^c n)$.

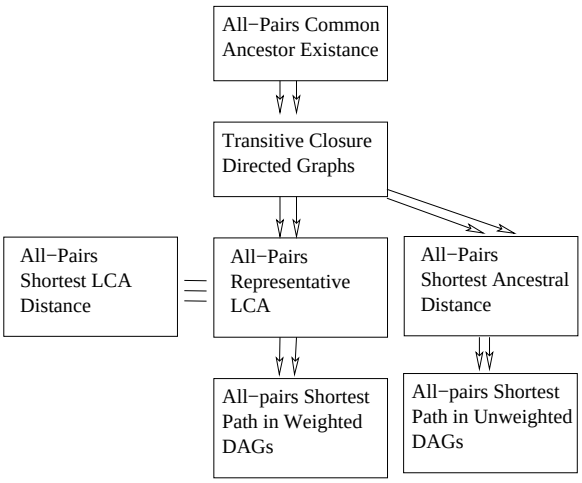

FIG. 1 Reduction chain of problems investigated in this paper. Arrows are in the reduction direction.

transitive-closure-based algorithm for sparse DAGs, whereas the transitive-closure-based algorithm outperforms the straightforward algorithm for dense DAGs. Our algorithm does not guarantee better asymptotic behavior, however it impressively outperforms the other two on all our test instances. Thus, the tools for optimal LCA in trees seem useful for constructing good LCA algorithms for DAGs.

1.2. Organization

The paper is organized as follows. In Section 2 we present Berkman and Vishkin's simple tree-LCA algorithm [4]. In Section 3 we define the lowest common ancestors in DAGs. In Section 4 we present an efficient algorithm to determine common-ancestor existence. Then we present an $o(n^3)$ algorithm and a near-matching lower bound for the all-pairs-LCA problem. Finally, in Appendix A we present experimental comparisons of several LCA algorithms for trees and DAGs.

2. LCA IN TREES

The first study of the LCA problem was done by Tarjan [26], who consider both the off-line and online setting for the LCA problem in trees. Harel and Tarjan gave important upper and lower bounds for the LCA problem [19]. In 1984 Gabow, Bentley, and Tarjan [14] showed that the LCA problem was linearly equivalent to the (one-dimensional) Range Minimum Query problem, which will be further discussed in this section. Gabow, Bentley, and Tarjan were concerned with multidimensional range searching, and did not present an optimal solution to the LCA problem but their work provides the basis for much of this section. In 1988 Schieber and Vishkin [25] introduced an LCA algorithm with optimal asymptotic bounds. Berkman and Vishkin present a PRAM algorithm that uses $\Theta(n)$ operations to preprocess the tree for answering queries in $O(\alpha(n))$ time [4]. The main technical difficulty of their work is in achieving the parallel bounds, and the sequential LCA algorithm presented in this section was known to them but omitted from their manuscript. An optimal dynamic algorithm was given by Cole and Hariharan [9]. Other LCA algorithms on trees can be found in [5, 27, 28].

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 that this folk wisdom is wrong. Stripped of its PRAM complications, Berkman and Vishkin's

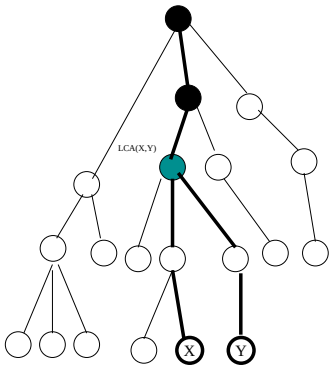

FIG. 2 The common ancestors of nodes x and y are marked in boldface. The $\text{LCA}(x, y)$ is the common ancestor of x and y that is farthest away from the root.

sequential algorithm, which we present here, is not only a *simplified* LCA algorithm, but a *simple* LCA algorithm.

The key to the algorithm lies in the work of Gabow, Bentley and Tarjan [14]. The Range Minimum Query problem turns out to have a cute solution using standard ideas such as table lookups (see e.g., [15]). This was the approach of Berkman and Vishkin [4] in their PRAM algorithm, and their optimal sequential LCA algorithm is an elegant and direct application of these ideas.

2.1. Algorithms for LCA in Trees

We begin by defining the *Lowest Common Ancestor (LCA) Problem* in trees formally; see Figure 2.

PROBLEM 2.1. *The* Lowest 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 lowest common ancestor of u and v in T, that is, it returns the node farthest 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 2.2. *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 $\text{RMQ}_A(i, j)$ 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 $\langle f(n),\ g(n)\rangle$.

Solutions to the LCA problem are derived from solutions to the RMQ problem. Thus, before proceeding, we present the reduction from the LCA problem in trees to the RMQ problem, as given by Gabow, Bentley, and Tarjan [14]. The following simple lemma establishes this reduction.

LEMMA 2.3. *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 $\Theta(n)$ term in the preprocessing comes from the time needed to create the soon-to-be-presented length $2n-1$ array, and the $\Theta(1)$ 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 2.1. *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 traversed in an Euler Tour of the tree T.[8] That is, $E[i]$ is the label of the ith node traversed 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[9]; formally, the representative of i is $\min\{j : E[j]=i\}$. Compute the Representative Array $R[1,\ldots,n]$, where $R[i]$ is the representative of node i.

Each of these three steps takes $\Theta(n)$ time, yielding $\Theta(n)$ 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 $E[R[u],\ldots,R[v]]$ (or $E[R[v],\ldots,R[u]]$).
- 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 2.1.)
- 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 $\Theta(n)$. Thus, the total preprocessing is $f(2n-1)+\Theta(n)$. To calculate the query time observe that an LCA query in this reduction uses one RMQ query in L and three array references at $\Theta(1)$ time each. The query thus takes time $g(2n-1)+\Theta(1)$, 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.

2.2. A Naïve Solution for RMQ

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

[8] 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 traversed 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.

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

2.3. A Faster RMQ Algorithm

We will improve the $\langle \Theta(n^2),\ \Theta(1)\rangle$-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 $M[i,j] = \min\{A[k] : k = i \ldots i + 2^j - 1\}$. Table M therefore has size $\Theta(n \log n)$, and we fill it in time $\Theta(n \log n)$ 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 $A[M[i,j-1]] \leq A[M[i+2^{j-1}-1, j-1]]$ 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 $k = \lceil \log(j - i + 1) \rceil$. 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 $\langle \Theta(n \log n),\ \Theta(1)\rangle$. The table is indexed using a $\langle$distance, array index$\rangle$ tuple. Notice that the total computation to answer an RMQ query involves two subtractions, two 2-dimensional array references, a minimum and a truncated-log operation. The truncated-log operation does not require a table lookup in most modern processors, and can be seen as finding the most significant bit of a word. Notice that we must have at least one array indexing operation in our algorithm, since Harel and Tarjan [19] showed that a pointer-algorithm LCA computation has a lower bound of $\Omega(\log \log n)$ operations.

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

2.4. A $\langle \Theta(n),\ \Theta(1)\rangle$-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 $\frac{\log n}{2}$~~. (Without loss of generality assume $\log n$ is even). 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 $\langle \Theta(n),\ \Theta(1)\rangle$. 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, then 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. (If i and j are not in the same block prefix minima and suffix minima suffice).

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 2.2. *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 few kinds of normalized blocks.

LEMMA 2.4. *There are $\Theta(\sqrt{n})$ 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 $2^{(1/2\cdot\log n)-1} = \Theta(\sqrt{n})$ such vectors. □

We are now basically done. We create $\Theta(\sqrt{n})$ tables, one for each possible normalized block. In each table, we put all $(\frac{\log n}{2})^2 = \Theta(\log^2 n)$ answers to all in-block queries. This gives a total of $\Theta(\sqrt{n}\log^2 n)$ total preprocessing of normalized block tables, and $\Theta(1)$ 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 normalized-block table lookup.

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

2.5. 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 $\langle \Theta(n^2), \Theta(1)\rangle$-time table-lookup algorithm for RMQ, and show how to sparsify the table to get a $\langle \Theta(n \log n), \Theta(1)\rangle$-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.

2.6. A Fast Algorithm for RMQ

We have a $\langle \Theta(n), \Theta(1)\rangle$ ± 1RMQ. Gabow and Tarjan [15] show that the general RMQ can be solved in the same complexity. They 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 and proof from [15] establishes the reduction from RMQ to LCA.

LEMMA 2.5. *If there is a $\langle \Theta(n), \Theta(1)\rangle$ solution for LCA, then there is a $\langle \Theta(n), \Theta(1)\rangle$ solution for RMQ.*

We will show that the $\Theta(n)$ term in the preprocessing comes from the time needed to build the Cartesian Tree of A and the $\Theta(1)$ term in the query comes from the time needed to convert 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 be at the end of the rightmost path of C_{i+1}, so we climb up the rightmost path

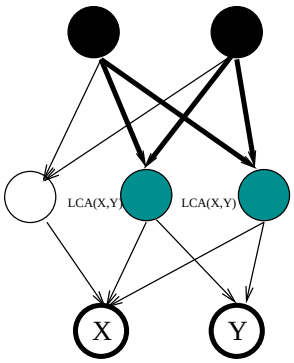

FIG. 3 The common ancestors of nodes x and y appear shaded. The set $\text{SLCA}(x, y)$ is composed of all common ancestors of x and y with no common ancestor descendants. The lowest common ancestors of x and y (appear lightly shaded) are exactly the elements of $\text{SLCA}(x, y)$.

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 $\Theta(n)$.

The reduction is as follows.

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

CLAIM 2.1. $\text{RMQ}_A(i, j) = \text{LCA}_C(i, j)$.

Proof: Consider the lowest 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 a 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 $\Theta(n)$ to build, and because C is an n node tree, LCA preprocessing takes $\Theta(n)$ time, for a total of $\Theta(n)$ time. The query then takes $\Theta(1)$, and we have completed the proof of the reduction. □

3. DEFINITIONS FOR LCA IN DAGS

We now present two equivalent definitions for the LCA in DAGs; see Figure 3. For the special case where the DAG is a tree, we obtain the standard tree-LCA definition.

DEFINITION 3.1. *Let $G = (V, E)$ be a DAG, and let $x, y \in V$. Let $G_{x,y}$ be the subgraph of G induced by the set of all common ancestors of x and y. Define $\text{SLCA}(x, y)$ to be the set of ~~out-~~degree 0 nodes (leafs) in $G_{x,y}$. The lowest common ancestors of x and y are the elements of $\text{SLCA}(x, y)$.*

Observe that there may be as many as $|V| - 2$ distinct lowest common ancestors of a given pair of nodes in DAGs, whereas the LCA in trees is unique.

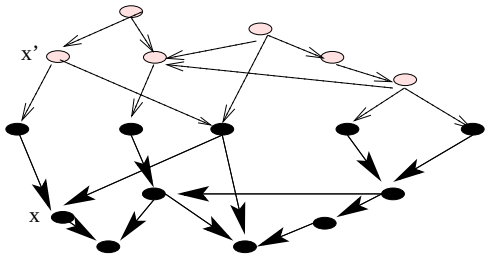

FIG. 4 Doubling the DAG by reflecting through the sources. The original graph has bold edges.

We introduce some terminology. A *maximum element* in a partially ordered set is an element m such that no member of the set is greater than m. The *transitive closure* $G_{tr} = (V, E_{tr})$ of a DAG $G = (V, E)$ is a graph such that $(i, j) \in E_{tr}$ if and only if there is a path from i to j in G. The transitive closure of any DAG G forms a partially ordered set.

Ait-Kaci, Boyer, Lincoln, and Nasr [1] describe the LCA is in terms of partially ordered sets.

DEFINITION 3.2. *For any DAG $G = (V, E)$, we define the partially ordered set $S = (V, \preceq)$ as follows: element $i \preceq j$ if and only if $i = j$ or (i, j) is in the transitive closure G_{tr} of G. Let* $\text{SLCA}(x, y)$ *be the set of the maximum elements of the common ancestor set $\{z | z \preceq x \wedge z \preceq y\} \subseteq V$. The lowest common ancestors of x and y are the elements of* $\text{SLCA}(x, y)$.

We answer LCA queries by returning a *representative element* from $\text{SLCA}(x, y)$. Typically, we find the representative LCA that is closest to x and y in the DAG. This closest element is useful in applications such as genealogy. In many common applications such as algorithms on semi-lattices, the set LCA consists of a single element.

We identify the representative element by defining the *depth* of a node x. Note that the following depth definition applies to both positive weighted and unweighted edges.

DEFINITION 3.3. *The* depth *of node x in a DAG,* $\text{depth}(x)$*, is the length of the longest path from a source to x.*

Based on node depth, we find an LCA using the following lemma:

LEMMA 3.1. *The node of greatest depth in a DAG that is an ancestor of both x and y is a lowest common ancestor of x and y.*

4. FINDING ALL-PAIRS LCA IN DAGS

We first give an algorithm for the all-pairs-common-ancestor-existence problem in DAGs. Then we give an algorithm for the all-pairs-representative-LCA problem. We conclude the section by showing that the all-pairs-LCA problem is transitive-closure hard.

In our algorithms we compute the answers to all (n choose 2) queries in the preprocessing stage. Then we answer queries by performing table lookups. We show how to build the binary common-ancestor-existence matrix in $\tilde{O}(n^{\omega})$ operations and the representative-LCA matrix in $\tilde{O}(n^{\frac{\omega+3}{2}})$ operations. The fastest known matrix-multiplication algorithm to date runs in $O(n^{\omega})$ where, $\omega \approx 2.376$ [10]. Thus, our all-pairs-common-ancestor-existence algorithm runs in time $\tilde{O}(n^{2.376})$, and our all-pairs-representative-LCA algorithm runs in time $\tilde{O}(n^{2.688})$.

4.1. The Common-Ancestor-Existence Algorithm

Pairs of nodes in a DAG may not have any common ancestors, in contrast to pairs of nodes in a tree. Here we show how to determine whether two nodes share common ancestors. In the representative-LCA computation, we can then

assume that LCA existence has been verified.

The preprocessing step consists of a transitive-closure computation on a graph F, which is constructed from G as follows: Reverse the edges in G to form the DAG G'. Combine these two graphs by merging the sinks of G' with the sources of G. We call the resulting graph F; see Figure 4. Every vertex v' in $F - G$ is a reflection of some v in G. The nodes x and y have a common ancestor if and only if (x', y) is in the transitive closure of F.

Thus, we obtain the following theorem:

THEOREM 4.1. *The ancestor-existence matrix can be computed in transitive-closure time.*

4.2. The All-Pairs LCA Algorithm

We solve the LCA problem by exploiting the close relationship between the LCA and all-pairs-shortest-path problems. By computing shortest paths, we can isolate a sublinear number of potential LCA nodes, from which we choose the node of greatest depth, which by Lemma 3.1 is an LCA.

In trees, the depth of $\text{LCA}(x, y)$ and the distance between the nodes $\text{dist}(x, y)$ are related as follows:

$$\text{dist}(x, y) = \text{depth}(x) + \text{depth}(y) - 2\,\text{depth}(\text{LCA}(x, y)). \tag{1}$$

This relationship holds in trees because of the following two properties:

PROPERTY 1. *The order $x \preceq y$ implies that* $\text{depth}(x) \leq \text{depth}(y)$.

PROPERTY 2. *For $x \preceq y$, the length of the shortest path from x to y is* $\text{depth}(y) - \text{depth}(x)$.

We define an *ancestral path* from x to y in G as a path that is composed of the shortest path from x up to some common ancestor z concatenated with the shortest path from z down to y. The path from x to z goes against the directions of the edges, and the path from z to y goes along the direction of the edges.

We construct a weighted DAG L, which reduces the representative-LCA problem to the shortest-path problem. The DAG L is constructed in two stages. First we add weights to G so that Properties 1 and 2 hold in G and so that *no two nodes have the same depth*. We use the weighted DAG G to construct weighted DAG L so that the ancestral paths in G are exactly the paths in L. Finding the length of the shortest path in L determines the depth of the representative LCA. Since no two nodes have the same depth, knowing the depth of the representative LCA uniquely determines the representative LCA.

We assign edge weights in G as follows. We begin by calculating node depths in the unweighted version of G according to Definition 3.3; thus Property 1 is satisfied. Then we find a linear extension as follows: We sort all the nodes in the (unweighted) graph G by their depth, arbitrarily breaking ties. Each node's order in this linear extension will be its *new* depth. Then we assign weights to all edges in G according to Property 2 to achieve the desired node depth. Specifically, given $(u, v) \in E$ we set $weight(v) = \text{depth}(v) - \text{depth}(u)$. This weight assignment guarantees that Properties 1 and 2 hold and that no two nodes have the same depth.

Next we show that the farthest common ancestor from the root is an LCA. We do so by defining the set $\text{SLCA}_h(x, y)$ and proving that it is a subset of $\text{SLCA}(x, y)$. Note that in our construction $\text{SLCA}_h(x, y)$ contains a single element.

DEFINITION 4.1. *Let $\text{SLCA}_h(x, y)$ be the set of common ancestors of x and y that have greatest depth. An $\text{LCA}_h(x, y)$ is an element of $\text{SLCA}_h(x, y)$.*

LEMMA 4.1. *The set of common ancestors of greatest depth is a subset of the set of lowest common ancestors. That is,* $\text{SLCA}_h(x, y) \subseteq \text{SLCA}(x, y)$.

Now we construct the weighted DAG L from the weighted DAG G. Let $G' = (V', E')$ be the graph constructed from G by reversing all edges in G. We combine G and G' by adding a zero-weight edge (v', v) for all $v' \in G'$ and $v \in G$. The resulting graph is L; see Figure 5. This construction ensures that the ancestral paths in G are maintained in L. We show in the following lemma that the shortest path from x' to y goes through the $\text{LCA}_h(x, y)$:

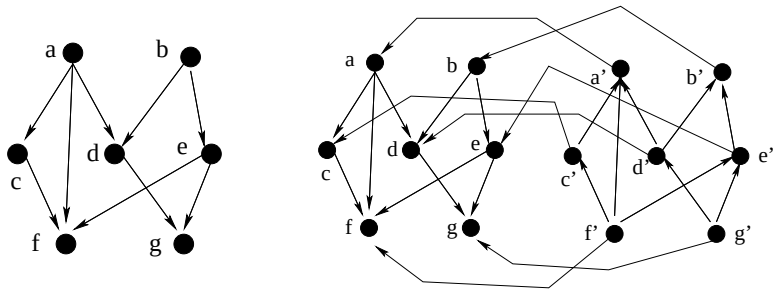

FIG. 5 The original DAG G is on the left. On the right we show L constructed from G.

LEMMA 4.2. *The shortest path from node x' to node y in graph L goes through the* $\text{LCA}_h(x, y)$.

Proof. We assume that x and y have a common ancestor in G and therefore that such a path exists in L. If x is a descendant of y, the length of the shortest path from x' to y (it goes through y') is $\text{depth}(x) - \text{depth}(y)$. Then y is the only $\text{LCA}(x, y)$ and the lemma holds. Otherwise, any path from x' to y must go through some common ancestor z and be of the form $x', \ldots, z', z, \ldots, y$, which is of length $\text{depth}(x) + \text{depth}(y) - 2\text{depth}(z)$. The shortest path then, passes through the common ancestor of greatest depth which is the $\text{LCA}_h(x, y)$. □

We use the all-pair approximate shortest path algorithm given by Zwick [29] to approximate the depth of the representative LCA. We choose this approximation algorithm because no $\tilde{o}(n^3)$ exact all-pairs shortest path algorithm is known for DAGs with large weights, and L may have edge weights as large as $n-1$. Zwick's algorithm approximates the shortest path to within $1 + \varepsilon$ using $\tilde{O}(\frac{n^\omega}{\varepsilon} \log \frac{W}{\varepsilon})$ operations, where W is an upper bound for the weight of the greatest edge. Given the approximate shortest path we deduce the approximate depth of the LCA using Equation 1.

Once we know the approximate depth of the LCA, we sift through the candidates to find the LCA itself. Since depths are unique and the shortest path can have length at most $2n-2$, there are $\Theta(\varepsilon n)$ LCA candidates of consecutive depths. In order to sift through the candidates we precompute the transitive closure of L to determine ancestry relation in constant time. We choose the LCA candidate of greatest depth that is a common ancestor.

We optimize the value for ε. Since there are $\Theta(n^2)$ pairs, and each pair requires examining $\Theta(\varepsilon n)$ nodes to find the common ancestor of greatest depth, we expend $\Theta(\varepsilon n^3)$ work to find the LCAs after we are given the all-pairs-approximate-shortest-path matrix. Computing all-pairs-approximate-shortest path requires $\tilde{O}(\frac{n^\omega}{\varepsilon} \log \frac{W}{\varepsilon})$ operations. Modulo logarithmic terms, we optimize for ε by equating terms and setting $\varepsilon = n^{\frac{\omega-3}{2}}$. Given this value for ε, our algorithm runs in time $\tilde{O}(n^{\frac{\omega+3}{2}})$. We obtain the following theorem:

THEOREM 4.2. *An all-pairs-LCA matrix for a DAG G can be found in $\tilde{O}(n^{\frac{\omega+3}{2}})$ time.*

Proof. Observe that the L has no edge weights greater than n since n is the depth of the deepest node in the graph. We set $\varepsilon = n^{\frac{\omega-3}{2}}$, which leads to a running time of $\tilde{O}(n^{\frac{\omega+3}{2}})$ for the approximate-shortest-path computation. The greatest possible distance error is $n^{\frac{\omega-1}{2}}$. Thus the algorithm identifies $2n^{\frac{\omega-1}{2}}$ possible LCA_h candidates for every pair. We perform a transitive-closure operation (using fast matrix multiplication) to allow for fast access to a node's ancestors. We find $A(x) \cap S$ and $A(y) \cap S$ in $O(n^{\frac{\omega-1}{2}})$ operations, and we identify the node of maximum depth in $O(n^{\frac{\omega-1}{2}})$ operations as well. Thus we use $O(n^{\frac{\omega-1}{2}})$ operations per pair, and the total running time of the algorithm is $\tilde{O}(n^{\frac{\omega+3}{2}})$. □

4.3. A Simpler Ancestor-List-Based All-Pairs LCA algorithm

In this section we give an $O(n^3)$ algorithm for the all-pairs-representative LCA. This algorithm is easily implementable, is efficient on low and high density DAGs, and performs well in our experiments. In contrast, the subcubic

algorithm from Section 4.2 uses the fast matrix-multiplication algorithm of Coppersmith and Winograd [10] and is considered impractical. Simpler algorithms that are easier to implement have inconsistent performance over DAG density ranges. We present an experimental study of the $O(n^3)$ algorithm in Section A.2. The algorithm proceeds as follows.

First add a node v to the DAG G, and add directed edges from v to all sources in G. The addition of v guarantees that every two nodes have an LCA. If v is reported as a representative LCA for nodes x and y then x and y have no common ancestors in G.

Preprocess G by partitioning the set of edges into two sets: S_1 is the set of edges of a spanning tree T of G, and S_2 is composed of the remaining edges, which make up the DAG $D = (V, S_2)$. Label the vertices of T by depth, breaking ties arbitrarily. The same labels are used for D. Create a list L_v for each vertex v, which contains the ancestors of v in D, including v itself. Proceed from vertex 0 until vertex $n-1$. Sort the lists according to the first appearance of their nodes in an Euler tour of T.

LEMMA 4.3. *The list-construction stage of the ancestor-list-based LCA algorithm is completed in* $\Theta((n-1)\Sigma_{x\in V}|L_x|) \in O(n^3)$ *operations.*

We compute pairwise LCA in DAG G by performing queries in the tree T. Let LCA_T stand for an LCA computation on the tree T, and let LCA_G stand for an LCA computation on the DAG G. We arrive at the following lemma:

LEMMA 4.4. *For nodes* $x, y \in G$*, a greatest depth node in the set* $\{\text{LCA}_T(u, v) : u \in L_x, v \in L_y\}$ *is an* $\text{LCA}_G(x, y)$.

$\text{LCA}_G(x, y)$ can be computed by making all pairwise LCA_T queries between L_x and L_y, and choosing the LCA_T of greatest depth. Each query uses $O(n^2)$ LCA_T operations, and the all-pairs computation is in $O(n^4)$. Next we show how to prune out unnecessary LCA queries in T, and compute LCA_G using $O(n)$ constant time LCA_T queries. This query speed up leads to an $O(n^3)$ all-pairs computation. Lemma 2.3 establishes the equivalence between $\text{LCA}_T(u, v)$ and $A(\text{RMQ}(E(u), E(v)))$, where $A[1, \ldots, 2n-1]$ in an array representation of the Euler tour of T, and $E(v)$ is the index of the first occurrence of node v in A. An $\text{LCA}_G(x, y)$ is a maximum-depth $\text{LCA}_T(u, v)$, where $u \in L_x$ and $v \in L_y$. The depth of $\text{LCA}_T(x, y)$ is greater or equal to the depth of $\text{LCA}_T(z, t)$ if $[x, y]$ is a subrange of $[z, t]$ in A. Therefore $\text{LCA}_T(z, t)$ can not be a new node of greatest depth in the pairwise LCA set, and the query $\text{LCA}_T(z, t)$ can be pruned out. Let L be the result of an order-preserving merge of L_x and L_y. To find $\text{LCA}_G(x, y)$ select the greatest depth node from $\{\text{LCA}_T(\ell_i, \ell_{i+1}) : \ell_i \in L_x, \ell_{i+1} \in L_y \text{ or } \ell_i \in L_y, \ell_{i+1} \in L_x\}$.

LEMMA 4.5. *A query* $\text{LCA}_G(x, y)$ *is completed in* $O(|L_x| + |L_y|)$ *operations.*

Proof. Let $A[1, \ldots, 2n-1]$ be an array representation of the Euler tour of T and $E(v)$ the index of the first occurrence of node v in A. For nodes $x, y \in G$, consider the two ancestor lists $L_x = x_1, \ldots, x_q$ and $L_y = y_1, \ldots, y_t$. According to Lemma 4.4 $\text{LCA}_G(x, y)$ is the node of greatest depth in $\{\text{LCA}_T(u, v) : u \in L_x, v \in L_y\}$, and according to Lemma 2.3 $\text{LCA}_T(u, v) = A[\text{RMQ}(E(u), E(v))]$. We observe that $A[\text{RMQ}(E(u), E(v))] \geq A[\text{RMQ}(E(u), E(\ell))]$ if $E(u) < E(v) < E(\ell)$. Without loss of generality let $E(x_i) < E(y_j)$ then $\text{LCA}_T(x_i, y_j)$ is the node of greatest depth in $\{\text{LCA}_T(x_i, v) : v \in y_j, \ldots y_t\}$. Let $L = \ell_1, \ldots, \ell_q$ be a merged list of L_x and L_y according to the first occurrence in the Euler tour of T. Then $\text{LCA}_G(x, y)$ is the node of greatest depth in $\{\text{LCA}_T(\ell_i, \ell_{i+1}) : \ell_i \in L_x, \ell_{i+1} \in L_y \text{ or } \ell_i \in L_y, \ell_{i+1} \in L_x\}$. □

THEOREM 4.3. *All-pairs-representative LCA is computed in* $O((n-1)\Sigma_{x\in V}|L_x|)$ *operations.*

4.4. A Lower Bound for All-Pair LCA

We show that the all-pairs-LCA problem in DAGs is as hard as transitive closure by reducing transitive closure in directed graphs to the all-pairs-LCA problem in DAGs. Begin by reducing transitive closure in directed graphs to transitive closure in DAGs, then reduce transitive closure in DAGs to the all-pairs-LCA problem in DAGs. The following lemma is a attributed to folklore, and its proof is omitted.

LEMMA 4.6. *Transitive Closure in DAGs is as hard as Transitive Closure in directed graphs.*

We next prove that the set of lowest common ancestors $\text{SLCA}(x, y)$ contains x and only x if and only if x is an ancestor of y:

LEMMA 4.7. $\text{SLCA}(x, y) = \{x\}$ *if and only if* $x \preceq y$.

Proof. Clearly $x \preceq y$ implies $x \in \text{SLCA}(x, y)$. The proof that x is the only element in $\text{SLCA}(x, y)$ follows by contradiction. Assume $q \neq x$ is an element of $\text{SLCA}(x, y)$. By the definition of $\text{SLCA}(x, y)$, $q \preceq x$ and $q \preceq y$. But, $q \prec x$ and is not a maximum element of $\{z | z \preceq x \vee z \preceq y\}$. Therefore q can not be in $\text{SLCA}(x, y)$. The only-if direction is trivial. □

Finally, we obtain the desired theorem:

THEOREM 4.4. *All-pairs LCA in DAGs is transitive closure hard.*

Proof. Computing transitive closure of a directed graph reduces to computing transitive closure of a DAG by Lemma 4.6. The elements of the transitive-closure matrix TC of a DAG can be described in the following way. $TC_{ij} = 1$ if i is an ancestor of j, and $TC_{ij} = 0$ otherwise. By Lemma 4.7, i is an ancestor of j if and only if $i = \text{LCA}(i, j)$. Then $TC_{ij} = 1$ if and only if $i = \text{LCA}(i, j)$ and $TC_{ij} = 0$ otherwise. □

ACKNOWLEDGMENT

We thank our anonymous reviewers for suggestions that have greatly improved the quality of this paper.

APPENDIX A: EXPERIMENTAL RESULTS FOR LCA IN TREES AND DAGS

We implemented a range of LCA algorithms for trees and DAGs. In this appendix we present a study of the performance of these algorithms on various input data. Our goal is to demonstrate ease of implementation[10] and investigate the practicality of the algorithms. For trees, we pinpoint the threshold single-child probability (skewness) when the query times of our algorithms are shorter than the query time of the naïve algorithm. For DAGs, we demonstrate that the ancestor-list-based all-pairs LCA algorithm given in Section 4 performs well on both low and high edge-density random DAGs.

The experiments are conducted on an Ultra-SPARC machine with 2 250MHZ CPUs, 1MB off-chip L2 cache and 2 GB memory running SunOS 5.6, and a 300MHZ Pentium-II with 256KB L2 cache and 384 MB memory running Red-Hat Linux 6.2. Studying the performance of these space-intensive algorithms on two different architectures helps us reach conclusions that are less system specific.

[10] Implementations are available at http://www.cs.pdx.edu/∼ps/code/soda01 (titled "All-Pairs Lowest Common Ancestors in Trees and Directed Acyclic Graphs") and http://www.cs.pdx.edu/∼ps/code/zack (titled "LCA Wizard").

A.1. Experiments on Trees

We tested three tree LCA algorithms: (1) a simple cache-optimized algorithm that uses no preprocessing, (2) the $\Theta(n \log n)$ preprocessing time and $\Theta(1)$ query time algorithm, and (3) the $\Theta(n)$ preprocessing time and $\Theta(1)$ query time algorithm. We refer to these algorithms as naïve, $\Theta(n \log n)$, and $\Theta(n)$ algorithms respectively.

Test Data Generation. To test the LCA algorithms we generated random binary trees, where each internal node in the tree has a single child with probability α. Thus, we use α as a knob that controls the depth and density of the tree. The nodes are stored in depth-first order to reduce the number of cache misses per query in the naïve algorithm. We report average query time for random queries performed on randomly generated trees of given depth.

Results. We compared the performance of the three algorithms on well balanced and skewed binary trees. The naïve LCA algorithm is best for small balanced trees. For larger trees of any kind, the $\Theta(n \log n)$ algorithm has the best query time. However, the preprocessing time for the $\Theta(n \log n)$ algorithm may dwarf the query time when the number of queries is small. The expected query time of the $\Theta(n)$ algorithm is less than that of the naïve algorithm only when the tree is very skewed. For binary trees of less than one million nodes, the query time of $\Theta(n)$ algorithm is faster (on our ultra SPARC machine) than that of the naïve algorithm only when the probability of a single child is greater than 0.93, implying that the expected tree depth is roughly $14 \log_2 n$. The gap in query time performance increases inversely with the probability of a second offspring.

Figures 6 and 7 show the query time for all three algorithms with various values of α and tree sizes. The query performance of the naïve algorithm depends on the tree's depth, which is a function of both α and N. The query performance of the $\Theta(n \log n)$ and the $\Theta(n)$ algorithms depends only on the number of table lookups performed. The choice of the appropriate tree-LCA algorithm depends on the number of nodes in the tree, the number of queries expected, and the depth of the tree constructed. Figures 6 and 7 also show the preprocessing time for the $\Theta(n)$ and the $\Theta(n \log n)$ algorithms for various trees sizes. Note that the preprocessing time does not depend on α.

There is a performance gap between the query time of the $\Theta(n)$ and the $\Theta(n \log n)$ algorithms, despite the fact that both queries do nothing more than a constant number of array lookups. This gap is because the $\Theta(n)$ algorithm query is composed of up to three queries of the $\Theta(n \log n)$ algorithm.

A.2. Experiments on DAGs

We evaluate our ancestor-list-based all-pairs-representative LCA algorithm on both sparse and dense graphs. We compare its performance to that of the following two algorithms:

- *Naïve* – The naïve algorithm simply walks up the DAG to find the ancestors of the queried nodes, from which it chooses a node of greatest depth. In the preprocessing stage, we traverse the DAG in breadth-first manner and assign depth to every node. This depth assignment requires $\Theta(n + m)$ operations. We expect this algorithm to perform well on sparse graphs, and especially well on star trees, where the LCA is found after at most 2 parent lookups and one comparison.

- *Matrix-Multiplication Transitive-Closure-Based LCA* – This algorithm computes the transitive closure of the DAG using matrix multiplication in the preprocessing stage to answer queries efficiently. To find the common ancestors of a given pair of nodes x and y, a binary AND on the rows x and y of the transitive-closure matrix is performed. The representative LCA is chosen by selecting a deepest common ancestor. This algorithm preprocesses the DAG using $\log n$ matrix multiplications, and answers queries in $\Theta(n)$ time. We use the PhiPac [6, 7] package for fast matrix multiplication. We expect this algorithm to perform well on dense graphs.

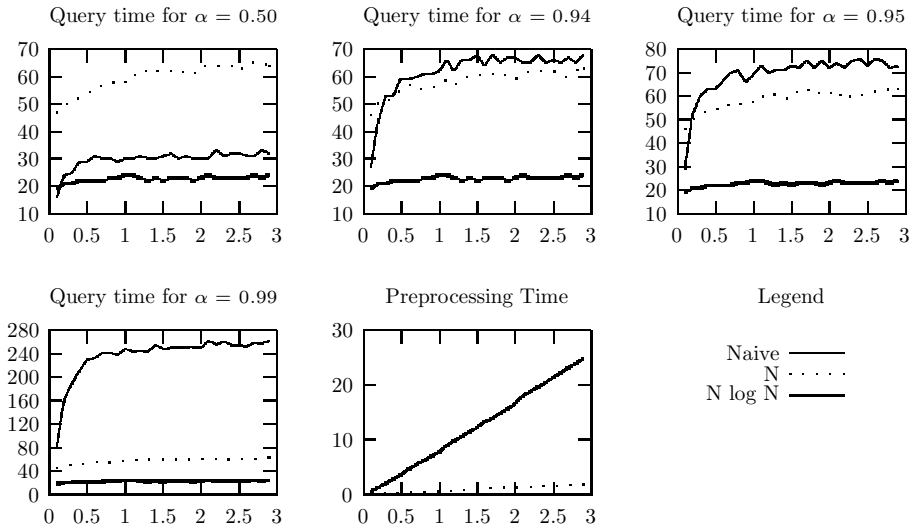

FIG. 6 Query time comparison (milliseconds) for skewed binary trees of different depths (expressed by the number of nodes in millions), on an Ultra-SPARC. The knob α controls the single-child probability (skewness) when generating the tree.

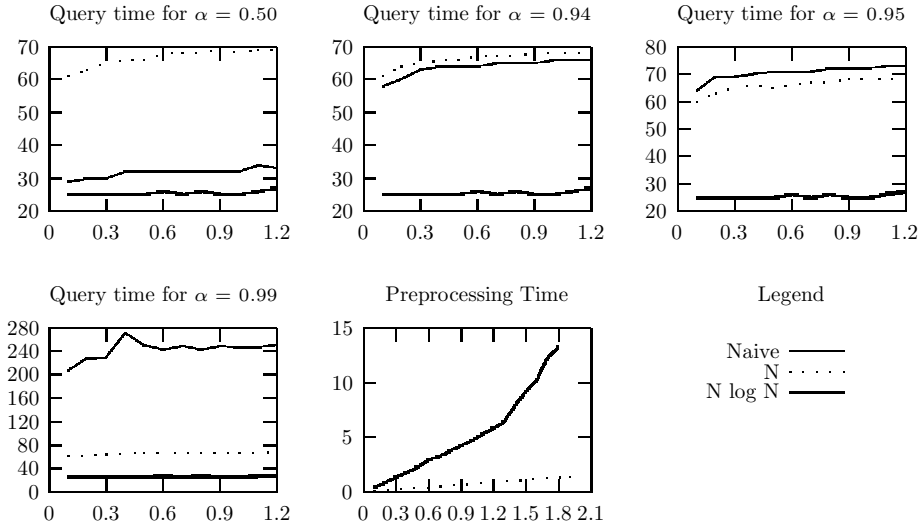

FIG. 7 Query time comparison (milliseconds) for skewed binary trees of different depths (expressed by the number of nodes in millions), on a Pentium-II. The knob α controls the single-child probability (skewness) when generating the tree.

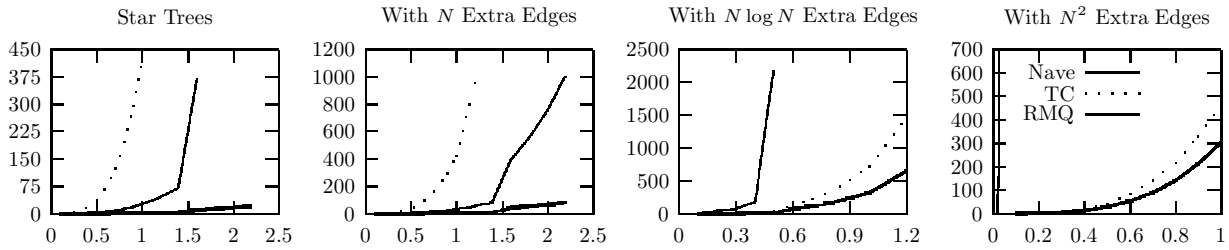

FIG. 8 All-pairs-LCA time (seconds) as a dependent on the number of nodes (expressed in thousands) for various DAG densities by the naïve, transitive closure, and ancestor-list-based LCA algorithms on an Ultra-SPARC. DAG density is varied by randomly adding edges to a star tree.

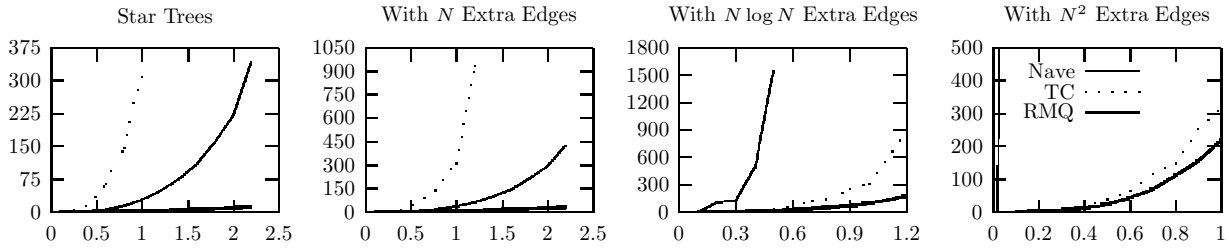

FIG. 9 All-pairs-LCA time (seconds) as a dependent on the number of nodes (expressed in thousands) for various DAG densities by the naïve, transitive closure, and ancestor-list-based LCA algorithms on a Pentium-II. DAG density is varied by randomly adding edges to a star tree.

Test Data Generation. We generated random DAGs of increasing density to test the efficiency of the three algorithms. A DAG is constructed by first generating a star and then adding extra edges between randomly chosen nodes. This method allows us to control the density of the DAG. Star trees are chosen as a starting point for the construction because they are the best-case instances for the naïve algorithm.

Results. We tested the three algorithms, naïve LCA, transitive-closure-based LCA and ancestor-list-based LCA, on the following DAG types: (1) star shaped DAGs, (2) star shaped DAGs augmented with n random edges, (3) star shaped DAGs augmented with $n \lg n$ random edges, and (4) complete DAGs. The results for all-pairs LCA are given in Figures 8 and 9. Although the ancestor-list-based LCA is not asymptotically better than the transitive-closure-based LCA, our experimental results suggest that it outperforms the other two algorithms.

The transitive-closure algorithm performs $\Theta(n)$ operations per query, independently of the density of the DAG. The naïve algorithm performs better than the transitive-closure algorithm in sparse DAGs, but its performance reduces dramatically in dense DAGs. The naïve algorithm traverses $O(n^2)$ edges per query, making the total running time for all-pairs LCA $O(n^4)$. The ancestor-list-based algorithm outperformed the naïve algorithm even on star trees, which are the best-case instances for the naïve algorithm; it outperformed the transitive-closure-based LCA algorithm on dense graphs, which are best-case instances for the transitive-closure-based LCA algorithm. Our experiments suggest that the ancestor-list-based algorithm has good performance for DAGs of all densities; it has optimal performance for trees and is the best for dense DAGs.

REFERENCES

[1] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and System*, 11(1):115–146, 1989.

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, Apr. 2000.

[3] M. A. Bender, G. Pemmasani, S. Skiena, and P. Sumazin. Finding least common ancestors in directed acyclic graphs. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 235–244, Jan. 2001.

[4] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, Apr. 1993.

[5] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, Apr. 1994.

[6] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Using PHiPAC to speed error back-propagation learning. In *Proceedings of ICASSP*, volume 5, pages 4153–4157, Munich, Germany, April 1997.

[8] V. Bouchitté and J.-X. Rampon. On-line algorithms for orders. *Theoretical Computer Science*, 175(2):225–238, 10 Apr. 1997.

[9] R. Cole and R. Hariharan. Dynamic LCA queries on trees. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 235–244, 1999.

[10] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 1–6, New York City, 25–27 May 1987.

[11] R. Ducournau and M. Habib. On some algorithms for multiple inheritance in object-oriented programming. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP'87 European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 243–252, Paris, France, 15–17 June 1987. Springer.

[12] A. Fall. *Reasoning with Taxonomies*. PhD thesis, Simon Fraser University, Burnaby, British Columbia, Canada, 1990.

[13] A. Fall. An abstract framework for taxonomic encoding. *Proceedings First International Symposium on Knowledge Retrieval, Use and Storage for Efficiency*, 1995.

[14] H. N. 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*, pages 135–143, 1984.

[15] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 246–251, 1983.

[16] A. Goldberg and D. Robson. *SmallTalk-80: The language and its implementation*. Addison Wesley, Reading, Mass., 1980.

[17] M. Habib, M. Huchard, and J. Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13(6):573–591, June 1995.

[18] M. Habib and L. Nourine. Structure for distributive lattices and applications. *Theoretical Computer Science*, 165(2):391–405, 1996.

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

[20] G.-V. Jourdan, J.-X. Rampon, and C. Jard. Computing on–line the lattice of maximal antichains of posets. *Order: A Journal on the Theory of Ordered Sets and Its Applications*, 11(2):197–210, 1994.

[21] D. McAllester. Boolean class expressions. *ACM SIGPLAN Notices*, 21(11):417–423, Nov. 1986.

[22] M. Morvan and L. Nourine. Generating minimal interval extentions. *Research Report No. 92-015, LIRMM Montpellier*, 1992.

[23] M. Nykänen and E. Ukkonen. Finding lowest common ancestors in arbitrarily directed trees. *Information Processing Letters*, 50(6):307–310, 27 June 1994.

[24] D. S. Parker. Partial order programming. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 260–266, Austin, Texas, 1989.

[25] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.

[26] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, Oct. 1979.

[27] B. Wang, J. Tsai, and Y. Chuang. The lowest common ancestor problem on a tree with unfixed root. *Information Sciences*, 119:125–130, 1999.

[28] Z. Wen. New algorithms for the LCA problem and the binary tree reconstruction problem. *Information Processing Letters*, 51(1):11–16, 1994.

[29] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Annual Symposium on Foundations of Computer Science*, pages 310–319, Palo Alto, California, USA, Oct. 1998.