Finding Least Common Ancestors in Directed Acyclic Graphs

Michael A. Bender* Giridhar Pemmasani Steven Skiena† Pavel Sumazin‡
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA
{bender,giri,skiena,psumazin}@cs.sunysb.edu

1 Introduction

One of the fundamental algorithmic problems on trees is how to find the *least common ancestor (LCA)* of a given pair of nodes. The LCA of nodes u and v in a tree is an 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 we preprocess T to answer LCA queries quickly on 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.

However, many similar combinatorial problems cannot be solved using the tree LCA algorithmic machinery because the ancestor queries are on more complicated directed structures. Examples of such combinatorial problems include preprocessing arbitrarily (unrooted) directed trees [NU94], computing the LCA on lattices (used to represent inheritance graphs) [AKBLN89], and finding modular coverings in inheritance graphs [DH87, HHS95].

In this paper we introduce a natural extension to all of these LCA problems. We develop algorithms for efficiently answering least common ancestor queries on *directed acyclic graphs* (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 ancestors of both u and v. (See Definitions 2.1 and 2.2.) We present an $o(n^3)$ preprocessing algorithm for answering least common ancestor queries on DAGs in constant time. We then initiate an experimental study of new and existing LCA algorithms on rooted trees and DAGs.

Least common ancestor queries on 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 [AKBLN89, Fal90, Fal95, GR80, HN96, McA86, Par87] and others. There has even been considerable interest in further decomposing inheritance graphs into modules that are efficient to query [DH87, HHS95]. The LCA operation is central to such object inheritance formalizations, because it is the natural method to resolve object dependence.
- *Analysis of Genealogical Data* – Algorithms for finding the LCA are useful for finding ancestral distances in the field of genealogy. In this field we are interested in a class of DAGs called pedigree graphs. Genealogical analysis of *pedigree graphs* is important for *linkage analysis* [JIS93, SGSJ94], a technique by which the genes causing genetic diseases can be identified. Linkage analysis requires quantification of the degree of inbreeding in a given pedigree, and the LCA supplies an alternative method for this calculation. The height of the LCA and the highest common descendant (the symmetric problem) provide information about inbreeding loops in the pedigree. In addition, many mathematical packages for analyzing pedigree only run on pedigree graphs with a bounded number of inbreeding loops and thus geneticists are forced to remove the loops [JIS93, SGSJ94] in an *ad hoc* manner. An algorithm based on LCA could automate this procedure.
- *Lattice Operations for Complex Systems* – Algorithms on lattices are used to model the dynamic and static behavior of complex systems arising in distributed computing [BR97, JRJ94, MN92].

*Supported in part by ISX Corporation and HRL Laboratories.
†Supported in part by NSF Grant CCR-9625669 and ONR Award N00149710589.
‡Supported by GAAN fellowship P200A97030199.

LCA queries arise in computing the covering of maximal ideal lattices.

1.1 Results

We summarize our theoretical and experimental results as follows.

Theoretical Results.

- We show how to preprocess the DAG for constant-time queries to answer whether nodes x and y have a common ancestor. We improve the naive algorithm from $O(n^3)$ to $\tilde{O}(n^\omega)$, where $\omega \approx 2.376$ is the exponent of the fastest known matrix multiplication algorithm [CW87].[1]
- We present an algorithm that solves the all-pair LCA problem for DAGs in $\tilde{O}(n^{\frac{\omega+3}{2}}) \approx \tilde{O}(n^{2.688})$ time. Thus we answer an open question posed by Nykänen and Ukkonen [NU94] in the affirmative; they ask whether the DAGs can be preprocessed in $o(n^3)$ operations for constant-time LCA queries. Before our work, no nontrivial lower bound was known for this problem, and the best known algorithm preprocesses the graph in $O(n^3)$ operations for $O(\log^2 n)$ queries [AKBLN89]. We obtain these tighter bounds by forging a relationship between the LCA and the all-pairs shortest path problem.
- We provide a near matching lower bound of $\tilde{O}(n^\omega)$ by reducing the transitive closure problem to all-pairs LCA in DAGs.
- We provide algorithms to calculate genealogical distances according to two distance measures, the shortest ancestral distance and the shortest LCA ancestral distance. We calculate the all-pairs shortest ancestral distance in $O(n^{2.575})$, and all-pairs shortest LCA ancestral distance in $\tilde{O}(n^{2.688})$.
- There has been considerable interest in fast (subcubic) algorithms for the all-pairs *shortest* path problem [Fre76, Sei92, AGM91, Zwi98, CZ97, CZ96, Zwi99]. We give subcubic bounds for the all-pairs *longest* path problem in DAGs and a matrix multiplication lower bound.

Experimental Results. We implement and measure the performance of various LCA algorithms for rooted trees and DAGs.

We compare three different tree LCA algorithms, (1) a naive algorithm that answers LCA queries by walking upward towards the root, (2) an optimal algorithm which uses $O(n)$ operations to preprocess the tree for constant-time LCA queries and (3) a sub-optimal algorithm that uses $O(n \log n)$ operations to preprocess the tree for constant-time LCA queries. Most $O(n)$ tree LCA algorithms are difficult to implement. However, the tree LCA algorithm of Bender and Farach-Colton [BFC00], which is a simple serialization of the PRAM LCA algorithm of Berkman and Vishkin [BV93] is easy to implement. The $O(n)$ and $O(n \log n)$ algorithms answer LCA queries by answering range minimum queries (RMQ) [BSV93].

We illustrate the trade-offs between various approaches. Our study suggests that the asymptotically optimal $O(n)$ preprocessing algorithm has better query time than the naive algorithm when the trees are skewed, whereas the naive algorithm has faster query time when the trees are more balanced. However, the $O(n \log n)$ preprocessing algorithm, which is used as a subroutine in the optimal algorithm, has the fastest query time in almost all our test cases.

We introduce a new easily implementable LCA algorithm for DAGs. Like the $O(n)$ and $O(n \log n)$ 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 transitive-closure-based algorithm for sparse DAGs, whereas the transitive closure-based algorithm outperforms the straightforward algorithm for dense DAGs. The RMQ-based algorithm does not guarantee better asymptotic behavior, however it impressively outperforms the other two approaches on all our test instances. Thus, the tools for optimal LCA in trees are useful for constructing good LCA algorithms for DAGs.

1.2 Related Work

Both off-line and online LCA problems on trees were studied by Tarjan [Tar79]. Harel and Tarjan showed that off-line LCA queries can be answered in constant time after only linear preprocessing of the tree T [HT84]. Schieber and Vishkin [SV88] introduced a new LCA algorithm with the same asymptotic bounds, but which was dramatically simpler. Berkman and Vishkin present a PRAM algorithm that uses $O(n)$ operations to preprocess the tree for answering queries in $O(\alpha(n))$ time [BV93]. The serialization of this algorithm yields an easily implementable optimal sequential algorithm, which is stated in [BFC00]. An optimal dynamic algorithm was given by Cole and Hariharan [CH99]. Other LCA algorithms on trees can be found in [BV94, WTC99, Wen94].

There has also been work on the LCA problem

[1] $f(n) = \tilde{O}(g(n))$ if $\exists c$ such that $f(n) = O(g(n) \log^c n)$.

for more general DAGs than rooted trees. Nykänen and Ukkonen [NU94] give a linear-time preprocessing, constant-time query algorithm for the LCA in arbitrarily directed trees. The idea to preprocess the object taxonomy in the compilation stage for fast LCA queries was first introduced by Ait-Kaci, Boyer, Lincoln and Nasr [AKBLN89]. They consider the problem of LCA on lattices, that are used to represent inheritance graphs. Ducournau, Habib, Huchard, and Spinrad [DH87, HHS95] consider the problem of finding modular coverings in inheritance graphs (the LCA can be used to find maximum covering). Their objective is to decompose lattices into modules that can be queried faster.

1.3 Organization

The paper is organized as follows. We begin Section 2 by defining the least common ancestors in DAGs. In Section 3 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. Problems on pedigree graphs, relevant to genealogical linkage studies, are discussed in Section 4. In Section 5 we consider the all-pairs longest path problem in DAGs. Finally, in Section 6 we present experimental comparisons of several LCA algorithms for trees and DAGs.

2 Definitions of LCA in DAGs

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

Definition 2.1. *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 $SLCA(x, y)$ to be the set of out-degree 0 nodes (leaves) in $G_{x,y}$. Then the least common ancestors of x and y are exactly the elements of $SLCA(x, y)$.*

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

Ait-Kaci, Boyer, Lincoln and Nasr [AKBLN89] proposed an alternative definition, where the LCA is described in terms of partially ordered sets. 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.

Definition 2.2. *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 $SLCA(x, y)$ be the set of the maximum elements of $\{z | z \preceq x \wedge z \preceq y\} \subseteq V$. Then the least common ancestors of x and y are exactly the elements of $SLCA(x, y)$.*

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

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

Definition 2.3. *The* height *of node x in a DAG, $height(x)$, is the length of the longest path from a source to x.*

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

Lemma 2.1. *The node of greatest height in a DAG that is an ancestor of both x and y is a least common ancestor of x and y.*

Proof. The proof is by contradiction. Let q be a common ancestor of greatest height. Assume q is not an LCA. This implies that there is a common ancestor z such that $q \prec z$, and that there is a path from q to z. By the height definition, $height(z) > height(q)$ which contradicts the assumption that q is an ancestor of greatest height. □

3 Finding All-Pairs LCA in DAGs

We now present an algorithm for the all-pairs common ancestor existence problem in DAGs. Then we show 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 $O(n^2)$ 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 LCA-representative matrix in $\tilde{O}(n^{\frac{\omega+3}{2}})$ operations. The fastest known matrix multiplication algorithm to date runs in $O(n^\omega)$ where $\omega \approx 2.376$ [CW87]. Thus our all-pairs common ancestor existence algorithm runs in time $\tilde{O}(n^{2.376})$, and our all-pairs LCA representative algorithm runs in time $\tilde{O}(n^{2.688})$.

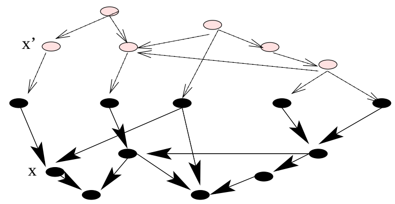

Figure 1: Doubling the DAG by reflecting through the sources. The original graph has solid edges.

3.1 The Common Ancestor Existence Algorithm

Common ancestor existence information is useful for the LCA representative computation, because it allows us to 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. The resulting graph is F; see Figure 1. Every vertex v' in $F - G$ is a reflection of some v in G. The nodes x and y have a common ancestor if (x', y) is in the transitive closure of F.

Thus we obtain the following theorem:

THEOREM 3.1. *The ancestor existence matrix can be computed in $\tilde{O}(n^\omega)$ time.*

Proof. F is constructed in $O(n^2)$ operations and is followed by a single transitive closure computation, which completes in $\tilde{O}(n^\omega)$ operations. Let S be the set of all sources in G, and S_v denote the sources that are ancestors of node v. Then, node $x' \in F$ is an ancestor of all nodes in S_x, and no node in $S - S_x$ is a decedent of x'. If $S_x \cup S_y = \emptyset$ then (x', y) is not in F_{tr}, and the common ancestor does not exist. Otherwise (x', y) is in F_{tr}, and a common ancestor exists. □

3.2 The All-Pairs LCA Algorithm

We solve the LCA problem by exploiting the close relationship between the LCA and the all-pairs shortest path problem. This relationship enables us to isolate a sublinear number of potential LCA nodes, from which we choose the node of greatest height (see Lemma 2.1).

In trees, the height of $LCA(x, y)$ and the distance between the nodes $dist(x, y)$ are related as follows:

$$dist(x,y) = height(x) + height(y) - 2 \times height(LCA(x,y)).$$

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

PROPERTY 3.1. *$x \preceq y$ implies that $height(x) \leq height(y)$.*

PROPERTY 3.2. *For $x \preceq y$, the length of the shortest path from x to y is $height(y) - height(x)$.*

We construct a weighted DAG L, which is used to reduce the representative LCA problem to the shortest path problem. We define an ancestral path from x to y in G as a path that is composed of the shortest path from x to some common ancestor z, concatenated with the shortest path from z to y. The DAG L satisfies the following requirements: (1) Properties 3.1 and 3.2 hold in L; (2) no two nodes in L have the same height; (3) all the ancestral paths in G are maintained in L; and (4) *only* the ancestral paths in G are maintained in L. Finding the length of the shortest path in L results in the discovery of a representative LCA, whose height can be reconstructed from the the length of the shortest path.

Assigning heights in G according to Definition 2.3 satisfies Property 3.1. We use a linear extension scheme described as follows: We sort the nodes in the (unweighted) graph G by their height, arbitrarily braking ties. Each node's order in this linear extension will be its new height. Now we assign weights to all edges in G according to Property 3.2. This height and weight assignment guarantees that requirements (1) and (2) are met.

Next we show that the highest common ancestor in this construction is an LCA. We do so by defining the set $SLCA_h(x, y)$ and proving that it is a subset of $SLCA(x, y)$. Note that in our construction $SLCA_h(x, y)$ will contain a single element.

DEFINITION 3.1. *$SLCA_h(x, y)$ is the set of nodes of greatest height that are ancestors of both x and y. An $LCA_h(x, y)$ is an element of $SLCA_h(x, y)$.*

LEMMA 3.1. *$SLCA_h(x, y) \subseteq SLCA(x, y)$.*

Proof. The proof is by contradiction. Assume node $q \in SLCA_h(x, y)$ and $q \notin SLCA(x, y)$. $q \preceq x$ and $q \preceq y$ by the definition of $SLCA_h(x, y)$. Since $q \notin SLCA(x, y)$, $\exists z$ such that $z \preceq x, z \preceq y$ and $q \prec z$, but since $q \prec z$, $height(q) < height(z)$ which contradicts the assumption that $q \in SLCA_h(x, y)$. □

Next we present the construction of L. Let $G' = (V', E')$ be the graph constructed from G by reversing all edges in G. Combine G and G' by adding an edge (v', v) for all $v' \in G'$ and $v \in G$, and let this edge be of weight 0. The resulting graph is L; see Figure 2. The shortest path in L from node x' to node y has length $dist(x, y)$ in G.

The construction of L satisfies requirements (3) and (4). We show that the shortest path from x' to y

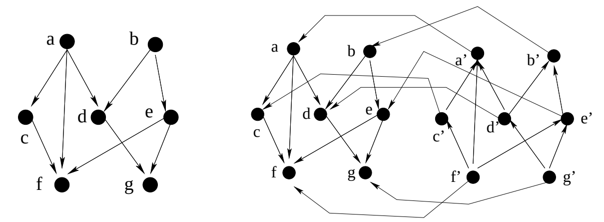

Figure 2: The original DAG G is on the left. On the right we show L constructed from G.

goes through the $LCA_h(x, y)$ in the following lemma:

LEMMA 3.2. *The shortest path from node x' to node y of graph L goes through the $LCA_h(x, y)$.*

Proof. We assume that x and y have a common ancestor in G, and therefore such a path exists in L. If x is a descendent of y, the length of the shortest path from x' to y' is $height(x) - height(y)$. Then y is the only $LCA(x, y)$ and the lemma is true. 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 $height(x)+height(y)-2\cdot height(z)$. The shortest path then, will go through the common ancestor of greatest height which is the $LCA_h(x, y)$. $\square$

We use the all-pair approximate shortest path algorithm given by Zwick [Zwi98] to approximate the height of the representative LCA. We choose this approximation algorithm because no subcubic exact all-pairs shortest path algorithm is known for DAGs with large weights, and L may have edge weights as large as $n-1$. Once we know the approximate height of the LCA_h, it is easier to sift through the nodes to find the LCA_h itself.

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. We find the approximate shortest path, and then examine the range of nodes whose heights are in the appropriate additive error bound. Since the shortest path can have length at most $2n-2$, there are $O(\varepsilon n)$ LCA_h candidates of consecutive heights. Let S be the set of these candidates, and let $T = A(x) \cap A(y) \cap S$, where $A(x)$ denotes the set of x's ancestors. The node $LCA_h(x, y) \in T$ and $max\{height(x) : x \in T\} = height(LCA_h(x, y))$. Thus finding $LCA_h(x, y)$ reduces to finding the node of maximum height in T.

In order to optimize we need to determine an appropriate value for ε. Since there are $O(n^2)$ pairs, and each pair requires examining $O(\varepsilon n)$ nodes, we expend $O(\varepsilon n^3)$ 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 3.2. *An all-pairs LCA matrix for a DAG G can be found in $\tilde{O}(n^{\frac{\omega+3}{2}})$ time.*

Proof. Observe that $\max\{w(i, j) \ : (i, j) \in E\} < n$ since n is the height of the highest 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 $2\cdot n^{\frac{\omega-1}{2}}$ possible LCA_h candidates for every pair. 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 height 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 $\tilde{O}(n^{\frac{\omega+3}{2}})$ $\square$

3.3 A Lower Bound for All-Pair LCA

We next show that the all-pairs LCA problem in DAGs is as hard as transitive closure by showing that it is as hard as transitive closure in directed graphs. We begin by explaining why transitive closure in DAGs is as hard as transitive closure in directed graphs:

LEMMA 3.3. *Transitive Closure in DAGs is as hard as Transitive Closure in directed graphs.*

Proof. All strongly connected components in a graph can be found in $O(n+m)$ time using a traversal algorithm. Contracting each strongly connected component into a single vertex yields a DAG. The transitive closure of an arbitrary directed graph can be computed by first contracting all strongly connected components into single nodes, computing the transitive closure on the resulting DAG, and then expanding the strong components into the full transitive closure matrix in $O(n^2)$ time. $\square$

We next prove that $LCA(x, y) = \{x\}$ if and only if x is an ancestor of y:

LEMMA 3.4. *$SLCA(x, y) = \{x\}$ if and only if $x \preceq y$.*

Proof. Clearly $x \preceq y$ implies $x \in SLCA(x, y)$. The proof that x is the only element in $SLCA(x, y)$ follows by contradiction. Assume $q \neq x$ is an element of $SLCA(x, y)$. By the definition of $SLCA(x, y)$, $q \preceq x$ and $q \preceq y$. Since $q \prec x$, it is not a maximum element of $\{z | z \preceq x \vee z \preceq y\}$, and can not be in $SLCA(x, y)$. $\square$

Finally, we obtain the following theorem:

THEOREM 3.3. *All-pairs LCA in DAGs is transitive closure hard.*

Proof. The directed graph G whose transitive closure we wish to compute reduces to a DAG by Lemma 3.3. 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 3.4, i is an ancestor of j if and only if $i = LCA(i, j)$. Then $TC_{ij} = 1$ if and only if $i = LCA(i, j)$ and $TC_{ij} = 0$ otherwise. □

4 Pedigree Graphs and Genealogy

Pedigree graphs, also known as family or hereditary trees, are a class of sparse DAGs used to represent ancestor relations. Each person has at most two parents, and hence a pedigree graph is a DAG where every node has in-degree at most two. Pedigree graphs are fundamental to *linkage analysis* [JIS93, SGSJ94], a technique by which the genes associated with genetic diseases can be identified. Linkage analysis depends on quantifying the degree of inbreeding in a given pedigree.

We are interested in two different measurements of inbreeding:

- The *shortest ancestral distance* between two nodes x and y returns the length of the shortest path through a common ancestor of x and y. Interestingly, this path does not necessarily pass through an LCA.
- The *shortest ancestral LCA distance* between two nodes x and y returns the length of the shortest path through an LCA of x and y.

Although these definitions seem closely related, they are not equivalent, as shown in Figure 3(b). For any two nodes, these distance measures can be used to describe the smallest inbreeding loop containing them. Inbreeding loops are formed by concatenating ancestral and descendant paths between nodes. We only show how to compute ancestral paths since descendant paths follow from symmetry. The shortest ancestral path is important for quantifying the genealogical relation between the pedigree members.

4.1 Shortest Ancestral Distance

In Section 3 we showed how to find a representative LCA and and the LCA inclusive shortest ancestral distance. Here we present a modified algorithm to find the shortest ancestral distance.

We modify L's construction by removing all edge weights. Finding the length of the shortest path in this modified graph is equivalent to finding the shortest ancestral distance. Since we use uniform edge weights we can use Zwick's exact all-pairs shortest path algorithm [Zwi98]. Zwick's algorithm finds the shortest path in $O(n^{2.575})$ operations. The following lemma uses arguments from Section 3.2:

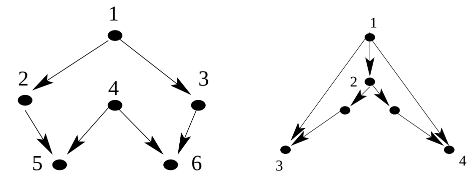

Figure 3: (a) $LCA(5, 6)$ is either 1 or 4. (b) $LCA(3, 4)$ is 2.

Lemma 4.1. *Given a graph G and nodes x and y, the shortest ancestral distance between x and y is the shortest distance between the associated nodes x' and y in the constructed graph.*

Thus we obtain the following theorem:

Theorem 4.1. *The all-pairs* shortest ancestral distance *is computed in $O(n^{2.575})$ time.*

4.2 Shortest LCA Ancestral Distance

We now show how to use the LCA algorithm given in Section 3.2 to calculate the shortest LCA ancestral distance. Figure 3 shows examples of pedigree that illustrate the difference between the two distance measures. The shortest LCA inclusive ancestral distance from node 5 to 6 in Figure 3(a) is 2. However, in the context of pedigree graphs we may consider not only topological information but also generational information. Specifically, we would like to be able to distinguish between nodes of equal height that are of different generations. Distinguishing between these nodes requires that all sources be labelled by generational height. For example, the information that node 1 in Figure 3(a) should be considered higher than node 4 can be embedded in the graph by setting the height of node 1 to be 1 and the height of node 4 to be 3. In our LCA algorithm we use a linear extension that takes into account generational information.

The all-pairs shortest LCA ancestral distance algorithm follows as a Corollary of Theorem 3.2:

Corollary 4.1. *The all-pairs shortest LCA ancestral distance can be computed in $\tilde{O}(n^{2.688})$ time.*

5 All-Pairs Longest Path

Finding the all-pairs *shortest* path matrix of a graph G is a basic problem in graph algorithms. The Floyd-Warshall algorithm runs in $O(n^3)$, and has been known since 1962. The first subcubic result is due to Fredman [Fre76]. Seidel [Sei92] and Alon, Galil and Margalit [AGM91] found $O(n^k)$ algorithms with $k < 3$, for undirected graphs and directed graphs respectively. Both solutions are based on fast matrix multiplication on rings.

Here we consider the problem of finding the all-pairs *longest* path matrix for unweighted DAGs. Each entry $M[x, y]$ is the length of the longest unweighted path from x to y. The problem is NP-complete for directed or undirected graphs by a simple reduction from Hamiltonian path. However, a simple breadth-first search algorithm solves the single-source (weighted) longest path problem in DAGs in $O(n+m)$ operations, which implies an $O(n^3)$ algorithm for the all-pairs unweighted problem.

Here we give subcubic bounds for the all-pairs unweighted longest path problem, and we provide a transitive closure lower bound. Recall that the LCA is naturally defined in terms of the height of each vertex, where the height is the length of the longest path from the root. Hence these simple results fall naturally from our previous techniques.

The transitive closure matrix can be derived from the all-pairs longest path matrix, and thus we obtain the following theorem:

THEOREM 5.1. *All-pairs longest path in DAGs is transitive closure hard.*

The following theorem shows how to compute the all-pairs unweighted longest path:

THEOREM 5.2. *The all-pairs unweighted longest path matrix for DAGS can be computed in* $\tilde{O}(n^{\frac{\omega+3}{2}})$ *time.*

Proof. We use the all-pairs shortest path algorithm proposed by [AGM97] on the appropriate graph to solve the all-pairs longest unweighted path problem. Alon et.al.'s algorithm solve all-pairs shortest path on graphs whose edge weights are $\{-1, 0, -1\}$. Given an unweighted DAG D, we assign each edge a weight of -1. $\square$

We leave as open the problem of finding an algorithm matching the lower bound, or a sub-cubic algorithm using lower precision arithmetic.

6 Experimental Results for LCA in Trees and DAGs

We implemented a range of LCA algorithms for trees and DAGs. In this section, we present a study of the performance of these algorithms on various input data. The experiments are conducted on an Ultra-SPARC with 2 GB memory running SunOS 5.6, and a Pentium-II with 384 MB memory running Linux 6.2.

6.1 Experiments on Trees

The LCA problem on trees has a reputation for being complicated. The first optimal solution (linear time preprocessing and constant time queries) by Harel and Tarjan [HT84] is difficult to implement, and even the simplified LCA by Schieber and Vishkin [SV88] is not easily implementable. Thus there are no experimental studies on this problem that we are aware of.

We tested three LCA algorithms: (1) a simple cache-optimized algorithm that uses no preprocessing, (2) an algorithm described in [BFC00, BV93] and elsewhere that has $O(n \log n)$ preprocessing time and $O(1)$ query time, and (3) an algorithm by Bender and Farach-Colton [BFC00] that has $O(n)$ preprocessing time and $O(1)$ query time. The algorithm of [BFC00] is a serialization of the PRAM algorithm of Berkman and Vishkin [BV93]. We refer to these algorithms as naive, $O(n \log n)$ and $O(n)$ algorithms respectively.

Test Data Generation. To test the efficiency of 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 height of the tree. The nodes are stored in depth-first order to reduce the number of cache misses per query in the naive algorithm.

Results. We compared the performance of the three algorithms on well balanced and skewed binary trees. The naive LCA algorithm is best for small balanced trees. For larger trees of any kind, the $O(n \log n)$ algorithm has the best query time. However, the preprocessing time for the $O(n \log n)$ algorithm may dwarf the query time when the number of nodes is small. The expected query time of the $O(n)$ algorithm is less than that of the naive algorithm when the tree is skewed. For binary trees of less than one million nodes, the $O(n)$ algorithm is faster than the naive algorithm when the probability of a single child is greater than 0.931.

There is a performance gap between the query time of the $O(n)$ and the $O(n \log n)$ algorithms, despite the fact that both queries do nothing more than a constant number of array lookups. This gap is due to the fact that the $O(n \log n)$ algorithm performs two array lookups for each query, whereas the $O(n)$ algorithm performs up to six array lookups. When the number of nodes in the tree is large, every array look up is likely to cause a cache miss.

Tables 1 and 2 show the query time for all three algorithms with various values of α and tree sizes. The query performance of the naive algorithm depends on the tree's height, which is a function of both α and N. The query performance of the $O(n \log n)$ and the $O(n)$ algorithms is based on the cache hit rate. The choice of the appropriate tree LCA algorithm depends on the number of nodes in the tree, the number of queries expected, and the

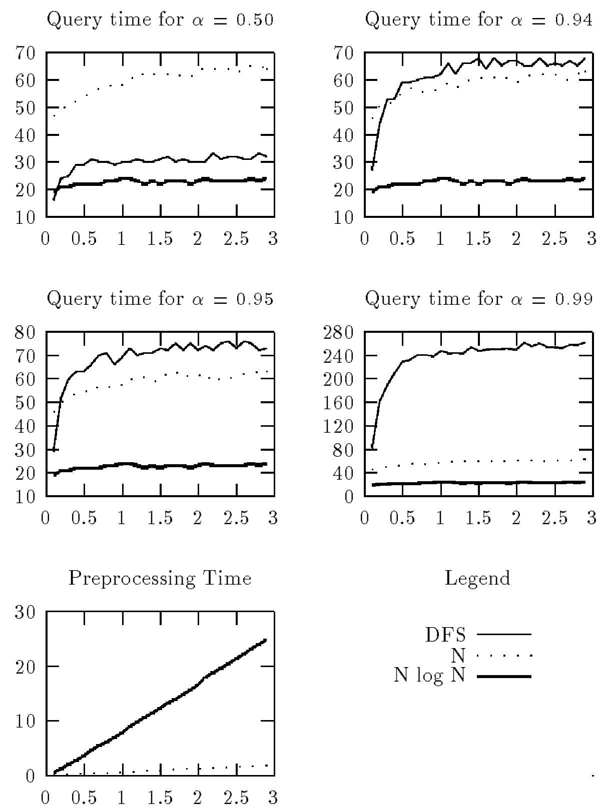

Table 1: Query time comparison (milliseconds) for skewed trees of different heights (expressed by the number of nodes in millions), on an Ultra-SPARC.

height of the tree constructed. Tables 1 and 2 also show the preprocessing time for the $O(n)$ and the $O(n \log n)$ algorithms for various trees sizes. Note that the preprocessing time does not depend on α.

For the configurations used, we observed that for balanced trees the cost of a single $O(n)$ query is equivalent to the cost of traversing 5 nodes by the naive algorithm. For $\alpha = 0.5, 0.95$ and 0.999, an $O(n)$ query is equivalent to the traversal of 9, 26, and 36 nodes respectively.

6.2 Experiments on DAGs

In this section we evaluate the following three algorithms for all pairs LCA on DAGs:

- *Naive* – The naive algorithm simply walks up the DAG to find the ancestors of the queried nodes, from which it chooses a node of greatest height. In the preprocessing stage, we traverse the DAG in breadth-first manner and assign height to every node. This height assignment requires $O(n+m)$ operations.
- *Transitive Closure-Based LCA* – This algorithm computes the transitive closure of the DAG in the

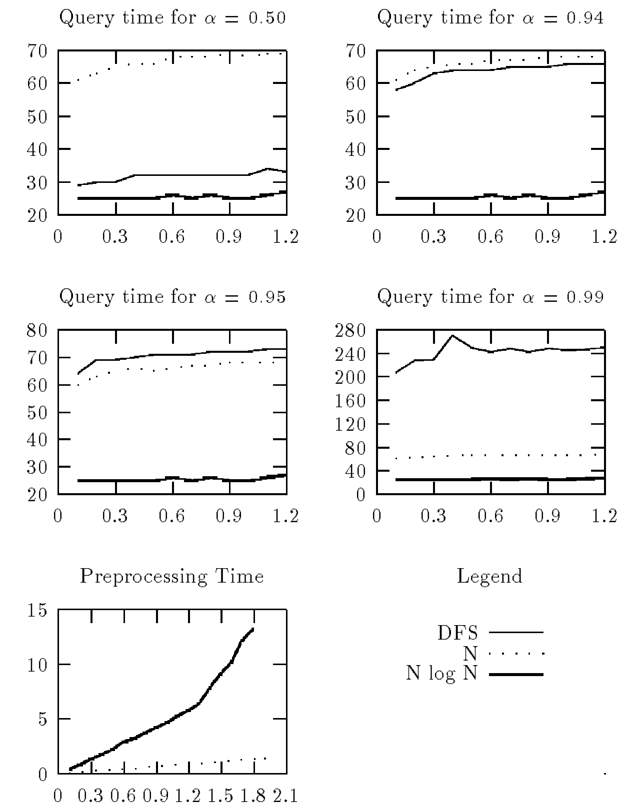

Table 2: Query time comparison (milliseconds) for skewed trees of different heights (expressed by the number of nodes in millions), on a Pentium-II.

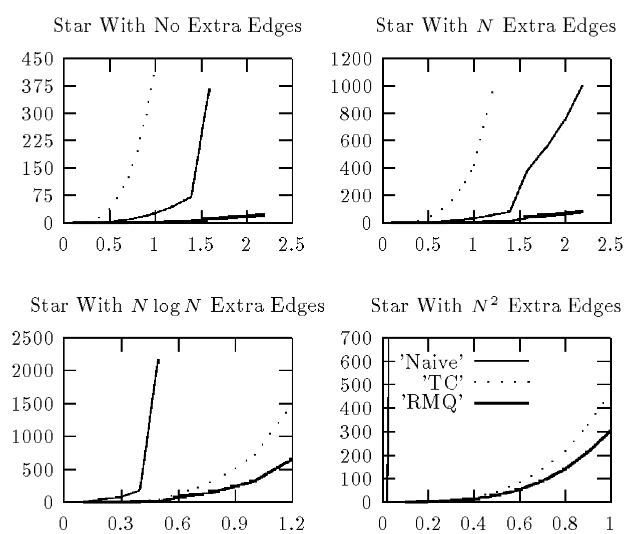

Table 3: All pairs LCA time (seconds) as a dependent on the number of nodes (expressed in thousands) for various DAG densities by the naive, transitive closure, and RMQ-based LCA algorithms on an Ultra-SPARC.

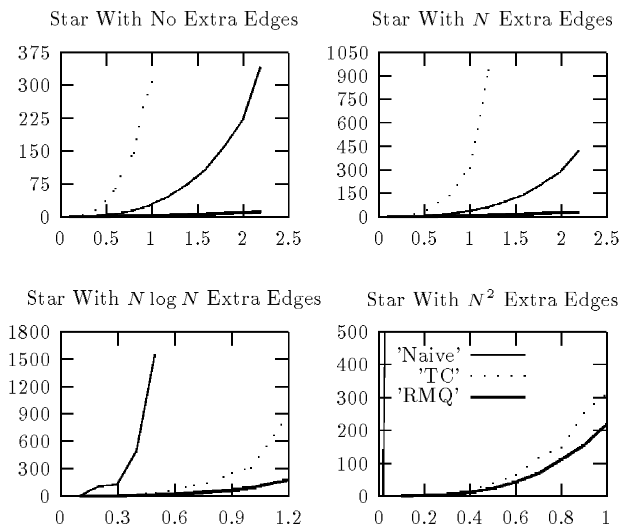

Table 4: All pairs LCA time (seconds) as a dependent on the number of nodes (expressed in thousands) for various DAG densities by the naive, transitive closure, and RMQ-based LCA algorithms on a Pentium-II.

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 highest common ancestor. This algorithm preprocesses the DAG using $\log n$ matrix multiplications, and answers queries in $O(n)$ time. We use the PhiPac [BAwCD97b, BAwCD97a] package for fast matrix multiplication.

- *Range Minimum Query-Based LCA* – RMQ efficiently finds the index of the minimal element in a given query interval, which is used to answer LCA queries in trees. We preprocess the DAG 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 induce the DAG $D = (V, S_2)$. Let $LCA_T(x,y)$ stand for LCA queries on the tree T, and let $LCA_G(x,y)$ stand for the LCA queries on the DAG G. A greatest height node in the set $\{LCA_T(u,v) : u \preceq x, v \preceq y\}$ is an $LCA_G(x,y)$. Maintaining the complete ancestor set for each node is unnecessary. Instead, with every node x we associate a list L_x, which is composed of x and all of its ancestors that have nonzero out-degree in D. Thus $LCA_G(x,y)$ is a highest node in the set $\{LCA_T(u,v) : u \in L_x, v \in L_y\}$. Observe that a Eulerian height ordering (equivalent to the RMQ array) of T's nodes can be used to trim the number of $LCA_T(u,v)$ queries from $O(n^2)$ to $O(n)$.

Test Data Generation. We generated random DAGs to test the efficiency of the three algorithms. A DAG is constructed by first generating a star and then adding extra random edges between randomly chosen nodes. This method allows us to control the density of the DAG.

Results. We tested the three algorithms - naive LCA, transitive closure-based LCA and RMQ 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 Tables 3 and 4. Although the RMQ based LCA is not theoretically better than the transitive closure-based LCA, our experimental results suggest that it performs better than the other two algorithms.

The transitive closure algorithm performs $O(n)$ operations per query, independantly of the density of the DAG. The naive algorithm performs better than the transitive closure algorithm in sparse DAGs, but its performance reduces dramatically in dense DAGs. The naive algorithm may traverse $O(n^2)$ edges per query, making the total running time for all-pairs LCA $O(n^4)$. The RMQ algorithm is the most flexible, has optimal performance for trees and is the best for dense DAGs.

References

[AGM91] Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. In *32nd Annual Symposium on Foundations of Computer Science*, pages 569–575, San Juan, Puerto Rico, 1–4 October 1991. IEEE.

[AGM97] Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, April 1997.

[AKBLN89] 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.

[BAwCD97a] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim 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.

[BAwCD97b] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Using PHiPAC to speed error back-propagation learning. In *Proceedings of ICASSP*, volume 5, pages 4153–4157, Munich, Germany, April 1997.

[BFC00] Michael Bender and M. Farach-Colton. The

LCA problem revisited. In *Latin American Theoretical Informatics*, pages 88–94, Punta del Este, Uruguay, 10–14 April 2000.

[BR97] Vincent Bouchitté and Jean-Xavier Rampon. On-line algorithms for orders. *Theor. Comput. Sci.*, 175(2):225–238, 10 April 1997.

[BSV93] Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, May 1993.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, April 1993.

[BV94] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, April 1994.

[CH99] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 235–244, 1999.

[CW87] Don Coppersmith and Shmuel 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.

[CZ96] E. Cohen and U. Zwick. All-pairs small-stretch paths. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 452–461, Burlington, Vermont, 14–16 October 1996.

[CZ97] E. Cohen and U. Zwick. All-pairs small-stretch paths. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 93–102, New Orleans, Louisiana, 5–7 January 1997.

[DH87] R. Ducournau and Michel Habib. On some algorithms for multiple inheritance in object-oriented programming. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry 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.

[Fal90] Andrew Fall. *Reasoning with Taxonomies.* PhD thesis, Simon Fraser University, Burnaby, British Columbia, Canada, 1990.

[Fal95] Andrew Fall. An abstract framework for taxonomic encoding. *Proc. First International Symposium on Knowledge Retrieval, Use and Storage for Efficiency*, 1995.

[Fre76] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, March 1976.

[GR80] A. Goldberg and D. Robson. *SmallTalk-80: The language and its implementation.* Addison Wesley, Reading, Mass., 1980.

[HHS95] Michel Habib, M. Huchard, and Jeremy Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13(6):573–591, June 1995.

[HN96] M. Habib and L. Nourine. Structure for distributive lattices and applications. *Theor. Comput. Sci.*, 165(2):391–405, 1996.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[JIS93] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[JRJ94] G.-V. Jourdan, J.-X. Rampon, and C. Jard. Computing on–line the lattice of maximal antichains of posets. *Order*, 1994. À paraître.

[McA86] D. McAllester. Boolean class expressions. *ACM SIGPLAN Notices*, 21(11):417–423, November 1986.

[MN92] M. Morvan and L. Nourine. Generating minimal interval extentions. *Research Report No. 92-015, LIRMM Montpellier*, 1992.

[NU94] Matti Nykänen and Esko Ukkonen. Finding lowest common ancestors in arbitrarily directed trees. *Inf. Process. Lett.*, 50(6):307–310, 27 June 1994.

[Par87] D.S. Parker. Partial order programming. *Technical Report CSD-870067, Computer Science Department, UCLA*, December 1987.

[Sei92] Raimund Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 745–749, Victoria, British Columbia, Canada, 4–6 May 1992.

[SGSJ94] A. A. Schffer, S. K. Gupta, K. Shriram, and R. W. Cottingham Jr. Avoiding recomputation in linkage analysis. *Human Heredity*, 44:225–237, 1994.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

[Tar79] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, October 1979.

[Wen94] Z. Wen. New algorithms for the LCA problem and the binary tree reconstruction problem. *Inf. Process. Lett.*, 51(1):11–16, 1994.

[WTC99] B. Wang, J. Tsai, and Y. Chuang. The lowest common ancestor problem on a tree with unfixed root. *Information Sciences*, 119:125–130, 1999.

[Yuv76] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Inf. Process. Lett.*, 4(6):155–156, March 1976.

[Zwi98] Uri 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, October 1998.

[Zwi99] Uri Zwick. All pairs lightest shortest paths, almost exact algorithms. In *Proc. of the 31th Ann. ACM Symp. on Theory of Computing*, pages 61–69, Atlanta, Georgia, USA, May 1999.