New Common Ancestor Problems in Trees and Directed Acyclic Graphs

Johannes Fischer*, Daniel H. Huson

Universität Tübingen, Center for Bioinformatics (ZBIT), Sand 14, D-72076 Tübingen

Abstract

We derive a new generalization of lowest common ancestors (LCAs) in dags, called the *lowest single common ancestor* (LSCA). We show how to preprocess a static dag in linear time such that subsequent LSCA-queries can be answered in constant time. The size is linear in the number of nodes.

We also consider a "fuzzy" variant of LSCA that allows to compute a node that is only an LSCA of a given percentage of the query nodes. The space and construction time of our scheme for fuzzy LSCAs is linear, whereas the query time has a sub-logarithmic slow-down. This "fuzzy" algorithm is also applicable to LCAs in trees, with the same complexities.

Key words: Algorithms, Data Structures, Computational Biology

1. Introduction

The *lowest common ancestor* (LCA) of a set of nodes in a static tree is a well-known concept [1] that arises in numerous computational tasks. The fact that the LCA of any two nodes can be determined in constant time, after a linear amount of preprocessing [9], is considered one of the classical results of computer science.

The LCA of two nodes u and v in a tree T is defined as the deepest node in T that is an ancestor of both u and v. This node, denote it by ℓ, can be

*Corresponding author.

Email addresses: fischer@informatik.uni-tuebingen.de (Johannes Fischer), huson@informatik.uni-tuebingen.de (Daniel H. Huson)

interpreted in at least two ways:

1. ℓ is an ancestor of both u and v, but no descendant of ℓ has this property.
2. ℓ lies on all root-to-u- and on all root-to-v-path, but no descendant of ℓ has this property.

Both of the above points are natural interpretations of the term "lowest" in the name "LCA," and they coincide in the case of trees. However, this is *not* the case for dags. As an example, look at nodes 3 and 4 in the dag of Fig. 1. Under interpretation (1), nodes 1 and 2 satisfy the definition of LCA, whereas under interpretation (2), only node 0 does. To distinguish between the two concepts in dags, we refer to interpretation (1) as the lowest common ancestors of u and v (in accordance to the literature), whereas interpretation (2) is referred to as the *lowest single common ancestor* (LSCA) of u and v.

In dags, only the first of the above interpretations (LCA) has been considered so far, but not the second one (LSCA). This is surprising, as LSCA has several interesting applications in computational biology, especially in "rooted phylogenetic networks." These are rooted, connected dags, whose leaves are labeled by biological taxa. They are used as a generalization of phylogenetic trees to model evolution in the presence of evolutionary events other than simple mutation and speciation [12]. They give rise to several applications of LSCA.

Firstly, if the network N was obtained as the "cluster network" of a set of rooted phylogenetic trees, all defined on the same set of taxa, then the LSCA of any two species s_1 and s_2 gives the evolutionary closest (hypothetical) species on which all input trees agree that it is an ancestor of s_1 and s_2. For the biologist, this is much better to interpret than the (probably many) species returned by LCA [12]. Secondly, if the network was obtained from trees on different, but overlapping taxon sets, then LSCA makes it possible to compute common ancestor relationships which are not represented in any of the input trees alone. Thirdly, recent algorithms for drawing rooted phylogenetic networks use a variant of LSCA to draw the network [10].

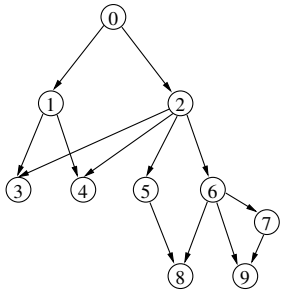

Figure 1: A directed acyclic graph G.

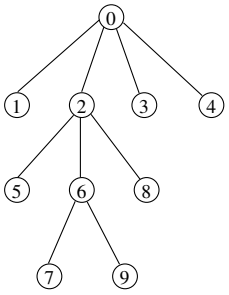

Figure 2: The LSA-tree T_G of the dag in Fig. 1.

1.1. Our Results

We propose to use LSCA instead of LCA. The first contribution of this article is summarized in the following theorem.

Theorem 1. *A static dag can be preprocessed in optimal $O(n+m)$ time into a data structure of size $O(n)$ such that subsequent LSCA-queries can be answered in constant time.*

This should be directly compared to the LCA-schemes in dags [3, 4, 6, 7, 14, 15], which have at least three computational drawbacks:

1. The preprocessing time is $\omega(n^2)$.
2. In order to answer LCA-queries in optimal time, no better solutions exist than storing the answers to all $\binom{n}{2}$ queries *explicitly* in $\Omega(n^2)$ space.
3. The only way to reduce the $O(n)$ output-size of a single LCA-query is to return an *arbitrary* member of the set of all LCAs (called *representative* LCA in the literature). However, this is *not* necessarily one with an additional useful property (such as minimal distance to the query nodes).

A possible objection against LSCA is that it is overly conservative, as the LSCA will always be an ancestor of any LCA. To account for this, in Sect. 4 we will allow for some kind of "fuzziness" by considering a *variant* of LSCA that requires to compute a node which lies *only on a certain percentage* of all root-to-W-paths, if W is the set of nodes for which the LSCA is computed.

Theorem 2. *A static tree or dag with n vertices and m edges can be preprocessed in $O(n+m)$ time such that* fuzzy-LSCA-queries *of the form "return the deepest node that lies on at least k root-to-W-paths" can be answered in $O(\min\{|W|\log n/\log\log n, n\})$ time, for any subset of nodes W, and any $k > |W|/2$.*

We emphasize that parameter k is specified only at *query* time, and must *not* be given to the preprocessing algorithm. An application where fuzziness is desired comes from metagenomics, where a main computational task is "taxonomical binning": given a set of environmental DNA sequences of unknown

taxonomical origin, assign each sequence to some node in a taxonomical tree or network, where the leaves represent different species and internal nodes represent higher order taxa such as Bacteria, Proteobacteria, Alpha-Proteobacteria etc. [11]. To *bin* a DNA sequence d, the sequence is first compared against a database of reference sequences to obtain a set of species W that have a segment of DNA that closely matches the sequence d. The LSCA is then computed and the read d is then assigned to the taxon associated with that node. However, due to *outlier* matches this could result in assigning a DNA sequence d to an overly high level taxon near the root of the taxonomy. To obtain a more specific assignment or "prediction" that is insensitive to outliers, one could replace the LSCA computation by the *fuzzy LSCA* computation to determine the LSCA of 90% of the matched species, say.

1.2. Outline

Having presented the necessary definitions and previous results in the following section, Sect. 3 and 4 are devoted to prove Thm. 1 and 2, respectively.

2. Definitions and Previous Results

Let $G = (V, E)$ denote a connected dag with vertices V and edges E. Let $n = |V|$ and $m = |E|$. We assume that G has a *unique* source-node (a node with in-degree 0) which we denote by $\perp$. If this is not the case, we add a new (virtual) *super-root* to V which is connected to all (original) source nodes of G. A *parent* node of $v \in V$ is any node w such $(w, v) \in E$. By *parents*(v) we denote the set of all parent nodes of v. A *tree node* is a node v with $|\mathit{parents}(v)| \leq 1$.

For trees, we have the well-known concept of lowest common ancestors [1].

Definition 1. *Given two nodes v and w in a tree, their* lowest common ancestor $\textsc{lca}_T(v, w)$ *is the node of maximum depth that is an ancestor of both v and w.*

As mentioned in the introduction, a static tree T can be preprocessed in linear time such that subsequent LCA-queries for a set of *two* nodes can be answered in constant time. This implies that the LCA of k nodes can be found in

$O(k)$ time, where the LCA for a set is defined inductively as $\text{LCA}_T(v_1, \ldots, v_k) = \text{LCA}_T(v_1, \text{LCA}_T(v_2, \ldots, v_k))$ for $k \geq 3$. For the dynamic case, we have:

Fact 1 ([8])**.** *An n-node tree can be maintained dynamically such that starting with the empty tree, a sequence of n leaf additions, interspersed with m LCA-computations, can be processed in $O(n+m)$ time and $O(n)$ space.*

Fact 2 ([2])**.** *A rooted tree on n nodes can be stored in $O(n)$ space such that each of the following operations can be processed in $O(\log n / \log \log n)$ time:*

- *mark*(v)*/unmark*(v)*:* mark *or* unmark *node* v*.*
- $\text{LMA}(v)$*: return the* lowest marked ancestor *of* v*.*

Note that the data structure from Fact 2 is not fully dynamic, as we require the tree topology be static.

3. A Preprocessing Scheme for Lowest Single Common Ancestors

Definition 2. *Let $v, w \in V$ be two nodes in G. Then the* lowest single common ancestor $\text{LSCA}_G(v, w)$ *of v and w is the unique node ℓ that lies on all paths from $\perp$ to v and on all paths from $\perp$ to w, but no descendant of ℓ has this property.*

In the special case of trees the notions of lowest single common ancestors and lowest common ancestors coincide, $\text{LCA}_T(v, w) = \text{LSCA}_T(v, w)$.

We remark that node ℓ in Def. 2 is well-defined, because if there are multiple nodes ℓ_i with the property that all paths from $\perp$ to v or w go through ℓ_i, these nodes must be ordered by the ancestor-relationship; hence, the *lowest* such node is *unique*. As with LCAs, the definition of LSCAs can be extended beyond pairs of nodes by setting $\text{LSCA}_G(w_1, \ldots, w_k) = \text{LSCA}_G(w_1, \text{LSCA}_G(w_2, \cdots, w_k))$.

Definition 3. *Let $v \neq \perp$ be a node in G. The* lowest single ancestor $\text{LSA}_G(v)$ *of v is the deepest node $\ell \neq v$ that lies on all all paths from $\perp$ to v.*

Note that $\text{LSA}_G(v)$ is equal to the unique parent of v if v is a tree node in G. We can now define the *LSA-tree* of a dag G [12].

Definition 4. *The* LSA-tree T_G *of* $G = (V, E)$ *is defined to have vertices* V*, and its edges are defined such that the parent node of* $v \neq \perp$ *is given by* $\text{LSA}_G(v)$*.*

It follows immediately from the definition of LSA that this yields a well-defined tree. See Fig. 2 for an example.

The next lemma will be fundamental to our scheme, as it gives the desired connection between LSCAs in G and LCAs in T_G.

Lemma 3. *Let* G *be a dag and* T_G *its corresponding LSA-tree. Further, let* $v, w \in V$ *be two arbitrary nodes in* G*. Then* $\text{LSCA}_G(v, w) = \text{LCA}_{T_G}(v, w)$*.*

Proof. Let $\ell = \text{LCA}_{T_G}(v, w)$. We need to show that (1) ℓ lies on all paths from $\perp$ to v or w in G, and (2) no descendant of ℓ has property (1).

Let $\ell = u_0 \to u_1 \to \cdots \to u_x = v$ be the (unique) path from ℓ to v in T_G. By the definition of LSA, $u_{i-1} = \text{LSA}_G(u_i)$ lies on all paths from $\perp$ to u_i for all $0 < i \leq x$, so we see that ℓ lies on all paths from $\perp$ to v. As the same is true for w, we get (1).

For proving (2), assume for the sake of contradiction that there is a descendant ℓ' of ℓ with property (1). Then ℓ' must lie on the path from ℓ to v (or w) in T_G, for otherwise ℓ' would not lie on all paths from $\perp$ to v (or w) in G. But this contradicts the fact that ℓ is the *lowest* node in T_G that is an ancestor of both v and w. □

In essence, Lemma 3 implies that all we have to do for constant-time LSCAs on G is to compute G's LSA-tree T_G and prepare it for constant-time LCA-queries. The next lemma suggests how the LSA-tree can be computed efficiently.

Lemma 4. *For any node* $v \neq \perp$*,* $\text{LSA}_G(v) = \text{LSCA}_G(\mathit{parents}_G(v))$*.*

Proof. Let $P = \mathit{parents}_G(v)$. First note that all paths from $\perp$ to v must go through some $p \in P$ and hence through $\ell = \text{LSCA}_G(P)$. For the sake of contradiction, assume there is a descendant ℓ' of ℓ such that $\ell' = \text{LSA}_G(v)$. Because $v \neq \perp$, there must be a non-empty path π from ℓ to some $p \in P$ which does not go through ℓ', for otherwise ℓ would not be the LSCA of P. Adding

v to the end of π gives a path π' from ℓ to v which does not go through ℓ', a contradiction. We thus conclude that ℓ is the LSA of v. □

Combining Lemmas 3 and 4, we see that $\text{LSA}_G(v) = \text{LCA}_{T_G}(\mathit{parents}(v))$. We can thus formulate the construction algorithm for the LSA-tree as follows. The idea is to build the LSA-tree in a top-down manner, as suggested by Lemma 4.

1. In a depth-first traversal of G, compute a topological sort L of G.
2. Initialize T_G to be the empty tree. T_G will be stored using the dynamic data structure for $O(1)$-LCAs from Fact 1.
3. For all $v \in V$ in the order of L: compute $\ell = \text{LCA}_{T_G}(\mathit{parents}(v))$ as the LSA of v, and add v to T_G as the child of ℓ.

Due to the order in which the vertices in G are processed, when visiting a node v in step 3, all nodes from $\mathit{parents}(v)$ have already been added to T_G, and hence the operation $\text{LCA}_{T_G}(\mathit{parents}(v))$ yields the correct result.

Let us now analyze the time complexity of the above algorithm. Steps 1–2 take $O(n + m)$ time in total. In each iteration of the loop in step 3, the time needed is $O(|\mathit{parents}(v)|)$. Because each node in V is visited exactly once, the overall time for step 3 is $O(n + m)$. This finishes the proof of Thm. 1

4. Fuzzy Lowest Common Ancestors

We now come to the problem of computing *fuzzy* lowest common ancestors in a static rooted tree $T = (V, E)$ with n nodes. As LSCA-queries are internally reduced to LCA-queries in the corresponding LSA-tree (see the previous section), this also captures the problem of computing *fuzzy* LSCAs in dags, and hence proves Thm. 2.

Definition 5. *Let T be a tree and let W denote an arbitrary set of nodes from T. Further, let k be an integer with $|W|/2 < k \leq |W|$. The* fuzzy lowest common ancestor *of W w.r.t. k is defined as the deepest node ℓ in T that is an ancestor of at least k nodes $v \in W$. This node ℓ is denoted by $\text{FLCA}_T(W, k)$.*

Throughout this section, let T_v denote the subtree of T rooted at $v \in V$.

4.1. A Naive Solution

First, find the *true* LCA ℓ of all nodes in W in $O(|W|)$ time. Then, in a depth-first traversal of T_ℓ, for each node v compute a number $\alpha(v, W)$, where $\alpha(v, W)$ counts the number of nodes from W in T_v, $\alpha(v, W) = |W \cap V(T_v)|$. The lowest node with $\alpha(v) \geq k$ is the node $\textsc{flca}_T(W, k)$.

The problem with this approach is that in the worst case, the size of T_ℓ is $O(n)$ — thus, answering FLCA-queries cannot be bounded better than $O(n)$, and the whole machinery used for constant-time (true) LCAs could have been avoided in the first place. We overcome this problem in the following section.

4.2. A Better Solution

The idea of an improved solution is to apply the naive algorithm from Sect. 4.1 to a *restricted* subtree of $T_{\text{LCA}(W)}$.

Definition 6. *The* LCA-skeleton-tree T_W^{LCA} *of* $W \subseteq V$ *is defined as follows. Its vertices* $V' \subseteq V$ *are defined such that* $v \in V'$ *iff* $\exists x, y \in W$ *such that* $v = \textsc{lca}_T(x, y)$*. Its edges* E' *are defined such that* $(v, w) \in E'$ *iff the unique path from* v *to* w *in* T *contains no nodes in* V' *other than* v *and* w*.*

Note that the number of nodes in T_W^{LCA} is $\Theta(|W|)$. The LCA-skeleton-tree is also called "the subtree of T induced by W" [5]. Because $\alpha(\cdot, W)$ can only change at nodes that are an LCA of two nodes $x, y \in W$, T_W^{LCA} contains all candidate nodes for $\textsc{flca}_T(W)$. Hence, it is enough to process T_W^{LCA} with the naive algorithm from Sect. 4.1 in order to answer $\textsc{flca}_T(W)$.

4.2.1. Building the LCA-Skeleton-Tree

It remains to show how to compute the LCA-skeleton-tree efficiently. Despite extensive research we could not find such an algorithm in the literature, although the concept of the LCA-skeleton-tree seems quite natural.

Algorithm 1 processes W element-wise and maintains the property that S is the LCA-skeleton-tree for the subset of W that has been processed so far. The variable ℓ always points to the root of S. We assume that T is preprocessed

Algorithm 1: Construction of the LCA-skeleton-tree T_W^{LCA} of W in T.

```
1  Let w_1 ∈ W be arbitrary. Set ℓ ← w_1. Initialize S with leaf w_1. Mark w_1.
2  foreach v ∈ W \ {w_1} do
3      if v is not marked then                          {otherwise nothing to do}
4          Add v to S and mark v. Compute ℓ' ← LCA_T(ℓ, v).
5          if ℓ' ≠ ℓ then
6              Add ℓ' and edges (ℓ', ℓ) and (ℓ', v) to S. Mark ℓ'. Set ℓ ← ℓ'.
7          else
8              Compute m ← LMA_T(v).
9              Let m' be the child of m (in T) on the path from m to v.
10             if there is an edge (m, w) in S s.th. w ∈ T_{m'} then
11                 Compute x ← LCA_T(v, w).
12                 Delete (m, w) from S. Add (m, x) and (x, w) to S. Mark x.
13                 if x ≠ v then add (x, v) to S.
14             else
15                 Add (m, v) to S.
16 Unmark all marked nodes in T.
17 return S
```

for marked-ancestor-queries with the data structure from Fact 2. During the construction, the marked nodes in T are exactly those nodes that are in S.

The aim of the outer for-loop (lines 2–16), where all $v \in W$ are successively inserted into S, is to make sure that $\text{LCA}(v, w)$ is present in S for all nodes w that are already in S. In line 3, we first calculate the LCA ℓ' of S's root ℓ and the new element v. The easy case is when ℓ' is different from ℓ, because then ℓ' will be the new root of S (lines 5–6), and $\text{LCA}(v, w) = \ell'$ for all nodes w that are already in S. Otherwise, we have to find out if there is an edge in S that has to be broken up by $\text{LCA}(v, w)$ for some node w already in S. To this end, we first compute the nearest (i.e., lowest) ancestor m of v that is already present in S by a marked ancestor query (line 8), and then check if there is an outgoing edge

$e = (m, w)$ from m that points "in the direction of v." Note that there can only be one such edge, for otherwise m would not be the nearest ancestor of v in S. If e exists, we know that $x = \text{LCA}(v, w)$ breaks up e into (m, x) and (x, w), and that v must be added as a child of x (lines 10–13). If there is no such edge e, v is simply added to S as a child of m (line 15).

4.2.2. Implementation Details

Two implementation details are in order at this point. The first is on line 9, where we need to find the child m' of m that lies on the path from m to v, without looking at all outgoing edges of m. This can be accomplished if we use the constant time LCA-algorithm presented by Jansson et al. [13], and execute $\text{LCA}_T(m, v)$. Since v is in T_m, the result of the LCA-query will be m itself. However, inspecting the proof of Lemma 3.2 in [13], we see that computing $\text{LCA}_T(m, v)$ is actually done by first finding the child m' of m that is on the path to v, and then moving to the parent of m'. Hence, we can find the correct child m' in constant time.

The second detail is on finding the (unique) edge (m, w) in line 10. This can be accomplished by *coupling* all edges (v, w) in S to the edge (v, w') in T, such that w' is the child of v that is on the path from v to w. Using the mechanism explained in the previous paragraph, w' can be found in constant time. Hence, if each edge added to S is coupled with its corresponding edge in T, then the edge (m, w) can be found in constant time by looking at the coupled edge of (m, m'), where m' is the node computed in line 9. Naturally, all edges from S need to be uncoupled once the computation of the LCA-skeleton-tree is done.

Observe that all operations in Alg. 1 can be implemented in $O(1)$ time, except marking the nodes and finding the LMA that take $O(\log n / \log \log n)$ time each. As each node in S is marked/unmarked exactly once and the size of S is $O(|W|)$, Thm. 2 follows.

5. Summary

We have introduced two new common ancestors operations in trees and dags, and given data structures for efficient query answering. This way, we have introduced the first sound definition of lowest common ancestors in dags (called LSCA) that allows for a linear preprocessing scheme with constant query time (Thm. 1). In applications where LSCAs are too restrictive one can use a relaxed version called fuzzy LSCA (Thm. 2).

Acknowledgments

We thank Regula Rupp for a fruitful discussion on this subject.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. FOCS*, pages 534–543. IEEE Computer Society, 1998.

[3] M. Baumgart, S. Eckhardt, J. Griebsch, S. Kosub, and J. Nowak. All-pairs ancestor problems in weighted dags. In *Proc. ESCAPE*, volume 4614 of *LNCS*, pages 282–293. Springer, 2007.

[4] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[5] R. Cole, M. Farach-Colton, R. Hariharan, T. M. Przytycka, and M. Thorup. An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. *SIAM J. Comput.*, 30(5):1385–1404, 2000.

[6] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theor. Comput. Sci.*, 380(1–2):37–46, 2007.

[7] S. Eckhardt, A. M. Mühling, and J. Nowak. Fast lowest common ancestor computations in dags. In *Proc. ESA*, volume 4698 of *LNCS*, pages 705–716. Springer, 2007.

[8] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. SODA*, pages 434–443. ACM/SIAM, 1990.

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

[10] D. H. Huson. Drawing rooted phylogenetic networks. *IEEE/ACM Transactions in Computational Biology and Bioinformatics*, 2008. (in press).

[11] D. H. Huson, A. F. Auch, J. Qi, and S. C. Schuster. MEGAN analysis of metagenomic data. *Genome Res.*, 17(3):377–386, 2007.

[12] D. H. Huson and R. Rupp. Summarizing multiple gene trees using cluster networks. In *Proc. WABI*, volume 5251 of *LNCS*, pages 296–305. Springer, 2008.

[13] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.

[14] M. Kowaluk and A. Lingas. Unique lowest common ancestors in dags are almost as easy as matrix multiplication. In *Proc. ESA*, volume 4698 of *LNCS*, pages 265–274. Springer, 2007.

[15] M. Kowaluk, A. Lingas, and J. Nowak. A path cover technique for LCAs in dags. In *Proc. SWAT*, volume 5124 of *LNCS*, pages 222–233. Springer, 2008.