Accepted Manuscript

A scalable approach to computing representative Lowest Common Ancestor in Directed Acyclic Graphs

Santanu Kumar Dash, Sven-Bodo Scholz, Stephan Herhut,
Bruce Christianson

PII: S0304-3975(13)00722-6
DOI: 10.1016/j.tcs.2013.09.030
Reference: TCS 9477

To appear in: *Theoretical Computer Science*

Received date: 27 June 2012
Revised date: 14 August 2013
Accepted date: 23 September 2013

Please cite this article in press as: S.K. Dash et al., A scalable approach to computing representative Lowest Common Ancestor in Directed Acyclic Graphs, *Theoretical Computer Science* (2013), http://dx.doi.org/10.1016/j.tcs.2013.09.030

A scalable approach to computing representative Lowest Common Ancestor in Directed Acyclic Graphs

Santanu Kumar Dash [1a], Sven-Bodo Scholz [2a], Stephan Herhut [3b], Bruce Christianson [4a]

[a] *School of Computer Science, University of Hertfordshire, Hatfield, UK*
[b] *Programming Systems Labs, Intel Labs, Santa Clara, CA., USA*

Abstract

LCA computation for vertex pairs in trees can be achieved in constant time after linear-time preprocessing. However, extension of these techniques to compute LCA for vertex-pairs in DAGs has been not possible due to the non-tree edges in a DAG. In this paper, we present an algorithm for computing the LCA for vertex pairs in a DAG which treats the DAG's spanning tree and its non-tree edges separately. Our approach enables us to tap the efficiency of existing LCA algorithms for trees. Furthermore, our algorithm decomposes the DAG into a set of component trees called clusters which significantly reduces the preprocessing necessary to incorporate non-tree edges in the LCA computation. Our algorithm seamlessly interpolates the performance graph between the best reported algorithms for trees and the best reported algorithms for DAGs depending on the incidence of non-tree edges in the DAG. Using the proposed techniques, it is possible to achieve near-linear preprocessing and constant query time for sparse DAGs.

Keywords: Lowest Common Ancestors, Directed Acyclic Graph

1. Introduction

The set of Lowest Common Ancestors (LCA) of two vertices $\mathtt{u}$ and $\mathtt{v}$ in a DAG is a set of vertices $L = \{\mathtt{l_1}, \mathtt{l_2} \cdots, \mathtt{l_n}\}$ such that all vertices in L are common ancestors of $\mathtt{u}$ and $\mathtt{v}$ and no other descendant of the vertices in L is an ancestor of $\mathtt{u}$ and $\mathtt{v}$ [1]. LCA queries find widespread application in the domain of programming languages for deciding object inheritance, in complex systems for lattice operations, in analysis of genealogical data,

computing maximum matching in graphs for string problems, in studying customer-provider relationships, etc.

A tree is a special case of a DAG; there is a unique LCA for all vertex-pairs in a tree. But vertex pairs in arbitrary DAGs may have multiple LCAs. In such a setting, a *representative LCA* is typically selected from the set of vertices satisfying the LCA properties. The initial approach to picking a representative LCA in the literature was to use the notion of depth of a vertex in the DAG [2, 3]. The depth was defined to be the longest hop distance of a vertex from the source of the DAG. In such a setting, it was possible for multiple vertices to have the same depth and ties were resolved arbitrarily to ensure that no two vertices have the same depth. Thus, it was possible to obtain a unique representative LCA for all vertex pairs. In later approaches [4, 5], a simpler approach was used to assigning depth values to vertices through topological ordering. In these works, the reachability matrix for the DAG was sorted according to the topological numbers of vertices and the representative LCA was defined to be the maximal witness of the boolean matrix product of the reachability matrix and its transpose. In other words, the representative LCA for a vertex-pair was nothing but the vertex with the highest topological number amongst the common ancestors of the vertex-pair. Similar to [4, 5], we define the representative LCA to be the vertex that has the highest topological number in the set of common ancestors of the vertex-pair in this paper.

The regular structure of trees and the unique LCA of vertex-pairs in trees make computation of LCA in trees relatively easier as compared to other kinds of graphs. LCA computation in trees can be computed in linear time using the *range minimum query* technique [6]. On the other hand, a rooted DAG contains an overlay of forward and cross edges on top of the edges of a spanning tree that covers the DAG. This additional layer of complexity inhibits the applicability of the simple and elegant RMQ technique to the case of DAGs. As a result, all of the reported techniques in the literature have resorted to computing the transitive closure of the entire DAG as a first step towards computing the LCA. By computing the closure, it is possible to easily identify the ancestors of vertices and consequently, the respresentative LCA for any given vertex-pair.

Computing the closure of a DAG is a computationally expensive operation. The fastest known algorithm for computing the closure relies on matrix multiplication and can be achieved in $O(n^{\omega})$ where $\omega(\sim 2.3)$ is the exponent of the fastest matrix multiplication algorithm reported in the literature

[7, 8, 9]. The additional drawback of this approach is that for sparse DAGs, where the structure is very similar to a tree, one is still forced to put up with computing the closure of the entire DAG. Ideally, one would hope for a technique that exploits the decomposition of the DAG into a spanning tree and a set of additional edges. LCA computation can then proceed by considering reachability information over the spanning tree and over the rest of the DAG separately. This would let us achieve a fast algorithm for computing pairwise LCAs in a sparse DAG.

In this paper, we present a technique to systematically decompose and pre-process a rooted DAG such that subsequent computation of representative LCA for any pair of vertices can be achieved in constant time. Reachability between vertices in our algorithm is represented as reachability over the spanning tree and/or reachability over additional non-tree edges. Consequently, we show that the LCA computation can be achieved by treating the tree and non-tree edges in isolation. We compute the transitive closure over tree edges using the range-interval labeling scheme (which has a linear complexity) [10] and the transitive closure for the non-tree edges using matrix multiplication. We further introduce the notion of a *cluster* which is a set of adjacent vertices with a single point of entry via a cross-edge. We demonstrate how the reachability information over non-tree edges can be abstracted to the level of clusters which further simplifies the LCA computation. More importantly, depending on the density of non-tree edges, the computational requirements of our algorithm interpolates seamlessly between the best reported techniques for trees and DAGs.

Thanks to our formulation of clusters, the computational costs of our approach are tied to the number of vertices with incoming or outgoing cross-edges. Specifically, if N_s and N_t are the number of vertices that have outgoing and incoming cross-edges respectively and $c = \max(N_s, N_t)$, our approach has a time complexity of $O(n+c^{2.575})$ and a space complexity of $O(n+c^2)$ where n is the number of vertices in the DAG. In contrast, techniques reported in the literature for DAGs currently have a time and space complexity of $O(n^{2.575})$ and $O(n^2)$ respectively. For all DAGs, $c < n$ and for many DAGs including sparse DAGs $c \ll n$. Therefore, our techniques can perform better than previously reported results on LCA computation. At the same time, depending on the value of c, the reported techniques can perform as efficiently as the best algorithm for LCA computation on trees (space and time complexity of $O(n)$) or the best algorithm for LCA computation on dense DAGs (time complexity of $O(n^{2.575})$ and space complexity of $O(n^2)$).

Theorem 1.1. *The representative least common ancestor of a vertex-pair in a DAG can be answered in constant time after $O(n + c^{2.575})$ preprocessing requiring $O(n + c^2)$ space.*

The rest of the paper is organized as follows. We discuss related work in section 2. In section 3, we discuss that any vertex `u` could reach another `v` either through the spanning tree or through a combination of spanning tree edges and cross edges. In the former case, we call `u` a tree ancestor (T) of `v` and in the latter case, we call `u` a cross ancestor (C) of `v`. Thereby, we categorize potential LCAs into one of the four categories - TT-PLCA, CT-PLCA, TC-PLCA and CC-PLCA - corresponding to the type of ancestral relationship between the potential LCA and the query vertices. The vertex with the highest topological number amongst these four PLCAs is the LCA of the query pair. In section 3, we discuss that the TT-PLCA can be computed by using the RMQ query on the spanning tree and show that the CT-PLCA need not be computed for a DAG. In section 4, we discuss techniques to identify the TC-PLCA for a vertex pair. In section 5, we discuss techniques to identify the CC-PLCA and conclude this paper in section 6.

2. Related Work

In this section, we give an overview of related work done in computing LCAs for vertex-pairs in trees and DAGs.

2.1. LCA computation in trees

While investigating multidimensional discrete range searching problems, the authors of [6] observed the equivalence between unidimensional range minimum searching and the LCA computation on Cartesian trees. The unidimensional range minimum query is defined as follows.

Definition Given an n-element array $A[1..n]$, the range minimum index query $RMQ_{idx}(i, j)$ returns the index of the smallest element between $A[i]$ and $A[j]$. For the sake of simplicity, we assume for our algorithm that the range minimum query (RMQ) takes i and j as argument and returns the element corresponding to the minimum index rather than the index.

Since the Cartesian tree is a binary tree, efficient schemes needed to be developed for the computation of LCA on nodes belonging to a generic

tree. To achieve this a labeling scheme for nodes was proposed in [11]. This scheme was able to answer LCA queries in constant time after a linear time preprocessing. However, the preprocessing for the algorithm presented in [11] remained complicated until some of the preprocessing steps were removed in [12]. A parallel approach to computing the LCA of two nodes using the simplified algorithms was also presented in [12].

A completely different approach to preprocessing trees for computing LCAs of two nodes was presented in [13]. The approach relied on the Euler tour of the tree [14] to generate a sequence of integers as an input to the preprocessing phase.

It was shown in [13] that the LCA of nodes `u` and `v` is always encountered between first visits to `u` and `v` during the Euler Tour of the tree. Let E store nodes in Euler Tour sequence and D store the depths of those nodes in the same sequence. For any two nodes `u` and `v`, let u_{idx} and v_{idx} denoted the indices of the first occurrence of these nodes in E. Then, $RMQ_{idx}(u_{idx}, v_{idx})$ on the array D returns the index of the LCA of the two nodes and $E[RMQ_{idx}(u_{idx}, v_{idx})]$ returns the LCA itself. However, it was observed in [13] that the RMQ_{idx} queries on the depth array is actually a restricted domain problem where consecutive entries differ by ± 1. This restricted domain property was exploited to develop efficient schemes for answering the $\pm 1 RMQ_{idx}$ and subsequently, the LCA query in constant time after linear time preprocessing [15] [2].

2.2. LCA computation in DAGs

Interest in LCA computation for vertex-pairs in DAG is recent. The LCA problem was initially studied in [3]. The authors reduced the all-pairs LCA problem to all pairs shortest distance query and proposed a solution that had a preprocessing time of $O(n^{2.688})$ and constant query time. Techniques from rectangular matrix multiplication discussed in [7] and [8] were used in [5] and [16] to further reduce the computational complexity of preprocessing to $O(n^{2.575})$.

Apart from the main results pertaining to LCA computation in DAGs, there have also been techniques developed to address special classes of DAGs. A path cover based approach to computing LCAs in DAGs having low width was discussed in [1]. The algorithm had a preprocessing time of $\tilde{O}(n^2 w(G))$[1]

[1] $\tilde{O}(f(n)) = O(f(n)\,\mathrm{polylog}(n))$

where $w(G)$ is the width of the DAG. This approach was also validated for DAGs having small depth and it was shown to possess the same worst case complexity as reported in [5].

For sparse DAGs, techniques to compute all-pair representative LCAs with a time complexity of $O(nm)$ were discussed in [5] and [16] where m is the number of edges in the graph. It was further shown in [17] that all-pair representative LCAs can be computed in $O(nm_{red})$ where m_{red} is the number of edges in the transitive reduction of the DAG. Based on the results regarding the number of strongly independent vertices in random DAGs [18], the authors of [17] note that the expected complexity is reduced to $O(n^2 \log n)$. However, the worst case complexity for this algorithm stands unchanged at $O(nm)$ because computation of transitive reduction itself takes $O(nm)$ time.

Similar to the techniques based on reachability matrices reported in [5] and [16], we also use matrix multiplication as the basic ingredient in our approach. Therefore, in this paper, we demonstrate the advantages of our algorithm by comparing it with the best reported algorithms based on matrix multiplication. As discussed earlier in this section, these algorithms have time and space costs of $O(n^{2.575})$ and $O(n^2)$ respectively [5, 16].

3. Identifying potential LCAs for a vertex pair

In this section, we give an overview of our approach to computing the LCA of a vertex pair in constant time after polynominal preprocessing. For the subsequent discussions, we assume that the DAG under consideration is rooted and static. If there are multiple parentless vertices in the DAG, we can always introduce a single parent for the parentless vertices to make the DAG rooted.

We perform a depth first walk of the DAG and classify the edges as tree, forward and cross edges [14]. This can be achieved by preordering and postordering the vertices in a DAG and has the same computational costs as a depth first traversal of the rooted DAG. For computing the LCA, the set of forward edges can be safely ignored. These edges introduce a redundant order between two vertices that are already connected.

3.1. Overview of our approach to computing representative LCAs

With forward edges eliminated from the DAG, we are now left to deal with tree edges and cross edges. Subsequently, whenever we refer to a DAG,

we assume that the DAG only contains tree and cross edges. For any vertex, we now have two kinds of ancestors. One kind, which we call *tree ancestors* reach the vertex through the spanning tree. The other kind, which we call *cross ancestors*, are all ancestors that are not tree ancestors.

We now give a brief overview of our approach to computing the respresentative LCA for two vertices `x` and `y`. Equations 1 and 2 show how the set of ancestors $A_{\mathtt{x}}$ and $A_{\mathtt{y}}$ for vertices `x` and `y` respectively are composed of tree ancestors (denoted by $A^t_{\mathtt{x}}$ and $A^t_{\mathtt{y}}$) and cross ancestors (denoted by $A^c_{\mathtt{x}}$ and $A^c_{\mathtt{y}}$) for the vertices.

$$
\begin{aligned}
A_{\mathtt{x}} &= A^t_{\mathtt{x}} \cup A^c_{\mathtt{x}} && (1)\\
A_{\mathtt{y}} &= A^t_{\mathtt{y}} \cup A^c_{\mathtt{y}} && (2)\\
LCA(x,y) &= \max_{topo}[A_{\mathtt{x}} \cap A_{\mathtt{y}}] && (3)\\
&= \max_{topo}[\,\{\,\max_{topo}(A^t_{\mathtt{x}} \cap A^t_{\mathtt{y}}),\, \max_{topo}(A^t_{\mathtt{x}} \cap A^c_{\mathtt{y}}),\\
&\quad \max_{topo}(A^c_{\mathtt{x}} \cap A^t_{\mathtt{y}}),\, \max_{topo}(A^c_{\mathtt{x}} \cap A^c_{\mathtt{y}})\,\}\,]\\
&= \max_{topo}\{\text{TT-PLCA, TC-PLCA, CT-PLCA, CC-PLCA}\} && (4)
\end{aligned}
$$

Similar to other reported techniques for LCA computation in DAGs , we assume the LCA for vertices `x` and `y` to be the vertex with the maximum topological number amongst the ancestors common to `x` and `y` [3] [5] [16]. This is used in equation 3. Equation 4 expands on equation 3 and shows how we can identify the representative LCA by shortlisting 4 potential LCAs (TT-PLCA, TC-PLCA, CT-PLCA, CC-PLCA) and then picking the one with the highest topological number.

We now show that if we rearrange the arguments of the LCA query such that the postorder number of the first argument is always greater than the postorder number of the second argument then we don't need to calculate the TC-PLCA in order to compute the LCA.

Definition For a vertex `v` in a DAG, `pre(v)` and `post(v)` denote the pre-order and post-order numbers for `v` in the spanning tree that covers the DAG.

Definition A query of the form LCA(`x`,`y`) is considered *argument-arranged* if `post(x)` > `post(y)`.

Lemma 3.1. *If* `pre(x)` > `pre(y)` *and* `post(x)` > `post(y)` *and* `x` *reaches* `y` *then* `x` *is a cross ancestor of* `y`.

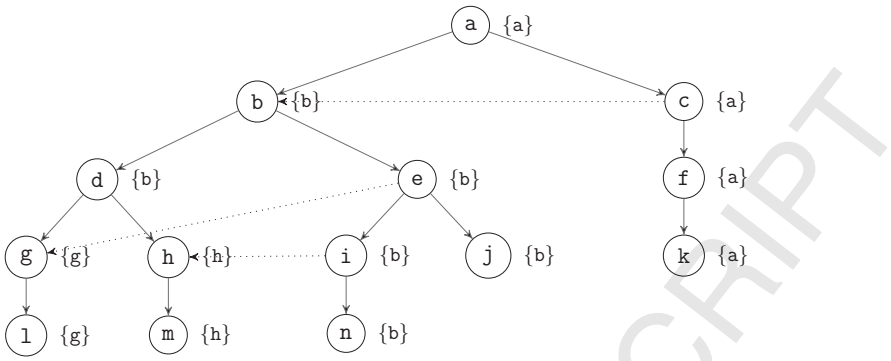


Figure 1: A directed acyclic graph with all vertices annotated with the corresponding clusterhead

Proof Straightforward. The proof follows directly from the manner in which preorder and postorder numbers are allocated during a depth first walk. □

Lemma 3.2. *It is not necessary to compute the* CT-PLCA *for an LCA query if the query is argument-arranged.*

Proof For an *argument-arranged* query LCA(x, y), there can be two cases.

- `pre(x)` < `pre(y)`: Since `post(x)` > `post(y)` by the virtue of argument-arrangement, it immediately follows that there is a path in the spanning tree from `x` to `y`. In this case, the LCA of `x` and `y` is `x`. Thus, the LCA of `x` and `y` can be easily computed during the computation of TT-PLCA and we do not need to consider the CT-PLCA for this case.

- `pre(x)` > `pre(y)`: Let `p` be an arbitrary cross ancestor of `x`. Then, `pre(p)` > `pre(x)` > `pre(y)` and `post(p)` > `post(x)` > `post(y)`. From lemma 3.1, it follows that if `p` reaches `y` then `p` is a cross-ancestor of `y` as well. It follows that $(A^c_{\mathtt{x}} \cap A^t_{\mathtt{y}}) = \phi$. Hence, we do not need to compute the CT-PLCA in this case as well.

Therefore, a simple arrangement of the arguments to the LCA query eliminates the need for computing the CT-PLCA. For the identifying TT-PLCA, the application of the RMQ technique to the spanning tree suffices. However, computing the TC-PLCA and the CC-PLCA is more involved and techniques to compute these are described in sections 4 and 5 respectively.

3.2. Decomposing a DAG into clusters

There are two kinds of vertices in the DAG; one kind has an incoming spanning tree edge and the other kind has incoming cross-edges in addition to the tree edge. We denote the set of vertices of the former kind as $\downarrow$ and the set of vertices of the latter kind as $\downarrow^{+c}$. For the vertices in $\downarrow^{+c}$, if we ignore the incoming edges to these vertices, the DAG can be seen as a composition of trees. The only way to reach a vertex in these trees from a vertex external to it is by passing through its root - a vertex that belongs to $\downarrow^{+c}$. We call these component trees of the DAG as *clusters* and the root of the cluster as the *clusterhead*.

Definition Clusters are component trees of a DAG obtained by discarding all incoming edges to vertices that have both incoming spanning tree edges and cross edges.

Fig. 1 shows the vertices of an example DAG annotated with clusterheads for the cluster to which they belong. After edge classification, cluster identification can be performed by a simple traversal of the spanning tree in $O(n)$ time where n is the number of vertices in the DAG.

If we are testing reachability from vertex `x` to vertex `y` and they belong to the same cluster, we only need to consider the edges of the spanning tree that covers the DAG. Otherwise, we have to additionally check for reachability from `x` to the clusterhead for `y` through a combination of tree and cross edges. In this context, the advantage that clusters offer is that we do not need to compute the transitive closure at the level of vertices but at the level of clusters; an approach that is significantly faster for sparse graphs [10].

Since the first step in computing the LCA is identifying common ancestors for the query vertices, reachability analysis has a direct bearing on the computation of the LCA. Due to the formulation of clusters, the computation of TC-PLCA and the CC-PLCA can be based on a combination of vertex labelling and small matrix lookups using the annotated labels in a manner similar to [10]. These small matrices are derived from a single matrix that captures the transitive reachability from cross edge sources to clusterheads. In the rest of this paper, for an argument-arranged LCA query, the annotation at the first argument is used to index the rows of these small matrices and the annotation at the second argument is used to index the columns of the small matrices.

4. Identifying the TC-PLCA

To compute TC-PLCA(x, y) one does not need to consider all ancestors of x in the spanning tree. Instead, it is sufficient to pick just 2 cross-edge sources (denoted as $s_<$ and $s_>$) which we call *proximals*.

Definition For the query TC-PLCA(x, y), the proximals are defined as the cross-edge sources that immediately precede and succeed x in the pre-order sequence of vertices and reach the clusterhead for y (hence, reach y itself). Evaluation of reachability between proximals and clusterheads considers both tree and cross edges in the DAG.

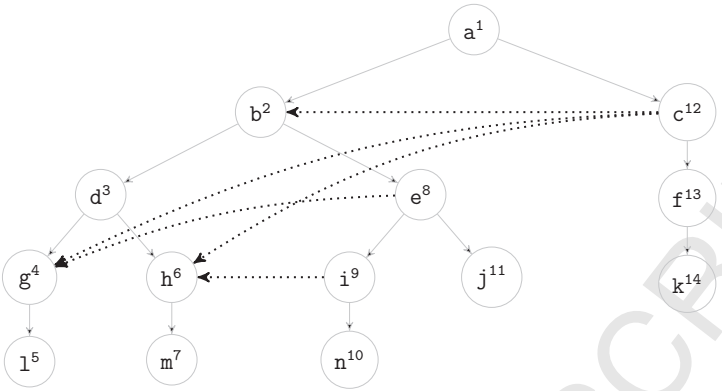
4.1. Picking appropriate proximals for a vertex

Let the TC-PLCA of two vertices x and y be denoted as l. l is y's cross ancestor and reaches y through a combination of tree and cross edges. Until we reach a cross-edge source in the path from l to y, the path is composed of tree-edges entirely. Therefore, if we compute the TT-PLCA of x with every cross-edge source that reaches y and pick the vertex with the maximum depth in the spanning tree amongst the computed TT-PLCAs, we obtain l. However, with the aid of lemma 4.1, we will show that it is not necessary to consider all cross-edge sources that reach y. Instead, it is sufficient to pick the proximals only.

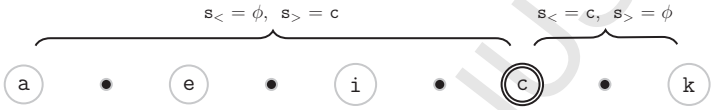
Lemma 4.1. *Let* [0,r] *be the range of preorder numbers of vertices in a DAG. For a given vertex* x*, the depth of* TT-PLCA(x,y) *in the spanning tree monotonically increases in the interval* [0, pre(x)] *and monotonically decreases thereafter.*

Proof Sketch. The TC-PLCA(x, y) in our case needs to be the lowest vertex in the spanning tree that reaches both x and a cross-edge source that reaches y. The TC-PLCA reaches every cross-edge source in the sub-tree rooted at the TC-PLCA. Therefore, it is easy to see that for the TC-PLCA to reach a cross-edge source outside the sub-tree, it is imperative for the TC-PLCA to be higher up in the tree. □

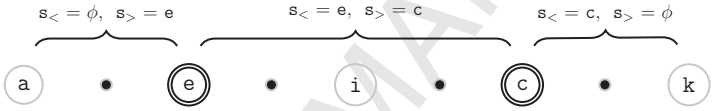
Let $\mathbb{S}_<$ be the set of cross-edge sources having a pre-order number less than x and reaching the clusterhead for y. The first proximal, which we denote as $s_<$, is the vertex with the highest pre-order number in $\mathbb{S}_<$. Similarly, let $\mathbb{S}_>$ be the set of cross-edge sources having a pre-order number greater



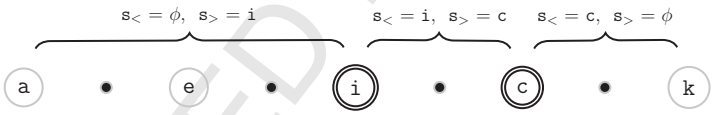
(a) Reachability between cross-edge sources and targets



(b) Proximals for all vertices for clusterhead b



(c) Proximals for all vertices for clusterhead g



(d) Proximals for all vertices for clusterhead h

Figure 2: Identifying proximals for all vertices for all clusterheads. Vertices are annotated with their pre-order numbers.

than x and reaching the clusterhead for y. The second proximal, which we denote as $s_>$, is the one with the lowest pre-order number in $\mathbb{S}_>$.

Identification of proximals simplifies the reachability information that needs to be captured for the TC-PLCA computation. Instead of considering reachability from one vertex to another, it is now sufficient to capture the transitive reachability information between cross-edge sources and clusterheads for the computing the TC-PLCA. This reduces the size of the reachability matrix that we need from a naive $O(n^2)$ to $O(c^2)$ where $c = max(N_s, N_t)$,

N_s and N_t being the number of cross-edge sources and cross-edge targets (clusterheads) respectively.

4.2. Variations in proximals

Answering arbitrary TC-PLCA queries requires us to annotate each vertex with proximals for each of the clusterheads. This is expensive and our next objective is to reduce the annotation overhead.

For a given vertex `y`, let all possible values of `x` in TC-PLCA(`x`,`y`) be written out in pre-order sequence. For all `x`'s in the pre-order sequence, the proximals change only when a cross edge source is encountered in the sequence. This is due to the fact that the proximals themselves are nothing but cross-edge sources.

This point is further illustrated in Fig. 2 which shows the variations in proximals for all vertices for all clusterheads. The solid dots in Fig. 2 represent intermediate vertices in the pre-order sequence. Subfig. 2(a) shows the reachability between cross-edge sources and cross-edge targets for our example graph and aids the understanding of Subfig. 2(b), 2(c) and 2(d). In Subfig. 2(b), 2(c) and 2(d) vertices are written out in pre-order sequence and cross-edge sources reaching clusterheads are marked with concentric circles. For each of the clusterheads, we also show how the values for proximals change as we run through the vertices written out in pre-order sequence.

Let us consider the cross-edge sources reaching clusterhead `h` in Subfig. 2(d). `i` and `c` are the two cross-edge sources reaching the clusterhead `h`. For the pre-order range [0, `pre(i)`], the first proximal $\mathtt{s}_<$ is undefined (denoted in the subfigure as ϕ). However, the second proximal $\mathtt{s}_>$ is defined as `i`. In the next sub-range (`pre(i)`, `pre(c)`] $\mathtt{s}_<$ is `i` and $\mathtt{s}_>$ is `c`. Finally, in the range (`pre(c)`, `pre(k)`] $\mathtt{s}_<$ is `c` and $\mathtt{s}_>$ is undefined.

The proximals for vertices in pre-order sequence vary only when a cross-edge source is encountered. This subtle observation enables us to deploy a labeling and indexing scheme for identifying the proximals for any vertex. Therefore, we can annotate each vertex `x` with an index that points to a cross-edge source which has the lowest pre-order number amongst cross-edge sources having pre-order numbers higher than `x`. Let the identified cross-edge source be denoted as `u`. Since there are no other cross-edge sources in the interval (`pre(x)`+1, `pre(u)`), the proximals are same for both `x` and `u`. Subsequently, we can get the proximals for `x` by looking up the proximals for `u`.

4.3. Building and indexing the TC-matrix

In order to be able to deploy a labeling and indexing scheme for identification of proximals, we first build a matrix called the TC-matrix which holds the proximal information for cross-edge sources. The rows of the TC-matrix are indexed by clusterheads and its columns are indexed by cross-edge sources. The TC-matrix for our example graph is shown in table 1. In this subsection, we first discuss techniques for constructing the TC-matrix. Subsequently, we also discuss techniques to annotate vertices with labels to index the TC-matrix.

The first step in computing the TC-matrix is to compute the transitive closure for the rechability information between cross edge sources and clusterheads. We multiply an adjacency matrix based on the cross-edges with a second matrix that captures reachability from clusterheads to cross-edge sources (through the spanning tree) to obtain an intermediate matrix γ. The result of the closure over γ shows reachability from one cross-edge source to another through a combination of cross and tree edges. We may need to further amend γ because some cross-edge sources may be reachable from another solely through the spanning tree. It is well known the transitive closure of an adjacency matrix has the same computational complexity as a matrix multiplication. Hence, obtaining the transitive closure of γ has the same computational complexity as a matrix multiplication. Creating a reachability matrix between cross-edge sources and clusterheads from γ is straightforward and can be obtained by observing the cross-edges. Let this reachability matrix be denoted as $\mathcal{M}$.

Definition $\mathcal{M}$ is a sub-matrix of the transitive closure matrix for the DAG that captures the reachability information between cross-edge sources and clusterheads.

The overall complexity of this reachability computation step can be limited to $O(c^{\omega})$ where ω is the exponent of the fastest matrix multiplication algorithm [8, 9].

Upon obtaining $\mathcal{M}$, we use algorithm 1 to obtain the TC-matrix. We scan through the list of all cross-edge sources reaching each clusterhead (cf. 4-5) and push the cross-edge source onto a stack after updating the value for $\mathtt{s}_<$ for it (cf. 6-7). The value of $\mathtt{s}_<$ for a cross-edge source is set to be the same as the $\mathtt{s}_<$ for the cross-edge source encountered immediately before it (denoted by $prev\_s_<$). Upon encountering a cross-edge source $\mathtt{s}$ that reaches

Algorithm 1 TC-matrix computation

```
 1: procedure ComputeTCPLCA(M)
 2:     prev_s< ← φ
 3:     Stack ← φ
 4:     for each clusterhead t do
 5:         for each cross edge source s do
 6:             s.s< ← prev_s<
 7:             Stack.push(s)
 8:             if s ⇝ t then
 9:                 while Stack is not empty do
10:                     v ← Stack.pop()
11:                     v.s> ← s
12:                 end while
13:                 prev_s< ← s
14:             end if
15:         end for
16:         while Stack is not empty do
17:             v ← Stack.pop()
18:             v.s> ← φ
19:         end while
20:     end for
21: end procedure
```

	e	i	c
b	{ϕ,c}	{ϕ,c}	{ϕ,c}
g	{ϕ,e}	{e,c}	{e,c}
h	{ϕ,i}	{ϕ,i}	{i,c}

Table 1: TC-matrix

the clusterhead, we pop all cross-edge sources on the stack to update the $\mathtt{s}_>$ values for them (cf. 8-14). Additionally, we also update the value for $prev\_s_<$ to s. This process continues until we reach the cross-edge source with the highest pre-order number. At this stage, if there are any additional cross-edge sources on the stack, we set the $\mathtt{s}_>$ value for these sources to be ϕ (cf. 16-19). Table 1 shows the TC-matrix for the DAG in Fig. 1 using this algorithm.

Algorithm 2 Labeling all vertices for indexing TC-matrix

```
1: Stack ← φ
2: procedure LabelVerticesForTCPLCA(G)
3:     LabelVertex(root(G), φ, φ)
4:     while Stack is not empty do
5:         v ← Stack.pop()
6:         v.colIdx ← φ
7:     end while
8: end procedure
9: procedure LabelVertex(n, rowIdx, colIdx)
10:    Stack.push(n)
11:    if n is a cross-edge source then
12:        if colIdx is φ then
13:            colIdx ← 0
14:        else
15:            colIdx ← colIdx + 1
16:        end if
17:        while Stack is not empty do
18:            v ← Stack.pop()
19:            v.colIdx ← colIdx
20:        end while
21:    end if
22:    if n is a clusterhead then
23:        if rowIdx is φ then
24:            rowIdx ← 0
25:        else
26:            rowIdx ← rowIdx + 1
27:        end if
28:    end if
29:    n.rowIdx ← rowIdx
30:    for each child of n in the spanning tree do
31:        LabelVertex(child, rowIdx, colIdx)
32:    end for
33: end procedure
```

In order to index the rows of the TC-matrix, we annotate vertices with a label for their clusterhead. In order to index the columns, we annotate

each vertex with a second label that is based on proximal information for the vertex. The vertices can be labeled in $O(n+m)$ using algorithm 2 where m is the number of edges in the DAG. We trigger this algorithm using the root of the spanning tree that covers the DAG (cf ll 3). The function LabelVertex is responsible for annotating the row and column indices at every vertex for accessing the TC-matrix. We annotate the row label (cf ll 10-21) and column label (cf ll 22-29) with the aid of the variables rowIdx and colIdx. Similar to algorithm 1, while annotating column labels, we keep pushing vertices onto the stack until a cross-edge source is encountered. Upon encountering a cross-edge source, we pop all vertices on the stack and label the vertices with a column index that corresponds to the encountered cross-edge source. Our example DAG with annotated with the column and row indices (in that order) is shown in Fig. 3.

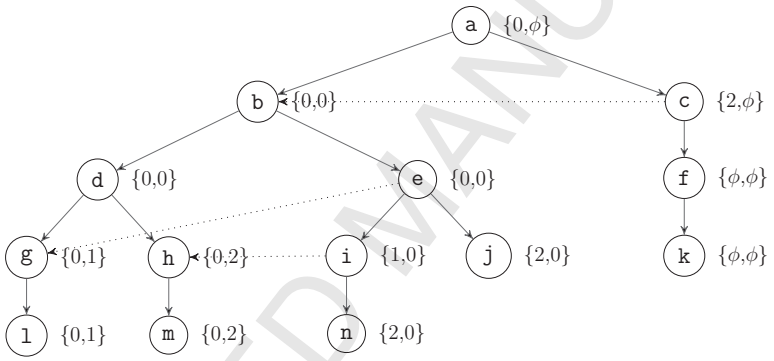


Figure 3: DAG vertices annotated with TC-matrix indices

5. Identifying the CC-PLCA

The CC-PLCA of a vertex pair has the highest topological number amongst the common cross ancestors that reach the pair. Computation of the CC-PLCA is done in three steps which are described below.

- Step 1 - We try to find out if any of cross-edge sources reach the both vertices in the query pair. If this is true, then the cross-edge source itself could be the CC-PLCA. For each query pair, we identify all cross-edge sources reaching both vertices and then choose one that has the highest topological number. We denote this vertex as τ. It may be the case

that τ does not exist as there no cross-edge source that reaches both vertices. Therefore, we also need to consider LCAs of the cross-edge sources reaching the vertices as detailed in the step 2.

- Step 2 - For a vertex pair {x,y} let the distinct cross edge sources $\mathtt{c_x}$ and $\mathtt{c_y}$ reach x and y respectively. The LCA of $\mathtt{c_x}$ and $\mathtt{c_y}$ could potentially be a CC-PLCA for the vertex pair. Let the candidate CC-PLCA identified in this manner be denoted as $\overline{\tau}$. If S_x and S_y denote the set of all cross edge sources reaching x and y respectively, $\overline{\tau}$ can be identified in two stages. In the first stage, we create a shortlist of vertices by taking one vertex each from S_x and S_y and computing their LCA. Let this shortlisted set of vertices be denoted as $S_{\overline{\tau}}$. In the second stage, we choose the vertex with the highest topological amongst the vertices in $S_{\overline{\tau}}$.

- Step 3 - We choose the vertex that has the higher topological number between τ and $\overline{\tau}$ which gives us the CC-PLCA for the query pair.

5.1. A simplified approach to computing $\overline{\tau}$

Instead of computing the pairwise LCAs as detailed in step 2 above, we can obtain $\overline{\tau}$ by computing the pairwise TT-PLCA.

Lemma 5.1. *For an LCA query* LCA(x,y)*, let S_x and S_y be the set of cross-edge sources reaching the clusterheads of* x *and* y*. Let, $S^t_{\overline{\tau}}$ denote the set of vertices obtained by computing* TT-PLCA *of all pairs of vertices $\mathtt{c_x}$ and $\mathtt{c_y}$ such that $\mathtt{c_x} \in S_x$ and $\mathtt{c_y} \in S_y$ and $\mathtt{c_x} \neq \mathtt{c_y}$. $\overline{\tau}$ can be obtained by the picking the vertex with the highest topological number in $S^t_{\overline{\tau}}$.*

Proof Vertices in S_x and S_y form a partial order due to the fact that the set of vertices in S_x and S_y is transitively closed. Let us consider two cross edge sources $\mathtt{c_x}$ and $\mathtt{c_y}$ from the sets S_x and S_y respectively. During the LCA computation of $\mathtt{c_x}$ and $\mathtt{c_y}$, we do not need to consider any cross ancestors of $\mathtt{c_x}$ and $\mathtt{c_y}$ towards identification of $\overline{\tau}$; they will be considered anyway when we consider other vertices in S_x and S_y and obtain their LCA. Therefore, it suffices to just compute the TT-PLCA of $\mathtt{c_x}$ and $\mathtt{c_y}$. This discussion can be inductively extended to all pair-wise combinations of a vertex each from S_x and S_y. Therefore, if $S^t_{\overline{\tau}}$ denote the set of vertices obtained by computing TT-PLCA of all pairs of vertices $\mathtt{c_x}$ and $\mathtt{c_y}$ such that $\mathtt{c_x} \in S_x$ and $\mathtt{c_y} \in S_y$ and $\mathtt{c_x} \neq \mathtt{c_y}$. $\overline{\tau}$ can be obtained by the picking the vertex with the highest topological number in $S^t_{\overline{\tau}}$. □

For the remainders of this discussion, we refer to vertices that are TT-PLCAs of cross edge sources as *extras*.

5.2. CC-PLCA computation for all pairs of clusterheads

So far, we have discussed the CC-PLCA computation for a given vertex pair. It is important to reiterate two aspects of the CC-PLCA problem at this stage. Firstly, we are interested in the CC-PLCA computation of all vertex-pairs instead of any given pair. Secondly, since any cross ancestor reaches a vertex through its clusterhead, it would be sufficient to compute the CC-PLCA of all pairs of clusterheads. In order to compute τ for all pairs of clusterheads, we need reachability information from cross-edge sources to clusterheads. Let us denote this matrix by $\mathcal{M}$. In order to compute $\overline{\tau}$, we need reachability information between *extras* and clusterheads. Let this information be encoded in another reachability matrix which we denote as $\mathcal{M}_x$.

Definition $\mathcal{M}_x$ is a sub-matrix of the transitive closure matrix for the DAG that captures the reachability information between *extras* and clusterheads.

The process of computing τ and $\overline{\tau}$ for all pairs of clusterheads from $\mathcal{M}$ and $\mathcal{M}_x$ respectively is straightforward. The details of computing the LCA from a reachability matrix can be found in [5]. The process is known as identification of the maximal witness in a boolean matrix product and has a best know runtime complexity of $O(c^{2.575})$ [5].

We have already computed $\mathcal{M}$ in section 4. We now discuss how $\mathcal{M}_x$ can be computed using $\mathcal{M}$ as an input. The initial step in the computation of $\mathcal{M}_x$ is to identify all *extras*. Simultaneously, we also need to keep track of which clusterheads are reached by which *extras*. One can naively enumerate the *extras* by obtaining pairwise TT-PLCA of all clusterheads. Since there are c cross-edge sources, the naive approach would entail $O(c^2)$ operations just to compute all TT-PLCAs. In addition for each of the TT-PLCA computation we have to keep track of clusterheads that the *extras* reach through reachability-set union operation. This would increase the worst-case complexity to $O(c^3)$. However, we will shortly show with the aid of a few lemmas that the algorithm can be simplified from a worst case complexity of $O(c^3)$ to $O(c^2 \log c)$. We first show through lemma 5.2 that it is not necessary to obtain pairwise TT-PLCA of all clusterheads.

Lemma 5.2. *Let $\mathcal{T}$ be a tree and the sequence $\mathcal{S} = \mathtt{v_1} \ldots \mathtt{v_p}$ be any p vertices from the tree written in post-order. Let* $\mathtt{l}$ *be the LCA of the nodes* $\mathtt{v_1}$ *and* $\mathtt{v_2}$ *and* $\mathtt{v_k}$ *be a vertex in* $\mathcal{S}$ *with the highest post-order number less than or equal to post*($\mathtt{l}$)*. Then,* LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = $\mathtt{l}$ *if* $2 \leq i \leq k$ *and* LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = LCA($\mathtt{l}$, $\mathtt{v_i}$) *if* $k < i \leq p$.

Proof Recall from the theory of post-order numbering for vertices in a tree that a vertex is numbered after numbering all its descendants. Therefore, if a vertex `x` is the ancestor of another vertex `y` then `x` is the ancestor of all other vertices that have post-order numbers in the range [`post(y)`, `post(x)`].

Case 1. $2 \leq i \leq k$: Let $\mathtt{l_i}$ = LCA($\mathtt{v_1}$, $\mathtt{v_i}$). In a tree, there is only one path from the root to every vertex which passes through all ancestors of vertex and there exists a total order amongst the ancestors of the vertex. We know that both $\mathtt{l_i}$ and $\mathtt{l}$ are ancestors of $\mathtt{v_1}$. So, there exists an order between $\mathtt{l_i}$ and $\mathtt{l}$. There are two cases possible, either $\mathtt{l_i}$ is an ancestor of $\mathtt{l}$ or $\mathtt{l_i}$ is a descendant of $\mathtt{l}$. We show by contradiction that neither is possible.

Since $\mathtt{post(v_1)} < \mathtt{post(v_i)} \leq \mathtt{post(v_k)} \leq \mathtt{post(l)}$ and $\mathtt{l}$ is an ancestor of $\mathtt{v_1}$, $\mathtt{l}$ is an ancestor of $\mathtt{v_i}$ as well. If $\mathtt{l}$ is a descendant of $\mathtt{l_i}$, then LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = $\mathtt{l}$. This is a contradiction since we know that LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = $\mathtt{l_i}$. Also, since $\mathtt{post(v_1)} < \mathtt{post(v_2)} < \mathtt{post(v_i)}$ and $\mathtt{l_i}$ is the ancestor of $\mathtt{v_1}$, $\mathtt{l_i}$ is an ancestor of $\mathtt{v_2}$ as well. If $\mathtt{l_i}$ is a descendant of $\mathtt{l}$, then LCA($\mathtt{v_1}$, $\mathtt{v_2}$) = $\mathtt{l_i}$. A contradiction again since we know that LCA($\mathtt{v_1}$, $\mathtt{v_2}$) = $\mathtt{l}$. Therefore, $\mathtt{l_i}$=$\mathtt{l}$.

Case 2. $k < i \leq p$: Once again, let $\mathtt{l_i}$ = LCA($\mathtt{v_1}$, $\mathtt{v_i}$). For this case, we have $\mathtt{post(l_i)} > \mathtt{post(v_i)} > \mathtt{post(l)} \geq \mathtt{post(v_k)} > \mathtt{post(v_1)}$. Also, we know that both $\mathtt{l_i}$ and $\mathtt{l}$ are ancestors of $\mathtt{v_1}$. There is a total order between $\mathtt{l_i}$ and $\mathtt{l}$. Since $\mathtt{post(l_i)} > \mathtt{post(l)}$, $\mathtt{l_i}$ must be an ancestor of $\mathtt{l}$ in the tree and all paths from $\mathtt{l_i}$ to $\mathtt{v_1}$ pass through $\mathtt{l}$. Thus, LCA($\mathtt{v_1}$, $\mathtt{v_i}$) can be rewritten as LCA($\mathtt{l}$, $\mathtt{v_i}$) for this case. □

Lemma 5.2 shows that if we have a sequence of vertices $\mathcal{S} = \mathtt{v_1} \ldots \mathtt{v_p}$ in post-order sequence, the list of unique TT-PLCAs for all vertex pairs can be obtained by a recursive operator. This operator computes the TT-PLCA of the first two vertices in the sequence, adds the TT-PLCA back into the sequence (according to its post-order number) and drops the first vertex from the sequence. Assuming that the operator terminates, it continues to run until it exhausts $\mathcal{S}$. Based on this observation, we formulate a recursive operator Λ to identify the set of *extras* and clusterheads reachable from these extras. We first give a formal presentation of Λ and then discuss its correctness and termination properties.

Definition Λ is an operation on the set of cross-edge sources $\mathcal{S}$ sorted in ascending order of their post-order numbers such that:

1. It calculates the TT-PLCA $\mathtt{l}$ of first two vertices in $\mathcal{S}$
2. It updates the clusterheads reachable from $\mathtt{l}$ with those reachable from the first two vertices in $\mathcal{S}$
3. It inserts $\mathtt{l}$ back into $\mathcal{S}$ while maintaining the vertex ordering in $\mathcal{S}$
4. It drops the first vertex in $\mathcal{S}$
5. If $\mathcal{S}$ has at least 2 elements, Λ calls itself with $\mathcal{S}$ as an argument otherwise Λ terminates

Lemma 5.3. Λ *correctly identifies all extras and all clusterheads reachable from these extras.*

Proof The TT-PLCA of $\mathtt{v_1}$ and $\mathtt{v_i}$ where $2 < i \leq k$ is $\mathtt{l}$. If $\mathtt{l}$ is not in $\mathcal{S}$ yet, we insert it in $\mathcal{S}$. We update the clusterheads reached by $\mathtt{l}$ with the clusterheads reached by $\mathtt{v_1}$ and $\mathtt{v_2}$. As Λ operates on $\mathcal{S}$, the pairwise TT-PLCA of $\mathtt{l}$ with vertices $\mathtt{v_{k+1}} \ldots \mathtt{v_p}$ will gives us the other *extras* that may arise due to $\mathtt{v_1}$. Therefore, we do not need $\mathtt{v_1}$ anymore and it can be dropped. Thus, Λ preserves the information about *extras*. If we assume termination of Λ (which will be proved later), at some stage $\mathtt{l}$ will become the first vertex in the sequence. If we find that $\mathtt{l}$ is not a cross-egde source, we add it to the set of *extras*.

Apart from reaching clusterheads directly, *extras* can also reach clusterheads transitively through other cross-edge sources reachable from them in the spanning tree. We need to show that when *extras* are dropped from $\mathcal{S}$, the identified set of clusterheads reachable from it is complete. Let $\mathtt{l_i}$ TT-PLCA of two vertices $\mathtt{v_i}$ and $\mathtt{v_{i+1}}$ such that $\mathtt{post(v_1)} < \mathtt{post(v_i)} < \mathtt{post(v_{i+1})} < \mathtt{post(l)}$. According to lemma 5.2, $\mathtt{l}$ also reaches $\mathtt{v_i}$ and $\mathtt{v_{i+1}}$. Therefore, $\mathtt{l_i}$ could be either $\mathtt{l}$ or one of its descendants in the spanning tree. If $\mathtt{l_i}$ is $\mathtt{l}$, reachable clusters from $\mathtt{l}$ are updated with those reachable from $\mathtt{v_i}$. Otherwise, clusterheads reachable from $\mathtt{l_i}$ are updated and $\mathtt{l_i}$ is inserted in $\mathcal{S}$ in a position between $\mathtt{v_{i+1}}$ and $\mathtt{l}$. $\mathtt{l_i}$ continues to remain in the sequence until it comes to the beginning of the sequence. Then, the information about clusterheads reachable from it is added to one its ancestors (which could be $\mathtt{l}$ or one of its descendants) and so on.

The discussion reveals a powerful property of Λ - no vertex is dropped without handing over clusterhead reachability information to a tree ancestor

of the vertex that is already in $\mathcal{S}$. More importantly, it is not possible for `l` to come to the head of the sequence until all the vertices that have post-order numbers in the range [`post(v1)`, `post(l)`] have been dealt with. As a result, we will ultimately reach a stage in Λ where all reachable clusterheads from `l` have been correctly identified. □

Lemma 5.4. *Λ terminates in $O(c)$ iterations.*

Proof Sketch. The proof follows straightforwardly from the observation that for a set of nodes in a tree, the number of unique LCAs generated through pairwise LCA operations on nodes in the set is no larger than the cardinality of the set. □

5.3. Algorithmic details

Having discussed the technical details of the process for computing the CC-PLCA, we are now ready to discuss the algorithmic details. In this subsection we present the algorithm that computes $\mathcal{M}_x$ from $\mathcal{M}$. The rest of the process for computing the CC-PLCA relies on computation of maximal witnesses in a boolean matrix product and can be found in the literature [5]. The algorithm discussed in the sub-section closely follows the theoretical discussions on CC-PLCA computation.

Algorithm 3 uses the reachability matrix $\mathcal{M}$ and the set of cross-edge sources C_s as input. It uses a priority queue as the basic data structure. The priority queue uses the post-order number as the ranking criteria. In the algorithm, we first initialize the priority queue with the set of cross-edge sources (cf ll 1-3). We dequeue a vertex `v1` and check whether it is one of *extras*. If it is, we add it to $\mathcal{M}_x$ (cf ll 6-9). Then we compute the TT-PLCA of `v1` and the head of the sequence $\mathcal{S}$. The TT-PLCA is denoted as `l`. We update the clusters reachable from the `l` as well (cf ll 12-13). Finally, we put back `l` in the sequence $\mathcal{S}$ (cf ll 14-18).

We know from lemma 5.4 that the outer loop in algorithm 3 runs $O(c)$ times. For each of the iterations, we insert a vertex in the sequence which takes $O(\log c)$ time because $\mathcal{S}$ is already sorted and we update the reachable clusterheads which takes another $O(c)$ time. So, the worst case time complexity for obtaining $\mathcal{M}_x$ from $\mathcal{M}$ is $O(c^2)$. At the same time, it is clearly evident that we do not need more space than $O(c^2)$. After obtaining $\mathcal{M}$ and $\mathcal{M}_x$, we just need to do 2 maximal witness of boolean matrix product operations and a comparison of two $c \times c$ matrices. Given that the maximal

Algorithm 3 Finding clusters reachable through CC-PLCAs that are not cross edge sources

```
1: for v ∈ C_s do                                  ▷ C_s is the set of cross-edge sources
2:     S.Enqueue(v)                                ▷ S is a priority queue
3: end for
4: M_x ← φ
5: while !S.empty() do
6:     v_1 ← Q.dequeue()
7:     if !v_1.isCrossSource() then
8:         M_x ← M_x ∪ v_1
9:     end if
10:    if !S.empty() then
11:        v_2 = S.head()
12:        l = TT_PLCA(v_1, v_2)
13:        l.clusters ← l.clusters ∪ getReachable(M, v_1)
14:        if v_2! = l then
15:            S.enqueue(l)
16:        end if
17:    end if
18: end while
```

witness operation has a time and space complexity of $O(c^{2.575})$ and $O(c^2)$ respectively, we can conclude that the CC-PLCA computation has a time and space complexity of $O(c^{2.575})$ and $O(c^2)$ respectively.

Table 2 shows the CC-PLCAs obtained for all pairs of clusterheads for our example graph. Indexing this matrix requires no further labelling since we have already annotated each vertex its corresponding clusterhead during the TC-PLCA computation. These clusterhead labels can be used to index table 2 as well.

The final step in computing the LCA of any two arbitary vertices is to

	b	g	h
b	b		
g	c	g	
h	c	e	h

Table 2: CC-PLCAs

return the vertex that has the highest topological number amongst the TT-PLCA, TC-PLCA, CC-PLCA. This operation can be achieved in constant time.

Theorem 5.5. *The representative least common ancestor of a vertex-pair in a DAG can be answered in constant time after $O(n + c^{2.575})$ preprocessing requiring $O(n + c^2)$ space.*

Discussion. In order to answer TC-PLCA and CC-PLCA queries, we use a combination of vertex labeling and small-matrix look-ups. Labeling of vertices for indexing TC-matrix and the CC-PLCAs matrix relies on a depth-first traversal of the DAG and can easily be integrated with the initial traversal of the DAG for edge classification. Similar to the depth-first traversal of a DAG, labeling has a time and space cost of $O(n+m)$ and $O(n)$ respectively.

The pre-processing phase of our algorithm computes the TC-matrix and the CC-PLCA matrix for answering TC-PLCA and CC-PLCA queries efficiently. The TC-matrix is computed from $\mathcal{M}$ with a time and space cost of $O(c^2)$. Computing $\mathcal{M}$ from the cross-edge information derived from the initial traversal of the DAG can be done using a sequence of matrix multiplications. This has a time and space cost of $O(c^\omega)$ and $O(c^2)$ respectively. Thus, the TC-matrix can be computed with an overall time and space cost of $O(c^\omega)$ and $O(c^2)$ respectively.

For the computation of the CC-PLCA matrix we need to perform an element-wise comparison of matrices that hold τ and $\overline{\tau}$ for all combinations of clusterheads. This operation can be done with a time and space cost of $O(c^2)$. The matrices that hold τ and $\overline{\tau}$ can be respectively obtained from $\mathcal{M}$ and $\mathcal{M}_x$ through the maximal witness of the boolean matrix product operation. This has a time and space cost of $O(c^{2.575})$ and $O(c^2)$ respectively. It has been further shown that $\mathcal{M}_x$ itself can be obtained from $\mathcal{M}$ in $O(c^2 \log c)$ time and $O(c^2)$ space. Thus, the overall time and space cost of obtaining the CC-PLCA matrix is $O(c^{2.575})$ and $O(c^2)$ respectively.

Finally, answering TT-PLCA queries in constant time requires us to preprocess the spanning tree that covers the DAG. This is achieved in $O(n)$ time and space using existing techniques for LCA computation in trees.

6. Conclusion

In this paper, we have proposed a fast and scalable technique to identify representative LCAs in a DAG. The computational requirement of our techn-

qiues scales itself based on the number of vertices in the DAG with incoming or outgoing cross edges. We achieved this by taking the spanning tree of the DAG as the base structure for our analysis and computing the transitive closure of the additional reachability information in the graph. Then, we identified potential LCAs depending on all possible types of paths that may exist between the potential LCA and the query vertex. The vertex with the maximum topological number amongst the PLCAs was identified as the representative LCA. The reported techniques provide best of both worlds in terms of computational requirements: LCA computation using our algorithm proceeds on trees and dense DAGs in the most efficient techniques reported currently for these structures in the literature. The computational requirements of our algorithm interpolate seamlessly for anything in-between these two categories. Also, unlike exisiting algorithms that compute the transitive closure of the entire DAG, we compute the closure for only cross-edge source and targets which renders our algorithms more efficient that those reported in the literature.

References

[1] Miroslaw Kowaluk, Andrzej Lingas, and Johannes Nowak. A path cover technique for LCAs in DAGs. In *SWAT '08: Proceedings of the 11th Scandinavian workshop on Algorithm Theory*, pages 222–233, 2008.

[2] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[3] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[4] Miroslaw Kowaluk and Andrzej Lingas. Unique lowest common ancestors in dags are almost as easy as matrix multiplication. In *Algorithms – ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 265–274. Springer Berlin Heidelberg, 2007.

[5] Artur Czumaj, Miroslaw Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, 2007.

[6] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, 1984.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[8] Don Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13(1):42 – 49, 1997.

[9] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th symposium on Theory of Computing*, STOC '12, pages 887–898, 2012.

[10] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 75, 2006.

[11] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[12] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[13] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22(2):221–242, 1993.

[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2009.

[15] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, 2002.

[16] Matthias Baumgart, Stefan Eckhardt, Jan Griebsch, Sven Kosub, and Johannes Nowak. All-pairs ancestor problems in weighted DAGs. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *Lecture Notes in Computer Science*, pages 282–293. 2007.

[17] Stefan Eckhardt, Andreas Mühling, and Johannes Nowak. Fast lowest common ancestor computations in DAGs. In *Algorithms ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 705–716. 2007.

[18] Amnon B. Barak and Paul Erdös. On the maximal number of strongly independent vertices in a random acyclic directed graph. *SIAM Journal on Algebraic and Discrete Methods*, 5(4):508–514, 1984.