Stockholms universitet

Transforming DAGs into "2-LCA DAGs" and generating BMGs from "2-LCA DAGs"

Omvandling av DAG:er till "2-LCA-DAG:er" och generering av BMG:er från "2-LCA DAG:er"

Emil Eriksson

Handledare: Marc Hellmuth

Examinator: Lars Arvestad

Inlämningsdatum: 2026-01-05

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

The main aim of this paper is to provide a polynomial time algorithm for transforming an arbitrary evolutionary directed acyclic graph (DAG) into a "2-LCA DAG" (i.e. a DAG with a unique least common ancestor between all pairs of leaves). We are not interested in finding a transformation that can preserve graph-related properties in the underlying evolutionary DAG, but rather in finding an efficient method that can be repeatedly used in order to generate large datasets of random "2-LCA DAGs". Large datasets of this kind are of particular interest for researchers who want to study properties and potentially derive conjectures related to Best Match Graphs (BMGs). Aside from this, we also derive polynomial time algorithms for generating random DAGs, identifying "2-LCA DAGs" and generating BMGs. The reader should be well aware that the underlying biology will not be considered. That is to say, only the mathematical properties of the evolutionary DAGs are considered of relevance. Nevertheless, we want to underline that the derived results have practical real-world application.

Smmanfattning

Huvudsyftet med denna artikel är att tillhandahålla en polynomisk tidsalgoritm för att transformera en godtycklig evolutionär riktad acyklisk graf (DAG) till en "2-LCA DAG" (en riktad acyklisk graf där varje par av löv har en unik lägsta gemensamma förfäder). Vi är inte intresserade i att finna en transformation som kan bevara graf-relaterade egenskaper i den underliggande evolutionära DAG:en, utan snarare att finna en effektiv metod som kan återupprepas för att producera stora datamängder av slumpmässigt genererade "2 LCA DAG:er". Stora datamängder av denna typ är av särskilt intresse för forskare som vill studera egenskaper och potentiellt härleda förmodningar relaterade till Best Match Grafer (BMG:er). Bortsett från detta, härleder vi även polynomiska tidsalgoritmer för att generera slumpmässiga DAG:er, identifiera "2-LCA DAG:er" och generera BMG:er. Läsaren bör vara väl medveten om att den underliggande biologin inte kommer att beaktas. Det vill säga, endast de matematiska egenskaperna hos de underliggande evolutionära DAG:erna anses vara relevanta. Trots det vill vi understryka att de härledda resultaten har praktisk tillämpning i verkliga situationer.

Contents

1 Introduction

In mathematical biology, directed acyclic graphs (DAGs) are used in order to model relationships between genes or species that traditional rooted trees can not fully capture, such as in the case of reticulate processes or horizontal gene transfers [3]. The leaves in these DAGs (i.e. the subset of vertices with no descendants) model the extant taxa (genes) while the remaining vertices represent their ancestors [1]. A fundamental concept that naturally appears when studying these DAGs is the least common ancestor, and it plays a vital role in understanding hierarchical relationships [4]. If we wish to distinguish genes from different species, we may provide leaves belonging to a distinct species with a unique colour. From these coloured evolutionary DAGs, we can then derive their corresponding "Best match graphs" which provide information about the evolutionary relationships between genes of different species. Edges in these graphs correspond to best matches between pairs of genes from different species, and they are determined from the underlying evoloutionary DAG. More precisely, given a DAG G, $y \in L(G)$ is sayd to be a best match of $x \in L(G)$ if and only if y and x have different colours, and there exists no other leaf $y' \in L(G)$ (with same colour as y) that has a "lower" least common ancestor with x than y [6]. The definition of best matches thus requires least common ancestors to be unique between pairs of leaves of different colours in the underlying evolutionary DAG, something which may not hold in general.

2 Assumptions

In this paper we restrict our attention to finite DAGs with single roots, more commonly refered to as finite networks [1]. Thus if nothing else is mentioned, whenever a DAG G is introduced, it is assumed to be a finite network.

3 Preliminaries

Before diving into the main content of this paper, we encourage the reader to read this "Preliminaries" section. It collects the definitions of all key mathematical objects and concepts used throughout the paper.

Definition 3.0.1. ***Tree***

Let $T = (V, E)$ be an undirected graph with vertex set V and edge set E. We say that T is a tree if and only if it is connected and acyclic.

Definition 3.0.2. ***Directed rooted tree***

If we instead change T to be a directed graph, we call T a directed rooted tree, if there exists a single root $r \in V$ (a vertex satisfying $indeg_T(r) = 0$) such that for every $v \in V \setminus \{r\}$, there is exactly one path from r to v. This definition implies that there are no upward arcs (directed edges towards the root) and hence no cycles in T.

Definition 3.0.3. ***DAG***

Let $G = (V, E)$ be a directed graph. We say that G is a DAG if it contains no cycles.

Observation 3.0.1. *As in the case of a directed rooted tree, a DAG with a single root $r \in V$ has the special property that every vertex $v \in V \setminus \{r\}$ is reachable from r [1]. In particular then, for such a DAG, the single root always serves as a common ancestor for any pair of vertices $u, v \in V \setminus \{r\}$.*

From here on, whenver a DAG G is introduced, it is assumed to be finite and to have a single root unless otherwise specified.

Definition 3.0.4. ***Topological sort***

We say that $v_1, ..., v_n$ ($n = |V|$) constitutes a topological sort of the vertex set V, in a directed graph G, if and only if for every edge $(v_i, v_j) \in E$ it holds that $i < j$. In other words, v_i must come before v_j in the above mentioned list [9]. In lemma 4.1.3, (under subsection 4.1.1) we prove an equivalence between a topological sort and a DAG.

Definition 3.0.5. ***Descendant and Ancestor***

Let $G = (V, E)$ be a DAG. We say that $u \in V$ is a descendant of $v \in V$ if and only if there is a directed path from v to u. If this is the case, we write $u \preceq_G v$. We say that u is an ancestor of v if and only if there is a directed path from u to v. If this is the case we write $v \preceq_G u$

Observation 3.0.2. *Note that two vertices in a DAG may be incomparable, in the sense that neither is an ancestor nor a descendant of the other. If that is the case, we write* $u||v$.

Definition 3.0.6. ***Leaf***

Let $G = (V, E)$ *be a DAG. We say that* $l \in V$ *is a leaf if and only if* $outdeg_G(l) = 0$. *The set of leaves of* G *is denoted* $L(G)$.

Observation 3.0.3. *One may show that since* G *is finite it follows that* $|L(G)| \geq 1$ *[3].*

Definition 3.0.7. ***Leaf-coloured DAG***

Let $G = (V, E)$ *be a DAG. We say that* G *is a leaf-coloured DAG, if every* $l \in L(G)$ *has been assigned a colour* $\sigma(l)$ *where* $\sigma : L(G) \to \mathbb{N}$. *Whenever we want to remark that* G *is a leaf-coloured DAG, we will write* (G, σ).

Definition 3.0.8. ***Least common ancestor***

Let $G = (V, E)$ *be a DAG and consider* $A \subseteq L(G)$. *A least common ancestor of* A *is a vertex* $v \in G$ *that is an ancestor of all leaves in* A *and that has no descendant satisfying the same property. The set* $LCA_G(A)$ *is the set of all such vertices. Throughout the remainder of this paper we will only concern ourselves with particular subsets* A *of leaves. Namely those for which* $|A| = 2$. *Supposing* $A = \{u, v\}$ *for a distinct pair* $u, v \in L(G)$, *we write* $LCA_G(u, v)$ *instead of the more general* $LCA_G(A)$ *for denoting the set of least common ancestors of* A. *[3]*

Definition 3.0.9. ***"2-LCA DAG"***

Let $G = (V, E)$ *be a DAG. We say that* G *is a "2-LCA DAG", if and only if for every pair of distinct leaves* $x, y \in L(G)$ *it holds that* $|LCA_G(x, y)| = 1$.

If G *happens to be leaf-coloured according to some map* $\sigma : L(G) \to \mathbb{N}$, *then we say that* (G, σ) *is a "2-LCA DAG", if and only if for every pair of distinct leaves* $x, y \in L(G) : \sigma(x) \neq \sigma(y)$ *it holds that* $|LCA_G(x, y)| = 1$.

Observation 3.0.4. *Since* $|LCA_G(x, y)| = 1$ *for these types of DAGs, we may abuse notation slightly and write* $LCA_G(x, y)$ *to represent the unique vertex that is a least common ancestor of* x *and* y *in* G

Definition 3.0.10. ***Cluster***

Let $G = (V, E)$ *be DAG and take* $v \in V$. *The set*

$C_G(v) = \{x \in L(G) | x \preceq_G v\}$

is called a cluster of G. *More precisely,* $C_G(v)$ *is the set containing all leaves reachable from* v.

Definition 3.0.11. ***Clustering system***

Let $G = (V, E)$ *be a DAG. The set*

$\mathscr{C} = \{C_G(v) | v \in V\}$

is called the clustering system of G. It is the collection of all unique clusters in G

Observation 3.0.5. *A clustering system is usually defined as a subset of the powerset 2^X, where X is some finite set that is grounded (i.e. $\{x\} \in \mathscr{C}$, $\emptyset \notin \mathscr{C}$) and that contains X [3]. What can be shown is that the set $\mathscr{C}$, introduced above, satisfies both properties (note in particular that enforcing a single root implies that $X \in \mathscr{C}$). Along with the fact that it is the only clustering system we will consider throughout this paper, we will simply call it the clustering system of G.*

Definition 3.0.12. ***Inclusion minimal cluster***

Let $G = (V, E)$ be a DAG. An inclusion minimal cluster containing a subset of vertices $U \subseteq L(G)$, is a cluster $C \in \mathscr{C}$ such that

(1) $U \subseteq C$

(2) $\nexists\, C' \in \mathscr{C} : C' \subset C$ *and* $U \subseteq C'$

Definition 3.0.13. ***Partial order***

We say that the relation R on a set S is a partial order on S, if for all $x, y, z \in S$, the following three properties hold:

(1) *Reflexive: xRx*

(2) *Antisymmetric: if xRy and yRx then $x = y$*

(3) *Transitive: if xRy and yRz then xRz [8]*

Definition 3.0.14. ***Partially ordered set***

We call (S, R) a partially ordered set if R is a partial order on S

Definition 3.0.15. ***Hasse diagram of (S,R)***

The Hasse diagram (denoted H) of the partially ordered set (S, R) is a graphical representation of (S, R), where the vertices are the elements of S, and a directed edge (u, v) (from u to v) is present whenever vRu and there is no other element $z \in S \setminus \{u, v\}$ such that vRz and zRu. [7]

Definition 3.0.16. ***Hasse diagram of*** $(\mathscr{C}, \subseteq)$

Let $G = (V, E)$ be a DAG and consider the clustering system $\mathscr{C}$ of G and the subset relation $\subseteq$. The fact that $(\mathscr{C}, \subseteq)$ is a partially ordered set, follows from basic set theory properties. In particular, the subset relation is reflexive, antisymmetric and transitive. Thus we can consider the Hasse diagram H of $(\mathscr{C}, \subseteq)$. Replacing R with the subset relation in definition 3.0.13 *gives us the following:*

For $C_1, C_2 \in \mathscr{C}$, an edge (C_1, C_2) is present in H

$\Leftrightarrow$

C_2 is a subset of C_1 and there exists no $C_3 \in \mathscr{C} \setminus \{C_1, C_2\}$ such that $C_2 \subsetneq C_3$ and $C_3 \subseteq C_1$.

Definition 3.0.17. ***Best match***

Let $G = (V, E)$ be a DAG and suppose (G, σ) (for some colour map $\sigma : L(G) \to \mathbb{N}$) is a "2-LCA DAG". We say that $y \in L(G)$ is a best match of $x \in L(G)$ in G if and only if the following two properties hold:

(1) $\sigma(x) \neq \sigma(y)$

(2) *For any $z \in L(G)$ with $\sigma(z) = \sigma(y) \Rightarrow LCA_G(x, y) \preceq_G LCA_G(x, z)$ or $LCA_G(x, y) || LCA_G(x, z)$*

One may informally interpret this as stating that amongst all leaves having the same colour as y (which is different from the colour of x), $LCA_G(x, y)$ is "closest" to x. [6]

Definition 3.0.18. ***Best match graph (BMG)***

Let $G = (V, E)$ be a DAG. The BMG $(G', \sigma) = ((V', E'), \sigma)$ of (G, σ) is the directed graph obtained from (G, σ) by taking $V' = L(G)$ and letting $e = (u, v) \in E'$ (edge from u to v) if and only if v is a best match of u in G. [6]

Observation 3.0.6. *From the definition of best matches, we may conclude that a BMG will not have any arcs between pairs of vertices with the same colour. Letting $R(\sigma)$ be the range of the function $\sigma : L(G) \to \mathbb{N}$, one may also show that for every $u \in V'$ and for every colour $s \in S = R(\sigma) \setminus \{\sigma(u)\}$, there is atleast one edge $(u, v) \in E'$ with $\sigma(v) = s$, but we leave the verification to the reader. The definitions of BMGs and best matches also allow for bi-directional edges, since we may have that two vertices $u, v \in V'$ are best matches of each other.*

4 Main

4.1 Generating a random DAG

Before any discussion related to the recognition and generation of random "2-LCA DAGs" or the generation of BMGs could take place, we needed methods for generating random finite DAGs with single roots. We arrived at two approaches. These approaches were then compared in terms of their space and time complexity. In section 4.1.1, we discuss the implementation of approach 1, and in section 4.1.2 we discuss the implementation of approach 2.

4.1.1 Approach 1: Topological sort + root-to-root connections

The first method involved initializing a set of vertices V (with a pre-defined, finite size) and an empty edge set E. The vertices in V were listed in some arbitrary order $v_1, ..., v_n$ where $n = |V|$. E was then expanded by considering random additions of directed edges of the form (v_i, v_j) where $1 \leq i < j \leq n$. That is to say, the directed edge (v_i, v_j) was added to E with some pre-defined probability $0 < p < 1$. As we will see in lemma 4.1.3, this algorithm produces a random DAG (with potentially multiple roots). To enforce a single root, we first considered any single ordering of the roots $r_1, ..., r_m \in V$ where $1 \leq m \leq n$ and preceded by adding the edges (r_1, r_j) where $2 \leq j \leq m$. What follows is a formal justification of the algorithm. The main result is captured in theorem 4.1.1, but first some useful lemmas.

Lemma 4.1.1. *Let $G = (V, E)$ be a DAG (not necessarily having a single root). Then there exists at least one vertex $v \in V$ with $indeg_G(v) = 0$. I.e. G contains at least one root vertex.*

Proof. Assume by contradiction that G has no vertex with in-degree 0. Pick any single vertex $v_0 \in V$ and construct a walk in the transposed graph G^T (same graph as G but edges are reversed) as follows:

> For $i \in \{0, 1, 2, ...\}$ go from v_i to v_{i+1} where v_{i+1} is any single vertex in the out-neighborhood of v_i
>
> Consider the infinite walk $v_0, v_1, v_2, v_3, ..., v_n, ...$

The constructed walk is well-defined since $indeg_G(v) \geq 1$ for all $v \in V$ and hence $outdeg_{G^T}(v) \geq 1$ for all $v \in V$. Furthermore, the walk must have repetition of vertices, since G and hence V is finite. Consider the first pair of indices i, j such that $i < j$

and $v_i = v_j = v$ for some $v \in V$. The walk $v_i, v_{i+1}, ..., v_j$ describes a cycle in G^T and hence $v_j, ..., v_{i+1}, v_i$ describes a cycle in G. But G is a DAG. It follows therefore that our assumption was false and that there must exist at least one vertex in G having in-degree 0.

□

Lemma 4.1.2. *Let $G = (V, E)$ be a DAG (not necessarily having a single root) and suppose $v \in V$ is such that $indeg_G(v) = 0$. Then the directed graph G', obtained from G by removing v and its outgoing edges, is also a DAG.*

Proof. Suppose by contradiction that G' is not a DAG. Thus there exists a cycle in G'. But all the edges in G' are also edges in G. Therefore there exists a cycle in G, contradicting the assumption that G is a DAG. Thus we conclude that our hypothesis was false and that G' is a DAG.

□

Lemma 4.1.3. *Let $G = (V, E)$ ($|V| = n$ for some $n \in \mathbb{N}$) be a finite directed graph. The vertex set V of G can be topologically sorted if and only if G is a DAG (not necessarily having a single root).*

Proof. ($\Rightarrow$) Suppose that the vertex set V of G can be topologically sorted, and let $v_1, ..., v_n$ be one such topological sort. Suppose also by contradiction that there exists a cycle in G. This cycle must start in one of the vertices v_j ($1 \leq j \leq n$) listed above and end in v_j. Furthermore the cycle must contain at least one intermediate vertex, since otherwise the self-edge (v_j, v_j) would be present in G and thus the mentioned list would not be a topological sort. Now, all intermediate vertices in the cycle are of the form v_k for some $j < k \leq n$ since $v_1, ..., v_n$ constitutes a topological sort of V. Consider the last intermediate vertex in the cycle. The cycle is achieved by moving from this vertex back to v_j. But this would mean that there existed an edge (v_i, v_j) for some $i > j$, contradicting the assumption that $v_1, ..., v_n$ constitutes a topological sort of V. Thus it follows that G must be acyclic.

($\Leftarrow$) Suppose G is a DAG. Thus G contains no cycles. Let $R \subseteq V$ denote the set of roots in G (i.e. those vertices in V for which $indeg_G(v) = 0$) and let $m = |R|$. From lemma 4.1.1 we have that $m \geq 1$. Consider now Kahn's algorithm, described in detail below [5].

1. Initialize a queue $Q = R$ and an empty List T.
2. While Q is non-empty, do:
    - a) Pick any $u \in Q$ and for each out-going edge $(u, v) \in E$ do:
        - i. Delete (u, v) from E.
        - ii. If the in-degree of v is now 0, add v to Q.

    b) Add u to the end of T and remove it from V.

3. Return T.

We claim that the algorithm terminates and that it returns a topological sort T of V in G. The former claim is a matter of establishing that Q must eventually be empty, which follows from the observations that a vertex can appear at most once in Q and that V is finite (note in particular that the first observation leads to the conclusion that the total number of dequeue operations is bounded above by |V|). The first observation needs to be formally justified and we argue as follows. Initially $Q = R$ and when a vertex is dequeued, it is removed from V and can thus not reappear in Q in any later stage. It follows that the algorithm indeed terminates. Now, to establish that the returned list T is a topological sort of V in G, we turn to a justification of the following invariant.

"Let V_i' be the set of removed vertices after iteration $i \geq 0$ of the while loop. Then the listed vertices in T (after iteration i) constitute a topological sort of V_i' in the induced subgraph $G[V_i']$."

First note that the set of removed vertices at a particular time point are precisely those vertices present in T at that time. For a vertex is present in T if and only if it has been removed from V. Thus T contains precisely the vertices in V_i' after iteration i. A formal proof of the invariant is now achieved by means of induction over iteration $i \geq 0$. If $i = 0$, then T is an empty list and obviously T is then a topological sort of $V_0' = \emptyset$ in $G[\emptyset]$. Now suppose the invariant holds after iteration $i = k \geq 0$ has been completed. During iteration $k+1$, a new vertex $u \in V$ with in-degree 0 is picked. In G (the starting graph), u can not be a predecessor of any of the listed vertices currently in T. For if we assume the contrary, then there is some $v \in T$ for which there exists a $u \sim v$ path in G. For any such path, let (x, y) be the first edge whose head (i.e. y, potentially equal to v) lies in T (such an edge must exist since $u \notin T$ but $v \in T$). Thus $y \in T$ but $x \notin T$. But if $x \notin T$, then the in-degree of y was at least 1 when it was added to T, since the edge (x, y) have not yet been removed. But the algorithm only allows for vertices with in-degree 0 to be added to T. Thus we conclude that in G, u can not be a predecessor of any vertices currently in T. What follows is that extending T by placing u at the end, yields a topological sort of V_{k+1}' in the induced subgraph $G[V_{k+1}']$.

The invariant, in itself, however, is not sufficient for a complete proof. We need also to verify that T contains all the vertices in V, after the last iteration of the while loop is completed. We argue as follows. Each update of G (i.e. a removal of a vertex with indegree 0 and removal of its out-going edges) yields another DAG and each such DAG has at least one vertex with in-degree 0. The former claim follows from repeated use of lemma 4.1.2, whereas the latter follows directly from lemma 4.1.1. Now, notice that Q contains precisely those vertices having indegree 0 in the latest reduced DAG, and by previous conclusions, it follows that a new vertex (i.e. a vertex not already in T) is picked from Q, so long as there are $0 < n \leq |V|$ vertices remaining. Therfore, the number of iterations is precisely $|V|$ and thus T contains all the vertices in V, after the last iteration is completed. This completes the proof.

□

Lemma 4.1.4. *Let $G = (V, E)$ be a DAG (not necessarily having a single root). Then, for any subset $U \subseteq V$, the induced subgraph $G[U]$ is a DAG.*

Proof. Consider a topological sort of V in G (exists because of lemma 4.1.3). Let $U = \{u_1, ..., u_n\}$ for some $n \leq |V|$, and suppose W.L.G that the vertices in U occur precisely in the order $u_1, ..., u_n$ in that particular topological sort. Thus there are no backward edges of the form (u_j, u_i) for any $1 \leq i < j \leq n$ in G and hence no such edges in $G[U]$. It follows that the list $u_1, ..., u_n$ is a topological sort of U in $G[U]$ and hence by lemma 4.1.3, $G[U]$ is a DAG.

□

Lemma 4.1.5. *Let $G = (V, E)$ be a DAG with roots $r_1, ..., r_m$ and remaining vertices $v_1, ..., v_n$, such that $|V| = m + n$. Let V' be the set of remaining vertices and suppose W.L.G that that the list $v_1, ..., v_n$ constitute a topological sort of V' in $G[V']$ (By lemma* 4.1.4 *such a topological sort exists since $G[V']$ is a DAG). Then $r_1, ..., r_m, v_1, ..., v_n$ is a topological sort of V in G.*

Proof. Since $v_1, ..., v_n$ is a topological sort of V' in $G[V']$, it follows that there are no edges of the form (v_j, v_i) for any $j > i$ in G. Thus the only types of edges that could cause the list $r_1, ..., r_m, v_1, ..., v_n$ to fail being a topological sort of V in G, are of the form (r_i, r_j) and (v_k, r_j) for $i \neq j$. But since $r_1, ..., r_m$ are the roots of G, for each $1 \leq i \leq m$, $indeg_G(r_i) = 0$. Thus no such edges exist in G and it follows that the list $r_1, ..., r_m, v_1, ..., v_n$ is a topological sort of V in G. This completes the proof. □

Theorem 4.1.1. *The method described in the opening paragraph of this section, produces a random finite DAG with a single root.*

Proof. First note that the first part of the method, which inolves adding random edges of the form $(v_i, v_j) : i < j$ to the list $v_1, ..., v_n$, produces a topological sort of V in G, and hence by lemma 4.1.3, G is a DAG with potentially more than one root. Consider now the set $W = \{w_1, ..., w_k\} = V \setminus R$, where $R = \{r_1, ..., r_m\}$ is the set of roots in G. Suppose $W.L.G$ that the list of vertices $w_1, ..., w_k$ constitute a topological sort of W in $G[W]$. From lemma 4.1.5, it follows then that $r_1, ..., r_m, w_1, ..., w_k$ is a toplogical sort of V, in the DAG obtained after applying the first step of the method. The second part of the method involves adding random edges of the form (r_1, r_j) for $2 \leq j \leq m$, and it is not hard to see that the list $r_1, ..., r_m, w_1, ..., w_k$ remains a topological sort of V, in the new directed graph. Thus this new directed graph is also a DAG (lemma 4.1.3). Moreover it is finite, since V and E are finite and only a finite number of edges were added to E. Lastly, it has a single root, namely r_1, and this completes the proof. □

4.1.2 Approach 2: Extension from Tree

The second method involved randomly inserting edges into a finite directed rooted tree, which was automatically generated by using the "Networkx" library in python. The idea was intuitive. If we start with a random directed rooted tree $T = (V, E)$ with vetices $v_1, ..., v_n$, listed in an arbitrary topological order, then we may add random edges of the form (v_i, v_j) for $i < j$ to produce a random DAG with a single root. The claim is justified in theorem 4.1.2.

Theorem 4.1.2. *Let $T = (V, E)$ be a directed rooted tree and $v_1, ..., v_n$ an arbitrary topological sort of V in T (which exists because T is a DAG). Then any addition of new, random edges of the form (v_i, v_j) for $1 \leq i < j \leq n$, extends T to a random finite DAG with a single root.*

Proof. First note that v_1 must be the single root of T. For if we suppose by contradiction that the root is not equal to v_1, then $indeg_G(v_1) \geq 1$. But then necessarily there must exist an edge (v_j, v_1) for which $j > 1$, contradicting the fact that $v_1, ..., v_n$ is a topological sort of V in G. Now, if all the additional random edges added to T are of the form (v_i, v_j) for $1 \leq i < j \leq n$, then trivially the provided list remains a topological sort of V in the new directed graph, after the transformation is complete. Thus it is a random DAG (lemma 4.1.3). Moreover, v_1 remains the unique root, since the transformation preserves the indegree of v_1, and either preservers or increases the indegree of the remaining vertices. Lastly, the new DAG is finite since it contains precisely the same number of vertices as T (a finite amount), and only a finite number of new edges could have been added to the finite set E. This completes the proof. $\square$

4.1.3 Pseudocode and a comparison in space and time complexity

To analyse the space and time complexity of the algorithms used for approach 1 and approach 2, we provide first, pseudocode for both algorithms.

Algorithm 1 Approach 1

```
1:  The input of the function below is the number of vertices $n$ and the proability of
    adding an edge $p$. The output is a random DAG $G$ on $n$ vertices
2:
3:  function DAGAPP1($n, p$):
4:      Initialize an empty directed graph $G = (V, E)$;
5:      Set $V = \{0, ..., n - 1\}$;
6:      for $i \in \{0, ..., n - 1\}$ do
7:          for $j \in \{i + 1, ..., n - 1\}$ do
8:              generate an integer $p_0$ uniformally on $[0, 1]$;
9:              if $p_0 \leq p$ then
10:                 Add the edge $(i, j)$ to $E$;
11:             end if
12:         end for
13:     end for
14:     Set $v_1 = 0$;
15:     Create a list R of all roots except $v_1$;
16:     for $r \in R$ do
17:         Add the edge $(v_1, r)$ to $E$;
18:     end for
19:     Return $G$
20: end function
```

Algorithm 2 Approach 2

```
1: Construct an undirected tree $T$ on $n$ vertices, using the networkx library;
2:
3: *The input of the function below is an undirected tree $T$ on $n$ vertices. The output
   is a directed rooted tree $T'$, the directed version of $T$.*
4:
5: function DRT(T):
6:     Set $r = 0$; (*choosing vertex labeled "0" to be the root*)
7:     Initialize the directed tree $T' = (V(T), E = \emptyset)$;
8:     Initialize an empty list $\beta$;

9:     *Start a BFS traversal from $r$, and whenever a new vertex $u$ is visited, add all
10:    edges of the form $(u, v)$ to the end of the list $\beta$, if and only if $\{u, v\} \in E(T)$
11:    and $v$ has not already been visited (note that the networkx library has a function
12:    which computes $\beta$ directly. Here we are basically just providing information
13:    about how the function works).*

14:    for $(u, v) \in \beta$ do
15:        add $(u, v)$ to $E$;
16:    end for
17:    Return $T'$
18: end function
19:
20: *The input of the function below is a directed rooted tree $T$ on $n$ vertices, and the
    probability $p$ of adding an edge between two vertices. The output is a random DAG
    on $n$ vertices. The algorithm is in-place, in the sense that we do not make a new
    directed graph object. We simply add edges to $E$ and return $T$ in the end.*
21:
22: function DAGAPP2(T, p)
23:     Generate a topological sort $\sigma_V$ of the vertex set $V$ in $T$;
24:     *(Noting here that $\sigma_V = \{v_1, ..., v_n\}$ where $n = |V|$)*
25:     for $i \in \{1, ..., n\}$ do
26:         for $j \in \{i + 1, ..., n\}$ do
27:             generate an integer $p_0$ uniformally on [0,1];
28:             if $p_0 \leq p$ then
29:                 Add an edge $(v_i, v_j)$ to $E$;
30:             end if
31:         end for
32:     end for
33:     Return $T$
34: end function
```

The correctness of both algorithms have for the most part already been established in subsections 4.1.1 and 4.1.2. But note that we have not discussed the correctness of the "DRT" function, used for approach 2. The primary reason being that the transformation of turning an undirected tree into a directed rooted tree, was not originally intended to be implemented manually. Rather, we thought that the networkx library would support a direct method for it. Thus before analyzing the space and time complexity of both algorithms, we provide first a formal justification of the correctness of the "DRT" function, used for approach 2.

Theorem 4.1.3. *The "DRT" function, used for approach* 2*, converts an undirected tree T on n vertices to a directed rooted tree T' on n vertices.*

Proof. The list β was generated using an built-in method (nx.bfsedges) from the networkx library. Therefore, we can assume that β contains the edges of T, ordered based on when they were traversed during a BFS traversal , starting from the chosen root r. Since $T' = (V, E)$ is generated by taking $V = V(T)$ and $E = \{(u, v) : (u, v) \in \beta\}$, it follows that T' is at least a directed graph on n vertices. To prove that T' is a directed rooted tree, we must show that

- The only root of T' is r.
- For each $v \in V \setminus \{r\}$, there is a single $r \sim v$ path in T'.

Now, r is a root in T', because starting a BFS traversal from r ensures that we will never add an edge of the form (v, r) to E for some $v \in V$. This follows from the fact that r will have been marked as visited before visiting any other $v \in V$. Moreover, for each $v \in V \setminus \{r\}$, there is a unique vertex $u \neq v \in V$ (possibly equal to r) closer to r in the unique $r \sim v$ path in T, such that $\{u, v\} \in E(T)$. Thus u is visited before v and, upon visiting $u \in V$, the edge (u, v) is added to E. Therefore $indeg_{T'}(v) \geq 1$, verifying that r is indeed a unique root. The last claim follows immediately from the fact that the unique undirected $r \sim v$ path in T, appears as a directed $r \sim v$ path in T', completing the proof.

□

We now turn to a formal justification of the space and time complexity of both algorithms.

Theorem 4.1.4. *Consider the algorithms used for approach* 1 *and approach* 2 *above. The worst case space and time complexity for both algorithms is $O(n^2)$.*

Proof. We analyze first the space complexity of algorithm 1. Creating a directed graph object which stores all vertices and edges requires $O(n^2)$ space in the worst case (recalling that the number of edges in a DAG is bounded above by $\frac{n(n-1)}{2}$). Storing the uniformally generated real number p_0 requires $O(1)$ space. Storing v_1 requires $O(1)$ space. Storing R requires $O(n)$ space in the worst case, since the number of roots is bounded above by n. What follows is that the worst case space complexity of algorithm 1 is $O(n^2)$. As for the time complexity of algorithm 1, initializing G as an empty directed graph is done in $O(1)$ time. Setting $V = \{1, ..., n\}$ is done in $O(n)$

time. Iterating over all pairs of vertices is done in $O(n^2)$ time. Generating p_0 is done in $O(1)$ time. Adding the single edge (i, j) is done in $O(1)$ time. Setting $v_1 = 0$ is done in $O(1)$ time. Creating a list R that stores all roots except v_1 is done in (the worst case) $O(n)$ time. Iterating over all $r \in R$ is done in (the worst case) $O(n)$ time. Adding the single edge (v_1, r) to E is done in $O(1)$ time. What follows is that the worst case time complexity of algorithm 1 is $O(n^2)$.

We now justify that the space and time complexity of algorithm 2 is also $O(n^2)$, starting with the space complexity. First note that total amount of space required by algorithm 2 is the maximum of the space complexities of the two functions. We start by analyzing the "DRT" function. Storing $r = 0$ requires $O(1)$ space. Storing the directed rooted tree T' requires $O(n)$ space, since a directed rooted tree on n vertices has exactly $n - 1$ edges. Generating β requires $O(n)$ space, $O(n)$ space for the BFS traversal and $O(n)$ space for storing all the edges in β. Thus the worst case space complexity of the DRT function is $O(n)$. As for the DAG_{APP2} function, generating and storing the topological sort σ_V requires $O(n)$ space. Storing the temporary variables v_i, v_j and p_0 requires $O(1)$ space. Lastly, note that since T is expanded to a DAG on a single root, storing T now requires $O(n^2)$ space. Thus the total space complexity for algorithm 2 is $O(n^2)$. The total running time of algorithm 2 is the sum of the running time of the "DRT" and the "DAG_{APP2}" function. We start by analyzing the running time of the "DRT" function. Storing $r = 0$ is done in $O(1)$ time. Creating the directed graph object $T' = (V(T), E = \emptyset)$ is done in $O(n)$ time. Generating β is done in $O(n)$ time (since BFS runs in $O(n)$ time whenever the input is a tree). Iterating over all the edges in β is done in $O(n)$ time. Adding the single edge (u, v) to E is done in $O(1)$ time. What follows is that the time complexity of the "DRT" function is $O(n)$. As for the "DAG_{APP2}" function, we note first that generating a topological sort σ_V is done in $O(n)$ time. Iterating over all pairs of vertices is done in $O(n^2)$ time . Generating an integer uniformally on $[0, 1]$ is done in $O(1)$ time. Lastly, adding the single edge (v_i, v_j) to E is done in $O(1)$ time. What follows is that the worst case time complexity of algorithm 2 is $O(n^2)$, completing the proof.

□

The conclusion that may be drawn from theorem 4.1.4 is that approach 1 or approach 2 can not be prefered over the other, on the basis of space and time complexity.

4.2 Recognizing a "2-LCA DAG"

The BMG of a leaf-coloured DAG (G, σ) is only well-defined, whenever, for any two distinct leaves $x, y \in L(G)$ with $\sigma(x) \neq \sigma(y)$ it holds that $|LCA_G(x, y)| = 1$. But this may of course not hold in general. Consider for instance the following leaf-coloured DAG:

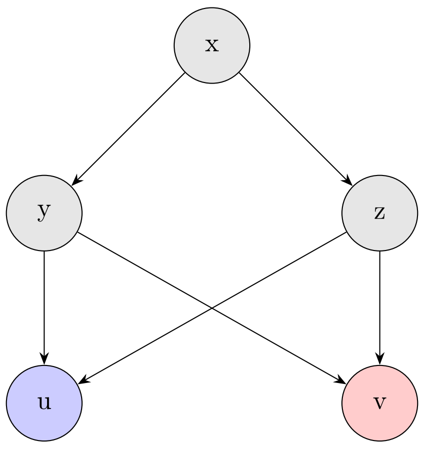

Figure 4.1: A leaf-coloured DAG (G, σ) with uniqueness criteria not satisfied

The two leaves are u and v and they have different colours (blue and red respectively). However, $LCA_G(u, v) = \{y, z\}$ and hence $|LCA_G(u, v)| = 2 > 1$. This sort of observation motivated use to find a polynomial time algorithm that takes as input any coloured DAG (G, σ) and returns a modified version of (G, σ) , namely (G', σ), where the uniqueness criteria is fulfilled. That is to say, our attention was directed towards finding a polynomial time algorithm for generating a special kind of "2-LCA DAG" from an arbitrary coloured DAG (G, σ). But before finding such an algorithm, we first justified that recognizing a "2-LCA DAG" could be done in polynomial time. The vericiation was done in 4 steps

1. Create a function that computes the set of all ancestors of a vertex $v \in V$
2. Create a function that uses the previous function, in order to compute the common ancestor graph $G[A_u \cap A_v]$ of two distinct vertices u and v, where A_u and A_v denote the set of all ancestors of u and v respectively.
3. Create a function that uses the previous function, in order to compute the set $LCA_G(u, v)$ for any two distinct vertices $u, v \in V$
4. Create a function that uses the previous function, in order to do the final check. I.e. iterating over all pairs of distinct leaves $u, v \in L(G)$ with different colours and checking whether $|LCA_G(u, v)| = 1$ holds or not.

4.2.1 Pseudocode for verification algorithm

Having described the basic idea behind the recognition algorithm, we provide below, pseudocode for it.

Algorithm 3 Verification algorithm

```
*The input of the function below is a vertex $v \in V$, the underlying DAG $G$, an ancestor set $A$ and a set of visited vertices $V'$. Note that $A$ and $V'$ are initialised as "None" objects. The output of the function is the set of all ancestors $A$ of the vertex $v$ in $G$.*

function AllAncestorsOf($v, G, A, V'$):
    if A = None then
        Set $A = \emptyset$;
    end if
    if V' = None then
        Set $V' = \emptyset$;
    end if
    for $u : (u, v) \in E$ do
        if $u \notin V'$ then
            $A \cup \{u\}$;
            $V' \cup \{u\}$;
            AllAncestorsOf(u, G, A, V');
        end if
    end for
    return A
end function

*The input of the function below are two distinct vertices $u, v \in V$ and the underlying DAG $G$. The output of the function is the induced subgraph $G[A_u \cap A_v]$, where $A_u$ and $A_v$ denote the set of all ancestors of $u$ and $v$ respectively.*

function CancestorGraph(u,v, G):
    $A_u \leftarrow AllAncestorsOf(u, G)$;
    $A_v \leftarrow AllAncestorsOf(v, G)$;
    $S \leftarrow \emptyset$;
    for $x \in A_u$ do
        if $x \in A_v$ then
            $S \cup \{x\}$;
        end if
    end for

    *(Note that $S = A_u \cap A_v$ after the above for loop is complete.)*

    Initialize an empty DAG $G'$;
    Set $V(G') = S$;

    for $x \in S$ do
        for $y \in S$ do
            if there exists an edge $(x, y) \in E$ then
                $E(G') \cup \{(x, y)\}$;
            end if
        end for
    end for
    return $G'$
end function

*The input of the function below are two distinct leaves $u, v \in L(G)$ and the underlying DAG $G$. The output of the function is the set $L = LCA_G(u, v)$.*
```

```
function LCA(u,v, G):
    $L \leftarrow \emptyset$;
    $G[A_u \cap A_v] \leftarrow$ CAncestorGraph(u,v,G);
    for $x \in A_u \cap A_v$ do
        if $outdeg_{G[A_u \cap A_v]}(x) = 0$ then
            $L \cup \{x\}$;
        end if
    end for
    return $L$
end function

The input of the function below is a leaf-coloured DAG $(G, \sigma)$. The output of the
function is True if $(G, \sigma)$ is a "2-LCA DAG" and False otherwise.

function IsValidDAG(G, $\sigma$):
    for $u \in L(G)$ do
        for $v \in L(G)$ do
            if $\sigma(u) \neq \sigma(v)$ then
                $LCA_G(u, v) \leftarrow$ LCA(u,v,G);
                if $|LCA_G(u, v)| \neq 1$ then
                    return False
                end if
            end if
        end for
    end for
    return True

    OBS! The above function does some repeated checks. More precisely, for every
    pair of distinct vertices $u, v \in L(G)$, 2 identical checks are done. This can be easily
    fixed by replacing the double for loop with "for u,v in combinations(L(G),2)". Also
    one may modify it for an input DAG that is not leaf-coloured. Then the LCA check
    is done over all pairs of distinct leaves.

end function
```

4.2.2 Correctness of verification algorithm

The correctness of algorithm 3 is mostly a matter of establishing that the set $LCA_G(u, v)$, for two distinct vertices $u, v \in V$, contains precisely those vertices $x \in V$ for which $outdegree_{G[A_u \cap A_v]}(x) = 0$ (see lemma 4.2.3). That is,

$LCA_G(u, v) = \{x \in V : outdeg_{G[A_u \cap A_v]}(x) = 0\}$.

But for a complete proof of correctness, we turn to formal justification of each of the functions.

Lemma 4.2.1. *The "AllAncestorsOf" function, correctly computes all ancestors of a vertex $v \in V$ in the DAG $G = (V, E)$.*

Proof. Let $A(v)$ be the set of all ancestors of v in G, and let us introduce the following parameter:

$L(v) = max_{x \in A(v)}$ (length of longest $x \sim v$ path).

Then for a complete proof, we may induct over $L(v) \geq 0$. Note that since G is finite, $L(v) < \infty$. Now if $L(v) = 0$, then consequently v has no ancestors and in particular no parents. Thus upon the first call of the function, the for loop is never entered and $A(v)$ is returned as an empty set, which is the correct output. Suppose now that the statement holds for all $u \in V$ for which $L(u) \leq k$ for some $k \in N$, and consider the case when $L(v) = k + 1$. Let $P(v)$ be the set of all parents of v, then the logic of the algorithm is to take

$A(v) = P(v) \cup (\cup_{u \in P(v)} A(u))$.

This is correct since each ancestor of v is either a parent of v or an ancestor of a parent of v. Thus to complete the proof, we need only to verify that the set $A(u)$, for each $u \in P(v)$ is correctly computed by the algorithm. But this follows immediately from the observation that $L(u) \leq k$ whenever $u \in P(v)$ (note that if $L(u) \geq k + 1$ then consquently $L(v) \geq k + 2$, a contradiction) and the induction hypothesis. This completes the proof.

□

Lemma 4.2.2. *The "CAncestorGraph" function correctly computes the common ancestor graph $G[A_u \cap A_v]$ of two distinct vertices $u, v \in V$, in the DAG $G = (V, E)$.*

Proof. By definition, the common ancestor graph $G' = G[A_u \cap A_v]$ is such that $V(G') = A_u \cap A_v$ and such that $E(G') = \{(x, y) \in E(G) : x, y \in V(G')\}$. The function generates $V(G')$ and $E(G')$ according to their definitions above. Moreover, they are both correctly computed, since A_u and A_v and hence $V(G') = A_u \cap A_v$ are correctly computed. This completes the proof. □

Lemma 4.2.3. *The "LCA" function correctly computes the set of least common ancestors $LCA_G(u, v)$ of two distinct vertices $u, v \in V$, in the DAG $G = (V, E)$.*

Proof. The underlying logic of the function is that

$LCA_G(u, v) = \{x \in V : outdeg_{G[A_u \cap A_v]}(x) = 0\}$.

We justify the above claim through the following series of equivalences

$x \in LCA_G(u, v) \iff x$ is a common ancestor of u and v and there is no descendant of x in G with the same property $\iff x \in A_u \cap A_v$ and there is no descendant y of

x s.t $y \in A_u \cap A_v \iff x \in A_u \cap A_v$ and there is no child y of x s.t $y \in A_u \cap A_v$ $\iff outdeg_{G[A_u \cap A_v]}(x) = 0$. This completes the proof.

□

Theorem 4.2.1. *The "IsValidDAG" function correctly identifies whether a given DAG G is a "2-LCA DAG" or not.*

Proof. The correctness follows immediately from the correctness of the "LCA" function. □

4.2.3 Space and time complexity analysis of verification algorithm

Our space and time complexity analysis will be similar to that of the space and time complexity analysis in subsection 4.1.3. But because we are now dealing with four functions, we make a seperate analysis for each, for the sake of clarity. In the end, our goal is to deduce the space and time complexity of the "IsValidDAG" function. We let $n = |V|$.

Lemma 4.2.4. *The worst case space complexity of the "AllAncestorsOf" function is $O(n)$ and the worst case time complexity is $O(n^2)$.*

Proof. The total space used, is the space allocated for storing vertices in A and V' as well as the space allocated for the recursive call stack. A and V' contain at most $n-1$ vertices (all except v itself). The space allocated for the recursive call stack is proportional to the maximum recursion depth, which is n in the worst case. What follows is that the total space complexity of the function is $O(n)$ in the worst case.

As for the running time, note that in the worst case, all $n-1$ remaining vertices (all vertices except v) are ancestors of v, each one visited once. When each such ancestor is visited, all its in-going edges are traversed. Since all checks are done in constant time, it follows that the total worst case time complexity is $O(n + |E|)$. Lastly, since $|E| \leq n^2$, we deduce that the total time complexity is $O(n^2)$ in the worst case.

□

Lemma 4.2.5. *The worst case space complexity of the "CAncestorGraph" function is $O(n^2)$ and the worst case time complexity is $O(n^2)$.*

Proof. The total space used, is the space allocated for storing vertices in A_u, A_v, S as well as the space allocated for storing G'. The worst case space complexity for storing A_u and A_v is equal to $O(n)$, which follows from the previous lemma. As for S, notice that S contains at most $n-2$ vertices (since in the worst case S contains all the vertices in G except for u and v), and hence requires $O(n)$ space to store in the worst case. The number of edges in G' is bounded above by $\binom{n-2}{2} \leq n^2$. What follows is that space complexity for storing G' is in the worst case $O(n^2)$. Thus the total worst case space complexity is equal to $O(n^2)$.

The running time of the function is the addition of the amount of time needed to generate A_u, A_v , S and G'. From previous results we know that generating A_u and A_v requires $O(n^2)$ time in the worst case. Intitializing S as an empty set requires $O(1)$ time. To expand S, we iterate over all pairs of elements from A_u and A_v, which is done in $O(n^2)$ time in the worst case, followed by a constant time operation of potentially expanding S. Thus generating S requires $O(n^2)$ time in the worst case. Lastly, generating G' is achieved by first initializing it as an empty directed graph object, which is done in $O(1)$ time. Setting $V(G') = S$ is done in $O(n)$ time in the worst case. To generate the set $E(G')$, we iterate over all pairs of elements from S, perform a constant time lookup and a potential constant time insertion. Thus generating G' also requires $O(n^2)$ time in the worst case. It follows that the worst case time complexity of the function is $O(n^2)$.

□

Lemma 4.2.6. *The worst case space complexity of the "LCA" function is $O(n^2)$ and the worst case time complexity is $O(n^2)$*

Proof. The worst case space complexity of the function is equal to the maximum of the worst case space complexities for storing L and $G[A_u \cap A_v]$. Notice that L contains at most $n-2$ vertices, which occurs in the case where $|A_u \cap A_v| = n-2$ and all the vertices in $G[A_u \cap A_v]$ have out-degree 0. Thus in the worst case, storing L requires $O(n)$ space. Storing $G[A_u \cap A_v]$ requires first calling the "CAncestorGraph" function and then assigning the result of the function to a variable. The former step requires $O(n^2)$ space in the worst case (follows from the previous lemma) and the latter also requires $O(n^2)$ space in the worst case (because the result of the function is a DAG where the number of edges is bounded above by n^2). Thus storing $G[A_u \cap A_v]$ requires $O(n^2)$ space in the worst case. What follows is that the total worst case space complexity of the function is equal to $O(n^2)$.

The worst case time complexity of the function is equal to the sum of the worst case time complexity for generating L and the worst case time complexity for generating $G[A_u \cap A_v]$. From the previous lemma, we know that the latter is equal to $O(n^2)$. The former is equal to $O(n)$, because generating L requires iterating over the elements of $A_u \cap A_v$ and then performing constant time checks and constant time operations. Thus the total worst case time complexity of the function is equal to $O(n^2)$.

□

The worst case space/time complexity of algorithm 3 is equal to the worst case space/time complexity of the "IsValidDag" function. Thus, we close off this subsection by formulating the worst case time/space complexity of this function as a theorem.

Theorem 4.2.2. *The worst case space complexity of the "IsValidDag" function is $O(n^2)$ and the worst case time complexity is $O(n^4)$.*

Proof. The worst case space complexity is equal to the worst case space complexity of storing $LCA_G(u, v)$, for each pair of leaves u, v such that $\sigma(u) \neq \sigma(v)$. To store $LCA_G(u, v)$, we first need to call the "LCA" function which will require $O(n^2)$ space in the worst case (follows from previous lemma). Assigning the result of the function to a variable, will then require an additional $O(n)$ space, because the number of vertices in the LCA-set is bounded above by $n - 2$. Thus it follows that the total worst case space complexity of the function (and hence the algorithm) is equal to $O(n^2)$.

As for the worst case time complexity, notice that we first iterate over all pairs of leaves, which is achieved in $O(n^2)$ time. This is then followed up with a constant time check (i.e. checking if colours match or not) and then potentially computing the LCA-set and the size of the LCA-set (depending on if the the "if" condition was passed or not). Computing the LCA-set is done in $O(n^2)$ time in the worst case (by the previous lemma) and checking its size is done in $O(n)$ time. Overall the worst case time complexity is therefore equal to $O(n^2(n^2 + n)) = O(n^4 + n^3) = O(n^4)$. What follows is that the worst case time complexity of the entire algorithm is $O(n^4)$. □

4.3 Transforming a random DAG into a "2-LCA DAG"

For transforming a random DAG into a "2-LCA DAG", we first considered applying an iterative procedure, which would involve repeatedly using the so-called $O-$ operator. An operator which removes a vertex from a directed graph and connects all its parents to all its children [3]. The basic idea was to iterate over all distinct pairs $\{u, v\} \subseteq L(G)$ and remove all but one vertex from the set $LCA_G(u, v)$, by applying the $O-$ operator. Indeed, this would locally help achieve the desired uniqueness criteria, but because future removals of vertices in other iterations could potentially change the set $LCA_G(u, v)$, we knew that this kind of method would face complications. Thus, we ruled it out.

4.3.1 Extension of Hasse diagram

The method we arrived at, involved deriving a specific extension $\mathscr{C}$' of the clustering system $\mathscr{C}$ (definition 3.0.11) followed by deriving the Hasse diagram of $(\mathscr{C}', \subseteq)$. This Hasse diagram would then correspond to the transformed DAG.

Before we describe the details of the transformation, let us recall the definitions of a cluster and a clustering system. For any DAG G, the set $C_G(v)$ is a cluster in G and it contains all reachable leaves from $v \in V$. The clustering system $\mathscr{C}$ of G, is the set of all such clusters. A useful way to illustrate the clustering system in a DAG, is to write down the set $C_G(v)$ next to each vertex $v \in V$. Consider now the hasse diagram

H of $(\mathscr{C}, \subseteq)$. Some natural questions are whether H is a finite DAG with a single root or not and whether it is a "2-LCA DAG" or not? As we now see, the answer to the first question turns out to be yes.

Lemma 4.3.1. *Let H be the Hasse diagram of the partially ordered set $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is the clustering system the DAG $G = (V, E)$, then H is finite.*

Proof. First observe that

$|\mathscr{C}| \leq 2^{|L(G|} \leq 2^{|V(G)|} < \infty$

where the first inequality follows from the fact that $\mathscr{C}$ is a subset of the power set $P(L(G))$, the second from the fact that $|L(G)| < |V|$ and the third from the fact that V is finite. By means of the number of edges being bounded above by $|\mathscr{C}|^2$ in H, we deduce that the edge set of H is also finite. It follows that H is finite.

$\square$

Lemma 4.3.2. *Let H be the Hasse diagram of the partially ordered set $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is the clustering system of the DAG $G = (V, E)$, then $C \in \mathscr{C}$ is an ancestor of $C' \in \mathscr{C}$ in H if and only $C' \subseteq C$.*

Proof. $(\Rightarrow)$ Suppose that C is an ancestor of C'. Thus there exists a finite path

$C \to C_1 \to ... \to C_n \to C'$: $n \in \mathbb{N}, C_1, ..., C_n \in \mathscr{C}$.

By the definition of H, we may therefore conclude that

$C' \subseteq C_n \subseteq ... \subseteq C_1 \subseteq C$.

Thus $C' \subseteq C$.

$(\Leftarrow)$ Suppose instead that $C' \subseteq C$ and consider the following set

$I(C', C) = \{\tilde{C} \in \mathscr{C} : C' \subset \tilde{C} \subset C\}$.

Expressed in words, $I(C', C)$ contains the set of clusters in $\mathscr{C}$ that are proper subsets of C and proper supersets of C'. From lemma 4.3.1, we know that $\mathscr{C}$ is finite and hence also $I(C', C)$ (note that $I(C', C) \subseteq \mathscr{C}$). We now establish that C is an ancestor of C' through induction over the size of $I(C', C)$. If $|I(C', C)| = 0$, then there is no larger subset of C than C' itself. Thus it follows from the definition of H, that C is a parent of C', and hence also an ancestor of C'. Suppose now that C is an ancestor of C' whenever $|I(C', C)| \leq n$ for some $n \in \mathbb{N}$, and consider the case $|I(C', C)| = n + 1$. Pick a cluster $C'' \in I(C', C)$ and introduce the sets $I(C', C'')$ and $I(C'', C)$ (defined in the same way as for $I(C', C)$). By means of $|I(C', C'')|, |I(C'', C)| \leq n$, we deduce from our induction hypothesis that C'' is an ancestor of C' and that C is ancestor of C''. Since the ancestral relation is transitive, it follows that C is an ancestor of C' and this completes the proof.

$\square$

Theorem 4.3.1. *Let $G = (V, E)$ be a DAG and $v_0 \in V$ its single root. Let H be the Hasse diagram of the partially ordered set $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is the clustering system of G, then H is a finite DAG with a single root. Moreover $L(H) = \{\{v\} : v \in L(G)\}\}$.*

Proof. Lemma 4.3.1 already establishes that H is finite. The fact that H is a DAG can be established through a proof by contradiction. Thus assume that H is cyclic and that it contains a cycle of the form

$C_1 \to C_2 \to ... \to C_n \to C_1$: $n \in N$ and $C_1, ..., C_n \in \mathscr{C}$.

By the definition of H, and the fact all clusters in $\mathscr{C}$ are different, we may therefore conclude that

$C_1 \subset C_n \subset C_{n-1} \subset ... \subset C_1$.

But then $C_1 \subset C_1$ and hence no such cycle can exist in H. Consider now the cluster $C \in \mathscr{C}$ that contains all the elements of $L(G)$. C is well-defined, for one may show that all leaves in G are reachable from the single root v_0. Moreover C can not be a subset of any of the other clusters in $\mathscr{C}$, because each such cluster contains fewer elements from $L(G)$. It follows that C does not have any ancestors and in particular no parents in H. Indeed then C is a root. C is the only root in H, beacuse each cluster $C' \in \mathscr{C} \setminus \{C\}$ is a subset of C and from lemma 4.3.2, a descendant of C, implying that $indeg_H C' \geq 1$. Lastly, the leaf set of H is precisely the set $L(H) = \{\{v\} : v \in L(G)\}\}$, because the singleton clusters (each containing a single leaf in G) are the smallest clusters in $\mathscr{C}$, and hence the only clusters that are not supersets of any other clusters in $\mathscr{C}$.

□

Though the Hasse diagram H of $(\mathscr{C}, \subseteq)$ turns out to be DAG with a single root, it is not true in general that it is a "2-LCA-DAG". Consider for instance, the following input DAG.

The Clustering system $\mathscr{C}$ for the DAG depicted in figure 4.2 is

$\mathscr{C} = \{\{v\}, \{x\}, \{y\}, \{u\}, \{v, x, y\}, \{x, y, u\}, \{v, x, y, u\}\}$

and one can easily show that the Hasse diagram H of $(\mathscr{C}, \subseteq)$, is isomorphic to the DAG depicted in figure 4.2, and hence not a "2-LCA DAG". However, notice that the extension $\mathscr{C}' = \mathscr{C} \cup \{x, y\}$ would turn H into a "2-LCA DAG", an observation that laid out the foundation for the algorithm described in theorem 4.3.2. Before turning to a formal justification of theorem 4.3.2, we first establish equivalence between an inclusion minimal cluster and a least common ancestor.

Lemma 4.3.3. *Let H be the Hasse diagram of the partially ordered set $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is the clustering system of the DAG G. $C \in \mathscr{C}$ is an inclusion minimal cluster containing two distinct $x, y \in L(G)$, if and only if it is a least common ancestor of the clusters $\{x\}, \{y\} \in L(H)$ in H.*

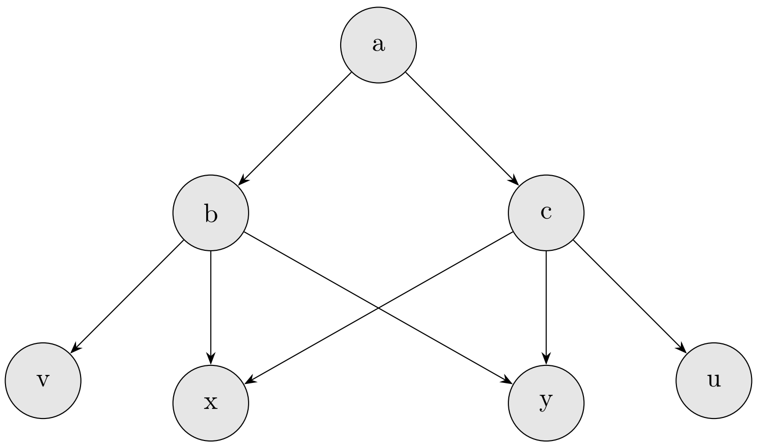

Figure 4.2: DAG for which the Hasse Diagram of $(\mathscr{C}, \subseteq)$ fails to be a "2-LCA DAG".

Proof. The proof is divided into two parts. The first part consists of showing that C is a cluster containing x and y, if and only if it is a common ancestor of $\{x\}$ and $\{y\}$. The second part consists of showing that there is no smaller cluster $C' \subset C$ containing x and y, if and only if there is no descendant of C that is a common ancestor of both $\{x\}$ and $\{y\}$. We have that

C is a cluster containing x and y $\iff$ The clusters $\{x\}$ and $\{y\}$ are subsets of C $\iff$ C is an ancestor of both $\{x\}$ and $\{y\}$ $\iff$ C is a common ancestor of $\{x\}$ and $\{y\}$

, where the first equivalence is a matter of basic set theory, the second a consequence of lemma 4.3.2, and the third a result of applying the definition of a common ancestor. Now,

$\nexists C' \in \mathscr{C}$ such that $C' \subset C$ and $\{x, y\} \subseteq C'$ $\iff$ $\nexists C' \in \mathscr{C}$ such that $C' \prec C$ and $\{x, y\} \subseteq C'$ $\iff$ $\nexists C' \in \mathscr{C}$ such that $C' \prec C$ and C' is a common ancestor of $\{x\}$ and $\{y\}$

where the first equivalence is a consequence of lemma 4.3.2, and the second a matter of applying the result for the first part of the proof. Combining both parts of the proof, now gives us that C is a least common ancestor of the clusters $\{x\}$ and $\{y\}$, and we are done. □

Theorem 4.3.2. *Let H be the Hasse diagram of the partially ordered set $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is the clustering system the DAG G. Consider the following algorithm that extends H to a new Hasse diagram H':*

1. *For all distinct $\{x\}, \{y\} \in L(H)$ do:*

If there is no unique inclusion minimal cluster containing x and y, then add the cluster $\{x, y\}$ to $\mathscr{C}$.

2. *Return the Hasse diagram H' of $(\mathscr{C}', \subseteq)$, where $\mathscr{C}'$ is the new clustering system obtained after the preceding for-loop.*

The algorithm above produces a finite "2-LCA DAG" H' , with the same single root as H and that satisfies $L(H') = L(H)$.

Proof. The first part of the proof consists of showing that H' remains a finite DAG with the same single root and that $L(H') = L(H)$. The fact that H' is finite follows from the fact that at most $\binom{|L(H)|}{2}$ new clusters were added to H. Thus both the vertex set (the extended clustering system) and the edge set of H' remain bounded above by some constant. The proof that H is a DAG in theorem 4.3.1, remains valid for any Hasse diagram and thus H' is a DAG. The element $C \in \mathscr{C}$ that contains all leaves in G, remains in $\mathscr{C}'$, and moreover it remains the only root, because the new clusters that were potentially added to $\mathscr{C}$, are also subsets of C. $L(H) = L(H')$ because of the same reasoning as in the proof of theorem 4.3.1.

What remains to show is that H' is a "2-LCA DAG". That is, we want to show that $|LCA_{H'}(\{x\}, \{y\})| = 1$ for each distinct pair $\{x\}, \{y\} \in L(H')$. To do this, we consider two cases:

(1) There was a unique inclusion minimal cluster in H containing x and y

(2) There was at least two inclusion minimal clusters in H containing x and y.

First observe that from lemma 4.3.3, we know that an inclusion minimal cluster is the same thing as a least common ancestor. Suppose now that (1) holds. Thus there was a unique least common ancestor $C \in \mathscr{C}$ of $\{x\}$ and $\{y\}$ in H, and we argue that it remains a unique least common ancestor in H' after the algorithm is complete, because of the following reasons:

- $C, \{x\}, \{y\}$ are all elements of $\mathscr{C}'$ and since C is a superset of $\{x\}, \{y\}$ it follows that C remains a common ancestor of $\{x\}$ and $\{y\}$ in H' (lemma 4.3.2).
- Any descendant of C in H, remains a descendant in H', and since none of them were common ancestors of $\{x\}$ and $\{y\}$ in H, the same holds in H'. Any potential new descendant of C added through the algorithm, contains a pair of leaves of which at least one of them is not x or y. Note in particular that the since (1) holds, the cluster $\{x, y\}$ is not included in H'. Thus it follows that any such new descendant is not a common ancestor of $\{x\}$ and $\{y\}$ in H'. What follows is that there can be no descendant of C that is a common ancestor of $\{x\}$ and $\{y\}$ in H'. This result, combined with the previous conclusions, gives us that C is least common ancestor.
- C is a unique least common ancestor of $\{x\}$ and $\{y\}$, because none of the potential new clusters in H' are common ancestors of $\{x\}$ and $\{y\}$, and the set of all other common ancestors of $\{x\}$ and $\{y\}$ (excluding C) are the same as in H, none of

which are inclusion minimal, and hence (lemma 4.3.3) none of which are least common ancestors.

From the above we deduce that any pair of leaves in H' satisfying (1), has a unique least common ancestor. Now suppose (2) holds. Thus the cluster $\{x, y\}$ is present in H'. We now argue that $\{x, y\}$ is a unique least common ancestor of $\{x\}$ and $\{y\}$ in H', because of the following reasons:

- $\{x, y\}$ is a superset of $\{x\}$ and $\{y\}$, and hence a common ancestor of them in H' (lemma 4.3.2)
- $\{x, y\}$ is a least common ancestor of $\{x\}$ and $\{y\}$ in H', because any proper subset of $\{x, y\}$ can not be a superset of both $\{x\}$ and $\{y\}$.
- $\{x, y\}$ is a unique least common ancestor, because it is the unique inclusion minimal cluster containing x and y in H'.

Thus for all distinct pairs $\{x\}, \{y\} \in L(H')$, it holds that $|LCA_{H'}(\{x\}, \{y\})| = 1$, and thus H' is "2-LCA DAG" and the proof is complete.

□

4.3.2 Pseudocode for Extension algorithm

The procedure described in theorem 4.3.2 , captures the bulk of the algorithm that we will soon provide pseudocode to. Naturally, the algorithm involves defining three main functions. One which outputs the clustering system $\mathscr{C}$, another which outputs the Hasse Diagram H, and a last function which outputs the extended Hasse Diagram H'. As we will see, the function which computes $\mathscr{C}$, uses a recursive strategy, which is quite intuitive. Note in particular that

$C_G(v) = \{v\}$

whenever $v \in L(G)$ and

$C_G(v) = \cup_{u:u \in N^+(v)} C_G(u)$

whenever $v \notin L(G)$ (obs! Here we define $N^+(v)$ to be the out-neighbourhood of v).

The function which computes the Hasse Diagram H, simply takes the clustering system $\mathscr{C}$ and builds H through an iterative procedure, corresponding to that described in definition 3.0.16. Lastly, the extended Hasse Diagram H' is built by deploying an iterative procedure corresponding to that described in theorem 4.3.2.

Algorithm 4 Extension algorithm

```
    The input of the function below is a DAG $G = (V, E)$, a vertex $v \in V$, a memoization table $M$ (initialized in such a way that $M[u] = None$ for every $u \in V$) and a list "visited" which tracks all visited vertices (initialized in such a way that visited$[u]=$ False for every $u \in V$). The function outputs the tuple $(C_G(v), M)$, where $M$ has been updated to satisfy the property: $M[u] = C_G(u)$ for every $u \in V$ such that $u \preceq v$.

    function Cg(G, v, M, visited):
        visited[v] $\leftarrow$ True;
        if $outdeg_G(v) = 0$ then
            $C_G(v) \leftarrow \{v\}$;
            $M[v] \leftarrow C_G(v)$;
            return $(C_G(v), M)$
        else
            $C_G(v) \leftarrow \{\}$;
            for $u \in N^+(v)$ do
                if visited[u] $=$ True then
                    $C_G(u) \leftarrow M[u]$;
                else
                    $C_G(u) \leftarrow$ Cg$(G, u, M, visited)[0]$;
                end if
                $C_G(v) \leftarrow C_G(v) \cup C_G(u)$;
            end for
            $M[v] \leftarrow C_G(v)$;
            return $(C_G(v), M)$
        end if
    end function

    The input of the function below is a DAG $G = (V, E)$. The output is the clustering system $\mathscr{C}$ of $G$

    function ClusteringSystem(G):
        Initialize a list visited of $n$ False entries;
        Initialize a hashmap M such that $M[v] = None$ for every $v \in V$;
        $r \leftarrow 0$;
        (r corresponds to the root in G)
        Cg(G, r, M, visited);
        (Above line calls for the previous function)
        Initialize an empty set handledclusters;
        Initialize an empty list $\mathscr{C}$;
        for $v \in$ M do
            if M[v] $\notin$ handledclusters then
                Add $M[v]$ to the set handledclusters;
                Append M[v] to $\mathscr{C}$;
            end if
        end for
        return $\mathscr{C}$
    end function
```

The input of the function below is a clustering system $\mathscr{C}$. The output is the Hasse diagram H of $(\mathscr{C}, \subseteq)$.

```
function H($\mathscr{C}$):
    Initialize an empty DAG H;
    Set $V(H) = \mathscr{C}$;
    for $C_1 \in \mathscr{C}$ do
        for $C_2 \in \mathscr{C}$ do
            edgebool ← True;
            if $C_2 \not\subset C_1$ then
                edgebool ← False;
            else
                for $C_3 \in \mathscr{C}$ do
                    if $C_2 \subset C_3 \subset C_1$ then
                        edgebool ← False;
                        break
                    end if
                end for
            end if
            if edgebool = True then
                $E(H) \leftarrow E(H) \cup \{(C_1, C_2)\}$;
            end if
        end for
    end for
    return H
end function
```

The input of the function below is a DAG $G = (V, E)$ and the clustering system $\mathscr{C}$ of G. The output is the extended Hasse diagram H' of $(\mathscr{C}', \subseteq)$.

```
function HPRIME(G, $\mathscr{C}$):
    Initialize $\mathscr{C}' = \mathscr{C}$;
    Set $L = L(G)$;
    for x,y in combinations(L,2) do
        (Above line means that we are iterating over all pairs of leaves)
        Initialize an empty list A;
        (The list above is going to store all clusters in $\mathscr{C}$, containing both $x$ and $y$)
        for $C \in \mathscr{C}$ do
            if $x \in C$ and $y \in C$ then
                Append $C$ to $A$;
            end if
        end for
        countincmin ← 0;
        (countincmin := number of inclusion minimal clusters containing $x$ and $y$)
        for $C_1 \in A$ do
            inclminbool ← True;
            for $C_2 \in A$ do
                if $C_2 \subset C_1$ then
                    inclminbool ← False;
                    break
                end if
            end for
            if inclminbool = True then
                countincmin ← countincmin + 1;
            end if
        end for
        if countincmin > 1 then
            $\mathscr{C}' \leftarrow \mathscr{C}' \cup \{\{x, y\}\}$;
        end if
    end for
    return H($\mathscr{C}'$)
```

We now turn to a formal justification of the Extension algorithm.

4.3.3 Correctness of Extension algorithm

Algorithm 4 relies heavily on the result of theorem 4.3.2. Thus it suffices to prove that the "Hprime" function, defined above, follows the iterative approach described in this theorem. That is to say,

- For each distinct $x, y \in L(G)$ pair, the "Hprime" function correctly decides whether to add the cluster $\{x, y\}$ to $\mathscr{C}$ or not.
- The "Hprime" function takes this extended clustering system $\mathscr{C}'$, and correctly derives the Hasse diagram of $(\mathscr{C}', \subseteq)$.

The first claim is justified if we can show that the inclusion minimality check is done correctly. The second claim is justified by verifying correctness of the "H" function (the function which returns the Hasse Diagram of $(\mathscr{C}, \subseteq)$, where $\mathscr{C}$ is some input clustering system). As per usual, the correctness of one function relies heavily on the correctness of the prior. Thus to make the proof complete, we provide proofs for the first three functions, to then finish off by justifying the correctness of the "Hprime" function in a theorem.

Lemma 4.3.4. *Given a DAG $G = (V, E)$ and any $v \in V$, the "Cg" function in Algorithm 4 outputs the tuple $(C_G(v), M)$, where M is a memoization table satisfying $M[u] = C_G(u)$ for all $u \in V$ such that $u \preceq v$.*

Proof. Let $\hat{C}_G(v)$ be the first part of the function's output when called on input v. What we wish to show is that $\hat{C}_G(v) = C_G(v)$ and that $M[u] = \hat{C}_G(u) = C_G(u)$ for all $u \in V$ such that $u \preceq v$. Note that the notation $\hat{C}_G(v)$ is introduced in order to not confuse it with the true value $C_G(v)$.

Define for any $u \in V$ the function $\delta : V \to \mathbb{N}$ such that

$\delta(u) = 0$ if $u \in L(G)$

and

$\delta(u) = 1 + max_{w \in N^+(u)} \delta(w)$ otherwise.

Thus $\delta(u)$ measures the maximum distance from u to a leaf.

We now prove the lemma by induction over $\delta(v)$.

Suppose $\delta(v) = 0$, then $v \in L(G)$, and the function sets $\hat{C}_G(v) = \{v\}$ and updates $M[v] = \{v\}$. It then returns the pair $(\hat{C}_G(v), M)$. Indeed $\hat{C}_G(v) = C_G(v)$ because if v is a leaf, v itself is the only reachable leaf from v. Moreover, there is no strict descendant of v and so the required property of M is also satisfied. This verifies the base case.

Now assume the statement holds for all $w \in V$ such that $\delta(w) \leq k$ for some $k \in \mathbb{N}$. That is to say, suppose for such a vertex w, that $\hat{C}_G(w) = C_G(w)$ and that M is updated in such a way that $M[u] = C_G(u)$ for all $u \in V$ such that $u \preceq w$. Consider now a vertex v satisfying $\delta(v) = k + 1$. Since $k \geq 0$, we can be sure that v is an inner vertex and hence that $N^+(v) \neq \emptyset$. Thus when v is explored, the "else" block is entered. What follows is an implementation of the recursive strategy discussed in the introduction of section 4.3.2. That is,

$C_G(v) = \cup_{w:w \in N^+(v)} C_G(w)$.

The above identity holds because a leaf is reachable from v if and only if it is reachable from at least one of v's children. Thus to verify that $\hat{C}_G(v) = C_G(v)$, it suffices to show that $\hat{C}_G(w) = C_G(w)$ for each $w \in N^+(v)$. The function computes $\hat{C}_G(w)$ in exactly one of two ways. If w has already been explored, the memoization table M satisfies $M[w] = \hat{C}_G(w)$, and we extract this value. If w has not been explored, we make a recursive call to retrieve $\hat{C}_G(w)$. Because $w \in N^+(v)$ and $\delta(v) = k + 1$, it follows that $\delta(w) \leq k$. Applying the induction hypothesis now gives $\hat{C}_G(w) = C_G(w)$, and it follows immediately that $\hat{C}_G(v) = C_G(v)$. Using the induction hypothesis again, we may conclude that after iterating over $w \in N^+(v)$, M satisfies $M[u] = C_G(u)$ for all $u \in V$ such that $u \preceq w$. Since $u \preceq v$ if and only if $u \preceq w$ for at least one of $w \in N^+(v)$, it follows that M stores $M[u] = C_G(u)$ for all $u \in V$ such that $u \preceq v$. Thus the statement holds for all $k \geq 0$, completing the proof.

□

Lemma 4.3.5. *The "Clusteringsystem" function in Algorithm* 4*, correctly computes the clustering system* $\mathscr{C}$ *of the input DAG* $G = (V, E)$.

Proof. By the previous lemma, we know that after the call "Cg(G, r,M, visited)" has been made, M satisfies the property $M[u] = C_G(u)$ for all $u \in V$ such that $u \preceq r$. But since G is a DAG with a single root r, the condition $u \preceq r$ is true for all $u \in V$. Thus M stores the values of $C_G(u)$ for all $u \in V$. The remainder of the function simply filters out any duplicates in M to retrieve the final output $\mathscr{C}$.

□

Lemma 4.3.6. *The "H" function in Algorithm* 4*, correctly computes the Hasse diagram* H *of* $(\mathscr{C}, \subseteq)$*, where* $\mathscr{C}$ *is some input clustering system.*

Proof. The function starts by setting $V(H) = \mathscr{C}$, which is correct because the vertices of H are precisely the clusters in $\mathscr{C}$. The remainder of the function builds the edge set $E(H)$ by making use of definition 3.0.16. Namely, for each pair of clusters $C_1, C_2 \in \mathscr{C}$, an edge (C_1, C_2) is added to $E(H)$, if and only if $C_2 \subset C_1$, and there is no intermediate cluster $C_3 \in \mathscr{C}$ such that $C_2 \subset C_3 \subset C_1$. This establishes correctness. □

Theorem 4.3.3. *The "Hprime" function in Algorithm* 4*, correctly computes the extended Hasse diagram* H' *of* $(\mathscr{C}', \subseteq)$.

Proof. What needs to be shown is that the implemented algorithm inside the "Hprime" function, aligns with the algorithm described in theorem 4.3.2. This amounts to proving that the extension of $\mathscr{C}$ is done correctly. For if this holds, the final return statement will return the extended Hasse Diagram H' of $(\mathscr{C}', \subseteq)$, based on the correctness of the "H" function, established in the previous lemma.

The function begins by iterating through all distinct pairs of leaves $x, y \in L(G)$ and for each such pair it computes the list A, containing those clusters $C \in \mathscr{C}$ for which $x, y \in C$. Then for each cluster $C_1 \in A$, it checks whether C_1 is an inclusion minimal cluster or not. The implemented logic being that C_1 is an inclusion minimal cluster if and only if there is no other cluster $C_2 \in A$ such that $C_2 \subset C_1$. This is precisely the definition of an inclusion minimal cluster (see definition 3.0.12). Thus we can be sure that the counter "countincmin" corresponds to the number of inclusion minimal clusters containing both x and y, after the for loop across all $C_1 \in A$ is complete. If this counter is strictly greater than 1, we add the cluster $\{x, y\}$ to $\mathscr{C}$, aligning with step 1 in theorem 4.3.2. It follows that the function correctly extends $\mathscr{C}$ to $\mathscr{C}'$. The correctness of the "Hprime" function now follows immediately from the correctness of the "H" function. □

4.3.4 Space and time complexity analysis of Extension algorithm

We now turn to a formal analysis of the worst case space and time complexity of algorithm 4. Our approach is as usual, we conclude the worst case space and time complexity of all functions before deducing the overall worst case space and time complexity. The main result is captured in theorem 4.3.4. We remark again that $n = |V|$.

Lemma 4.3.7. *The worst case space complexity of the "Cg" function is $O(n^2)$ and the worst case time complexity is $O(n^3)$*

Proof. There are three main sources contributing to the space complexity of the "Cg" function. These include:

1. The list "visited", which tracks visited/unvisited vertices.
2. The memoization table M, storing $C_G(u)$ for each $u \in V$ such that $u \preceq v$. OBS! If $u \npreceq v$ then M satisfies $M[u] =$ None.
3. The recursive call stack.

The first requires $O(n)$ space , since the "visited" list contains n `False`/`True` entries. The second requires $O(n^2)$ space in the worst case, and we argue as follows: For each $u \in V$ it holds that

$|C_G(u)| = 1$ if $n = 1$ (u is the single vertex in V and hence also the only leaf.)

and

$|C_G(u)| \leq n-1$ otherwise (A finite DAG $G = (V, E)$ with a single root has at most $n-1$ leaves whenever $n \geq 2$).

Thus we are lead to the concluson $|C_G(u)| \leq n$, and the claim follows from the fact that there are n keys in M mapping to sets of size $\leq n$. At any time point during execution, the size of the recursive call stack is bounded above by n, meaning that it requires a worst case space complexity of $O(n)$ to maintain. Summarizing this analysis, we arrive at the conclusion that the worst case space complexity is

$max(O(n), O(n^2), O(n) = O(n^2)$.

Regarding the time complexity, the total time cost can essentially be divided into two components:

1. Traversal cost. I.e. the time it takes to iterate over all vertices and edges
2. Union cost. I.e. the time it takes to perform all the updates $C_G(v) \leftarrow C_G(v) \cup C_G(u)$

The implemented traversal method is recursive DFS and therefore the worst case traversal cost is $O(n^2)$ (keeping in mind that $|E| \leq n^2$). As for the union cost, we argue that since $|C_G(u)| \leq n$, each union operation has a worst case time complexity of $O(n)$. Since there are $|E|$ edges in G and each vertex $v \in V$ is visited exactly once, $|E|$ such union operations are performed. This gives a worst case union cost time complexity of $O(n^3)$. Thus the overall worst case time complexity is $O(n^2) + O(n^3) = O(n^3)$.

□

Lemma 4.3.8. *The worst case space complexity of the "Clusteringsystem" function is $O(n^2)$ and the worst case time complexity is $O(n^3)$*

Proof. For the "Clusteringsystem" function, there are two additional sources of space introduced, aside from the three listed in lemma 4.3.7. These include the set "handledclusters" as well as the the list $\mathscr{C}$. Both of which, in the worst case, can store up to n clusters, contributing to $O(n)$ space. Thus the total worst case space complexity remains at $O(n^2)$.

As for the running time, note that initializing the list "visited" and the memoization table M are both $O(n)$ operations. Setting the root $r = 0$ is $O(1)$. Calling the "Cg" function is in the worst case $O(n^3)$ (which follows from previous the previous lemma). Lastly, computing "handledclusters" and $\mathscr{C}$ is $O(n)$ (we need only to iterate over the n keys in M and perform constant time lookups and insertions). It follows that the total worst case time complexity is

$O(n) + O(n) + O(1) + O(n^3) + O(n) = O(n^3)$.

Completing the proof.

□

Lemma 4.3.9. *The worst case space complexity of the "H" function is $O(n^2)$ and the worst case time complexity is $O(n^4)$, whenever the input is the clustering system $\mathscr{C}$ of a DAG $G = (V, E)$.*

Proof. The space complexity of the "H" function is completely determined based on how much space is required to store H. The vertex set $V(H) = \mathscr{C}$ requires $O(n)$ space in the worst case (recalling $|\mathscr{C}| \leq n$). As for the edge set $E(H)$, we may still conclude that $|E(H)| \leq \frac{n \cdot (n-1)}{2}$, since H is a DAG on at most n vertices. It follows that the worst case space complexity of the function is $O(n^2)$. (OBS! Because of the properties imposed on H, there is reason to believe that the bound for $|E(H)|$ could be lowered significantly. If such is the case, the worst case space complexity might be of lower order than n^2. We discuss this briefly in chapter 5, the discussion section.)

As for the running time, note that the first two lines together is $O(n)$, $O(1)$ for initializing an empty DAG H and $O(n)$ for setting $V(H) = \mathscr{C}$. The double for loop over the set $\mathscr{C}$ is $O(n^2)$ and this proceeds in the following way

1. The first subset condition is checked in $O(n)$ time.
2. If the first "if" condition was not sastified, we initiate another for loop over $\mathscr{C}$ and perform two subset checks, checking whether both $C_2 \subset C_3$ and $C_3 \subset C_1$ holds or not. Thus the code inside the "else" statement is $O(n^2)$.
3. If the "edgebool" variable is set to `True` after the two preceding steps, we add an edge in $O(1)$ time.

It follows that the total worst time complexity is

$$O(n) + O(n^2(n + n^2 + 1)) = O(n) + O(n^3 + n^4 + n^2) = O(n) + O(n^4) = O(n^4).$$

Completing the proof.

□

Lemma 4.3.10. *The worst case space complexity of the "Hprime" function is $O(n^4)$ and the worst case time complexity is $O(n^7)$.*

Proof. There are 4 main sources contributing to the space complexity of the "Hprime" function. These include:

1. The list $\mathscr{C}'$.
2. The list of leaves L.
3. The local list A, stored when iterating over a single pair of leaves.
4. The function call $H(\mathscr{C}')$.

The first requires $O(n^2)$ space, because in the worst case, we may add up to $\binom{n}{2} \leq n^2$ new clusters to $\mathscr{C}$. The second requires $O(n)$ space because $|L(G)| \leq n$. The third requires $O(n)$ space because $|A| \leq n - |L(G)| \leq n$ (A is a subset of $\mathscr{C}$, which has size at most n, and all singleton sets $\{u\}$, $u \in L(G)$ are contained in $\mathscr{C}$ and non of these contain both x and y). The fourth requires $O(n^4)$ space. For if we change the input $\mathscr{C}$ to $\mathscr{C}'$ in lemma 4.3.9, then by means of $|\mathscr{C}'| \leq n^2$, an upper bound for the number of edges in the Hasse diagram H' of $(\mathscr{C}', \subseteq)$ is n^4 (again, this bound could potentially be much lower). Thus we deduce that the overall worst case space complexity is

$max(O(n^2), O(n), O(n), O(n^4)) = O(n^4)$.

As for the running time, we also have 4 contributing factors. These include:

1. Initializing the list $\mathscr{C}'$ by setting $\mathscr{C}' = \mathscr{C}$.
2. Storing the set of leaves in the list L.
3. Expanding $\mathscr{C}$ to $\mathscr{C}'$.
4. Calling $H(\mathscr{C}')$.

The first and second factors are both executed in $O(n)$ time in the worst case (note that for the first, we need to make a copy of $\mathscr{C}$ in python, so as to not overwrite the contents of $\mathscr{C}$). As for the third, iterating over pairs of distinct leaves $x, y \in L(G)$ is done in $O(n^2)$ time in the worst case. The for loop is then followed up with the following:

1. An iteration over all elements $C \in \mathscr{C}$ ($O(n)$), a check if $x, y \in C$ ($O(n)$, recalling that $|C| \leq n$) and then potentially appending C to A ($O(1)$). Overall this procedure is done in $O(n^2)$ time.
2. A double for loop over the elements of A ($O(n^2)$, recalling the bound $|A| \leq n$) , a check if $C_2 \subset C_1$ ($O(n)$), a check if "inclminbool = `True`" or not ($O(1)$), followed by potentially incrementing "countincmin" by 1 ($O(1)$). Overall this procedure is done in $O(n^3)$ time.
3. Appending the set $\{x, y\}$ to $\mathscr{C}$ if "countincmin" > 1 ($O(1)$).

Thus the third contributing factor has a worst case time complexity of

$O(n^2(n^2 + n^3 + 1)) = O(n^4 + n^5 + n^2) = O(n^5)$.

Lastly, we argue that the function call $H(\mathscr{C}')$ has a worst case time complexity of $O(n^7)$. For if we change the input $\mathscr{C}$ to $\mathscr{C}'$ in lemma 4.3.9, then we must take into account that $|\mathscr{C}'| \leq n^2$, but that the size of every $C \in \mathscr{C}'$ remains bounded above by n. What follows is that the overall worst case time complexity is

$O(n) + O(n) + O(n^5) + O(n^7) = O(n^7)$

and the proof is complete.

$\square$

Theorem 4.3.4. *The worst case space complexity of Algorithm 4 is $O(n^4)$ and the worst case time complexity is $O(n^7)$*

Proof. Retrieving H' from the DAG $G = (V, E)$ consists of two steps. These are as follows:

1. Extract the clusteringsystem $\mathscr{C}$ by calling "Clusteringsystem(G)" ($O(n^2)$ space, $O(n^3)$ time)
2. Derive H' by calling "Hprime(G, $\mathscr{C}$)" ($O(n^4)$ space, $O(n^7)$ time).

Thus it follows immediately that the total worst case space complexity is

$max(O(n^2), O(n^4)) = O(n^4)$

and that the total worst case time complexity is

$O(n^3) + O(n^7) = O(n^7)$

and the proof is complete.

□

4.3.5 Tweaking the extension algorithm for leaf coloured input DAGs

So far, we had derived a method for transforming a randomly generated DAG $G = (V, E)$ into a "2-LCA DAG" (namely the extended Hasse diagram H') , where the leaves of G had not yet been coloured. But before we could head into finding an algorithm for generating "BMGs", we needed to consider such input DAGs. For any randomly generated, leaf-coloured input DAG (G, σ) (not necasarily a "2-LCA DAG"), we thus considered a transformation which involved generating an extended Hasse diagram H' with the following extra condition:

$|LCA_{H'}(\{x\}, \{y\})| = 1$ for all distinct pairs $\{x\}, \{y\} \in L(H') : \sigma(x) \neq \sigma(y)$.

Indeed this was already attained by algorithm 4, since it achieved the above uniqueness property for all pairs of leaves in H'. However, we wanted to remove redundant fixes, so as to speed up the average running time. In particular, given the leaf-coloured DAG (G, σ), it was not of interest to achieve $|LCA_{H'}(\{x\}, \{y\})| = 1$ for distinct pairs $x, y \in L(G)$ for which $\sigma(x) = \sigma(y)$. To this end, we considered the following tweak of the "Hprime" function in algorithm 4.

Algorithm 5 Extension algorithm with a small tweak

```
    A leaf-coloured DAG $(G, \sigma)$ and the clustering system $\mathscr{C}$ of $G$.

    function HPRIMETWEAK(G, $\sigma$, $\mathscr{C}$):
        Initialize $\mathscr{C}' = \mathscr{C}$;
        Set $L = L(G)$;
        for x,y in combinations(L,2) do
            (Above line means that we are iterating over all pairs of distinct leaves)
            if $\sigma(x) = \sigma(y)$ then
                Continue
            end if
            Initialize an empty list A;
            (The list above is going to store all clusters in $\mathscr{C}$, containing both $x$ and $y$)
            for $C \in \mathscr{C}$ do
                if $x \in C$ and $y \in C$ then
                    Append $C$ to A;
                end if
            end for
            countincmin = 0;
            (countincmin := number of inclusion minimal clusters containing $x$ and $y$)
            for $C_1 \in A$ do
                inclminbool ← True;
                for $C_2 \in A$ do
                    if $C_2 \subset C_1$ then
                        inclminbool ← False;
                        Break
                    end if
                end for
                if inclminbool = True then
                    countincmin ← countincmin + 1;
                end if
            end for
            if countincmin > 1 then
                $\mathscr{C}' \leftarrow \mathscr{C}' \cup \{\{x, y\}\}$;
            end if
        end for
        return H($\mathscr{C}'$)
    end function
```

Correctness is easily established by considering a distinct leaf pair $\{x\}, \{y\} \in L(H')$ such that $\sigma(x) \neq \sigma(y)$, and applying the same strategy used in theorem 4.3.2 to prove that $|LCA_{H'}(\{x\}, \{y\})| = 1$. The reader may also convince themselves that the remaining properties imposed on H', namely that H' is finite, has the same single root as H and satisfies $L(H') = L(H)$, still hold.

It should be mentioned that this tweak of the "Hprime" function does not lower the worst case space or time complexity. Because in the case where each leaf has been provided a unique colour, algorithm 5 is completely equivalent to algorithm 4.

Lastly, let $x' \in L(H')$ be such that $x' = \{x\}$ for $x \in L(G)$, then by defining the function $\sigma' : L(H') \to \mathbb{N}$ as

$\sigma'(x') = \sigma'(\{x\}) = \sigma(x)$

it holds that

$|LCA_{H'}(x', y')| = 1$ for all $x', y' \in L(H') : \sigma'(x') \neq \sigma'(y')$.

Indeed then, the leaf coloured DAG (H', σ') is a "2-LCA DAG" and in particular, the BMG of (H', σ') is well-defined.

4.4 Generating the BMG of a "2-LCA DAG"

At this point, our method for generating a random "2-LCA DAG" could be described as follows:

1. Generate a random DAG $G = (V, E)$ by either using approach 1 or approach 2, described in detail in subsections 4.1.1 and 4.1.2 respectively.
2. Check if G $((G, \sigma))$ is a "2-LCA DAG" by applying the "Verification algorithm", described in section 4.2.
3. If G $((G, \sigma))$ succeeds in being a "2-LCA DAG", then return it. Otherwise, run the "Extension algorithm" described in section 4.3 to produce a new DAG H' $((H', \sigma'))$ that is a "2-LCA DAG".

To construct the BMG of a randomly generated, leaf-coloured "2-LCA DAG" (G, σ), we reverted to definitions 3.0.17 and 3.0.18. We arrived at the following algorithm.

4.4.1 Pseudocode for BMG algorithm

Algorithm 6 BMG algorithm

```
    The input of the function below is a DAG $G = (V, E)$ and two distinct vertices
    $x, y \in V$. The output is True if $y$ is a strict descendant of $x$ in $G$ and False
    otherwise.

    function IsStrictDesc($G, x, y$):
        $A_y \leftarrow AllAncestorsOf(y, G)$;
        if $x \in A_y$ then
            return True
        end if
        return False
    end function

    The input of the function below is a "2-LCA DAG" $(G, \sigma)$. The output is the BMG
    $(G', \sigma)$ of $(G, \sigma)$.

    function BMG($G, \sigma$):
        Initialize an empty DAG $G' = (V' = \emptyset, E' = \emptyset)$;
        Set $V' = L(G)$;
        for $x \in L(G)$ do
            for $y \in L(G)$ do
                if $\sigma(x) \neq \sigma(y)$ then
                    edgebool $\leftarrow$ True;
                    for $z \in L(G)$ do
                        if $z \neq y$ and $\sigma(y) = \sigma(z)$ then
                            $u := LCA(x, y, G)[0]$;
                            $v := LCA(x, z, G)[0]$;
                            (Calling the LCA function, mentioned in section 4.2)
                            if $IsStrictDesc(G, u, v)$ then
                                edgebool $\leftarrow$ False;
                                break
                            end if
                        end if
                    end for
                    if edgebool = True then
                        $E' \leftarrow E' \cup \{x, y\}$;
                    end if
                end if
            end for
        end for
        return $(G', \sigma)$
    end function
```

We now turn to a formal justification of the BMG algorithm.

4.4.2 Correctness of BMG algorithm

The correctness of the BMG algorithm is established by proving correctness of the "BMG" function. But first we must verify correctness of the "IsStrictDesc" function. The main result is captured in theorem 4.4.1

Lemma 4.4.1. *Let $G = (V, E)$ be a DAG. The "IsStrictDesc" function correctly checks if $y \in V$ is a strict descendant of $x \in V$ in G or not.*

Proof. The correctness follows immediately from the correctness of the "AllAncestorsOf" function and the fact that

$y \prec x$ if and only if $x \in A_y$.

□

Theorem 4.4.1. *The "BMG" function correctly computes the BMG (G', σ) of the "2-LCA DAG" (G, σ).*

Proof. It suffices to show that the function constructs V' and E' correctly. Since it sets $V' = L(G)$, the vertex set is correct. As for the edge set, we want to show that for all $x, y \in L(G)$, the arc (x, y) is added to E' if and only if y is a best match of x. But this follows immediately from the fact that the best match check is done in accordance with definition 3.0.17. Note in particular that the second condition in definition 3.0.17 may be exchanged with

There exists no $z \in L(G)$ for which $\sigma(z) = \sigma(y)$ and $LCA_G(x, z) \prec LCA_G(x, y)$.

Thus E' is correctly computed and the proof is complete.

□

4.4.3 Space and time complexity analysis of the BMG algorithm

As per usual, we derive the space and time complexity of the algorithm by deducing the space and time complexity of each function.

Lemma 4.4.2. *The worst case space complexity of the "IsStrictDesc" function is $O(n)$ and the worst case time complexity is $O(n^2)$.*

Proof. The amount of space used is determined entirely from the amount of space used when calling the "AllAncestorsOf" function, which has a worst case space complexity of $O(n)$ (lemma 4.2.4). Thus the worst case space complexity of the "IsStrictDesc" function is $O(n)$. To determine the running time, notice that calling the "AllAncestorsOf" function is in the worst case $O(n^2)$ (lemma 4.2.4), and that checking whether $x \in A_y$ or not, is in the worst case $O(n)$ (the size of A_y is at most $n-1$). It follows that the total worst case time complexity of the "IsStrictDesc" function is $O(n) + O(n^2) = O(n^2)$.

□

Theorem 4.4.2. *The worst case space complexity of the "BMG" function is $O(n^2)$ and the worst case time complexity is $O(n^5)$.*

Proof. There are three main factors contributing to the space complexity of the "BMG" function. These include:

1. Storing G'.
2. Storing the set of leaves in G.
3. The space required when calling the LCA function.

The first has a worst case space complexity of $O(n^2)$,which follows from the observation that the number of leaves in G is bounded above by $n-1$ for $n > 1$. These leaves make up the vertex set of G', and in the case where all have a unique colour and are best matches of each other, the size of $E(G')$ is at most $2 \cdot \binom{n-1}{2} \leq n^2$ (bound holds for $n \geq 1$). The second has a worst case space complexity of $O(n)$ and the third $O(n^2)$ (follows from lemma 4.2.6). Thus it follows that the total worst case space complexity of the "BMG" function is

$max(O(n^2), O(n), O(n^2)) = O(n^2)$.

In terms of the time complexity, we remark that initializing an empty DAG is done in $O(1)$ time and that setting $V' = L(G)$ is done in $O(n)$ time. As for the remainder of the code, the double for loop over the set of leaves is done in $O(n^2)$ time, and this is followed up with the following:

1. Checking whether $\sigma(x) \neq \sigma(y)$ holds or not ($O(1)$).

   (All steps below conditioned on previous check holding `True`.)
2. Storing the variable "edgebool" which is initially set to `True` ($O(1)$).
3. Iterating over the set of leaves a third time ($O(n)$).

   (Steps $4-7$ are done inside the for loop entered in step 3.)
4. Checking whether both $z \neq y$ and $\sigma(z) = \sigma(y)$ hold or not ($O(1)$).

   (Steps $5-7$ conditioned on the previous check holding `True`.)
5. Storing u and v, where u is the unique least common ancestor of x and y, and v the unique least common ancestor of x and z ($O(n^2)$, follows from lemma 4.2.6).
6. Checking whether v is a strict descendant of u ($O(n^2)$, follows from lemma 4.4.2).

   (Step 7 conditioned on previous check holding `True`.)
7. Resetting the value of "edgebool" to `False` ($O(1)$).

8. Checking whether the value of the "edgebool" variable is `True` (($O(1)$).

   (Step 9 conditioned on previous check holding `True`.)

9. Adding the edge (x, y) to $E(G')$ ($O(1)$).

Thus we deduce that this part of the code, in the worst case, runs in

$O(n^2(n(n^2 + n^2) + 1)) = O(n^5 + n^5 + n^2) = O(n^5)$ time.

From which it follows that that the total worst case time complexity is

$O(1) + O(n) + O(n^5) = O(n^5)$

and the proof is complete.

$\square$

5 Discussion

Though we have sucessfully derived an algorithm for transforming a random DAG into a "2-LCA DAG", we remark again that the transformation does not take into account in preserving properties in the underlying evolutionary DAG, something which may be of relevance, depending on the situation at hand. Suppose for instance that a researcher which to remove redundancies in a large dataset of DAGs (so as to produce "2-LCA DAGs") that otherwise contain important evolutionary information. To this end, an open research topic is to find a polynomial time algorithm for achieving this kind of transformation, given a set of structures/relationships to be preserved.

With regards to the running time of the various functions, the reader might raise immediate suspicion. Especially since functions like "IsValidDAG", "H", "Hprime" and "BMG" all have worst case time complexties equal to or exceeding $O(n^4)$, with "Hprime" reaching as high as $O(n^7)$. However, because of structual dependencies, a formal average case analysis would most likely result in deriving lower time complexities. To provide some insight into this argument, consider for instance the "IsValid-Dag" function. A worst case analysis of this function builds on the idea that both the number of leaves and the maximum number of ancestors of a leaf in an underlying DAG G can be considered to be $O(n)$. Roughly speaking, the combination of these factors describe a very specific DAG type. Namely a DAG with the following structural properties:

- Wide at the bottom (a lot of leaves).
- Tall (long ancestral chains)

While it is possible for such DAGs to be generated under both approaches, the underlying structural dependencies make their simultaneous occurence unlikely in practice. Especially for large values of n. For if the number of leaves is close to n, then a high fraction of the vertices are leaves, and we can expect a wide but shallow DAG (short ancestral chains). On the other hand, if the maximum number of ancestors of a leaf is close to n, then a high fraction of the vertices are internal vertices, and we can expect a narrow but tall DAG. While this is in no way a rigorous argument , it provides motivation to derive the average case time complexity of the "IsValidDAG" function, using either approach 1 or approach 2 as a probabilisitc model, and potentially prove that the average case time complexity is lower than the worst case time complexity. However, we do want to emphasise that there are known quicker algorithms for determining whether or not each pair of vertices in a DAG have a unique least common ancestor or not. In [2], Mirsolaw Kowaluk and Andrzej Lingas prove the problem to be solveable in $O(n^{\omega} log(n))$ time where $w < 2.376$ is the exponent of the fastest known algorithm for multiplication of two nxn matrices. As for the "Hprime" function, note that the factor n^7 arises as a consequence of $|\mathscr{C}'| \leq n^2$, where the worst case is one for which $|L(G)| = O(n)$ and all the clusters of the form $\{x, y\}$ (for distinct leaf pairs $x, y \in L(G)$) are added to $\mathscr{C}$. A natural question however, is if for an average case, the size of C' can be considered of lower order than n^2? To this end, let us construct an experiment that goes as follows:

1. for $p \in \{0.1, 0.3, 0.6, 0.9\}$ do:
    a) for $n \in \{20, 50, 100, 400\}$ do:
        i. for each pair (p, n), generate 100 random non-"2-LCA DAGs" on n vertices and where the probaility of adding an edge is p . Let $G_{i,p,n}$ denote the i-th $(1 \leq i \leq 100)$ generated DAG on n vertices and where the probability of adding an edge was p.
        ii. For each DAG $G_{i,p,n}$, compute the extended clustering system $\mathscr{C}'_{i,p,n}$ and store the value of $|\mathscr{C}'_{i,p,n}|$.
        iii. For each pair (p, n) compute $\rho_{p,n} = \frac{\mu_{p,n}}{n^2}$ where $\mu_{p,n} = avg_{1 \leq i \leq 100} |\mathscr{C}'_{i,p,n}|$.
    b) For fixed p, investigate the behaviour of $\rho_{p,n}$ as n grows. If it tends to 0 or decreases rapidly, we have reason to believe that on average, the size of $\mathscr{C}'$ can be considered of lower order than n^2, whenever edges are added with proability p.
2. Start by generating DAGs using approach 1 and then repeat the experiment by generating DAGs using approach 2.

We summarize the result of our experiment in four tables:

p	$\boldsymbol{\mu_{p,5}}$	$\boldsymbol{\mu_{p,10}}$	$\boldsymbol{\mu_{p,20}}$	$\boldsymbol{\mu_{p,50}}$
0.1	3.00	7.44	14.25	40.82
0.3	3.00	5.65	6.76	7.10
0.6	3.00	3.60	3.50	3.72
0.9	3.00	3.02	3.02	3.03

Table 5.1: Values of $\mu_{p,n}$ when DAGs are generated under approach 1

p	$\boldsymbol{\mu_{p,5}}$	$\boldsymbol{\mu_{p,10}}$	$\boldsymbol{\mu_{p,20}}$	$\boldsymbol{\mu_{p,50}}$
0.1	3.00	5.44	8.92	11.28
0.3	3.00	4.30	4.38	4.74
0.6	3.00	3.29	3.40	3.26
0.9	3.00	3.01	3.00	3.00

Table 5.2: Values of $\mu_{p,n}$ when DAGs are generated under approach 2

p	$\boldsymbol{\rho_{p,5}}$	$\boldsymbol{\rho_{p,10}}$	$\boldsymbol{\rho_{p,20}}$	$\boldsymbol{\rho_{p,50}}$
0.1	0.12	0.07	0.04	0.02
0.3	0.12	0.06	0.02	0.003
0.6	0.12	0.04	0.009	0.001
0.9	0.12	0.03	0.008	0.001

Table 5.3: Values of $\rho_{p,n}$ when DAGs are generated under approach 1

p	$\boldsymbol{\rho_{p,5}}$	$\boldsymbol{\rho_{p,10}}$	$\boldsymbol{\rho_{p,20}}$	$\boldsymbol{\rho_{p,50}}$
0.1	0.12	0.05	0.02	0.005
0.3	0.12	0.04	0.01	0.002
0.6	0.12	0.03	0.008	0.001
0.9	0.12	0.03	0.007	0.001

Table 5.4: Values of $\rho_{p,n}$ when DAGs are generated under approach 2

Clearly, the empirical results captured in tables 5.3 and 5.4 suggest that on average, the size of $\mathscr{C}'$ is of lower order than n^2, no matter the probability p of adding an edge, and whether approach 1 or 2 was used to generate a DAG (noting in particular that all $\rho_{p,n}$ tend to 0 as the value of n increases). Consequently, it seems to be the case that the average case time complexity is lower than $O(n^7)$. But once again, the claim can only be verified with a theoretical proof and we leave such matters as a topic for future research. Nevertheless, our current algorithm for generating the extended Hasse diagrams, need significant improvements in running time if one wishes to construct large datasets of "2-LCA DAGs" with a high number of vertices (For such a purpose, however, there is a method that involves not using the "Hprime" function at all. We discuss this in the last paragraph of this section). To close off our discussion on the running time of some of our implemented functions, let us consider the implementation of the BMG function, which was proven to run in $O(n^5)$ time in theorem 4.4.2. As in the case of the "IsValidDAG" function, the factor n^5 arises partly because both the number of leaves and the maximum number of ancestors of a leaf can be considered to be $O(n)$. However, we have already intuitevely argued that these bounds may be reduced in an average case analysis. Thus it seems reasonable that a rigorous average case analysis of the BMG function, would result in concluding that the average case time complexity is lower than $O(n^5)$. We also remark that optimizing the "LCA" function and the "AllAncestorsOf" function would help reduce even the worst case time complexity.

On the topic of space complexity, we concluded that the worst case space complexity of the "Hprime" function is $O(n^4)$. The reason being that if $\mathscr{C}(|\mathscr{C}| \leq n)$ is the input clustering system when calling the "H" function, then by means of H being a DAG, an upper bound for the number of edges in H is simply n^2. Consequently, if the the input is the extended clustering system $\mathscr{C}'(|\mathscr{C}'| \leq n^2)$, the upper bound increases to

n^4. This analysis, however, ignores additional structural constraints imposed on H. For example that:

- $\{x\} \in \mathscr{C}$ for all $x \in L(G)$.
- For $C_1, C_2 \in \mathscr{C}$, $(C_1, C_2) \in E(H)$ if and only if there is no intermediate cluster $C_3 \in \mathscr{C} \setminus \{C_1, C_2\}$ such that $C_2 \subsetneq C_3 \subsetneq C_1$.

Since singleton clusters are pairwise incomparable under set inclusion, the first constraint together with the second greatly reduces the number of possible edges in H, suggesting that the naive bound for $|E(H)|$ (and hence also $|E(H')|$) may be much lower than n^2 (n^4 for $|E(H')|$). But we leave a formal justification as an open research topic.

In the aim of deriving conjectures and results related to BMGs, one would most certainly demand some sort of control over the types of underlying "2-LCA DAGs" generated. Especially if one wishes to detect patterns. Some natural questions are:

- For fixed values of p and n, are there any significant differences between an average "2-LCA DAG" generated using approach 1 and one generated using approach 2, in terms of expected number of edges or the types of subgraphs present?
- If we fix an approach for generating the starting DAGs, how does varying p and n affect what kind of "2-LCA DAG" we can expect to generate on average? How many vertices and edges does it have? What subgraphs are present? etc.
- For fixed values of p and n , are there any significant differences between the induced probability distributions of approach 1 and approach 2? For instance, is it more likely to generate a certain family of DAGs using approach 1 than approach 2?

Answering these types of questions would provide a researcher with a toolbox for generating datasets of DAGs (in particular "2-LCA DAGs") of desired type.

We have already mentioned the major limitation of the "Hprime" function, whose purpose is to transform a non-"2-LCA DAG" into a "2-LCA DAG". Namely that it runs in $O(n^7)$ time in a worst case scenario. However, we may add that for the purpose of generating large datasets of "2-LCA DAGs", the "Hprime" function may be completely ignored. Another much more intuitive method is as follows:

1. Decide values of N, n and p where N is the size of the dataset to be generated (i.e. the number of "2-LCA DAGs"), n the number of vertices that each "2-LCA DAG" will contain and p the probability of adding an edge.
2. Produce a DAG (using either approach 1 or approach 2) and check whether it is a "2-LCA DAG" or not (achieved by running the "IsValidDAG" function). If it satisfies the condition, store the DAG. If not, disgard it.
3. Keep applying step 2 until N "2-LCA DAGs" have been stored.

The iterative procedure captured by steps 2 and 3 could easily be achieved by using a while loop. Moreover, since in each step, the probability of generating a "2-LCA DAG" is non-zero, we can be sure that the while loop eventually terminates. However, a question arises regarding the expected number of iterations, which depend entirely on the values of p and n as well as the induced probability distributions of the two approaches. Letting I be the number of iterations, it may be the case that $E[I]$ (expected number of iterations) is large. Perhaps large enough to diqualify the method entirely. Thus deriving $E[I]$ remains an open research problem of interest.

6 Conclusion

We summarize the results of our research as follows:

In the aim of constructing a method for converting a random DAG into a "2-LCA DAG", we have successfully derived a polynomial time algorithm:

1. Generate a random DAG G on a specified number of vertices n and on a specified probability p of adding an edge, using either approach 1 or approach 2.
2. Check whether G is a "2-LCA DAG" or not by running the "IsValidDAG" function.
    a) If it is a "2-LCA DAG", then simply return G
    b) If not, then extend the Hasse diagram H of $(\mathscr{C}, \subseteq)$ to H' by running the "Hprime" function. The result of theorem 4.3.2 ensures that H' is a "2-LCA DAG". Thus we can return H'.

Our current implementations of the "IsValidDAG" and"Hprime" functions , however, are not efficient in terms of their running time, which was proven to be $O(n^4)$ (theorem 4.2.2) and $O(n^7)$ respectively (lemma 4.3.10). Thus, it is of relevance to optimize these functions if one wishes to use them for recognizing and transforming large DAGs (i.e. DAGs on many vertices and edges).

Aside from deriving a method for transforming DAGs into "2-LCA DAGs", we have also briefly discussed "Best match graphs" and how they may be generated from leaf-coloured "2-LCA DAGs". The provided algorithm (Algorithm 6) quite literally follows definitions 3.0.17 and 3.0.18. However, just as in the case with the "IsValidDAG" and "Hprime" function, the "BMG" function has a high worst case time complexity, $O(n^5)$, and this would need to be reduced if one wishes to generate the "BMG" of a large leaf-coloured "2-LCA DAG" .

References

[1] Marc Hellmuth, David Schaller, and Peter F. Stadler. Clustering systems of phylogenetic networks. *Theory in Biosciences*, 142:301–358, 2023.

[2] Miroslaw Kowaluk and Andrzej Lingas. Unique lowest common ancestors in dags are almost as easy as matrix multiplication. volume 4698, pages 265–274, 2007.

[3] Anna Lindeberg and Marc Hellmuth. Simplifying and characterizing dags and phylogenetic networks via least common ancestor constraints. *Bulletin of Mathematical Biology*, 87, 2025.

[4] Anna Lindeberg, Bruno J. Schmidt, Manoj Changat, Ameera Vaheeda Shanavas, Peter F. Stadler, and Marc Hellmuth. Global least common ancestor (lca) networks, 2025.

[5] Bianca De Melo. Topological sorting. Cs560 research paper, Illinois Institute of Technology, Computer Science Department, 2012.

[6] David Schaller, Peter F. Stadler, and Marc Hellmuth. Complexity of modification problems for best match graphs. *Theoretical Computer Science*, 865:63–84, 2021.

[7] Eric W. Weisstein. Hasse diagram. From MathWorld–A Wolfram Resource, 2025.

[8] Chih-Cheng Rex Yuan and Bow-Yaw Wang. Sat-solving the poset cover problem. *arXiv*, 2025.

[9] Hao Zuo, Jinshen Jiang, and Yun Zhou. Dagor: Learning dags via topological sorts and qr factorization. *Mathematics*, 12, 2024.

Datalogi
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Matematiska institutionen