Constructing Incompatibility Graphs of Pairs of Trees in Optimal Output-Sensitive Time

Manuel Lafond
Université de Sherbrooke, Canada

Abstract

We present an output-sensitive algorithm for constructing incompatibility graphs between pairs of rooted or unrooted phylogenetic trees, in which edges represent incompatible clusters or splits. Incompatibility graphs capture conflicting evolutionary signals and play an important role in applications such as phylogenetic network reconstruction, supertree inference, and BHV distance computation. Existing approaches typically require $O(n^3/w)$ time using bitset operations, where n is the number of taxa and w is the machine word size. We introduce a new algorithm based on lowest common ancestor mappings that constructs the incompatibility graph of two rooted trees in optimal $O(n + d)$ time, where d is the number of incompatibility edges. The method is extended to unrooted trees and trees with different leaf sets while preserving the same complexity, while also being relatively simple to implement. Experimental results on random and simulated phylogenetic trees show substantial practical speedups over existing implementations, particularly on sparse incompatibility graphs, which commonly arise in large datasets.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Applied computing → Computational biology

Keywords and phrases Phylogenetics, graph theory, output-sensitive algorithms, clusters, splits, incompatibility

Digital Object Identifier 10.4230/LIPIcs.WABI.2026.25

Supplementary Material
Software (Code and data): `https://github.com/manuellafond/incompa-tree/`

Funding ML acknowledges financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC).

1 Introduction

In phylogenetics, the study of cluster and split incompatibility offers important insight into how and why conflicting evolutionary signals arise. In this setting, clusters (subsets of taxa forming clades in a rooted tree) and splits (bipartitions of taxa separated by an edge in an unrooted tree) offer complementary ways to represent phylogenies, and incompatibilities among them reflect discordance between inferred relationships. Such contradictory signals across different data sources or inference methods are often not merely noise, but can point to complex evolutionary processes. Indeed, they have important applications ranging from improving the accuracy of phylogenetic tree reconstruction [24, 9, 10] to identifying events such as horizontal gene transfer, hybridization, and recombination [14, 26, 20].

In many situations, clusters of interest come from two or more (rooted) trees, which are are possibly built from different tools or represent different evolutionary hypotheses. The incompatibility graph of two trees T, T' has a vertex for every internal node of T and T', and two vertices share an edge if their corresponding nodes have incompatible clusters, i.e., no tree can contain both clusters simultaneously. In unrooted trees, the same type of graph exists, but with clusters being replaced by splits. In this work, our aim is to reconstruct incompatibility graphs of pairs of rooted or unrooted trees as fast as possible.

Incompatibility graphs have several applications in phylogenetics. They are notably used by the CASS algorithm [26], which aims to reconstruct a phylogenetic network of small level that displays a given set of clusters. The incompatibility graph is an important component of this approach, since its connected components can be handled separately and, importantly, correspond to the biconnected components (blobs) of the output network. While the approach can deal with arbitrary sets of clusters in a heuristic manner, the CASS algorithm was shown to correctly minimize the level of the output network if it is given the set of clusters of two trees [17]. Therefore, faster incompatibility graph algorithms for this case can be useful for network reconstruction.

Another important motivation of incompatibility graphs is their central role in the Owen-Provan algorithm [23], a celebrated polynomial-time algorithm to compute the *BHV distance* between edge-weighted unrooted trees [3]. This distance projects such trees in a space where dimensions correspond to splits and asks for a shortest geodesic path in this space. The algorithm starts with the incompatibility graph G of splits of the two trees and computes up to n vertex covers on G, which determine successive directions of the desired geodesic path (this is *very* roughly speaking, see [23] for details). The current implementation [22] builds G by comparing all pairs of nodes while using bitsets for set operations, which altogether takes time $O(n^3/w)$ (n is the number of leaves and w is the word size). Each subsequent vertex cover computation takes time $O(n^3)$. Although the graph construction is not the bottleneck in theory, in practice it can sometimes dominate the running time (in our experiments, we saw that it does when the trees share many splits). Moreover, the vertex cover computations could be accelerated by recent improved flow techniques [8], and the incompatibility graph construction may eventually become both the theoretical and practical bottleneck.

Let us also note that a maximum independent set of the incompatibility graph yields a set of splits or clusters that fit into a tree. This corresponds to a notion of supertree sometimes known as a Maximum Split Fit [24, 9]. Although finding such a supertree is NP-hard [1], some heuristics do make use of the incompatibility graph (see e.g. [10]). On a similar note, we remark that *loose consensus trees*, which contain all the clusters from a set of trees that have no incompatibility at all [5], can be built directly from the incompatibility graphs, as it suffices to combine clusters corresponding to isolated vertices. There already exists an optimal linear-time algorithm to build loose consensus trees [16], but fast incompatibility graph construction algorithms can nonetheless provide alternative approaches.

In this work, our aim is to accelerate the construction of the incompatibility graph between two trees T and T', which is used by the aforementioned methods. We are not aware of any algorithm that achieves a better time than $O(n^3/w)$ as mentioned above, although $O(n^2)$ seems doable with appropriate data structures. On the other hand, it would appear that any algorithm needs to run in time $\Omega(n^2)$, because the incompatibility graph may have a quadratic number of edges and we need to write those edges to the output.

A more refined lower bound can be established though. If the graph has d edges, then the total size of the output graph is $O(n + d)$, which is a more accurate lower bound to aim for. An algorithm whose running time is dependent on the size of the input *and* the size of the output is called an *output-sensitive algorithm*. A recent example of this type of algorithm was developed by Weller [27], who showed that one can enumerate all conflicting rooted triples between a pair of trees in optimal time $O(n + t)$, with t the number of conflicting triples. In our work, we show that incompatiblity graphs can indeed be computed in optimal output-sensitive time $O(n+d)$. Note that the number of edges d of the output is not given as input, so the algorithm runs in optimal time while being oblivious to the lower bound. The algorithm works on rooted trees by applying various techniques based on the *lca-mapping* borrowed from gene tree and species tree reconciliation [4, 11, 6, 18]. We then show how this algorithm can be used on unrooted trees with the same running time.

While the complexity and correctness analysis of the algorithm are not trivial, it is not difficult to implement. We evaluated the efficiency gains of our C++ implementation against the basic $O(n^3/w)$ time algorithm. For the latter, we used our own implementation as well as the one provided by the authors of the Owen-Provan algorithm. All methods were first tested on random trees, and as we show, the density of the incompatibility graphs of random trees decreases rapidly as the number of leaves grows. This means that the number of edges d gets closer to linear as n grows, making the $O(n+d)$ complexity especially relevant in this case – as demonstrated by our empirical gains of efficiency of several orders of magnitude. We also evaluate trees that, instead of being completely random, are more likely to occur in evolution, by means of a comparison on simulated gene trees using SimPhy [21]. We studied varying levels of incomplete lineage sorting, which we show are correlated with the density of incompatibility graphs. Again, the results demonstrate that significant gains in time can be obtained with our new algorithm.

The implementation of the algorithm and all data necessary to reproduce the experiments are available at `https://github.com/manuellafond/incompa-tree/`.

2 Preliminary notions

We begin by focusing on rooted trees. We thus use the term *tree* to refer to a rooted tree. We deal with unrooted trees in a much later section, where we shall introduce the additional necessary definitions. In this setting, a tree T is a directed acyclic graph that has a single node $\rho(T)$ of indegree 0 called the *root*, such that every vertex other than the root has exactly one in-neighbor, called its *parent*. We also assume no node of T has exactly one child (in particular, there is no node of indegree 1 and outdegree 1), and we do not assume binary trees. The parent of a node $v \in V(T)$ is denoted $parent_T(v)$, and we put $parent_T(\rho(T)) = \perp$, which we interpret as undefined. We write $V(T)$ for the set of nodes of T. A *leaf* of T is a node of outdegree 0. We denote by $L(T)$ the set of leaves of T. We also write $I(T) = V(T) \setminus L(T)$, which is the set of *internal nodes* of T. Note, $\rho(T)$ is included in $I(T)$, unless T has only one node (in that case, the root counts as a leaf).

The out-neighbors of a node v of T are called the *children* of v and are denoted $ch_T(v)$. For $u, v \in V(T)$, we write $u \preceq_T v$ if u is a *descendant* of v, i.e., there is a directed path from v to u in T. We call v an ancestor of u, noting that v is a descendant and an ancestor of itself. We may write $u \prec_T v$ if $u \neq v$ also holds, and call u a *strict descendant* of v. Two nodes u, v are *incomparable* if u is not a descendant nor an ancestor of v.

The *cluster* of a node $u \in V(T)$, denoted $C_T(u)$, is the set of all leaves that descend from u. That is, $C_T(u) = \{\ell \in L(T) : \ell \preceq_T u\}$. We also denote $C(T) = \{C_T(u) : u \in V(T)\}$ for the set of clusters of T. Observe that if $v \in I(T)$, then $|C_T(v)| \geq 2$ because v has at least two children. Figure 1 illustrates two trees whose internal nodes are labeled by their cluster.

Let $Z \subseteq V(T)$. We define $lca_T(Z)$ as the lowest common ancestor of Z, that is, the node u of T that satisfies $z \preceq_T u$ for all $z \in Z$, such that u is at maximum distance from the root of T with this property. If $|Z| = 2$, we may write $lca_T(x, y)$ instead of $lca_T(\{x, y\})$, where $\{x, y\} = Z$. When we mention the notion of an *lca query*, we mean that we would like to obtain $lca_T(x, y)$ for two given nodes x, y.

Incompatibility graphs. Two sets A, B are *compatible* if there exists a tree T such that A and B are both in $C(T)$. Otherwise, A, B are *incompatible*. Note, a set with a single element is compatible with any set. Let $\mathcal{C}$ be a collection of sets. We say that $\mathcal{C}$ is *compatible* if there is a tree in which each element of $\mathcal{C}$ is a cluster. The following results are well-known.

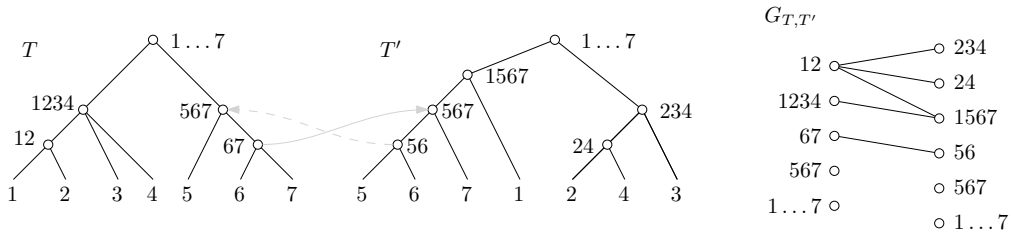

Figure 1 Two trees T, T' with $L(T) = L(T') = \{1, 2, 3, 4, 5, 6, 7\}$ and their incompatibility graph $G_{T,T'}$. Each internal node u of T (resp. x of T') is labeled with $C_T(u)$ (resp. $C_{T'}(x)$). The vertices of $G_{T,T'}$ are labeled with their corresponding cluster. The light gray arrows between T and T' represent the lca-map of two nodes. The node u of T whose cluster is $\{6, 7\}$ has $\vec{\mu}(u) = lca_{T'}(\{6, 7\})$ and the node x of T' whose cluster is $\{5, 6\}$ has $\overleftarrow{\nu}(x) = lca_T(\{5, 6\})$.

▶ **Proposition 1** ([25, 13]). *Two sets A, B are compatible if and only if at least one of $A \cap B = \emptyset, A \subseteq B$, or $B \subseteq A$ is true.*

Moreover, a collection of sets $\mathcal{C}$ is compatible if and only if every two elements $A, B \in \mathcal{C}$ are compatible. Moreover, if $\mathcal{C}$ is compatible, there exists a tree T satisfying $C(T) = \mathcal{C}$.

Let T, T' be two trees satisfying $I(T) \cap I(T') = \emptyset$. Two nodes $u \in I(T), x \in I(T')$ are *incompatible* if $C_T(u)$ and $C_{T'}(x)$ are incompatible. The *incompatibility graph* of T, T', denoted $G_{T,T'}$, is the undirected bipartite graph with vertex set $V(G_{T,T'}) = I(T) \cup I(T')$ and edge set

$$E(G_{T,T'}) = \{ux : u \in I(T), x \in I(T'), u \text{ and } x \text{ are incompatible}\}.$$

See Figure 1 for an illustration. For example, the node of T with cluster $\{1, 2, 3, 4\}$ is incompatible with the node of T' with cluster $\{1, 5, 6, 7\}$ because the two sets intersect and none is a subset of the other. On the other hand, $\{1, 2, 3, 4\}$ is compatible with every other cluster of T' because it either has no intersection with them, contains them, or is contained by them.

Let $n = \max(|L(T)|, |L(T')|)$. There is a straightforward $O(n^3)$ time algorithm to compute $G_{T,T'}$. First, compute $C_T(u)$ for each $u \in V(T)$, which can easily be done in total time $O(n^2)$. The clusters $C_{T'}(x)$ of T' can be computed similarly. Then for each $u \in V(T)$ and each $x \in V(T')$, check whether one of $C_T(u) \cap C_{T'}(x) = \emptyset, C_T(u) \subseteq C_{T'}(x)$, or $C_{T'}(x) \subseteq C_T(u)$ holds, and if not add $\{u, x\}$ to $E(G_{T,T'})$. The latter check takes time $O(n)$ and there are $O(n^2)$ pairs to check, which results in time $O(n^3)$. If the clusters are represented as bit vectors on n/w words each of size w, then the last check can be accelerated to time $O(n/w)$, for a total of $O(n^3/w)$. We aim for a running time of $O(|V(G_{T,T'})| + |E(G_{T,T'})|)$.

To finish this section, our algorithm will make use of the famous $O(n)$ time pre-processing algorithm that allows answering lca queries in constant time.

▶ **Theorem 2** ([15, 2]). *Let T be a tree. Then there exists a data structure that, after a pre-processing time of $O(|L(T)|)$ on input T, can answer any lca query on T in time $O(1)$.*

3 Computing incompatibility graphs

We assume for now that we are given two (rooted) trees T and T' with the same set of leaves $L(T) = L(T')$. We will show how to deal with unrooted trees in Section 3.3, and how to deal with trees having different leaf sets in Section 3.4.

We denote $n = |L(T)| = |L(T')|$. We also let $d = |E(G_{T,T'})|$ denote the number of edges in the incompatibility graph. Recall that d is not given and is thus not known in advance – our algorithm needs to run in time $O(n + d)$ without prior knowledge of d.

3.1 Incompatibility through lca-maps

We now introduce the notion of lca-maps to characterize incompatible nodes.

- The *lca-map from T to T'* is the map $\vec{\mu} : V(T) \to V(T')$ satisfying $\vec{\mu}(u) = lca_{T'}(C_T(u))$ for all $u \in V(T)$.
- The *lca-map from T' to T* is the map $\overleftarrow{\nu} : V(T') \to V(T)$ satisfying $\overleftarrow{\nu}(x) = lca_T(C_{T'}(x))$ for all $x \in V(T')$.

In Figure 1, the gray arrow from T to T' illustrates $\vec{\mu}(u)$, where $C_T(u) = \{6, 7\}$, and the dashed arrow from T' to T illustrates $\overleftarrow{\nu}(x)$, where $C_{T'}(x) = \{5, 6\}$.

Note that since we assume $L(T) = L(T')$, $\vec{\mu}$ and $\overleftarrow{\nu}$ are well-defined. Notation-wise, we use arrows above the map names to help remember their "direction", i.e., if we imagine T to the left of T', $\vec{\mu}$ maps nodes from the left tree T to the right tree T', and $\overleftarrow{\nu}$ maps nodes the other way around. We assume that $\vec{\mu}$ and $\overleftarrow{\nu}$ are stored in a lookup table, which can be achieved for instance by assigning a unique identifier to each node in $V(T) \cup V(T')$ between 1 and $|V(T) \cup V(T')|$. This allows accessing each value $\vec{\mu}(u)$ or $\overleftarrow{\nu}(x)$ in constant time, once the table is constructed.

Note that $\vec{\mu}$ is monotone, in the sense that $u \preceq_T v$ implies $\vec{\mu}(u) \preceq_{T'} \vec{\mu}(v)$. Likewise, $\overleftarrow{\nu}$ is monotone. We will also use the following observation.

▶ **Observation 3.** *For any $u \in V(T)$, $C_T(u) \subseteq C_{T'}(\vec{\mu}(u))$. Moreover, $C_T(u)$ is not a subset of $C_{T'}(y)$ for any child $y \in ch_{T'}(\vec{\mu}(u))$.*

Likewise, for any $x \in V(T')$, $C_{T'}(x) \subseteq C_T(\overleftarrow{\nu}(x))$. Moreover, $C_{T'}(x)$ is not a subset of $C_T(w)$ for any child $w \in ch_T(\overleftarrow{\nu}(x))$.

Proof. For $u \in V(T)$, $C_T(u) \subseteq C_{T'}(\vec{\mu}(u))$ because by the definition of $\vec{\mu}$, $\vec{\mu}(u)$ has every element of $C_T(u)$ in its descendants. Let $y \in ch_{T'}(\vec{\mu}(u))$. Then y cannot be an ancestor of every element of $C_T(u)$, since otherwise y would be a lower common ancestor of $C_T(u)$ than $\vec{\mu}(u)$. Thus $C_T(u)$ cannot be a subset of $C_{T'}(y)$. The second statement of the observation about $x \in V(T')$ can be argued in the same manner. ◀

It is well-known from the area of gene tree and species tree reconciliation that the lca-maps can be computed in linear time, see for example [7, 28].

The lca-maps lead us to a useful characterization of node incompatibility. The conditions of the next lemma are illustrated in Figure 1, where the light gray arrows explain why $\{6, 7\}$ and $\{5, 6\}$ are incompatible.

▶ **Lemma 4.** *Two nodes $v \in I(T)$ and $x \in I(T')$ are incompatible if and only if all the following hold:*

(1) $C_T(v) \cap C_{T'}(x) \neq \emptyset$*;*

(2) $x \prec_{T'} \vec{\mu}(v)$*;*

(3) $v \prec_T \overleftarrow{\nu}(x)$*.*

Proof. Let us start with the forward direction. If v, x are incompatible, this means that $C_T(v)$ and $C_{T'}(x)$ are incompatible sets. In that case, condition (1) $C_T(v) \cap C_{T'}(x) \neq \emptyset$ must hold by Proposition 1. For (2), first note that we have $C_T(v) \subseteq C_{T'}(\vec{\mu}(v))$ by Observation 3. Let $z \in C_T(v) \cap C_{T'}(x)$ (such a z exists by (1)). Then z is in both the cluster of x and the

cluster of $\vec{\mu}(v)$. Thus both x and $\vec{\mu}(v)$ are ancestors of z. This implies that x and $\vec{\mu}(u)$ cannot be incomparable. Now suppose that x is not a strict descendant of $\vec{\mu}(v)$. Then x must be an ancestor of $\vec{\mu}(v)$, with equality possible. This implies $C_T(v) \subseteq C_{T'}(\vec{\mu}(v)) \subseteq C_{T'}(x)$, which is a contradiction since it would mean that v and x are compatible by Proposition 1. Thus x can only be a strict descendant of $\vec{\mu}(v)$. For (3), a symmetric argument shows that v must be a strict descendant of $\overleftarrow{\nu}(x)$. Thus all required properties hold.

Conversely, suppose that v and x satisfy conditions 1-2-3 of the statement. We show that none of the compatibility conditions of Proposition 1 is true. We already know by condition (1) of the lemma that $C_T(v) \cap C_{T'}(x) \neq \emptyset$. Moreover, because $|C_T(v)| \geq 2$ and because $\vec{\mu}(v)$ is the lowest common ancestor of $C_T(v)$, $\vec{\mu}(v)$ has at least two distinct children y_1, y_2 that are ancestors of at least one leaf in $C_T(v)$ (if $\vec{\mu}(v)$ had only one child that is an ancestor of all $C_T(v)$, this would contradict Observation 3). Since (2) holds, x is a strict descendant of $\vec{\mu}(v)$, which means that one such child, say y_1, does not have x as a descendant nor an ancestor, and thus $C_{T'}(x) \cap C_{T'}(y_1) = \emptyset$. Therefore, $C_T(v) \setminus C_{T'}(x)$ contains some element of $C_T(v) \cap C_{T'}(y_1) \neq \emptyset$, and we deduce that $C_T(v)$ is not a subset of $C_{T'}(x)$. By the same reasoning, condition (3) stating that $v \prec_T \overleftarrow{\nu}(x)$ implies that $\overleftarrow{\nu}(x)$ has a child w_1 that is incomparable with v, such that $C_T(w_1)$ contains an element of $C_{T'}(x)$ that is not in $C_T(v)$. Thus $C_{T'}(x)$ is not a subset of $C_T(v)$. None of the conditions of Proposition 1 hold, and thus v and x are not compatible. ◀

The previous lemma suggests an algorithm to build $G_{T,T'}$. For each $v \in I(T)$, we can start a traversal of T' at $\vec{\mu}(v)$ and explore its strict descendants x that satisfy $v \prec_T \overleftarrow{\nu}(x)$ and $C_T(v) \cap C_{T'}(x) \neq \emptyset$. All such nodes satisfy the conditions of Lemma 4, and the time spent to check the conditions can be "charged" to an edge inserted into $G_{T,T'}$. When we reach a node x such that $\overleftarrow{\nu}(x)$ is not a strict ancestor of v, or such that $C_T(v) \cap C_{T'}(x) = \emptyset$, we do not explore the descendants of x further. This would work correctly, but unfortunately there are two problems with this idea. First, we are not aware of an efficient pre-processing to determine quickly whether $C_T(v) \cap C_{T'}(x) = \emptyset$. Moreover, even if such a pre-processing existed, stopping when we reach a node compatible with v implies that we waste time making compatibility checks that do not lead to edges in $G_{T,T'}$. This wasted time thus cannot be "charged" to the edges of $G_{T,T'}$, and the number of wasted compatibility checks could be arbitrarily larger than d, which is an obstacle to our desired output-sensitive running time.

Nonetheless, the characterization Lemma 4 is still useful, as it is the basis for the following alternate characterization, which sidesteps the need for set containment and intersection computations. The lemma identifies two conditions for nodes $v \in V(T)$ and $x \in V(T')$ to be incompatible, which are illustrated in Figure 2. See the caption therein for an intuition of the necessity of having at least one of those conditions. It turns out that these two are also the *only* possible reasons for incompatibility.

▶ **Lemma 5.** *Two nodes $v \in I(T)$ and $x \in I(T')$ are incompatible if and only if $v \prec_T \overleftarrow{\nu}(x)$, and in addition at least one of the following is true:*

(a) *v has a child w such that $\vec{\mu}(w) \preceq_{T'} x \prec_{T'} \vec{\mu}(v)$; or*

(b) *v has a child w such that w and x are incompatible.*

Proof. For the forward direction, suppose that v, x are incompatible. Then conditions 1-2-3 of Lemma 4 are true. In particular, Lemma 4.(3) holds and therefore $v \prec_T \overleftarrow{\nu}(x)$ holds. We must next show that at least one (a) or (b) is true. If condition (a) holds, we are done, so let us assume this is not the case. Therefore, we assume that v has no child w such that $\vec{\mu}(w) \preceq_{T'} x \prec_{T'} \vec{\mu}(v)$. In other words, for any child w of v, either $\vec{\mu}(w) \preceq_{T'} x$ is false, or $x \prec_{T'} \vec{\mu}(v)$ is false. By Lemma 4.(2) and the incompatibility of v, x, we know that $x \prec_{T'} \vec{\mu}(v)$ is true. Therefore, $\vec{\mu}(w) \preceq_{T'} x$ is false for every child w of v.

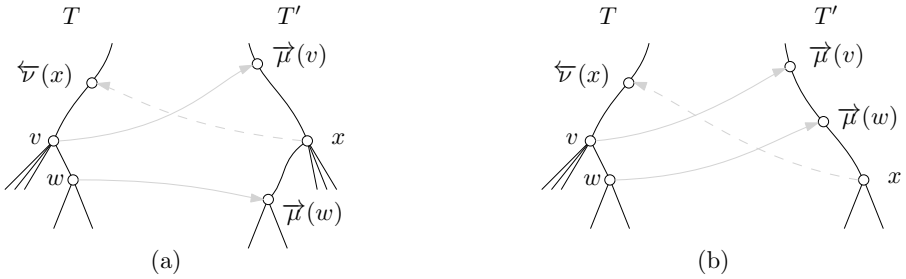

Figure 2 The two situations that make v, x incompatible in Lemma 5. Both situations require that $v \prec_T \overleftarrow{\nu}(x)$. (a) If v has a child w with $\overrightarrow{\mu}(w) \preceq_{T'} x \prec_{T'} \overrightarrow{\mu}(v)$, then $C_{T'}(x)$ contains $C_T(w)$ and therefore intersects with $C_T(v)$, and in addition $x \prec_{T'} \overrightarrow{\mu}(v)$ holds, so we have all conditions of Lemma 4. (b) If v has a child w that is incompatible with x, then Lemma 4 applies to w and x. Thus $C_{T'}(x)$ intersects $C_T(w)$ and therefore intersects $C_T(v)$, and we have $x \prec_{T'} \overrightarrow{\mu}(w) \preceq_{T'} \overrightarrow{\mu}(v)$, and again we invoke Lemma 4 to deduce that v, x are incompatible.

We want to show that condition (b) holds. By Lemma 4.(1), there exists $z \in C_T(v) \cap C_{T'}(x)$ since the latter is non-empty. Let w be the child of v such that $z \preceq_T w$ (this w must exist). We argue that conditions 1-2-3 of Lemma 4 hold on w and x. Note that $z \in C_T(w) \cap C_{T'}(x)$, so the intersection is non-empty and Lemma 4.(1) holds. For condition (2), in T', both $\overrightarrow{\mu}(w)$ and x are ancestors of z, so $\overrightarrow{\mu}(w)$ and x cannot be incomparable. At the same time, we are assuming that $\overrightarrow{\mu}(w) \preceq_{T'} x$ is false. It follows that $x \prec_{T'} \overrightarrow{\mu}(w)$. As for condition (3), we have $w \prec_T \overleftarrow{\nu}(x)$ because $w \prec_T v$ and $v \prec_T \overleftarrow{\nu}(x)$ (using Lemma 4.(3) on v and x). All conditions of Lemma 4 have been gathered and we can deduce that w and x are incompatible, and thus condition (b) holds.

For the converse direction, suppose that $v \prec_T \overleftarrow{\nu}(x)$ and that one of (a) or (b) holds. In particular, note that Lemma 4.(3) is true. Assume that condition (a) holds, so that $\overrightarrow{\mu}(w) \preceq_{T'} x \prec_{T'} \overrightarrow{\mu}(v)$ for some $w \in ch_T(v)$. In particular, Lemma 4.(2) is true. Moreover, $\overrightarrow{\mu}(w) \preceq_{T'} x$ and Observation 3 imply that $C_T(w) \subseteq C_{T'}(\overrightarrow{\mu}(w)) \subseteq C_{T'}(x)$. This implies that $C_{T'}(x) \cap C_T(v) \neq \emptyset$, which is the condition Lemma 4.(1). We have all the conditions of Lemma 4 and therefore v, x are incompatible. So assume that that (b) holds, so that x is incompatible with a child w of v. By Lemma 4, $x \prec_{T'} \overrightarrow{\mu}(w)$ and thus $x \prec_{T'} \overrightarrow{\mu}(v)$ by the monotonicity of $\overrightarrow{\mu}$. Also by Lemma 4, $C_{T'}(x) \cap C_T(w) \neq \emptyset$ which implies $C_{T'}(x) \cap C_T(v) \neq \emptyset$. We already had $v \prec_T \overleftarrow{\nu}(x)$ and again, all the conditions of Lemma 4 have been gathered and so v, x are incompatible. ◀

3.2 The algorithm

We need one last ingredient before proceeding. Let $x \in V(T')$. We define $diff\_anc(x)$ as the closest ancestor y of x that satisfies $\overleftarrow{\nu}(y) \neq \overleftarrow{\nu}(x)$ (by closest, we mean at minimum distance from x). If no such ancestor exists, then $diff\_anc(x) = \perp$, which is interpreted as undefined. The values of $diff\_anc$ for every $x \in V(T')$ can easily be pre-processed in time $O(n)$ in a preorder traversal. That is, in a top-down fashion we can use the recurrence that puts $diff\_anc(\rho(T')) = \perp$, and for a non-root node x we have

$$diff\_anc(x) = \begin{cases} diff\_anc(parent_{T'}(x)) & \text{if } \overleftarrow{\nu}(x) = \overleftarrow{\nu}(parent_{T'}(x)), \\ parent_{T'}(x) & \text{otherwise.} \end{cases}$$

The pseudo-code of our procedure is given in Algorithm 1 (note, we use G instead of $G_{T,T'}$ in the algorithm). At a high level, our algorithm uses Lemma 5 as follows. It iterates over each $v \in I(T)$, and for each such v it checks for incompatibilities that could arise from

either condition (a) or (b). Condition (b) is the easiest to implement and is performed by the function *check_incomp_child*. In essence, condition (b) says that we can look at every x incompatible with a child w of v, and if $v \prec_T \overleftarrow{\nu}(x)$, then the node x is automatically incompatible with v. The algorithm maintains an array Q of visited nodes to prevent adding the same edge multiple times. That is, while iterating on v, we have $Q[x] = v$ if and only the edge $\{v, x\}$ was added (and the same Q is re-used for all v's).

Algorithm 1 The main algorithm to build $G_{T,T'}$, denoted G in the algorithm.

```
1  function build_incompatibility_graph(T, T')
2      Pre-process T, T' for O(1) lca queries
3      Construct the lca-maps μ⃗ and ν⃖
4      Pre-process T' for O(1) diff_anc(x) queries
5      Let G be a graph with vertex set V(T) ∪ V(T') and no edge
6      Let Q be a global array of length |V(T')|, with values initialized to −1
7      for each v ∈ I(T) in a postorder traversal do
8          check_between(v, G)
9          check_incomp_child(v, G)
10
11 function check_between(v, G)
12     for each child w of v do
13         x = μ⃗(w)
14         while x ≠ ⊥ and ν⃖(x) ⪯_T v do
15             x = diff_anc(x)
16         while x ≠ ⊥ and x ≺_T' μ⃗(v) and Q[x] ≠ v do
17             Add {v, x} to E(G)
18             Q[x] = v
19             x = parent_T'(x)
20
21 function check_incomp_child(v, G)
22     for each child w of v do
23         for each edge {w, x} incident to w in G do
24             if v ≺_T ν⃖(x) and Q[x] ≠ v then
25                 Add {v, x} to E(G)
26                 Q[x] = v
```

Condition (a) is implemented in the function *check_between*. The condition says that we also need to do a search from every $\vec{\mu}(w)$, and look for ancestors x of $\vec{\mu}(w)$ satisfying $x \prec_{T'} \vec{\mu}(v)$ and $v \prec_T \overleftarrow{\nu}(x)$. We thus need a fast way to start from $\vec{\mu}(w)$ and "jump" to its first ancestor x satisfying $v \prec_T \overleftarrow{\nu}(x)$, if any. Thanks to the fast $diff\_anc$ queries and some arguing, the loop on line 14 makes a constant number of iterations and only take time $O(1)$. Once we have found this first ancestor x, we traverse all ancestors x' of x that also satisfy $x' \prec_{T'} \vec{\mu}(v)$ and stop when this becomes false. To visualize this, consider Figure 2.(a). If we imagine that x in the subfigure is the first ancestor of $\vec{\mu}(w)$ satisfying $v \prec_T \overleftarrow{\nu}(x)$, then every node on the path from x to $\vec{\mu}(v)$, excluding the latter, is incompatible with v. Therefore, the check for condition (a) can be seen as traversing a set of subpaths of T' that contain incompatible nodes with v. We just need to ensure that we do not waste time going through the same subpaths multiple times, which is the role of the Q array.

We can now state our main result.

▶ **Theorem 6.** *Algorithm 1 correctly computes the incompatibility graph $G_{T,T'}$ of two trees T, T' with the same leaves in time $O(n+d)$, where n is the number of leaves of the input trees, and d is the number of edges of $G_{T,T'}$.*

Proof sketch. The full proof is a bit lengthy, so it can be found in the Appendix. Here we provide a sketch. The correctness essentially follows from Lemma 5. If v, x satisfy condition (a) of the lemma, then *check_between* will catch it and add the edge $\{v, x\}$, and if it satisfies condition (b), then *check_incomp_children* will catch it. There are some subtleties though. In particular, it is important for correctness to perform *check_between* before *check_incomp_child*. This is because we would not want *check_incomp_child* to put some $Q[x] = v$, which could later prevent *check_between* to explore subpaths above that x (see the correctness proof for details).

For the running time, we already know that every pre-processing step takes time $O(n)$. Consider the time spent in *check_between*. It loops through the children of each node $v \in V(T)$, and overall throughout the execution of the whole algorithm, this corresponds to going through every parent-child relation of T, of which there are $O(n)$. The **while** loop on line 14 starts at $x = \vec{\mu}(w)$, and then loops through ancestors until $v \prec_T \overleftarrow{\nu}(x)$ (or $x = \perp$). We can argue that the loop makes at most three iterations, because in the worst case we start with $\overleftarrow{\nu}(x) = w$, then after one $diff\_anc$ we get $\overleftarrow{\nu}(x) = v$ or above, and then after one more $diff\_anc$, $\overleftarrow{\nu}(x)$ if definitely a strict ancestor of v (or $\perp$). Therefore the loop actually takes time $O(1)$ per w. As for the loop on line 16, every time we enter it, we add an edge to the graph G, and thus the total time spent in this loop throughout the whole execution is $O(d)$ (the array Q of "visited" nodes is needed here to ensure we add each edge exactly once). Note, checking whether $v \prec_T \overleftarrow{\nu}(x)$ can be done in time $O(1)$ using lca queries. Hence *check_between* takes time $O(n+d)$. Then, *check_incomp_child* also enumerates the $O(n)$ parent-child relations, and the loop on line 23 amounts to enumerating every edge of G throughout the whole execution, of which there are $O(d)$. The code inside that loop then takes constant time, so that subfunction also takes time $O(n+d)$. ◀

3.3 Extending to splits and unrooted trees

We now deal with unrooted trees, which have splits instead of clusters. The short version for the reader that is in a hurry: to obtain the incompatibility graph of splits of two unrooted trees on the same leaves, we root both trees at the same arbitrary taxon and run our algorithm (note, a one-sentence version of this idea appears in [12]). This seems to be a folklore procedure, but we could not find a proof of correctness of this way of doing, so we provide the details below.

An *unrooted tree* U is an acyclic connected graph with no vertex of degree 2. We write $V(U)$ and $E(U)$ for the set of vertices and edges of U, respectively. We also write $L(U)$ for the set of leaves of U, i.e., its set of vertices that have a single neighbor. A *split* of a set X is a partition $\{A, B\}$ of X, so that A, B are non-empty and non-intersecting and $A \cup B = X$. We denote such a split by $A|B$, noting that $B = X \setminus A$ and $A = X \setminus B$. Given an unrooted tree U, removing an edge $uv \in E(U)$ results in two connected components, and the set of leaves present in these components results in a split $A|B$ of $L(U)$, which we call the *split of* uv and denote by $S_U(uv)$. The set of splits of U is $S(U) = \{S_U(uv) : uv \in E(U)\}$.

Two splits $A|B$ and $C|D$ of a set X are *compatible splits* if there exists an unrooted tree U with $L(U) = X$ that contains both splits. Given two unrooted trees U, U', the *split incompatibility graph of* U *and* U' is the graph $G^{\mathsf{u}}_{U,U'}$ whose vertex set is $E(U) \cup E(U')$, with an edge $\{uv, xy\} \in E(G^{\mathsf{u}}_{U,U'})$ if and only if $S_U(uv)$ and $S_{U'}(xy)$ are incompatible splits (note, the u in the superscript of $G^{\mathsf{u}}_{U,U'}$ stands for "unrooted").

The following is well-known.

▶ **Proposition 7** ([25]). *Two splits $A|B, C|D$ of some set X are compatible splits if and only if at least one of the intersections $A \cap C, A \cap D, B \cap C, B \cap D$ is empty.*

The next result shows how computing $G^{\mathsf{u}}_{U,U'}$ from unrooted trees reduces to computing $G_{T,T'}$ on rooted trees, for an appropriate rooting, with the same running time. See the Appendix for the proof.

▶ **Proposition 8.** *Let U, U' be two unrooted trees with $L(U) = L(U') = \{1, 2, \ldots, n\}$ with $n \geq 3$, and let T, T' be the rooted trees obtained from U and U', respectively, by assigning the neighbor of leaf 1 as the root in both trees.*

Let $uv \in E(U)$ and $xy \in E(U')$, where u is the parent of v in T and x is the parent of y in T'. Then $S_U(uv)$ and $S_{U'}(xy)$ are compatible splits if and only if $C_T(v)$ and $C_{T'}(y)$ are compatible clusters.

3.4 Extending to trees with different leafsets

Going back to rooted trees, we now discuss how to deal with trees on different leaf sets. We show that this is mostly a small technicality, as we can add the missing leaves to the two trees without altering the incompatibility graph.

Suppose that $L(T) \neq L(T')$. If $L(T') \setminus L(T) \neq \emptyset$, define a tree $\tilde{T}$ obtained from T as follows. Start from T, add a new root r and new leaves $L(T') \setminus L(T)$, and add $\{\rho(T)\} \cup (L(T') \setminus L(T))$ as children of r. If $L(T') \setminus L(T) = \emptyset$, define $\tilde{T} = T$. Define $\tilde{T}'$ from T' in a similar manner, i.e., add a new root to T' whose children are $\rho(T')$ and $L(T) \setminus L(T')$ if the latter is non-empty, and otherwise define $\tilde{T}' = T'$. Note that if we create a new root in either tree, it has at least two children, so no internal node with a single child is created (proof in Appendix).

▶ **Proposition 9.** *Let T, T' be two trees and let $\tilde{T}, \tilde{T}'$ be defined as above. Then $|V(G_{\tilde{T},\tilde{T}'})| \leq |V(G_{T,T'})| + 2$. Moreover, ux is an edge of $G_{T,T'}$ if and only if ux is an edge of $G_{\tilde{T},\tilde{T}'}$.*

We can easily construct $\tilde{T}$ and $\tilde{T}'$ from T and T' in time $O(n)$. It then suffices to construct $G_{\tilde{T},\tilde{T}'}$ and remove $\rho(\tilde{T})$ and $\rho(\tilde{T}')$ to obtain $G_{T,T'}$. Since the two incompatibility graphs have the same size (number of vertices plus edges) with the exception of two isolated vertices, the complexity of building $G_{\tilde{T},\tilde{T}'}$ is the same as that of $G_{T,T'}$.

3.5 Extending to more than two trees

As stated in the introduction, there are some approaches that use the incompability graph of possibly more than two trees. Let $\mathcal{T} = \{T_1, \ldots, T_k\}$ be a set of k trees. The graph $G_{\mathcal{T}}$ has vertex set $\bigcup_{i=1}^{k} I(T_i)$ and edge set $\bigcup_{1 \leq i < j \leq k} E(G_{T_i,T_j})$. Let us denote by n the maximum number of leaves in one of the input trees. The naive implementation goes through every pair of trees and adds the incompatibility edges between them, which takes time $O(k^2 n^3/w)$ to construct $G_{\mathcal{T}}$.

If we let $d_{ij} = E(G_{T_i,T_j})$ for each distinct $i, j \in \{1, 2, \ldots, k\}$, and denote by d the number of edges in $G_{\mathcal{T}}$, we can iterate over every pair of trees T_i, T_j and add the edges found by our approach. This takes time $O(\sum_{1 \leq i < j \leq k}(n + d_{ij})) = O(k^2 n + \sum_{1 \leq i < j \leq k} d_{ij}) = O(k^2 n + d)$, with d the number of edges of $G_{\mathcal{T}}$.

While this is potentially much faster than the naive idea, this complexity does not coincide with the lower bound. Indeed, with k trees the size of the input is $O(kn)$ and the size of the output is $O(kn + d)$. We do not know whether this lower bound can be achieved. This would be surprising, as there is not even enough time to perform any kind of linear pre-processing

between the pairs of trees, even though we may need to add edges between vertices from all distinct pairs of trees. Perhaps some clever ideas can be borrowed from $O(nk)$ time algorithms for tree consensus, which achieve optimal time $O(nk)$ by iterating over the trees one by one while maintaining a single tree that captures the necessary information from the previous trees. This appears extremely difficult, if not impossible, so we leave this as an open problem.

Open problem. Find an algorithm that construct the incompatibility graph of k trees with n leaves in time $O(nk + d)$, or determine that no such algorithm exists.

Note, if the pairs of input trees have $\Omega(n)$ incompatibility edges on average, then $d \in \Omega(k^2 n)$, in which case $O(k^2 n + d)$ is optimal. So, an $O(nk + d)$ time algorithm would need to have improved complexity specifically on instances where pairs of trees have $o(n)$ incompatibility edges, on average.

4 Experiments

The main purpose of our experiments is quantify empirically the efficiency gains of Algorithm 1. We tested two C++ implementations for the computation of incompatibility graphs. The first is the implementation of Algorithm 1, which we call the *fast* implementation. The second is the $O(n^3/w)$ time algorithm that compares each pair of clusters from the two trees, as described at the end of Section 2. For faster intersection and set containment tests, we stored clusters using the efficient EWAHBoolArray implementation from [19], which are compressed bit arrays. We call this the *naive* implementation[1]. Note, we did not implement the pre-processing for $O(1)$ lca queries, and instead use a basic $O(h)$ time algorithm for lca queries, where h is the height of the tree. Profiling showed that these lca queries consist of less than 1% of the running time.

To limit possible biases that could arise from using only our own implementations, we also included the incompatibility graph computation used in the Owen-Provan algorithm [23] in our comparisons – we used the authors' implementation provided here: [22]. The latter implements the polynomial-time algorithm to compute the BHV distance between trees mentioned in the introduction. Computing the incompatibility graph is the first step, with the second step consisting of vertex cover computations. Since we are only interested in the first step, we made a minor modification to the code to calculate only the time needed to compute the incompatibility graph. As a small remark, the algorithm has a pre-processing step that calculates splits that are common to both trees. This can sometimes accelerate computation, since only splits that are on the same side of a common edge need to be compared for incompatibility. As we also count the pre-processing time in our implementation, we included the time to calculate the time to find common splits of the Owen-Provan algorithm[2]. The reader should bear in mind that the authors of the Owen-Provan algorithm did not aim to optimize the incompatibility graph step, and it is implemented in Java while our code is C++, so the comparisons should be taken with a grain of salt. All experiments were conducted on a standard laptop with 16 GB of RAM and a 3.2 GHz processor using a single CPU core.

[1] We also implemented a third version which represents clusters using a standard `std::set` object from C++. This was used to validate that all three methods produced the same incompatibility graph, on all pairs of trees (which was the case). We do not report the times for that third implementation, as it is about 10 times slower than the bit array version.

[2] To be fully specific, the timer starts at the beginning in the *getGeodesic* function, and stops right after the function *getIncidenceMatrix* has returned.

4.1 Datasets

We analyzed random trees and simulated gene trees with incomplete lineage sorting.

Random trees. For each $n \in \{20, 50, 100, 200, 500, 1000\}$, we generated 50 random binary trees with n leaves. We used the standard randomization process where we start with a root containing all leaf labels, then choose a random partition into two non-empty sets, add two children to the root and pass them the respective labels sets, and then repeat recursively. In each tree, we then added a new leaf adjacent to the root, to simulate the reduction from unrooted to rooted trees described in Proposition 8.

With each implementation and each n, we computed the incompatibility graph for each of the $\binom{50}{2} = 1225$ pairs of trees that we generated.

Simulated trees. In order to analyze trees that are more likely to occur in evolution, we generated gene trees with SimPhy [21] under varying levels of incomplete lineage sorting (ILS). We turned off gene duplication, loss, and transfer events. In a nutshell, using SimPhy as such generates a species tree and then gene trees that evolve inside of it. With no ILS, all gene trees are identical to the species tree. As the amount of ILS increases, the gene trees maintain alleles for longer periods of time in the species branches, making them become increasingly different from the species tree, and from each other. The level of ILS therefore has an impact on the number of incompatibilities created between the gene trees. In SimPhy, the *population* parameter controls the amount of ILS, and we observed incompatibilities only at populations from 10^6 and onwards.

So, for each $n \in \{20, 50, 100, 200, 500, 1000\}$ and each population $p \in \{10^6, 10^7, 10^8, 10^9\}$, we simulated 50 gene trees with SimPhy. As before, with each implementation and each n, we computed the incompatibility graph for each of the $\binom{50}{2} = 1225$ pairs of trees.

4.2 Results on random trees

The total running time to compare all pairs of random trees is shown in Figure 3. The x-axis is the number of leaves and the y-axis is the number of seconds needed to compare all pairs of random trees.

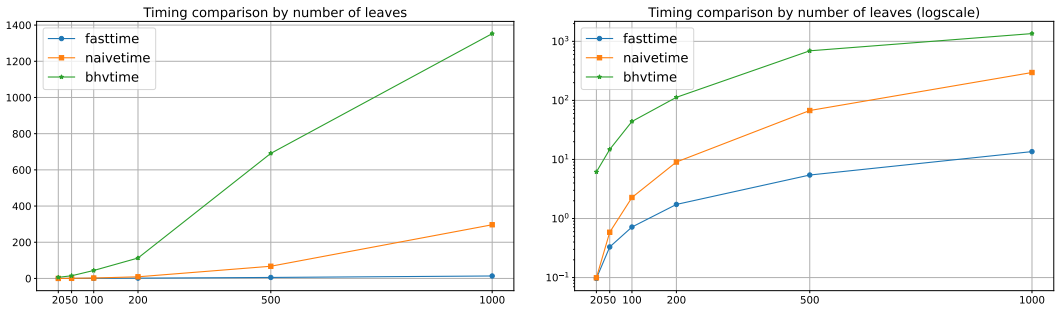

Figure 3 Total running time to compare all $\binom{50}{2}$ pairs of random trees, per number of leaves. **x-axis**: number of leaves. **y-axis**: total time in seconds. On the left, a standard scale is used for the y-axis. On the right, the same data but using a logarithmic scale for the y-axis.

It is not surprising to see that the *fast* implementation is much faster, especially as the trees get larger. The magnitude of the speedup of our algorithm is easier to see using a logarithmic scale (on the right). The *fast* algorithm tends to be about 10 to 30 times

times faster than our *naive* implementation, and around 100 times faster than the BHV implementation. Notably, with the most interesting case of $n = 1000$, the times to compare all pairs of trees for $fast, naive$ and bhv were respectively $13, 296$, and 1352 seconds.

Density analysis: large trees have sparse incompatibility graphs. We define the density of an incompatibility graph $G_{T,T'}$ as $\frac{|E(G_{T,T'})|}{|I(T)|\cdot|I(T')|}$. This is the number of edges of $G_{T,T'}$ divided by the maximum possible number of edges. Figure 4 illustrates the average density of the random trees as a function of n.

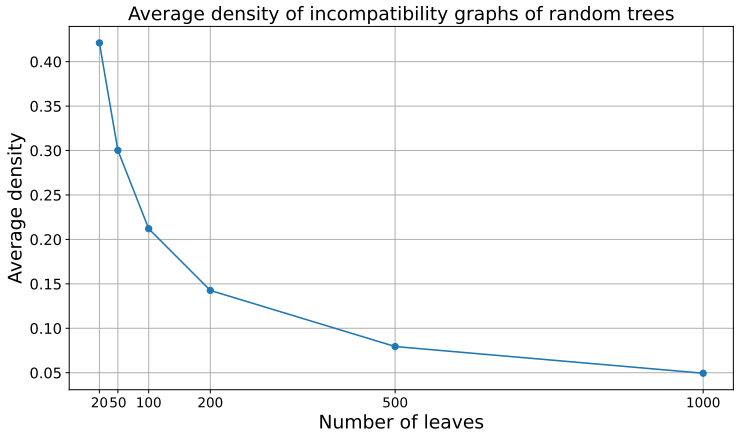

Figure 4 Average density $G_{T,T'}$ for the pairs of random trees, with respect to the number of leaves.

The figure shows that the proportion of incompatibilities decreases quickly as we increase the size of the trees. We are not aware of any study that establishes the probability of two clusters from random trees to be incompatible, but there is at least an intuition that explains this. In a binary tree, "most" clusters are expected to be very small, in the sense that clusters of small constant size represent a large proportion of the clusters. As n gets larger, such constant-size clusters are more likely to be non-intersecting with the clusters from the other tree. In other words, small clusters dominate the proportion of clusters, and the proportion of clusters they are incompatible with goes down with n.

One way to see this more concretely is to think incompatibilities between cherries in the trees, i.e., nodes whose children are both leaves, and therefore whose cluster has size 2. For example, if T, T' are fully balanced binary trees, roughly half of the internal nodes in either tree are cherries. The cluster $\{x, y\}$ of a cherry of T intersects only at most two cherries of T', so the *proportion* of cherries of T' that $\{x, y\}$ is incompatible with vanishes very quickly as n grows. Then, if we consider clusters of size at most 4, they make up about 75% of the clusters in either tree, and using the same argument, the proportion of possible incompatibilities with subtrees of size 4 decreases as n grows, and so on.

Characterizing the expected density of $G_{T,T'}$ of two random trees T, T' as a function of n is an interesting side question, but it is outside the scope of this paper. Nonetheless, we would argue that the fact that the density goes down with n is a strong argument in favor of our algorithm. It means d has a lesser impact in the $O(n + d)$ complexity when n gets large.

4.3 Results on simulated gene trees

Running time analysis. Figure 5 shows the total running times on Simphy gene trees, where there is one curve for each combination of method and population. The tendencies are the same as in the random trees (notice, the curves are clustered by method for all population

sizes). We notice that the $fast$ implementation is slightly slower on larger populations. This is very likely because those produced denser graphs and therefore larger values of d, as shown in the next subsection.

In contrast, the bhv implementation was noticeably faster on larger populations (the gap in the plot between population 10^9 and the others is significant as it uses a logarithmic scale). This required investigating the code for an explanation. It turns out that the bhv implementation is slower when the given trees are similar. This is implementation-specific and outside the scope of this paper, but in a nutshell, we found that this is caused by the pre-processing that enumerates common splits. If any such split exists, the algorithm repeats recursively on the matching subtrees on each side of the common split, thereby starting two new common split enumerations on the two smaller pairs of subtrees. So basically, this common splits enumeration happens as often as the number of common splits, making it slower on similar trees. This also explains why the total bhv times on gene trees are higher than on the random trees, because random trees seem to rarely share the exact same splits.

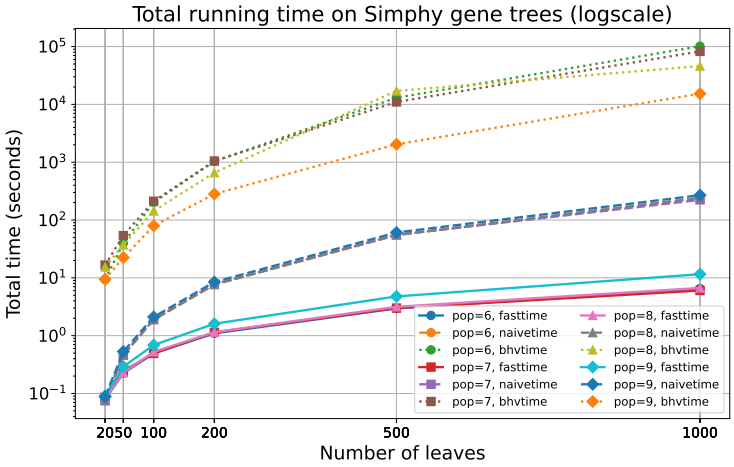

Figure 5 Total running time to compare all $\binom{50}{2}$ pairs of gene trees, for each population, per number of leaves (in the legend, $pop = X$ means a population of 10^X). **x-axis**: number of leaves. **y-axis**: total time in seconds. Full lines as $fast$, dashed lines are $naive$, and dotted lines are bhv. Note, the y-axis uses a logarithmic scale (the standard scale plot did not add anything meaningful, so we omit it).

Density analysis. Figure 6 shows the average density of the incompatibility graphs on the SimPhy datasets, for each number of leaves and population size. As in the random trees, we again observe a decrease in density as n grows, highlighting once again the importance of output-sensitive algorithms for large n. Note, the curve for population size $pop = 1e6$ behaves a bit more erratically – this is probably because for the vast majority of pairs of trees T, T', the graph $G_{T,T'}$ has 0 edges, so the few graphs that do have edges affect the average in more pronounced ways.

We also find it interesting that the amount of ILS in the trees has a direct impact on the proportion of incompatibilities. Although this is not surprising, it highlights that incompatibility graphs capture biologically meaningful disagreement between trees, rather than merely representing an abstract combinatorial object, which further motivates the need for efficient algorithms in downstream applications such as BHV distance computations and network inference.

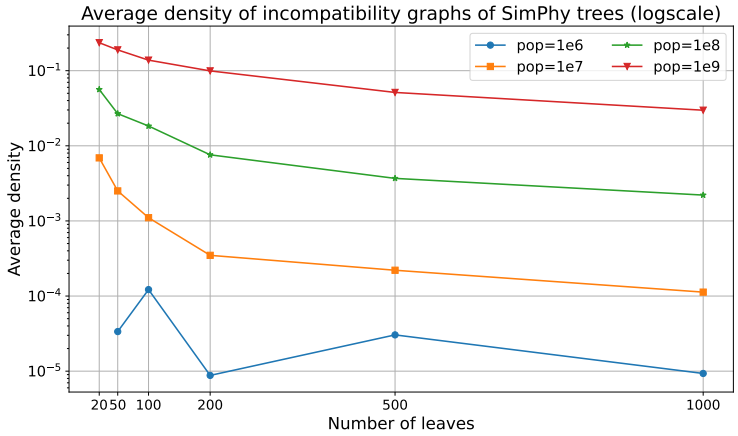

Figure 6 Average density of $G_{T,T'}$ for the pairs of SimPhy gene trees, with respect to n the number of leaves. There is one curve for each population size, from $1e6 = 10^6$ to $1e9 = 10^9$. For $pop = 1e6$ and $n = 20$, there is no point because the average density is 0, which cannot be displayed on a logarithmic scale.

5 Conclusion

In this work, we presented the first output-sensitive algorithm for constructing incompatibility graphs between pairs of rooted or unrooted phylogenetic trees. Beyond the theoretical improvement, our experiments demonstrate that incompatibility graphs arising from random and simulated data are often sparse in practice, making the output-sensitive complexity especially relevant for large datasets. In particular, we observed substantial empirical speedups over existing implementations used in BHV distance computations.

The algorithm opens several directions for future work. On the practical side, it would be interesting to integrate the method directly into software for BHV distance computation, phylogenetic network reconstruction, and supertree inference, in order to evaluate its impact on large-scale biological datasets. From a theoretical perspective, several questions remain open. Most notably, we do not know whether the incompatibility graph of k trees can be constructed in optimal time $O(nk + d)$, where d is the number of incompatibility edges. Another interesting direction is to better understand the expected structure and density of incompatibility graphs under probabilistic models of tree evolution, particularly in the presence of processes such as incomplete lineage sorting, hybridization, or horizontal gene transfer.

References

1 Jonathan Badger, Paul Kearney, Ming Li, John Tsang, and Tao Jiang. Selecting the branches for an evolutionary tree.: A polynomial time approximation scheme. *Journal of Algorithms*, 51(1):1–14, 2004. doi:10.1016/S0196-6774(03)00086-5.

2 Michael A Bender and Martin Farach-Colton. The lca problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

3 Louis J Billera, Susan P Holmes, and Karen Vogtmann. Geometry of the space of phylogenetic trees. *Advances in Applied Mathematics*, 27(4):733–767, 2001. doi:10.1006/AAMA.2001.0759.

4 Paola Bonizzoni, Gianluca Della Vedova, and Riccardo Dondi. Reconciling a gene tree to a species tree under the duplication cost model. *Theoretical computer science*, 347(1-2):36–53, 2005. doi:10.1016/J.TCS.2005.05.016.

5 Kåre Bremer. Combinable component consensus. *Cladistics*, 6(4):369–372, 1990.

6 Cedric Chauve and Nadia El-Mabrouk. New perspectives on gene family evolution: losses in reconciliation and a link with supertrees. In *Annual International Conference on Research in Computational Molecular Biology*, pages 46–58. Springer, 2009.

7 Kevin Chen, Dannie Durand, and Martin Farach-Colton. Notung: dating gene duplications using gene family trees. In *Proceedings of the fourth annual international conference on Computational molecular biology*, pages 96–106, 2000. doi:10.1145/332306.332351.

8 Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *Journal of the ACM*, 72(3):1–103, 2025. doi:10.1145/3728631.

9 Chris J Creevey and James O McInerney. Clann: investigating phylogenetic information through supertree analyses. *Bioinformatics*, 21(3):390–392, 2005. doi:10.1093/BIOINFORMATICS/BTI020.

10 Markus Fleischauer and Sebastian Böcker. Bad clade deletion supertrees: a fast and accurate supertree algorithm. *Molecular biology and evolution*, 34(9):2408–2421, 2017.

11 Pawel Górecki and Jerzy Tiuryn. Dls-trees: a model of evolutionary scenarios. *Theoretical computer science*, 359(1-3):378–399, 2006. doi:10.1016/J.TCS.2006.05.019.

12 Dan Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21(1):19–28, 1991. doi:10.1002/NET.3230210104.

13 Dan Gusfield. *ReCombinatorics: the algorithmics of ancestral recombination graphs and explicit phylogenetic networks*. MIT press, 2014.

14 Dan Gusfield, Satish Eddhu, and Charles Langley. Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *Journal of bioinformatics and computational biology*, 2(01):173–213, 2004. doi:10.1142/S0219720004000521.

15 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

16 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM (JACM)*, 63(3):1–24, 2016. doi:10.1145/2925985.

17 Steven Kelk, Celine Scornavacca, and Leo Van Iersel. On the elusiveness of clusters. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(2):517–534, 2011. doi:10.1109/TCBB.2011.128.

18 Manuel Lafond, Krister M Swenson, and Nadia El-Mabrouk. An optimal reconciliation algorithm for gene trees with polytomies. In *International Workshop on Algorithms in Bioinformatics*, pages 106–122. Springer, 2012. doi:10.1007/978-3-642-33122-0_9.

19 Daniel Lemire. Ewahboolarray, 2026. Accessed: 2026-05-09. URL: https://github.com/lemire/EWAHBoolArray.

20 Alitzel López Sánchez and Manuel Lafond. Galled perfect transfer networks. In *RECOMB International Workshop on Comparative Genomics*, pages 24–43. Springer, 2024. doi:10.1007/978-3-031-58072-7_2.

21 Diego Mallo, Leonardo de Oliveira Martins, and David Posada. Simphy: phylogenomic simulation of gene, locus, and species trees. *Systematic biology*, 65(2):334–344, 2016.

22 Megan Owen. Bhvtreespace, 2026. Accessed: 2026-05-09. URL: https://github.com/megan-owen/BHVtreespace.

23 Megan Owen and J Scott Provan. A fast algorithm for computing geodesic distances in tree space. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(1):2–13, 2010. doi:10.1109/TCBB.2010.3.

24 Andy Purvis. A composite estimate of primate phylogeny. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 348(1326):405–421, 1995.

25 Charles Semple, Mike Steel, et al. *Phylogenetics*, volume 24. Oxford University Press on Demand, 2003.

26 Leo Van Iersel, Steven Kelk, Regula Rupp, and Daniel Huson. Phylogenetic networks do not need to be complex: using fewer reticulations to represent conflicting clusters. *Bioinformatics*, 26(12):i124–i131, 2010.

27 Mathias Weller. Listing conflicting triples in optimal time. In *16th Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, page 144, 2018.

28 Christian M Zmasek and Sean R Eddy. A simple algorithm to infer gene duplication and speciation events on a gene tree. *Bioinformatics*, 17(9):821–828, 2001. doi:10.1093/BIOINFORMATICS/17.9.821.

A Proofs for Section 3 (Computing incompatibility graphs)

▶ **Theorem 6.** *Algorithm 1 correctly computes the incompatibility graph $G_{T,T'}$ of two trees T, T' with the same leaves in time $O(n + d)$, where n is the number of leaves of the input trees, and d is the number of edges of $G_{T,T'}$.*

Proof. Let T, T' be the two input trees. We first consider the time complexity of the algorithm, then prove correctness afterwards.

Running time analysis. Let us first observe that for any two nodes $u, v \in V(T)$, we have $u \preceq_T v$ if and only if $lca_T(u, v) = v$. Thus constant time lca queries allow checking whether $u \preceq_T v$ in time $O(1)$. We can also check if $u \prec_T v$ by simply ensuring that $u \preceq_T v$ and $u \neq v$. The same holds for ancestry relations in T'.

Let us now consider the pre-processing steps of the algorithm. Pre-processing for $O(1)$ lca queries, constructing $\vec{\mu}$ and $\overleftarrow{\nu}$, and pre-processing for $O(1)$ $diff\_anc$ queries are already known to be doable in time $O(n)$. For the graph G, we assume that it is stored as an adjacency list. Specifically, we assume that each node in $V(T) \cup V(T')$ is assigned a unique integer between 1 and $|V(T) \cup V(T')|$, which is $O(n)$. To initialize G, we create an array of length $|V(T)|$, with the i-th element intended to contain the list of neighbors of the i-th vertex of T, with each element of the array consisting of an empty list initially. This initialization takes time $O(n)$, and we assume that inserting an element in the i-th list takes constant time. However, we do *not* assume that checking whether an element is present in the i-th list takes constant time, which is why we use the global array Q. Speaking of which, this array also take time $O(n)$ to initialize.

Let us next argue that any edge $\{v, x\}$ gets added at most once during the course of the whole algorithm. This is because edges incident to v are only added when we iterate on v in the main procedure. During that iteration, we always check that $Q[x] \neq v$ before adding $\{v, x\}$ to G, and if we do add the edge we set $Q[x] = v$ right after, which ensures that the edge cannot be added a second time later on. Thus we add every edge of the incompatibility graph once, and so the number of times that we go through the instruction "Add $\{v, x\}$ to $E(G)$" in either subfunction is exactly $|E(G)| = d$. In particular, this implies that in total, throughout the execution of the whole algorithm, we spend a total time of $O(d)$ inside the loop of line 16 in *check_between*, because each iteration adds an edge and each edge is added once (and the conditions and instructions of the loop are easily seen to be doable in constant time).

With this in mind, we now focus on the time needed to call $check\_between(v, G)$ for a node $v \in V(T)$. We make $|ch_T(v)|$ iterations of the main loop that iterates on the children of v. For one such child w, consider the loop "*while* $x \neq \perp$ and $\overleftarrow{\nu}(x) \preceq_T v$" on line 14. We claim that for any v and w, we spend time $O(1)$ in this loop. Before the first iteration, $x = \vec{\mu}(w)$, and so $\overleftarrow{\nu}(x)$ must be w or one of its ancestors (this is because $\vec{\mu}(w) = x$ implies that $C_{T'}(x)$ contains all of $C_T(w)$, and $\overleftarrow{\nu}(x)$ is thus an ancestor of all elements of $C_T(w)$).

This means that before entering the loop, either $\overleftarrow{\nu}(x)$ is w, or v, or a strict ancestor of v. If $\overleftarrow{\nu}(x) = w$, then in the next iteration, the value of $\overleftarrow{\nu}(x)$ must change by the definition of $diff\_anc$, and therefore after 0 or 1 iteration we know that $\overleftarrow{\nu}(x)$ is v or one of its strict ancestors (or $\perp$ and we exit the loop). Another iteration is required at this stage only if $\overleftarrow{\nu}(x) = v$. The next iteration after that ensures that $v \prec_T \overleftarrow{\nu}(x)$ or $\overleftarrow{\nu}(x) = \perp$, so we make at most two iterations of this loop, each taking $O(1)$ time by our pre-processing for fast $diff\_anc$ queries. Thus if we omit the time spent inside the loop on line 16, *check_between* takes constant time per child w of v and therefore a total time of $O(|ch_T(v)|)$ on input v (this includes the time we spend to check the conditions of the loop on line 16 without entering it). Overall, adding in the time $O(d)$ spent in the loop of line 16, we total time spent in *check_between* over all $v \in V(T)$ is

$$O(d + \sum_{v \in V(T)} |ch_T(v)|) = O(d + n),$$

which is because $\sum_{v \in V(T)} |ch_T(v)|$ is just the number of edges of T, and this number is $O(n)$.

Now let us consider the time spent in *check_incomp_child*. For $w \in V(T)$, we denote by $deg_G(w)$ the number of edges incident to w in G. Again, we make $|ch_T(v)|$ iterations of its main loop. For each child w of v, we first go through the loop on line 23, which iterates through the $deg_G(w)$ edges $\{w, x\}$ incident to w in G (note, the enumeration takes time $O(deg_G(w))$ since G is implemented as an adjacency list). For each such edge $\{w, x\}$, the check for $v \prec_T \overleftarrow{\nu}(x)$ and $Q_a[x] \neq v$, and if needed adding $\{v, x\}$ to $E(G)$ and putting $Q_a[x]$ all take $O(1)$ time. Therefore, the loop on line 23 takes time $O(deg_G(w))$ per child w. For input v, this results in a time of $O(|ch_T(v)| + \sum_{w \in ch_T(v)} deg_G(w))$, which overall across T is

$$O\left(\sum_{v \in V(T)} \left(|ch_T(v)| + \sum_{w \in ch_T(v)} deg_G(w)\right)\right) = O(n + d).$$

To see why, notice that $\sum_{v \in V(T)} |ch_T(v)|$ is the number of edges of T and thus $O(n)$, and in the sum $\sum_{v \in V(T)} \sum_{w \in ch_T(v)} deg_G(w)$, we have that for any $w \in V(T)$, the term $deg_G(w)$ is added at most once, which is when the first summation over v is iterating on the parent v of w.

By summing the $O(n)$ time spent in pre-processing, plus the $O(n + d)$ time spent in *check_between*, plus the $O(n + d)$ time spent in *check_incompat_child*, we get a running time of $O(n + d)$.

Correctness. Let us now argue correctness. We prove by induction on the postordering of $V(T)$ that for each $v \in V(T)$, the algorithm adds an edge $\{v, x\}$ if and only if v, x are incompatible. Note that this statement allows v to be a leaf. As a base case, suppose that v is a leaf of T. Then v is not incompatible with any node of T'. The functions *check_between* and *check_incomp_child* are not even called on input v, so the algorithm adds no edge incident to v in G, which is correct.

Now assume that v is not a leaf and assume by induction that in G, the edges incident to each child w of v are correct.

Suppose that the algorithm adds an edge $\{v, x\}$. Assume first that the edge is added in the call to *check_between* on input v, while iterating over some child w of v. That iteration starts with $x = \vec{\mu}(w)$. At this moment, Observation 3 implies that $C_{T'}(x) \cap C_T(w) \neq \emptyset$, which in turn implies $C_{T'}(x) \cap C_T(v) \neq \emptyset$. Then the loop on line 14 maintains that property, since x is updated to become one of its ancestors (unless we get $x = \perp$, but we exclude this

case since it would become impossible to add an edge in the other loop of line 16). The same property also holds across every iteration of loop on line 16, so every value of $x \neq \perp$ encountered in the iteration on w satisfies $C_{T'}(x) \cap C_T(v) \neq \emptyset$.

When we exit the loop on line 14, since we assume $x \neq \perp$ we get that $\overleftarrow{\nu}(x) \preceq_T v$ is false after the loop. Since $C_{T'}(x) \cap C_T(v)$ is non-empty, $\overleftarrow{\nu}(x)$ is an ancestor of some descendant of v and thus $\overleftarrow{\nu}(x), v$ cannot be incomparable. It follows that $v \prec_T \overleftarrow{\nu}(x)$. The latter property is maintained after every iteration of the loop on line 16, when x is updated to its parent, because $\overleftarrow{\nu}(x) \preceq_T \overleftarrow{\nu}(parent_{T'}(x))$ (as long as $x \neq \perp$). Then, if we add an edge on line 17, the condition $x \prec_{T'} \vec{\mu}(v)$ is verified explicitly. Thus if we reach that line, we have $C_{T'}(x) \cap C_T(v) \neq \emptyset$, $v \prec_T \overleftarrow{\nu}(x)$, and $x \prec_{T'} \vec{\mu}(v)$. These are the conditions of Lemma 4, and v, x are incompatible, as desired.

So assume that the algorithm adds $\{v, x\}$ in a call to *check_incomp_child*, while iterating over some child w of v. Thus v has a child w such that $\{w, x\} \in E(G)$, and so by induction w, x are incompatible, and $v \prec_T \overleftarrow{\nu}(x)$ if checked explicitly. By Lemma 5 v, x are incompatible and adding $\{v, x\}$ is correct. Therefore, all edges added by the algorithm are correct.

Conversely, we argue that if v, x' are incompatible, then the algorithm adds the edge $\{v, x'\}$ (we use x' here to distinguish it from the variable x used by the algorithm). Note for later reference that for any $x \in V(T')$, $\{v, x\}$ is added by the algorithm if and only $Q[x]$ is set to v at some point (because when entering *check_between* on input v, no previous iteration could have put the value of v in Q, and then $Q[x]$ is set to v precisely after adding $\{v, x\}$ in either subroutine).

Now, by Lemma 5, the incompatibility of v, x' implies that $v \prec_T \overleftarrow{\nu}(x')$ and v, x' satisfy one of the two conditions of the lemma. Suppose first that condition (a) is true, i.e., v has a child w such that $\vec{\mu}(w) \preceq_{T'} x' \prec_{T'} \vec{\mu}(v)$. We claim that $\{v, x'\}$ gets added by *check_between*. The main difficulty of the proof is to argue that the $Q[x] \neq v$ condition does not make us miss edges.

Let $w_1, w_2, \ldots, w_k$ be the children of v, in the order that the function *check_between* iterates on them. Note that for $i = 1, \ldots, k$, when iterating on w_i, all the values of x encountered in *check_between* are ancestors of $\vec{\mu}(w_i)$. Thus after iterating on w_i, having $Q[x] = v$ is only possible if x is an ancestor of at least one of $w_1, \ldots, w_i$. Now let j be the minimum index such that x' is an ancestor of $\vec{\mu}(w_j)$ (such a j exists since $\vec{\mu}(w) \preceq_{T'} x'$ for some w). Recall that by Lemma 5, we know that $v \prec_T \overleftarrow{\nu}(x')$. When iterating on w_j, the loop of $diff\_anc$ reassignments on line 14 starts with $x = \vec{\mu}(w_j)$ and will stop at some node x that descends from x' (with equality possible). Since $x \preceq_{T'} x'$, none of $w_1, \ldots, w_{j-1}$ has x as an ancestor (otherwise it would have x' as an ancestor), and therefore $Q[x] \neq v$. The same is true for any ancestor of x. Moreover, we have $x' \prec_{T'} \vec{\mu}(v)$ by Lemma 4, and thus $x \prec_{T'} \vec{\mu}(v)$ as well, and the same holds for every node on the path between x and x'. It follows that when iterating on w_j, the algorithm will enter the loop of line 16 for x and all of its ancestors up to x', and therefore $\{v, x'\}$ will correctly get added to $E(G)$.

Suppose instead that condition (b) is satisfied, i.e., v has a child w such that w, x' are incompatible. If $\{v, x'\}$ gets added by *check_between*, we are done, so assume otherwise. Consider the call to *check_incomp_child* on input v, when we iterate on w. The edge $\{w, x\}$ is present by induction and the algorithm will enter the loop of line 23 on edge $\{w, x\}$. If $Q[x] = v$, the edge was added at an earlier moment and we are done. Since v, x' are incompatible, $v \prec_T \overleftarrow{\nu}(x')$ by Lemma 5. We therefore enter the "if" statement and add $\{v, x'\}$, as desired.

We have thus shown that v, x are incompatible if and only if the algorithm adds $\{v, x\}$ to G, which concludes the correctness proof. ◀

▶ **Proposition 8.** *Let U, U' be two unrooted trees with $L(U) = L(U') = \{1, 2, \ldots, n\}$ with $n \geq 3$, and let T, T' be the rooted trees obtained from U and U', respectively, by assigning the neighbor of leaf 1 as the root in both trees.*

Let $uv \in E(U)$ and $xy \in E(U')$, where u is the parent of v in T and x is the parent of y in T'. Then $S_U(uv)$ and $S_{U'}(xy)$ are compatible splits if and only if $C_T(v)$ and $C_{T'}(y)$ are compatible clusters.

Proof. First note that the condition $n \geq 3$ ensures that the neighbor of leaf 1 in both U and U' is not a leaf, and thus that the rootings are valid (in particular, since no node of U and U' has degree 2, the internal nodes of T and T' all have at least two children).

Let $A|B$ be the split of uv in U, with $B = C_T(v)$, and let $C|D$ be the split of xy in U', with $D = C_{T'}(y)$. If B contains 1, then $B = \{1\}$ because 1 is a child of the root. In that case, one of $B \cap C$ or $B \cap D$ is empty because one of C or D does not contain 1, and the two splits are compatible. Moreover, B and D are compatible clusters because a cluster with a single element is compatible with any cluster. Thus our statement holds when B contains 1. Using the same arguments, it also holds if D contains 1.

Assume from now on that B and D both do not contain 1. Suppose that $A|B$ and $C|D$ are compatible splits. Then by Proposition 7, one of the intersections $A \cap C, A \cap D, B \cap C, B \cap D$ is empty. Note that $A \cap C \neq \emptyset$ since both A and C contain the leaf 1, because $A = L(U) \setminus B$ and $C = L(U') \setminus D$. If $A \cap D = \emptyset$, then every element of D must be in B, and B, D are compatible clusters by Proposition 1 since $D \subseteq B$. If $C \cap B = \emptyset$, then likewise $B \subseteq D$ and B, D are compatible clusters. Finally, if $B \cap D = \emptyset$, then B, D are compatible clusters directly by Proposition 1. In all cases, $A|B, C|D$ being compatible implies that $B = C_T(v)$ and $D = C_{T'}(y)$ are compatible.

Conversely, suppose that B, D are compatible clusters. By Proposition 1, either $B \cap D = \emptyset$, or $B \subseteq D$, or $D \subseteq B$. If $B \cap D = \emptyset$, then $A|B, C|D$ are compatible splits by Proposition 7. If $B \subseteq D$, then $B \cap C = \emptyset$ and the two splits are compatible, and if $D \subseteq B$ then $A \cap D = \emptyset$ and the two splits are again compatible. This concludes the proof. ◀

▶ **Proposition 9.** *Let T, T' be two trees and let $\tilde{T}, \tilde{T}'$ be defined as above. Then $|V(G_{\tilde{T},\tilde{T}'})| \leq |V(G_{T,T'})| + 2$. Moreover, ux is an edge of $G_{T,T'}$ if and only if ux is an edge of $G_{\tilde{T},\tilde{T}'}$.*

Proof. The bound on the number of vertices is obvious because $\tilde{T}$ has at most one more internal node than T, and $\tilde{T}'$ has at most one more internal node than T'. For the other statement, suppose first that $u \in I(T)$ and $x \in I(T')$. Notice that $C_T(u) = C_{\tilde{T}}(u)$ and $C_{T'}(x) = C_{\tilde{T}'}(x)$. Therefore for such pairs of vertices, $ux \in E(G_{T,T'})$ if and only if $E(G_{\tilde{T},\tilde{T}'})$.

Now suppose that $u \in I(\tilde{T}) \setminus I(T)$. Then it must be that $u = \rho(\tilde{T})$. By construction, we have $C_{\tilde{T}}(u) = L(T) \cup L(T')$. Also, for any $x \in I(\tilde{T}')$, we have $C_{\tilde{T}'}(x) \subseteq L(T) \cup L(T') = C_{\tilde{T}}(u)$. Thus u and x are compatible. Therefore, u is an isolated vertex in $G_{\tilde{T},\tilde{T}'}$. Using a symmetric argument, we get that any $x \in I(\tilde{T}') \setminus I(T')$ must be the root of $\tilde{T}'$ and an isolated vertex in $G_{\tilde{T},\tilde{T}'}$. It follows that the edge sets of $G_{T,T'}$ and $G_{\tilde{T},\tilde{T}'}$ are identical. ◀