Algorithmica (2021) 83:1786–1828
https://doi.org/10.1007/s00453-021-00802-1

Computing the Rooted Triplet Distance Between Phylogenetic Networks

Jesper Jansson[1] · Konstantinos Mampentzidis[2] · Ramesh Rajaby[3] · Wing-Kin Sung[3,4]

Received: 29 June 2020 / Accepted: 13 January 2021 / Published online: 16 March 2021

Abstract

The *rooted triplet distance* measures the structural dissimilarity of two phylogenetic trees or phylogenetic networks by counting the number of rooted phylogenetic trees with exactly three leaf labels (called *rooted triplets*, or *triplets* for short) that occur as embedded subtrees in one, but not both, of them. Suppose that $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$ are phylogenetic networks over a common leaf label set of size n, that N_i has level k_i and maximum in-degree d_i for $i \in \{1, 2\}$, and that the networks' out-degrees are unbounded. Write $N = \max(|V_1|, |V_2|)$, $M = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$, and $d = \max(d_1, d_2)$. Previous work has shown how to compute the rooted triplet distance between N_1 and N_2 in $O(n \log n)$ time in the special case $k \leq 1$. For $k > 1$, no efficient algorithms are known; applying a classic method from 1980 by Fortune *et al.* in a direct way leads to a running time of $\Omega(N^6 n^3)$ and the only existing non-trivial algorithm imposes restrictions on the networks' in- and out-degrees (in particular, it does not work when non-binary vertices are allowed). In this article, we develop two new algorithms with no such restrictions. Their running times are $O(N^2 M + n^3)$ and $O(M + Nk^2 d^2 + n^3)$, respectively. We also provide implementations of our algorithms, evaluate their performance on simulated and real datasets, and make some observations on the limitations of the current definition of the rooted triplet distance in practice. Our prototype implementations have been packaged into the first publicly available software for computing the rooted triplet distance between unrestricted networks of arbitrary levels.

Keywords Phylogenetic network comparison · Rooted triplet distance · Fan graph · Resolved graph · Block tree · Contracted block network · Implementation

---

A preliminary version of this article appeared in *Proceedings of the Thirtieth International Workshop on Combinatorial Algorithms* (IWOCA 2019), Lecture Notes in Computer Science, Vol. 11638, pp. 290–303, Springer Nature Switzerland AG, 2019.

✉ Jesper Jansson
jesper.jansson@polyu.edu.hk

Extended author information available on the last page of the article

1 Introduction

1.1 Background

Phylogenetic trees are commonly used in biology to represent evolutionary relationships, with the leaves corresponding to species that exist today and internal vertices to ancestor species that existed in the past [1]. When studying the evolution of a fixed set of species, different available data and tree reconstruction methods can lead to trees that look structurally different. Quantifying this difference is essential to make better evolutionary inferences, which has led to the proposal of several phylogenetic tree distance measures in the literature. For example, to evaluate the accuracy of a tree reconstruction method $\mathcal{M}$, one can perform the following steps a number of times [2]: First generate a random phylogenetic tree T and let a sequence evolve down the edges of T according to some chosen model of sequence evolution, then apply the method $\mathcal{M}$ to reconstruct a tree T', and finally measure the distance between T and T'. Some phylogenetic tree distance measures that are based on counting how many times certain features differ in the two trees are the Robinson-Foulds distance [3], the rooted triplet distance [4] for rooted trees, and the unrooted quartet distance [5] for unrooted trees. Other distance measures are the nearest-neighbor interchange distance (introduced independently in [6] and [7]), the path-length-difference distance [8], the subtree prune-and-regraft distance [9], the maximum agreement subtree [10], and the subtree moving tree edit distance [11].

The rooted phylogenetic network model is an extension of the rooted phylogenetic tree model that allows internal vertices to have more than just one parent [12]. Such networks can describe more complex evolutionary relationships involving reticulation events such as horizontal gene transfer and hybridization. As in the case of phylogenetic trees, it is useful to have distance measures for comparing phylogenetic networks. Therefore, in this article, we study a natural generalization [13] of the rooted triplet distance for phylogenetic trees to rooted phylogenetic networks and present two new algorithms for computing it.

1.2 Problem Definitions

For any vertex u in a directed acyclic graph, let $in(u)$ and $out(u)$ be the in-degree and out-degree of u. The vertex u is called a *leaf* if $out(u) = 0$, and an *internal vertex* if $out(u) \geq 1$. Formally, a *rooted phylogenetic network* N' is a directed acyclic graph with one vertex of in-degree 0 (from here on called the *root of* N' and denoted by $r(N')$), distinctly labeled leaves, and no vertices with both in-degree 1 and out-degree 1. A vertex u in N' is called a *reticulation vertex* if $in(u) \geq 2$ holds. If N' has no reticulation vertices, i.e., if all vertices in N' have in-degree at most 1, then N' is a *rooted phylogenetic tree*. Below, when referring to a "tree", we imply a "rooted phylogenetic tree", and when referring to a "network", we imply a "rooted phylogenetic network".

For the rest of this subsection, suppose that N' is a network. A directed edge from a vertex u to a vertex v in N' is denoted by $u \rightarrow v$. A path from u to v in N' is denoted by $u \rightsquigarrow v$. Let the *height* of u, written as $h(u)$, be the number of edges in a longest path from u to a leaf in N'. By definition, if v is a parent of u in N' then $h(v) > h(u)$. We will use $\mathcal{L}(N')$ to refer to both the set of leaves in N' as well as to the set of leaf labels in N' since they are in one-to-one correspondence.

The *level of a network* was introduced by Choy *et al.* [14] as a parameter to measure the treelikeness of a network, with the special case of a level-0 network being a tree and a level-1 network a so-called *galled tree* [15] in which all underlying cycles are disjoint. The level is defined as follows. Let $U(N')$ be the undirected graph created by replacing every directed edge in N' with an undirected edge. An undirected, connected graph H is called *biconnected* if it has no vertex whose removal makes H disconnected. A maximal subgraph of $U(N')$ that is biconnected is called a *biconnected component of* $U(N')$. (Observe that the biconnected components of $U(N')$ are edge-disjoint but not necessarily vertex-disjoint.) For any biconnected component of $U(N')$, its corresponding subgraph in N' will be referred to as a *block of* N'. We say that N' is a *level-k network*, or equivalently N' *has level k*, if every block of N' contains at most k reticulation vertices. Figure 1 shows a level-2 and a level-3 network.

If B is a block of N' consisting of more than two vertices and one edge and B contains at most one vertex that has one or more outgoing edges to vertices not belonging to B then B is called *uninformative*. See Fig. 2 for an illustration.

Next, a *rooted triplet* τ is a tree with three leaves. If it is binary we say that τ is a *rooted resolved triplet*, and if it is non-binary we say that τ is a *rooted fan triplet*. We say that the rooted fan triplet $x|y|z$ is *consistent with* N' if and only if there exists a vertex u in N' such that there are three directed paths of non-zero length from u to x, from u to y, and from u to z that are vertex-disjoint except for in u. Similarly, we say that the rooted resolved triplet $xy|z$ is *consistent with* N' if and only if N' contains two vertices u and v such that there are four directed paths of non-zero length from u to v, from v to x, from v to y, and from u to z that are vertex-disjoint except for in u and v, and furthermore, the path from u to z does not pass through v. See Fig. 1 for an example. From here on, by "disjoint paths"

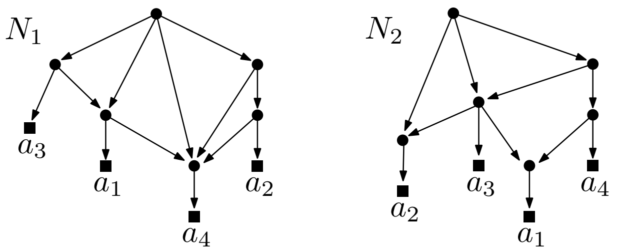


Fig. 1 N_1 is a level-2 network and N_2 is a level-3 network with $\mathcal{L}(N_1) = \mathcal{L}(N_2) = \{a_1, a_2, a_3, a_4\}$. In this example, $D(N_1, N_2) = 6$. Some shared triplets are: $a_1|a_2|a_4$, $a_3a_4|a_2$, $a_1a_3|a_2$. Some triplets consistent with only one network are: $a_1|a_3|a_4$, $a_2a_3|a_1$

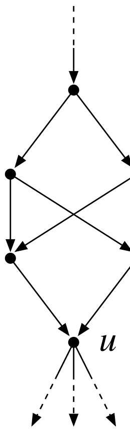


Fig. 2 The block drawn with solid edges is an uninformative block because it only has one vertex u with outgoing edges to vertices not in the block

we imply “vertex-disjoint paths of non-zero length”. Moreover, when referring to a “triplet”, we imply a “rooted triplet”.

Given two networks $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$ built on the same leaf label set Λ, the *rooted triplet distance* $D(N_1, N_2)$, or *triplet distance* for short, is the number of triplets over Λ that are consistent with exactly one of N_1 and N_2. Let $S(N_1, N_2)$ be the total number of *shared triplets*, i.e., triplets that are consistent with both N_1 and N_2. Then:

$$D(N_1, N_2) = S(N_1, N_1) + S(N_2, N_2) - 2S(N_1, N_2) \tag{1.1}$$

Note that a shared triplet contributes a +1 to $S(N_1, N_1)$, $S(N_2, N_2)$, and $S(N_1, N_2)$, e.g., the triplet $a_1|a_2|a_4$ in Fig. 1. On the other hand, a triplet from either network that is not shared contributes a +1 to either $S(N_1, N_1)$ or $S(N_2, N_2)$, and a 0 to $S(N_1, N_2)$. As an example, $a_1|a_3|a_4$ in Fig. 1 contributes a +1 to $S(N_1, N_1)$ and a 0 to $S(N_2, N_2)$ and $S(N_1, N_2)$.

Let $S_r(N_1, N_2)$ and $S_f(N_1, N_2)$ be the number of shared resolved and shared fan triplets, respectively. Then $S(N_1, N_2) = S_r(N_1, N_2) + S_f(N_1, N_2)$, which implies that $D(N_1, N_2)$ can be obtained by considering shared resolved triplets and shared fan triplets separately.

The rest of this article is focused on how to compute $D(N_1, N_2)$ efficiently. We shall use the following notation to express the time complexities of various algorithms. For $i \in \{1, 2\}$, the network N_i has vertex set V_i and edge set E_i. The level of N_i is k_i and the maximum in-degree taken over all vertices in N_i is d_i. We assume that the two given networks N_1 and N_2 have the same leaf label set Λ, and write $n = |\Lambda|$, $N = \max(|V_1|, |V_2|)$, $M = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$, and $d = \max(d_1, d_2)$.

To simplify the descriptions of the algorithms, we will also assume that: (i) there is no vertex u satisfying both $in(u) > 1$ and $out(u) = 0$, i.e., all leaves have in-degree at most 1; and (ii) there are no uninformative blocks in N_1 and N_2. Assumption (i) is justified because every leaf u with in-degree larger than 1 can be replaced by an internal vertex to which a leaf with the same leaf label as u is attached, and the resulting network will be consistent with exactly the same triplets as before. Assumption (ii) is justified because first each uninformative block can be replaced by an edge, and then each vertex with in-degree 1 and out-degree 1 can be eliminated by contracting its outgoing edge; the resulting network will be consistent with the same triplets as the original network. If necessary, checking the input networks N_1 and N_2 and modifying them to ensure that they comply with (i) and (ii) before running the algorithms takes $O(M)$ time, e.g., by using Hopcroft-Tarjan's algorithm [16] to identify the biconnected components of $U(N_1)$ and $U(N_2)$.

1.3 Previous Work

The rooted triplet distance was introduced by Dobson [4] in 1975 for trees, and generalized to networks by Gambette and Huber [13] in 2012. See also [17, Section 3.2] for a short discussion about the definition.

Table 1 lists the time complexities of some previously known algorithms and our new ones for computing $D(N_1, N_2)$. When $k = 0$, both N_1 and N_2 are trees. This case has been extensively studied in the literature [4, 18–24], with the most efficient algorithms in theory and practice [19, 20, 24] running in $\mathrm{O}(n \log n)$ time. For $k = 1$, an $\mathrm{O}(n^{2.687})$-time algorithm based on counting 3-cycles in an auxiliary graph was given in [17], and a faster, $O(n \log n)$-time algorithm that transforms the input to a constant number of instances with $k = 0$ was given in [25]. All of these algorithms allow the vertices in the input networks to have arbitrary

Table 1 Previous and new results for computing $D(N_1, N_2)$, where N_1 and N_2 are two phylogenetic networks built on the same leaf label set Λ

Year	Reference	k	In- and out-degrees	Time complexity
1980	Fortune *et al.* [26]	Arbitrary	Arbitrary	$\Omega(N^6 n^3)$
2010	Byrka *et al.* [27]	Arbitrary	Binary	$\mathrm{O}(N + Nk^2 + n^3)$
2013	Brodal *et al.* [19]	0	Arbitrary	$\mathrm{O}(n \log n)$
2019	Jansson *et al.* [25]	1	Arbitrary	$\mathrm{O}(n \log n)$
2020	New	Arbitrary	Arbitrary	$\mathrm{O}(N^2 M + n^3)$
2020	New	Arbitrary	Arbitrary	$\mathrm{O}(M + Nk^2 d^2 + n^3)$

Notation: $n = |\Lambda|$ is the number of leaf labels, $N = \max(|V_1|, |V_2|)$ is the maximum number of vertices, $M = \max(|E_1|, |E_2|)$ is the maximum number of edges, $k = \max(k_1, k_2)$ is the maximum level, and $d = \max(d_1, d_2)$ is the maximum in-degree of the two networks degrees. Moreover, software implementations of the fast algorithms for $k = 0$ and $k = 1$ are available [20, 23–25].

For $k > 1$, much less is known. In a special "binary degree" case where the phylogenetic networks' roots have out-degree 2 and all other internal vertices have either in-degree 2 and out-degree 1, or in-degree 1 and out-degree 2, one can adapt a technique developed by Byrka *et al.* [27] for a problem related to finding a network consistent with as many resolved triplets as possible from a given set. They showed how to preprocess any fixed network $N' = (V, E)$ satisfying the binary degree constraints so that checking if a resolved triplet is consistent with N' can be done efficiently. Below, we shall refer to this preprocessing as constructing a data structure $\mathcal{D}$ such that $\mathcal{D}$ can be used to determine whether any specified resolved triplet is consistent with N' in O(1) time. The proof of Lemma 2 in [27] showed how to build $\mathcal{D}$ in $O(|V|^3)$ time. According to Remark 1 in [27], this can be further improved to $O(|V| + |V|k^2)$, where k is the level of N'. The rooted triplet distance can thus be computed in $O(N + Nk^2 + n^3)$ time in a straightforward way when N_1 and N_2 obey the special binary degree constraints. A limitation of $\mathcal{D}$ is that it can only support consistency queries for resolved triplets, while a network with no restrictions on the vertices' degrees may also contain fan triplets.

In the general case, when N_1 and N_2 have unbounded degrees and unbounded levels, it is possible to compute $D(N_1, N_2)$ by iterating over all $4\binom{n}{3}$ triplets, and for each such triplet applying the classic directed acyclic graph pattern matching algorithm in [26] to determine its consistency with N_1 and N_2. However, this leads to a time complexity of $\Omega(N^6 n^3)$. To see this, let P in Theorem 3 in [26] be a resolved triplet and G a phylogenetic network N_i with $|V_i|$ vertices. P has two internal nodes and four edges, so the algorithm will consider $\binom{|V_i|}{2}$ ways of mapping the two internal nodes of P to vertices in N_i, and for each one, construct a configuration graph G' with $\Omega((|V_i| + 1)^4)$ vertices and look for a path in G'. Hence, the algorithm will use $\Omega(|V_i|^6)$ time for each resolved triplet to check if it occurs in N_i, i.e., $\Omega(N^6 n^3)$ time in total.

1.4 New Results

Here, we develop two algorithms that significantly improve upon the time complexity of computing the rooted triplet distance in the general, unbounded case. The running time of our first algorithm is $O(N^2 M + n^3)$. One key insight is that a technique of Perl and Shiloach for identifying two disjoint paths between two pairs of vertices in a directed acyclic graph [28] can be extended to check if a fan triplet or a resolved triplet is embedded in a phylogenetic network, leading to the useful concepts of a *fan graph* and a *resolved graph*. Our second algorithm then augments these ideas with so-called *block trees* and *contracted block networks* to obtain a running time of $O(M + Nk^2 d^2 + n^3)$. Neither algorithm has a strictly better time complexity than the other one for all possible inputs. In the special case where N_1 and N_2 follow the binary degree constraints of Byrka *et al.* [27], the time complexity reduces to $\mathrm{O}(N + Nk^2 + n^3)$, matching the bound in [27].

We also provide implementations of our algorithms, evaluate their performance on simulated and real datasets, and make some observations on the limitations of the current definition of the rooted triplet distance in practice. Our prototype implementations have been packaged into the first publicly available software for computing the triplet distance between two unrestricted networks of arbitrary levels.

1.5 Organization of the Article

Section 2 describes our first new algorithm and Sect. 3 the second one. Section 4 presents an implementation of both our algorithms and experiments illustrating their practical performance. Finally, Sect. 5 gives some concluding remarks.

2 A First Approach

This section presents an algorithm that computes $D(N_1, N_2)$ in $\mathrm{O}(N^2M + n^3)$ time.

Overview. The algorithm consists of a preprocessing step and a triplet distance computation step. For the preprocessing step, we extend a technique introduced by Perl and Shiloach [28] to construct suitably defined auxiliary graphs that compactly encode disjoint paths within N_1 and N_2. Two graphs, the *fan graph* and *resolved graph*, are created that enable us to check the consistency of any fan triplet and any resolved triplet, respectively, with N_1 and N_2 in $\mathrm{O}(1)$ time. In the triplet distance computation step, we compute $D(N_1, N_2)$ by iterating over all possible $4\binom{n}{3}$ triplets and using the fan and resolved graphs to check the consistency of each triplet with N_1 and N_2 efficiently.

2.1 Preprocessing

Let $G = (V, E)$ be a directed acyclic graph and s_1, t_1, s_2, and t_2 four vertices in G. Perl and Shiloach [28] gave an algorithm that can find two vertex-disjoint paths, one from s_1 to t_1 and one from s_2 to t_2, in $O(|V||E|)$ time or determine that no such pair of paths exists. They achieve this by creating a directed graph $G' = (V', E')$ in $\mathrm{O}(|V||E|)$ time, with the property that the existence of such a pair of vertex-disjoint paths in G is equivalent to the existence of a directed path from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G', where $\langle s_1, s_2\rangle$ and $\langle t_1, t_2\rangle$ are vertices in G'. A fan triplet or resolved triplet involves more than two vertex-disjoint paths, and below we show how to extend the technique by Perl and Shiloach [28] to determine if a given network has the necessary vertex-disjoint paths that would imply the consistency of a given triplet with the network.

2.1.1 The Fan Graph

For any network $N_i = (V_i, E_i)$, let its *fan graph* $N_i^f = (V_i^f, E_i^f)$ be a graph such that $V_i^f = \{s\} \cup \{(u, v, w) \mid u, v, w \in V_i,\ u \neq v,\ u \neq w,\ v \neq w\}$ and E_i^f includes the following directed edges:

1. $\{(u_1, v_1, w_1) \rightarrow (u_2, v_1, w_1) \mid u_1 \rightarrow u_2 \in E_i,\ h(u_1) \geq \max(h(v_1), h(w_1))\}$
2. $\{(u_1, v_1, w_1) \rightarrow (u_1, v_2, w_1) \mid v_1 \rightarrow v_2 \in E_i,\ h(v_1) \geq \max(h(u_1), h(w_1))\}$
3. $\{(u_1, v_1, w_1) \rightarrow (u_1, v_1, w_2) \mid w_1 \rightarrow w_2 \in E_i,\ h(w_1) \geq \max(h(u_1), h(v_1))\}$
4. $\{s \rightarrow (u, v, w) \mid u \rightarrow v \in E_i,\ u \rightarrow w \in E_i\}$

Every 3-tuple of vertices from N_i with distinct entries is represented by a vertex in N_i^f. Refer to Fig. 3 for an example. Note that N_i^f contains $O(|V_i|^3)$ vertices and $O(|V_i|^2|E_i|)$ edges, and can be constructed in $O(|V_i|^2|E_i|)$ time. It also has the property described in the following lemma, which generalizes Theorem 3.1 in [28].

Lemma 2.1 *Consider a network N_i and its fan graph $N_i^f = (V_i^f, E_i^f)$. For any three different leaves x, y, and z in N_i, vertex s can reach vertex (x, y, z) in N_i^f if and only if the fan triplet $x|y|z$ is consistent with N_i.*

Proof ($\leftarrow$) Let $x|y|z$ be any fan triplet consistent with N_i. By definition, there exists an internal vertex q in N_i and three disjoint paths (except for in q), one from q to x, one from q to y, and one from q to z. Denote these paths by $(q, x_0, x_1, \dots, x_a)$, $(q, y_0, y_1, \dots, y_b)$, and $(q, z_0, z_1, \dots, z_c)$, where $x_a = x$, $y_b = y$, and $z_c = z$. Then N_i^f also contains the following three paths:

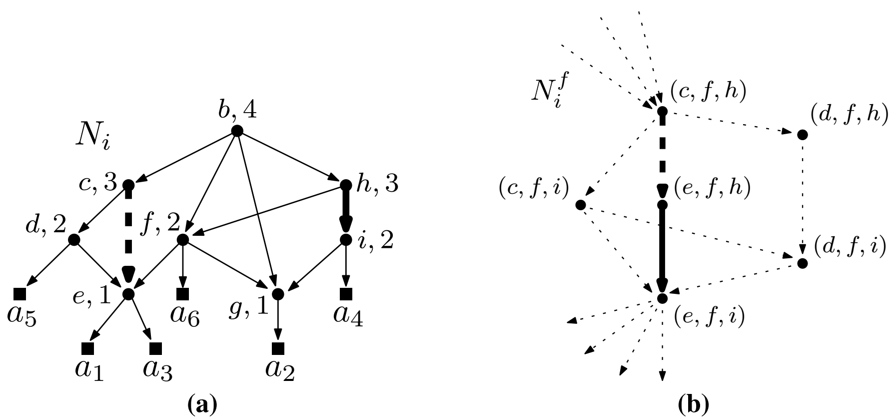


Fig. 3 Illustrating the fan graph. **a** An example network N_i. Every internal vertex is labeled by a letter and its height. **b** Consider the triplet $a_3|a_6|a_4$. Lemma 2.1 implies that it is consistent with N_i because there is a path $(s, (b,f,h), (c,f,h), (e,f,h), (e,f,i), (e,a_6,i), (e,a_6,a_4), (a_3,a_6,a_4))$ in the fan graph N_i^f. A small part of N_i^f is drawn here, with the two directed edges $(c,f,h) \rightarrow (e,f,h)$ and $(e,f,h) \rightarrow (e,f,i)$ in the path from s to (a_3, a_6, a_4) indicated

- $(s, (q, y_0, z_0))$: This can be seen from $q \to y_0 \in E_i$ and $q \to z_0 \in E_i$.
- $((q, y_0, z_0), (x_0, y_0, z_0))$: This follows from the fact that $q \to x_0 \in E_i$ and $h(q) > h(y_0), h(z_0)$.
- $((x_0, y_0, z_0), \ldots, (x_a, y_b, z_c))$: This is because $h(x_0) > h(x_1) > \ldots > h(x_a)$, $h(y_0) > h(y_1) > \ldots > h(y_b)$, and $h(z_0) > h(z_1) > \ldots > h(z_c)$ hold, and $(x_0, \ldots, x_a)$, $(y_0, \ldots, y_b)$, and $(z_0, \ldots, z_c)$ are paths in N_i.

By concatenating the three paths above, we get a path in N_i^f from s to (x, y, z).

($\rightarrow$) Because s can reach (x, y, z) in N_i^f, there exists a path P in N_i^f of the form $P = (s, (x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_t, y_t, z_t))$, where $x_t = x$, $y_t = y$, and $z_t = z$. Let $S_1 = (x_1, x_2, \ldots, x_t)$, $S_2 = (y_1, y_2, \ldots, y_t)$, and $S_3 = (z_1, z_2, \ldots, z_t)$, where $x_t = x$, $y_t = y$, and $z_t = z$, be three sequences of vertices from N_i obtained from P.

We prove by induction that the three paths obtained by following the sequences S_1, S_2, and S_3 are disjoint paths in N_i. Consider any $j \in \{1, 2, \ldots, t\}$. When $j = t$, all three vertices x_t, y_t, and z_t are different according to the definition of V_i^f. For $j < t$, by the inductive hypothesis we have that $(x_{j+1}, \ldots, x_t)$, $(y_{j+1}, \ldots, y_t)$ and $(z_{j+1}, \ldots, z_t)$ yield disjoint paths. In addition, by the definition of the fan graph N_i^f, for every $j \in \{1, 2, \ldots, t-1\}$, one of the following three cases holds: (1) $x_j \neq x_{j+1}$ only, (2) $y_j \neq y_{j+1}$ only, and (3) $z_j \neq z_{j+1}$ only. In case (1), note that $y_j = y_{j+1}$ and $z_j = z_{j+1}$, which means that $(x_{j+1}, \ldots, x_t)$, $(y_j, \ldots, y_t)$ and $(z_j, \ldots, z_t)$ yield disjoint paths. We now show that x_j cannot appear in any of these three paths. It holds that $h(x_j) \geq \max(h(y_j), h(z_j))$, so for $\mu \geq j+1$ and $y_\mu \neq y_j$, we have $h(x_j) > h(y_\mu)$. Similarly, for $\mu \geq j+1$ and $z_\mu \neq z_j$, we have $h(x_j) > h(z_\mu)$. Together with the fact that x_j , y_j, and z_j are different according to the definition of N_i^f, we deduce that the three paths obtained from $(x_j, \ldots, x_t)$, $(y_j, \ldots, y_t)$, and $(z_j, \ldots, z_t)$ are disjoint. Cases (2) and (3) can be argued in the same way. Thus, following S_1, S_2, and S_3 yields three disjoint paths.

Finally, since P contains a directed edge from s to (x_1, y_1, z_1), N_i contains an edge from x_1 to y_1 and an edge from x_1 to z_1. Therefore, the three paths in N_i that start at the internal vertex x_1 and then follow the sequences S_1, S_2, and S_3, respectively, are disjoint paths (except for in x_1) to x, y, and z. By definition, $x|y|z$ is consistent with N_i. $\square$

Corollary 2.2 *Let N_i be a given network and r' a dummy leaf attached to $r(N_i)$. For any two different leaves x and y in N_i that are not r', there are two paths from $r(N_i)$ to x and y that are disjoint, except for in $r(N_i)$, if and only if s can reach (r', x, y) in N_i^f.*

2.1.2 The Resolved Graph

For any network N_i, let its *resolved graph* $N_i^r = (V_i^r, E_i^r)$ be a graph such that $V_i^r = \{s\} \cup \{(u, v) \mid u, v \in V_i, u \neq v\} \cup \{(u, v, w) \mid u, v, w \in V_i, u \neq v, u \neq w, v \neq w\}$ and E_i^r includes the following directed edges:

1. $\{s \to (u, v) \mid u \to v \in E_i\}$
2. $\{(u_1, v_1) \to (u_2, v_1) \mid u_1 \to u_2 \in E_i,\ h(u_1) \geq h(v_1)\}$
3. $\{(u_1, v_1) \to (u_1, v_2) \mid v_1 \to v_2 \in E_i,\ h(v_1) \geq h(u_1)\}$
4. $\{(u, v) \to (u, v, w) \mid v \to w \in E_i,\ h(v) \geq h(u)\}$
5. $\{(u_1, v_1, w_1) \to (u_2, v_1, w_1) \mid u_1 \to u_2 \in E_i,\ h(u_1) \geq \max(h(v_1), h(w_1))\}$
6. $\{(u_1, v_1, w_1) \to (u_1, v_2, w_1) \mid v_1 \to v_2 \in E_i,\ h(v_1) \geq \max(h(u_1), h(w_1))\}$
7. $\{(u_1, v_1, w_1) \to (u_1, v_1, w_2) \mid w_1 \to w_2 \in E_i,\ h(w_1) \geq \max(h(u_1), h(v_1))\}$

Note that N_i^r contains $O(|V_i|^3)$ vertices and $O(|V_i|^2|E_i|)$ edges, can be constructed in $O(|V_i|^2|E_i|)$ time, and has the property described in the following lemma:

Lemma 2.3 *Consider a network N_i and its resolved graph $N_i^r = (V_i^r, E_i^r)$. For any three different leaves x, y, and z in N_i, vertex s can reach vertex (x, y, z) in N_i^r if and only if the resolved triplet $yz|x$ is consistent with N_i.*

Proof ($\leftarrow$) If $yz|x$ is consistent with N_i then N_i contains three paths of the following form: (1) $(x_0, x_1, \dots, x_a)$; (2) $(x_0, y_1, \dots, y_j, y_{j+1}, \dots, y_b)$; and (3) $(y_j, z_1, \dots, z_c)$; such that the three paths are vertex-disjoint except for in x_0 and y_j, the first path does not pass through y_j, and it holds that $x_a = x$, $y_b = y$, and $z_c = z$.

Let x_μ be a vertex on the first path satisfying $h(x_{\mu-1}) > h(y_j) \geq h(x_\mu)$. Then $(s, (x_0, y_1), \dots, (x_\mu, y_j), (x_\mu, y_j, z_1), \dots, (x_a, y_b, z_c))$ is a path in N_i^r.

($\rightarrow$) If there is a path from s to (x, y, z) in N_i^r, it must be of the form $(s, (x_1, y_1), (x_2, y_2), \dots, (x_q, y_q), (x_{q+1}, y_{q+1}, z_{q+1}), \dots, (x_t, y_t, z_t))$, with $x_t = x$, $y_t = y$, and $z_t = z$. By the definitions, we have $x_1 \to y_1 \in E_i$, $x_q = x_{q+1}$, $y_q = y_{q+1}$, and $y_q \to z_{q+1} \in E_i$. Define three sequences of vertices from N_i as follows: $S_1 = (x_1, x_2, \dots, x_t)$, $S_2 = (y_1, y_2, \dots, y_t)$, and $S_3 = (z_{q+1}, z_{q+2}, \dots, z_t)$.

We claim that following the sequences S_1, S_2, and S_3 yields three disjoint paths in N_i. (This claim is shown below.) The claim and the fact that N_i^r contains an edge from s to (x_1, y_1) and an edge from (x_q, y_q) to $(x_{q+1}, y_{q+1}, z_{q+1})$ then imply that N_i contains a path from x_1 to x, a path from x_1 to y_q, a path from y_q to y, and a path from y_q to z that make $yz|x$ consistent with N_i.

To prove the claim, we show that the paths obtained by following the sequences of vertices listed below are disjoint:

(a) $(x_1, x_2, \dots, x_q)$ and $(y_1, y_2, \dots, y_q)$
(b) $(x_{q+1}, x_{q+2}, \dots, x_t)$, $(y_{q+1}, y_{q+2}, \dots, y_t)$, and $(z_{q+1}, z_{q+2}, \dots, z_t)$
(c) $(x_1, x_2, \dots, x_q)$ and $(y_{q+1}, y_{q+2}, \dots, y_t)$
(d) $(x_1, x_2, \dots, x_q)$ and $(z_{q+1}, z_{q+2}, \dots, z_t)$
(e) $(y_1, y_2, \dots, y_q)$ and $(z_{q+1}, z_{q+2}, \dots, z_t)$
(f) $(y_1, y_2, \dots, y_q)$ and $(x_{q+1}, x_{q+2}, \dots, x_t)$

To prove that the paths obtained by following the sequences in (a) are disjoint we use induction. By the definition of N_i^r, we know that $x_q \neq y_q$. For the inductive hypothesis, assume that the paths obtained from $(x_{j+1}, \dots, x_q)$ and $(y_{j+1}, \dots, y_q)$ are disjoint. Again by definition, there are two cases: (1) $x_j \neq x_{j+1}$ only; and

(2) $y_j \neq y_{j+1}$ only. For (1), we have $y_j = y_{j+1}$ and $h(x_j) \geq h(y_j)$, thus for $\mu > j+1$ and $y_\mu \neq y_j$, we have $h(x_j) > h(y_\mu)$. Together with $x_j \neq y_j$, we can see that x_j does not appear in $(y_j, \dots, y_q)$. Case (2) can be handled in the same way. Thus, the paths from (a) are disjoint.

For (b), the induction proof from the proof of Lemma 2.1 immediately implies that the three paths are disjoint.

To show that the paths obtained from (c) are disjoint, let $j \in \{1, \dots, q\}$ be the largest index such that $x_j \neq x_q$. We know from the paths in (b) that $x_q = x_{q+1}$ does not appear in $(y_{q+1}, \dots, y_t)$, so we only need to prove that $(x_1, \dots, x_j)$ is disjoint from $(y_{q+1}, \dots, y_t)$. Because $x_j \neq x_q$, there exists some $\mu \in \{1, \dots, q\}$ such that $(x_j, y_\mu) \rightarrow (x_q, y_\mu)$ is in the path from s to (x, y, z). By definition $x_j \neq y_\mu$ and $h(x_j) \geq h(y_\mu)$. We consider the following two cases: (1) $h(x_j) > h(y_\mu)$ and (2) $h(x_j) = h(y_\mu)$. In case (1), because of $h(x_1), \dots, h(x_j) > h(y_\mu), \dots, h(y_t)$, the paths from (c) are disjoint. In case (2), let $g \in \{1, \dots, j\}$ be the maximum index such that $x_g \neq x_j$. Since $h(x_g) > h(x_j) = h(y_\mu)$, using the same argument as in (1), we have that $(x_1, \dots, x_g)$ and $(y_\mu, \dots, y_t)$ are disjoint. It only remains to show that x_j does not appear in $(y_\mu, \dots, y_t)$. If we assume that x_j appears in $(y_\mu, \dots, y_t)$ then because $y_\mu \neq x_j$, we would have $h(y_\mu) > h(x_j)$, which leads to a contradiction.

For the paths from (d), similar arguments as in (c) can be applied since $y_q \rightarrow z_{q+1} \in E_i$, $x_q = x_{q+1}$, and $x_{q+1} \neq z_{q+1}$.

To show that the paths from (e) are disjoint, because $y_q \rightarrow z_{q+1} \in E_i$, we have $h(y_1), \dots, h(y_q) > h(z_{q+1}), \dots, h(z_t)$, meaning that the paths from (e) are disjoint.

Finally, to show that the paths from (f) are disjoint, by definition we have $x_q = x_{q+1}$ and $h(y_q) \geq h(x_q)$. So for every $\mu > q+1$ and $x_\mu \neq x_q$, it holds that $h(y_q) > h(x_\mu)$. Since we also have that $x_q \neq y_q$, the paths from (f) are disjoint. □

Corollary 2.4 *Let N_i be a given network and r' a dummy leaf attached to $r(N_i)$. For any two different leaves x and y in N_i that are not r', there are two paths from some internal vertex $z \neq r(N_i)$ in N_i to x and y that are disjoint, except for in z, if and only if s can reach (r', x, y) in N_i^r.*

2.1.3 The Fan Table and the Resolved Table

Given N_i^f and N_i^r, we define the $n \times n \times n$ *fan table* A_i^f and the $n \times n \times n$ *resolved table* A_i^r as follows. For any three different leaves x, y, and z, $A_i^f[x][y][z] = 1$ if the fan triplet $x|y|z$ is consistent with N_i and $A_i^f[x][y][z] = 0$ otherwise. Similarly, $A_i^r[x][y][z] = 1$ if the resolved triplet $x|yz$ is consistent with N_i and $A_i^r[x][y][z] = 0$ otherwise.

With the help of Lemmas 2.1 and 2.3, both A_i^f and A_i^r can be precomputed by depth-first traversals (starting from s) of N_i^f and N_i^r. More precisely, $A_i^f[x][y][z] = 1$ if s can reach (x, y, z) in N_i^f and 0 otherwise, and $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r and 0 otherwise.

Since N_i^f and N_i^r have $O(|V_i|^3)$ vertices and $O(|V_i|^2|E_i|)$ edges, the time needed to build A_i^f and A_i^r by depth-first traversals is $O(|V_i|^3 + |V_i|^2|E_i|) = O(|V_i|^2|E_i|)$.

2.2 Triplet Distance Computation

Algorithm 1 summarizes the steps for computing the triplet distance between two networks N_1 and N_2. The main procedure, $D()$, uses Equation (1.1) to calculate $D(N_1, N_2)$. It first builds the fan table A_i^f and the resolved table A_i^r for each N_i, $i \in \{1, 2\}$, in a preprocessing step, and then relies on the procedure $S()$ for counting shared triplets. The shared fan triplets and shared resolved triplets are counted by iterating over all possible triplets and using the fan and resolved tables to determine the consistency of any triplet with each of the two networks. The correctness is ensured by Lemmas 2.1 and 2.3.

Algorithm 1 Computing $D(N_1, N_2)$ by using the data structures from Section 2.

1: **procedure** PREPROCESSING(N_1, N_2) ▷ Building the data structures
2:     **for** $i \in \{1, 2\}$ **do**
3:         Build $N_i^f = (V_i^f, E_i^f)$ and $N_i^r = (V_i^r, E_i^r)$
4:         Let A_i^f, A_i^r be $n \times n \times n$ tables initialized with 0 entries
5:         **for** $x, y, z \in \Lambda$ with $x \neq y$, $x \neq z$, $y \neq z$ **do**
6:             $A_i^f[x][y][z] = 1$ if s can reach (x, y, z) in N_i^f
7:             $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r
8:     **return** $(A_1^r, A_1^f, A_2^r, A_2^f)$

9: **procedure** $S_f(A_1^f, A_2^f)$ ▷ Finding the shared fan triplets
10:     $sharedF = 0$
11:     **for** $X \subseteq \Lambda$ with $|X| = 3$ **do**
12:         Let x, y, z be the elements of X in any order
13:         **if** $A_1^f[x][y][z] = A_2^f[x][y][z] = 1$ **then** $sharedF = sharedF + 1$
14:     **return** $sharedF$

15: **procedure** $S_r(A_1^r, A_2^r)$ ▷ Finding the shared resolved triplets
16:     $sharedR = 0$
17:     **for** $X \subseteq \Lambda$ with $|X| = 3$ **do**
18:         Let x, y, z be the elements of X in any order
19:         **if** $A_1^r[z][x][y] = A_2^r[z][x][y] = 1$ **then** $sharedR = sharedR + 1$
20:         **if** $A_1^r[y][x][z] = A_2^r[y][x][z] = 1$ **then** $sharedR = sharedR + 1$
21:         **if** $A_1^r[x][y][z] = A_2^r[x][y][z] = 1$ **then** $sharedR = sharedR + 1$
22:     **return** $sharedR$

23: **procedure** $S(A_1^r, A_1^f, A_2^r, A_2^f)$ ▷ Finding the shared triplets
24:     **return** $S_f(A_1^f, A_2^f) + S_r(A_1^r, A_2^r)$

25: **procedure** $D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))$ ▷ Computing $D(N_1, N_2)$
26:     $(A_1^r, A_1^f, A_2^r, A_2^f) =$ PREPROCESSING(N_1, N_2)
27:     **return** $S(A_1^r, A_1^f, A_1^r, A_1^f)$ + $S(A_2^r, A_2^f, A_2^r, A_2^f)$ - $2S(A_1^r, A_1^f, A_2^r, A_2^f)$

To analyze the running time, building the data structures N_i^r and N_i^f for $i \in \{1, 2\}$ on line 3 takes $O(|V_1|^2|E_1| + |V_2|^2|E_2|)$ time. Building the tables A_i^r and A_i^f on lines 4'7 requires $O(|V_1|^2|E_1| + |V_2|^2|E_2|)$ time as well. After the preprocessing is finished, the procedures $S_f()$ and $S_r()$ take $O(n^3)$ time because each of the $4\binom{n}{3} = O(n^3)$ triplets can be checked in O(1) time by table lookups. Hence, the total running time of the algorithm becomes $O(|V_1|^2|E_1| + |V_2|^2|E_2| + n^3)$. By the definitions of N and M (see Sect. 1), the time complexity is $O(N^2M + n^3)$. We have obtained the following theorem:

Theorem 2.5 *The triplet distance between two networks N_1 and N_2 can be computed in $O(N^2M + n^3)$ time.*

3 A Second Approach

In this section, we show how to compute $D(N_1, N_2)$ in $O(M + Nk^2d^2 + n^3)$ time.

Overview. Algorithm 1 in the previous section computed $D(N_1, N_2)$ by iterating over all possible triplets and using the fan and resolved tables for N_1 and N_2 to identify which triplets were consistent with both networks. To refine this idea, for every block of N_i, we will define a network of approximately the same size as the block, which we call a *contracted block network*. For every such contracted block network, we build a fan and resolved graph and the corresponding fan and resolved table. Furthermore, by replacing the blocks of N_i by single vertices, we obtain a tree structure called the *block tree*. The new algorithm in this section combines the block tree and all the fan and resolved tables of the contracted block networks of N_i to efficiently determine whether or not any specified triplet is consistent with N_i.

3.1 Preprocessing

Let N_i be a network. Note that every block B of N_i contains one vertex whose height is greater than the heights of all other vertices in B. This vertex will be called the *root of B* and denoted by $r(B)$. If B contains only one edge $u \to v$ and $v \in \mathcal{L}(N_i)$ then B is called a *leaf block*; otherwise, B is called a $non-leafblock$. Recall from Sect. 1.2 that we assume without loss of generality that: (i) all leaves have in-degree at most 1 (so that every leaf has a leaf block); and (ii) the input networks have no uninformative blocks. Lemma 3.1 presents an important property of the blocks in N_i.

Lemma 3.1 *All blocks of a given network N_i are edge-disjoint.*

Proof For the purpose of obtaining a contradiction, suppose that N_i has two different blocks $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ that share an edge. Define $B = (V_1 \cup V_2, E_1 \cup E_2)$. Let $U(B_1)$, $U(B_2)$, and $U(B)$ be the subgraphs of $U(N_i)$ corresponding to B_1, B_2, and B. Since $U(B_1)$ and $U(B_2)$ are connected graphs that share an edge, $U(B)$ is also connected. Furthermore, if any vertex is removed from B,

$U(B)$ will still be connected. Therefore, $U(B_1)$ and $U(B_2)$ are not maximal biconnected subgraphs of $U(N_i)$, which means B_1 and B_2 are not blocks of N_i. Hence, we have reached a contradiction and the lemma follows. □

3.1.1 The Block Tree

From a high-level perspective, we will remove the cycles in $U(N_i)$ by replacing the non-leaf blocks by internal nodes to obtain a rooted tree on the leaf label set $\mathcal{L}(N_i)$. A similar idea was previously used by Choy *et al.* in the proof of Lemma 2 in [14] to bound the number of reticulation vertices in a network, and later by Byrka *et al.* [27] to efficiently check if a resolved triplet is consistent with a network. Below, we will show that it is also useful for checking if a fan triplet is consistent with a network.

Formally, let $T_i = (V', E')$ be a rooted tree, from now on referred to as the *block tree*, with vertex set V' and edge set E' constructed as follows:

1. For every block B_j in N_i, create a vertex b_j in T_i.
2. Let B_1, B_2 be two blocks in N_i with $r(B_1) \neq r(B_2)$. If $r(B_2)$ is also a vertex in B_1 then create the edge $b_1 \to b_2$ in T_i.
3. Create a root vertex r in T_i. For every block B_j that has $r(N_i)$ as a root, create the edge $r \to b_j$ in T_i.
4. If B_j is a leaf block, rename b_j in T_i by the label of the leaf in B_j.

Figure 4 gives an example of a network N_i and its block tree T_i. The set of blocks in N_i and the vertex set $V' - r(T_i)$, i.e., the set of all vertices of T_i except the root, are in one-to-one correspondence. An edge $b_1 \to b_2$ in T_i means that the corresponding blocks B_1 and B_2 in N_i do not have the same root and the root vertex $r(B_2)$ is a shared vertex between B_1 and B_2. Note that by the definition of a block, an edge connecting two vertices can define a block of its own (for example, block B_9 in Fig. 4).

The following lemma states some properties of T_i.

Lemma 3.2 *Let $T_i = (V', E')$ be the block tree of a given network N_i. The block tree T_i is a rooted tree that has n leaves, $|V'| = O(n)$, and $|E'| = O(n)$.*

Proof We start by showing that T_i is a rooted tree. Since every edge of T_i is directed, T_i is a directed graph. Let $U(T_i)$ be the undirected version of that graph. Since $U(N_i)$ is connected, $U(T_i)$ is connected as well according to the construction. Next, we prove that T_i is a tree by contradiction. Suppose that $U(T_i)$ has a cycle. Then there exists a vertex b in T_i with $in(b) > 1$. If B is the corresponding block of b in N_i, this in turn implies the existence of two different blocks B_1 and B_2 in N_i such that $r(B) \neq r(B_1)$ and $r(B) \neq r(B_2)$, and with $r(B)$ being a vertex in both B_1 and B_2. By the definition of N_i, the root $r(N_i)$ has a path to every vertex in N_i, so $r(B_1)$ and $r(B_2)$ must have a common ancestor. This means that the two blocks B_1 and B_2 could be merged to create an even larger block that contains both of them, contradicting that B_1 and B_2 are blocks of N_i. Thus, T_i is a rooted tree.

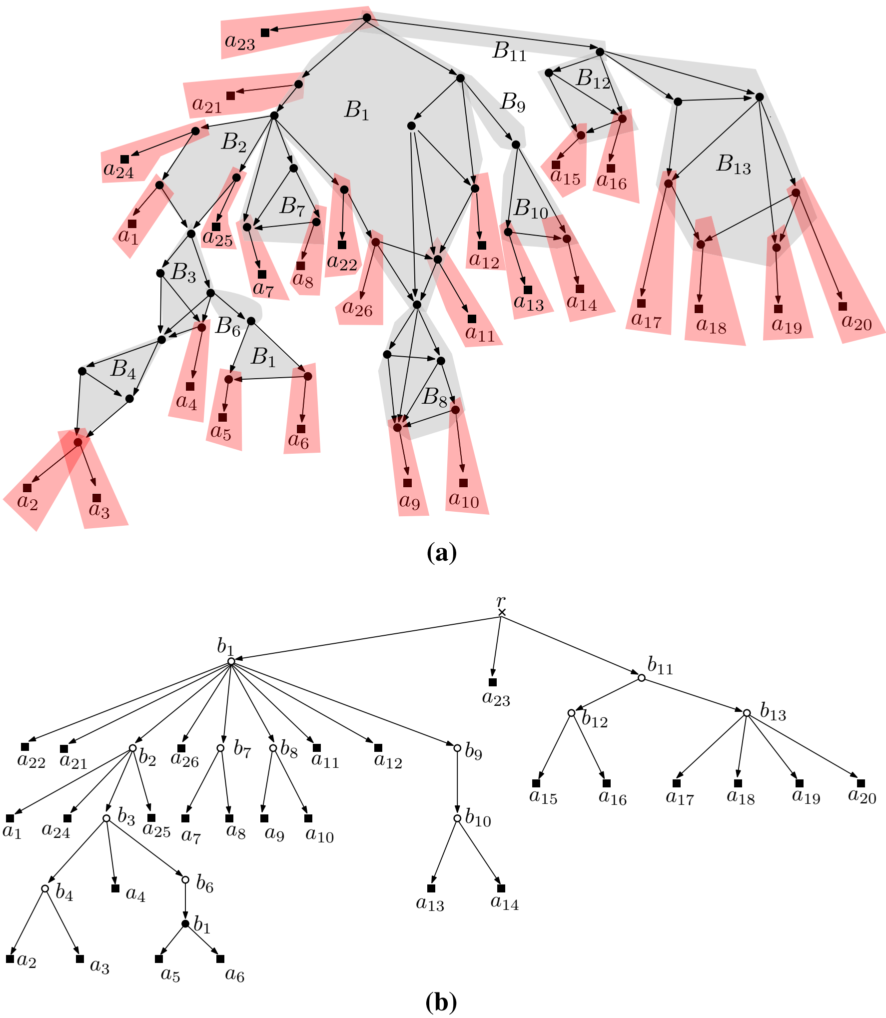


Fig. 4 **a** An example network N_i. The blocks containing leaves are highlighted in red. All other blocks are colored gray. **b** The corresponding block tree T_i

Next, we count the number of vertices and edges in T_i. By assumption (i) mentioned above, there are no leaves with in-degree greater than 1 in N_i. Thus, N_i contains n leaf blocks and there will be exactly n leaves in T_i. To count the internal vertices in T_i, we distinguish between vertices having in-degree 1 and out-degree 1, from now on referred to as *extra vertices*, and *non-extra vertices*. First, to count the non-extra vertices in T_i, observe that if we were to contract its extra vertices, i.e., add an edge from the parent of every such vertex u to the child of u and then remove u, we would obtain a tree $T_i'' = (V'', E'')$ with n leaves in which every internal vertex has in-degree 1 and out-degree at least 2. This means that $|V''| = \mathrm{O}(n)$ and $|E''| = \mathrm{O}(n)$. Secondly, to count the extra vertices, observe that any extra vertex corresponds to an uninformative block in N_i or a non-leaf block of N_i containing a single edge. By assumption (ii) above, N_i has no uninformative blocks. By the definition of a network, N_i has no vertex u with $in(u) = out(u) = 1$, so every extra vertex in T_i must be the parent of at least one non-extra vertex. Because T_i is a tree, no two extra vertices are parents of the same non-extra vertex. If follows that there are $O(n)$ extra vertices in T_i. In total, the number of vertices and edges in T_i is given by $|V'| = O(n)$ and $|E'| = O(n)$. □

Since the set of blocks of N_i and the set $V' - r(T_i)$ are in one-to-one correspondence, we also have:

Corollary 3.3 *The network N_i contains $O(n)$ blocks.*

The following lemma shows that the block tree T_i can be built efficiently:

Lemma 3.4 *The block tree $T_i = (V', E')$ of a given network N_i can be constructed in* $O(|E_i|)$ *time.*

Proof Constructing T_i when the blocks of N_i are given is performed by scanning the vertices of N_i and the list of components that every vertex belongs to, while adding edges to T_i according to the definition of V' and E'. This requires $O(|V_i|)$ time. Finding the blocks takes $O(|E_i|)$ time by applying the algorithm by Hopcroft and Tarjan in [16]. Lastly, $|V_i| \leq |E_i|$ because N_i is a connected graph, so we can build T_i in $O(|E_i|)$ time. □

3.1.2 Contracted Block Networks

Each block in N_i can be viewed as a network, to which we may apply the techniques from Sect. 2 for detecting those triplets that are anchored within. To be able to do so, we first take each block B, make some adjustments to it as described next, and call the resulting network C_B *the contracted block network of N_i corresponding to block B*. See Figs. 5 and 6 for an example of the construction.

For a given network N_i, a block B in N_i, and a vertex u in B, initialize L_B^u as the set of leaves that can be reached from u without using edges in B. For example, for the block B shown in Fig. 5, $L_B^{v_3} = \{a_5, a_6, a_7, a_{19}\}$ and $L_B^{v_{10}} = \{a_{15}\}$. Next, construct the network $C_B = (V', E')$ with vertex set V' and edge set E' and update the L_B^u-sets by applying the following operations:

1. Let C_B be a copy of N_i.
2. Delete every edge and vertex from C_B that is not in B.
3. For every edge $u_1 \rightarrow u_2$ in C_B, if $in(u_1) = out(u_1) = in(u_2) = out(u_2) = 1$ then contract the edge as follows: Let $u_2 \rightarrow u_3$ be the edge outgoing from u_2, create an edge $u_1 \rightarrow u_3$, delete u_2 and its two incident edges, and let $L_B^{u_1} = L_B^{u_1} \cup L_B^{u_2}$.

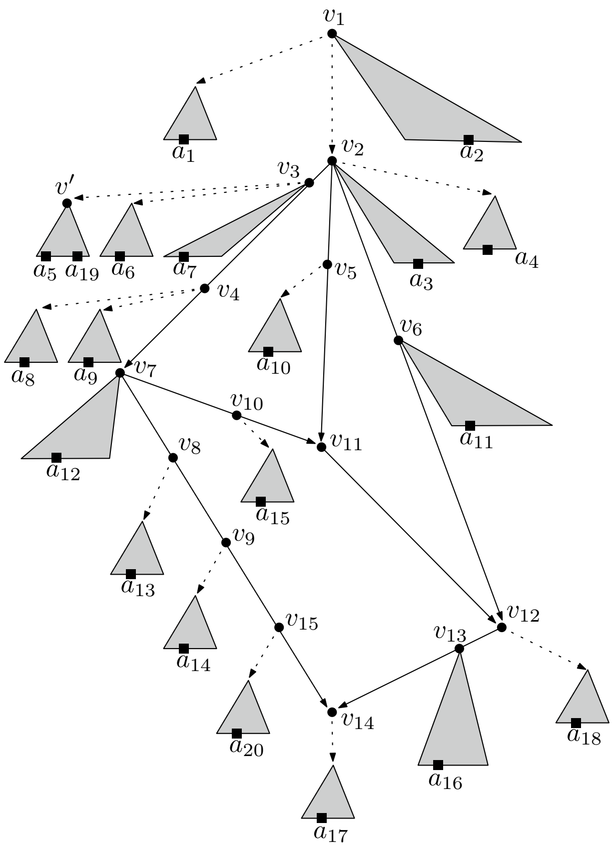


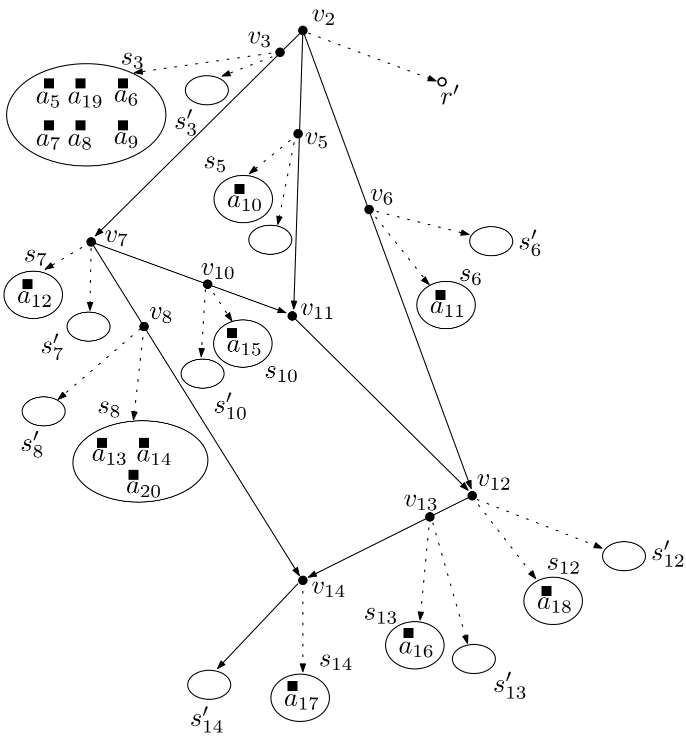
Fig. 5 In this example, N_i is a level-3 network that contains a block B whose vertices are $v_2, v_3, \ldots, v_{15}$ and whose edges are drawn with solid lines. Here, $r(B) = v_2$

4. For every vertex u_j in C_B with $L_B^{u_j} \neq \emptyset$, attach a child leaf s_j representing the set of leaves $L_B^{u_j}$. Also attach another child leaf s_j' called a *copy leaf*, to be used later on to count triplets.
5. Insert an artificial leaf r' as a child of the root $r(C_B)$.

Observe that every edge between two internal vertices in C_B corresponds to a path in B. For example, the edge $v_8 \to v_{14}$ in Fig. 6 corresponds to the path $(v_8, v_9, v_{15}, v_{14})$ in Fig. 5, while the edge $v_{13} \to v_{14}$ corresponds to the length-1 path (v_{13}, v_{14}).

The following lemma bounds the size of C_B:

Lemma 3.5 *Let N_i be a network, B a block in N_i, and $C_B = (V', E')$ the contracted block network of N_i that corresponds to block B. It holds that* $|V'| = \mathrm{O}(k_i d_i + 1)$, $|E'| = \mathrm{O}(k_i d_i + 1)$, *and* $|\mathcal{L}(C_B)| = \mathrm{O}(k_i d_i + 1)$.



(a) C_B

Fig. 6 The contracted block network C_B for the block B from Fig. 5. The internal vertices v_3 and v_4 in B have been merged in C_B, and similarly for v_8, v_9, and v_{15}. The set of leaves in C_B is $\{s_i, s'_i : i \in \{3, 5, 6, 7, 8, 10, 12, 13, 14\}\}$

Proof If $k_i = 0$ then B consists of a single edge of N_i, meaning that C_B is a binary tree on three leaves (a leaf of the form s_j, its copy leaf s'_j, and the artificial leaf r'). In this case, $|V'| = 5, |E'| = 4$, and $|\mathcal{L}(C_B)| = 3$.

If $k_i \geq 1$, there are two possibilities. If B contains only one edge then C_B is a binary tree on three leaves as in the case $k_i = 0$ above. Otherwise, proceed as follows to derive the bounds. Call a non-reticulation vertex of C_B that is the parent of at least two internal vertices of C_B a *branching vertex* (e.g., v_2 and v_7 in Fig. 6), and a non-reticulation vertex of C_B that is the parent of exactly one internal vertex a *path vertex* (e.g., v_3, v_5, v_6, v_8, v_{10}, and v_{13} in Fig. 6). We apply a technique from [27] to count the branching vertices and note that every branching vertex is the beginning of at least one new directed path that has to end at a reticulation vertex. Since each reticulation vertex can end at most d_i such paths and there are at most k_i reticulation vertices in C_B, the number of branching vertices is at most $k_i d_i$. Every path vertex is the parent of either a branching vertex or a reticulation vertex, and every reticulation vertex has at most d_i parents, so the number of path vertices is at most $2k_i d_i$. Therefore, the total number of internal vertices is at most $k_i(3d_i + 1)$. Next, at most two leaves are attached to each internal vertex, so $|\mathcal{L}(C_B)| \le 2k_i(3d_i + 1)$ and $|V'| \le 3k_i(3d_i + 1)$. As for the edges, there are at most $k_i d_i$ edges ending at reticulation vertices, at most $k_i d_i$ edges ending at branching vertices, at most $2k_i d_i$ edges ending at path vertices, and $|\mathcal{L}(C_B)|$ edges ending at leaves. Adding them together gives $|E'| \le 10k_i d_i + 2k_i$.

Hence, the lemma statement holds for every $k_i \ge 0$. □

3.1.3 Constructing All Contracted Block Networks Efficiently

We first introduce some additional notation. For a given network N_i and a block B in N_i, a leaf x in N_i is said to *associate with* B if there exists a vertex u in B such that $u \neq r(B)$ and $x \in L_B^u$. As an example, in Fig. 5, the leaf a_{16} associates with B, but the leaves a_2 and a_3 do not associate with B. For any leaf x associated with a block B of N_i, define:

- $q_B(x)$: The vertex in B from which there is a path to x that does not use any edges in B. That is, $x \in L_B^{q_B(x)}$.
- $p_B(x)$: The leaf in C_B representing x.
- $p'_B(x)$: The copy leaf of $p_B(x)$.

For example, in Figs. 5 and 6, $q_B(a_5) = v_3$, $p_B(a_5) = s_3$, $p'_B(a_5) = s'_3$, $q_B(a_8) = v_4$, and $p_B(a_8) = s_3$.

Lemma 3.1 yields an algorithm for constructing all block networks of N_i in $\mathrm{O}(|E_i|)$ time. As shown in the next lemma, by properly relabeling the leaves of N_i and using an additional $\mathrm{O}(n^2)$ time, it is possible to build the block networks so that we can subsequently compute, for any block B and any leaf $l \in \mathcal{L}(N_i)$, the values of $q_B(l)$ and $p_B(l)$ in $\mathrm{O}(1)$ time.

Lemma 3.6 *For any network N_i, all the contracted block networks of N_i can be computed in $\mathrm{O}(|E_i| + n^2)$ time, after which $q_B(l)$ and $p_B(l)$ for any block B and any leaf $l \in \mathcal{L}(N_i)$ can be retrieved in $\mathrm{O}(1)$ time.*

Proof Perform the following steps:

1. Identify all the blocks of N_i. Let $B_1 \dots, B_s$ be the blocks of N_i and let the corresponding vertex sets be $V(B_1), \dots, V(B_s)$. Note that for every $j \in \{1, \dots, s\}$, it holds that $V(B_j) \subseteq V_i$.
2. The leaves of N_i are relabeled as follows. A leaf receives the label i, where $i \in \{1, 2, \dots, n\}$, if it is the $i - th$ leaf in order that is discovered by a depth-first traversal of N_i. This traversal starts from $r(N_i)$. Let u be a vertex in N_i and part of the blocks $B_1, \dots, B_j$. Let B' be the block from $B_1, \dots, B_j$, such that $r(B')$ has the largest height among all roots of $B_1, \dots, B_j$. During the traversal, every child u' of u that is not part of B' is visited first. This is to ensure that the labels in $L_{B'}^u$ are consecutive and defined by a range of numbers $[u_{left}, u_{right}]$.

3. For every $j \in \{1, \dots, s\}$ the process of building $C_{B_j} = (V_j, E_j)$ is initialized as follows. Set $V_j = V(B_j)$. For every edge $u \to v$ in E_i, if both u and v are in V_j then add that edge to E_j. Finally, for any vertex u_1 in V_j, if $L_{B_j}^{u_1} \neq \emptyset$ create the leaf s_1 representing $L_{B_j}^{u_1}$, the copy leaf s'_1, add the edges $u_1 \to s_1$ and $u_1 \to s'_1$ to E_j, and set $Q_{B_j}[l] = u_1$ for every $l \in \{u_{left}, \dots, u_{right}\}$.
4. For every $j \in \{1, \dots, s\}$ the edges of C_{B_j} are contracted, following the definition of a contracted block network. While performing the contraction, for every $j \in \{1, \dots, s\}$, we build the table $P_{B_j}[1, \dots, n]$, defined so that for every $l \in \{1, \dots, n\}$ we have $P_{B_j}[l] = p_{B_j}(l)$. The value of $P_{B_j}[l]$ is updated once the final set in which the leaf l will reside has been determined. After contracting all the edges, we also add the artificial leaf r'_j.

Step 1 is performed by using the algorithm from [16], which takes $O(|E_i|)$ time. Step 2 is performed by a depth-first traversal of N_i, thus requiring $O(|E_i|)$ time as well. Since the blocks of N_i are edge-disjoint (see Lemma 3.1), we have $\sum_{j=1}^{s} |E_j| \leq |E_i|$, thus the time spent on adding and contracting vertices and edges in steps 3 and 4 is $O(|E_i|)$. For every contracted block network C_B, we spend $O(n)$ time to update the Q- and P-tables. By Corollary 3.3, there are $O(n)$ blocks, so the time needed to update every Q- and P-table is $O(n^2)$. Hence, the total time taken is $O(|E_i| + n^2)$. □

Finally, for any block B in N_i, we denote the fan graph of its contracted block network C_B by C_B^f and the resolved graph of C_B by C_B^r. Moreover, we let A_B^f be the fan table of C_B and A_B^r the resolved table of C_B. The following lemma bounds the time required to build C_B^f, C_B^r, A_B^f, and A_B^r for all the blocks of a network N_i.

Lemma 3.7 *Given a network N_i and all of its contracted block networks, building C_B^f, C_B^r, A_B^f, and A_B^r for every block B of N_i takes $O(|V_i|(k_i^2 d_i^2 + 1))$ time in total.*

Proof We simply apply the method from Sect. 2 to each contracted block network. To analyze the time that this will take, let $\{B_1, B_2, \dots, B_t\}$ be the blocks in N_i. For each block B_x in N_i, let $b(x)$ be the number of vertices in B_x, $c(x)$ the number of vertices in the contracted block network C_{B_x}, and $e(x)$ the number of edges in the contracted block network C_{B_x}.

We first express the total size of the contracted block networks in terms of N. When C_{B_x} is constructed from B_x, each vertex in B_x will either be deleted or remain and introduce at most two leaves, so $c(x) \leq 3 \cdot b(x)$. Next, since the blocks decompose N_i into edge-disjoint subgraphs by Lemma 3.1, and the total number of times that blocks overlap each other is equal to the number of edges E' in the block tree T_i, we have $\sum_{x=1}^{t} b(x) \leq |V_i| + |E'|$. By Lemma 3.2, $|E'| = O(n)$. Then, using $n \leq |V_i|$ gives $\sum_{x=1}^{t} c(x) \leq 3 \cdot \sum_{x=1}^{t} b(x) = O(|V_i|)$.

Now, we analyze the total time for all the blocks. According to Sect. 2, building each $C_{B_x}^f$, $C_{B_x}^r$, $A_{B_x}^f$, and $A_{B_x}^r$ takes $O(c(x)^2 e(x))$ time. The total time is thus

$\sum_{x=1}^{t} O(c(x)^2 e(x))$. Lemma 3.5 says that $c(x) = O(k_i d_i + 1)$ and $e(x) = O(k_i d_i + 1)$, so we can rewrite the total time needed as $O(\sum_{x=1}^{t} (k_i d_i + 1)^2 c(x)) = O((k_i d_i + 1)^2 \sum_{x=1}^{t} c(x)) = O(|V_i|(k_i^2 d_i^2 + 1))$. □

3.2 Checking If a Triplet is Consistent with a Network

Sections 3.2.1 and 3.2.2 below describe how to determine if any given fan or resolved triplet, respectively, is consistent with N_i in O(1) time, assuming that the data structures from Sect. 3.1 have already been built.

A more precise definition of triplet consistency that can associate specific locations in the network to triplets that are consistent with it will be needed in this section. Let B be a block of a network N_i. We say that $x|y|z$ is a *fan triplet consistent with B* if and only if there exists a vertex u in B such that there are three directed paths in N_i from u to x, from u to y, and from u to z that are disjoint except for in u. We also say that $x|y|z$ *is rooted at u* in B. Since u belongs to N_i, this means that $x|y|z$ is rooted at u in N_i as well. Next, we say that $xy|z$ is a *resolved triplet consistent with B* if and only if there exist two vertices u and v ($u \neq v$) in B such that there are four directed paths in N_i from u to v, from v to x, from v to y, and from u to z that are disjoint except for in u and v, and the path from u to z does not pass through v. Moreover, we say that $xy|z$ is *rooted at u and v* in B and in N_i.

Observe that if $x|y|z$ is a fan triplet consistent with a block B, then it is also consistent with N_i. In the same way, if $xy|z$ is a resolved triplet consistent with B, it is also consistent with N_i.

3.2.1 Checking a Fan Triplet

First, we show how to determine if a given fan triplet $x|y|z$ is consistent with a given block B (Lemma 3.8). The procedure, named IsFanInBlock, requires that the lowest common ancestor (in the block tree T_i) of x and y, the lowest common ancestor of x and z, and the lowest common ancestor of y and z are the same, and that this node corresponds to the block B being examined.

After that, the procedure IsFanInBlock is used as a subroutine in another procedure, named IsFan, to determine if a given fan triplet $x|y|z$ is consistent with a network (Lemma 3.9). Whenever IsFanInBlock's requirement on the lowest common ancestors cannot be met, IsFan instead considers the different cases for the locations of the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i. Since every vertex in T_i except $r(T_i)$ corresponds to a block in N_i, it can then apply the available data structures to determine if N_i has the necessary disjoint paths.

Algorithm 2 Checking if $x|y|z$ is consistent with B, assuming that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i corresponds to B.

procedure IsFanInBlock($x|y|z$, N_i, B, C_B^f)
    For every $l \in \{x, y, z\}$, let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i
    if $p_x = p_y = p_z$ **then**
        if $h_x = h_y = h_z$ **then return true** ▷ e.g., $a_5|a_6|a_7$ in Fig. 3.2
        if $((h_x = h_y) \wedge (h_x > h_z)) \vee ((h_x = h_z) \wedge (h_x > h_y)) \vee ((h_y = h_z) \wedge (h_y > h_x))$ **then return true** ▷ e.g., $a_5|a_6|a_8$ in Fig. 3.2
        if $h_x \neq h_y \neq h_z$ **then return false** ▷ e.g., $a_{13}|a_{14}|a_{20}$ in Fig. 3.2
    if $((p_x = p_y) \wedge (p_x \neq p_z)) \vee ((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$ **then** assume w.l.o.g. that $(p_x = p_y) \wedge (p_x \neq p_z)$
        if $h_x = h_y$ **then**
            if $\exists s \rightsquigarrow (p'_x, p_x, p_z)$ in C_B^f **then**
                return true ▷ e.g., $a_8|a_9|a_{15}$ in Fig. 3.2
            else return false ▷ e.g., $a_8|a_9|a_{11}$ in Fig. 3.2
        else return false ▷ e.g., $a_7|a_8|a_{15}$ in Fig. 3.2
    if $p_x \neq p_y \neq p_z$ **then**
        if $\exists s \rightsquigarrow (p_x, p_y, p_z)$ in C_B^f **then**
            return true ▷ e.g., $a_8|a_{11}|a_{16}$ in Fig. 3.2
        else return false ▷ e.g., $a_{14}|a_{16}|a_{17}$ in Fig. 3.2

Lemma 3.8 *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 3.7 has been performed on N_i. Consider any $x, y, z \in \Lambda$ such that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) is a node w in T_i. If $w \neq r(T_i)$, Algorithm 2 determines whether or not the fan triplet $x|y|z$ is consistent with the block B in N_i corresponding to w in $O(1)$ time.*

Proof For every $l \in \{x, y, z\}$, we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i. By construction (see Lemmas 3.4 and 3.6), we know that p_x, p_y, and p_z are not the root of C_B. The algorithm uses the tables Q and P to check all the possible cases for the values of p_x, p_y, p_z, q_x, q_y, and q_z, and return a true or false value, indicating a positive and a negative answer respectively. We have the following cases:

1. $p_x = p_y = p_z$:

(a) $h_x = h_y = h_z$: We have $q_x = q_y = q_z$ and $x|y|z$ is rooted at q_x. Hence, $x|y|z$ is consistent with B (e.g., $a_5|a_6|a_7$ in Fig. 5).
(b) $((h_x = h_y) \wedge (h_x > h_z)) \vee ((h_x = h_z) \wedge (h_x > h_y)) \vee ((h_y = h_z) \wedge (h_y > h_x))$. W.l.o.g., assume true for $((h_x = h_y) \wedge (h_x > h_z))$: Then, we have $q_x = q_y \wedge q_x \neq q_z$ and $x|y|z$ is rooted at q_x. Hence, $x|y|z$ is consistent with B (e.g., $a_5|a_6|a_8$ in Fig. 5).
(c) $h_x \neq h_y \neq h_z$: Then $q_x \neq q_y \neq q_z$, thus $x|y|z$ is not consistent with B (e.g., $a_{13}|a_{14}|a_{20}$ in Fig. 5).

2. $((p_x = p_y) \wedge (p_x \neq p_z)) \vee ((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$. W.l.o.g., assume true for $(p_x = p_y \wedge p_x \neq p_z)$:

   (a) $h_x = h_y$: We have $q_x = q_y$. If $p'_x|p_x|p_z$ is a fan triplet in C_B, then $x|y|z$ is rooted at q_x, thus $x|y|z$ is consistent with B (e.g., $a_8|a_9|a_{15}$ in Fig. 5). If $p'_x|p_x|p_z$ is not a fan triplet in C_B, $x|y|z$ is not rooted at any vertex in B, thus $x|y|z$ is not consistent with B (e.g., $a_8|a_9|a_{11}$ in Fig. 5).
   (b) $h_x \neq h_y$: Then $q_x \neq q_y$ and either q_x or q_y was contracted when creating C_B. Moreover, both x and y are now in the set of leaves defined by p_x. Since we also have $p_z \neq p_x$, the triplet $x|y|z$ is not consistent with B (e.g., $a_7|a_8|a_{15}$ in Fig. 5).

3. $p_x \neq p_y \neq p_z$: If $p_x|p_y|p_z$ is consistent with C_B, then there exists a vertex u in B such that $x|y|z$ is rooted at u. Hence, $x|y|z$ is consistent with B (e.g., $a_8|a_{11}|a_{16}$ in Fig. 5). If $p_x|p_y|p_z$ is not consistent with C_B, $x|y|z$ is not rooted at any vertex in B, thus $x|y|z$ is not consistent with B (e.g., $a_{14}|a_{16}|a_{17}$ in Fig. 5).

In every case above, testing if a fan triplet is consistent with C_B translates to finding a path that starts from s in C_B^f and ends in a vertex of C_B^f defined by the leaves of the fan triplet. Hence, every case can be handled in O(1) time. In Algorithm 2, the above cases are summarized in a procedure. $\square$

Algorithm 3 Checking if $x|y|z$ is consistent with N_i.

```
procedure IsFan(x|y|z, N_i, T_i)
    if x|y|z is consistent with T_i then
        w ← lca(x, y, z)
        if w is the root of T_i then return true                      ▷ e.g., a_23|a_9|a_20 in Fig. 3.1
        else let B be the block of w
            return IsFanInBlock(x|y|z, N_i, B, C_B^f)                 ▷ e.g., a_3|a_9|a_12 in Fig. 3.1
    if xy|z or xz|y or yz|x is consistent with T_i then
        Assume w.l.o.g. that xy|z is consistent with T_i
        w ← lca(x, y)
        μ ← lca(x, z)
        if μ is the parent of w in T_i then
            Let B be the block of w
            Let F be the block of μ
            if p_B(x) = p_B(y) then
                return false                                          ▷ e.g., a_2|a_3|a_4 in Fig. 3.1
            else
                if μ is the root of T_i then
                    if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^f then
                        return true                                   ▷ e.g., a_1|a_11|a_15 in Fig. 3.1
                    else return false                                 ▷ e.g., a_12|a_13|a_15 in Fig. 3.1
                else
                    if p_F(x) = p_F(z) then
                        if h_F(z) ≤ h_F(x) then
                            if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^f then
                                return true                           ▷ e.g., a_1|a_4|a_8 in Fig. 3.1
                            else return false                         ▷ e.g., a_1|a_24|a_8 in Fig. 3.1
                        else return false                             ▷ e.g., a_1|a_4|a_21 in Fig. 3.1
                    else
                        if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^f
                                and ∃s ⇝ (p_F(x), p'_F(x), p_F(z)) in C_F^f then
                                return true                           ▷ e.g., a_1|a_4|a_9 in Fig. 3.1
                        else return false                             ▷ e.g., a_1|a_4|a_12 in Fig. 3.1
        else return false                                             ▷ e.g., a_2|a_4|a_13 in Fig. 3.1
```

Lemma 3.9 *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 3.7 has been performed on N_i. For any $x, y, z \in \Lambda$, Algorithm 3 determines whether or not the fan triplet $x|y|z$ is consistent with N_i in* $O(1)$ *time.*

Proof For a block B of N_i and a vertex u in B that can reach a leaf x of N_i, define $h_B(x)$ to be the height of $q_B(x)$ in N_i. In Algorithm 3 we have the procedure for testing the consistency of the fan triplet $x|y|z$. It considers the following cases:

1. $x|y|z$ is consistent with T_i: Let w be the lowest common ancestor of x, y, and z in T_i.

(a) $w = r(T_i)$: $x|y|z$ is rooted at $r(N_i)$, thus $x|y|z$ is consistent with N_i (e.g., $a_{23}|a_9|a_{20}$ in Fig. 4).
(b) $w \neq r(T_i)$: w corresponds to a block B in N_i, thus we use Lemma 3.8 to determine if $x|y|z$ is consistent with B. If $x|y|z$ is consistent with B, then it is also consistent with N_i. If $x|y|z$ is not consistent with B, then it is not consistent with N_i (e.g., $a_3|a_9|a_{12}$ in Fig. 4).

2. $xy|z \vee xz|y \vee yz|x$ is consistent with T_i. Assume w.l.o.g. that $xy|z$ is consistent with T_i. Let $w = lca(x, y)$ in T_i and $\mu = lca(x, z)$ in T_i, and let B be the block in N_i corresponding to w and F the block in N_i corresponding to μ:

(a) μ is not the parent of w in T_i: then $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_2|a_4|a_{13}$ in Fig. 4).
(b) μ is the parent of w in T_i. By the definition of T_i, B is rooted at a vertex u of F that is not $r(F)$:

i. $(p_B(x) = p_B(y))$: then $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_2|a_3|a_4$ in Fig. 4).
ii. $(p_B(x) \neq p_B(y)) \wedge (\mu = r(T_i))$: If $r'|p_B(x)|p_B(y)$ is consistent with C_B, where r' is the dummy leaf in C_B (see Corollary 2.2), then $x|y|z$ is rooted at $r(N_i)$, thus $x|y|z$ is consistent with N_i (e.g., $a_1|a_{11}|a_{15}$ in Fig. 4). Otherwise, $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_{12}|a_{13}|a_{15}$ in Fig. 4).
iii. $(p_B(x) \neq p_B(y)) \wedge (\mu \neq r(T_i))$:

A. $(p_F(x) = p_F(z)) \wedge (h_F(z) \leq h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$, thus $h_F(x) = h_F(y)$. Using Corollary 2.2, if $r'|p_B(x)|p_B(y)$ is a fan triplet in C_B, where r' is the dummy leaf in C_B, then $x|y|z$ is rooted at $q_F(x)$, thus $x|y|z$ is a fan triplet in N_i (e.g., $a_1|a_4|a_8$ in Fig. 4). Otherwise, $x|y|z$ is not rooted at any vertex in N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_1|a_{24}|a_8$ in Fig. 4).
B. $(p_F(x) = p_F(z)) \wedge (h_F(z) > h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$ and $h_F(x) = h_F(y)$. Hence, $x|y|z$ is not consistent with N_i (e.g., $a_1|a_4|a_{21}$ in Fig. 4).
C. $p_F(x) \neq p_F(z)$: Using Corollary 2.2, if $r'|p_B(x)|p_B(y)$ is a fan triplet in C_B, where r' is the dummy leaf in C_B, and $p_F(x)|p'_F(x)|p_F(z)$ is a fan triplet in C_F, then $x|y|z$ is rooted at $q_F(x)$. Hence, $x|y|z$ is consistent with N_i (e.g., $a_1|a_4|a_9$ in Fig. 4). Otherwise, $x|y|z$ is not rooted at any vertex of N_i, thus $x|y|z$ is not consistent with N_i (e.g., $a_1|a_4|a_{12}$ in Fig. 4).

□

3.2.2 Checking a Resolved Triplet

The strategy for determining if a given resolved triplet $xy|z$ is consistent with a network is analogous to the case of fan triplets just described. The procedure IsResolvedInBlock (see Lemma 3.10) first considers consistency with a block B in the case where it holds in the block tree T_i that the lowest common ancestor of x and y, the lowest common ancestor of x and z, and the lowest common ancestor of y and z are the same. Next, the procedure IsResolved (see Lemma 3.11) uses IsResolvedInBlock and the available data structures to take care of the general case.

Algorithm 4 Checking if $xy|z$ is consistent with B, assuming that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) in T_i corresponds to B.

procedure IsResolvedInBlock($xy|z$, N_i, B, C_B^r, C_B^f)
  For every $l \in \{x, y, z\}$, let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i
  if $p_x = p_y = p_z$ **then**
    if $(h_z > h_x) \wedge (h_z > h_y)$ **then return true** ▷ e.g., $a_8a_9|a_6$ in Fig. 3.2
    else return false ▷ e.g., $a_8a_6|a_9$ in Fig. 3.2
  if $p_x = p_y \wedge p_x \neq p_z$ **then**
    if $\exists s \rightsquigarrow (p_z, p_x, p'_x)$ in C_B^r **then**
      return true ▷ e.g., $a_5a_8|a_{17}$ in Fig. 3.2
    else return false ▷ e.g., $a_5a_8|a_{15}$ in Fig. 3.2
  if $((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$ **then**
    assume w.l.o.g. that $(p_x = p_z) \wedge (p_x \neq p_y)$
    if $h_z > h_x$ **then**
      if $\exists s \rightsquigarrow (p_x, p'_x, p_y)$ in C_B^f **then**
        return true ▷ e.g., $a_{14}a_{17}|a_{13}$ in Fig. 3.2
      else return false ▷ e.g., $a_{14}a_{16}|a_{13}$ in Fig. 3.2
    else return false ▷ e.g., $a_{14}a_{17}|a_{20}$ in Fig. 3.2
  if $p_x \neq p_y \neq p_z$ **then**
    if $\exists s \rightsquigarrow (p_z, p_x, p_y)$ in C_B^r **then**
      return true ▷ e.g., $a_{12}a_{13}|a_{18}$ in Fig. 3.2
    else return false ▷ e.g., $a_{12}a_{18}|a_{13}$ in Fig. 3.2

Lemma 3.10 *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 3.7 has been performed on N_i. Consider any $x, y, z \in \Lambda$ such that the lowest common ancestor of every pair (x, y), (x, z), and (y, z) is a node w in T_i. If $w \neq r(T_i)$, Algorithm 4 determines whether or not the resolved triplet $xy|z$ is consistent with the block B in N_i corresponding to w in $O(1)$ time.*

Proof Like in the case of fan triplets in Lemma 3.8, for every $l \in \{x, y, z\}$, we let $p_l = p_B(l)$, $p'_l = p'_B(l)$, $q_l = q_B(l)$, and h_l be the height of q_l in N_i. By construction (see Lemmas 3.4 and 3.6), we know that p_x, p_y, and p_z are not the root of C_B. The algorithm uses the tables Q and P to check all the possible cases for the values of p_x, p_y, p_z, q_x, q_y, and q_z, and return a true or false value, indicating a positive and a negative answer respectively. We have the following cases:

1. $p_x = p_y = p_z$:

   1. $(h_z > h_x) \wedge (h_z > h_y)$. W.l.o.g., let $h_x \geq h_y$: Then, $xy|z$ is rooted at q_z and q_x, thus $xy|z$ is a resolved triplet in B (e.g., $a_8a_9|a_6$ in Fig. 5).
   2. $(h_z \leq h_x) \vee (h_z \leq h_y)$: Because $p_x = p_y = p_z$, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_8a_6|a_9$ in Fig. 5).

3. $(p_x = p_y) \wedge (p_x \neq p_z)$. W.l.o.g., assume $h_x \geq h_y$: If $p'_xp_x|p_z$ is consistent with C_B, there exists $u \neq q_x$ in B such that $xy|z$ is rooted at u and q_x in B. Hence, $xy|z$ is consistent with B (e.g., $a_5a_8|a_{17}$ in Fig. 5). If $p'_xp_x|p_z$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_5a_8|a_{15}$ in Fig. 5).
4. $((p_x = p_z) \wedge (p_x \neq p_y)) \vee ((p_y = p_z) \wedge (p_y \neq p_x))$. W.l.o.g., assume $(p_x = p_z) \wedge (p_x \neq p_y)$:

   1. $h_z > h_x$: If $p'_x|p_x|p_y$ is a fan triplet in C_B, then $xy|z$ is rooted at q_z and q_x, thus $xy|z$ is consistent with B (e.g., $a_{14}a_{17}|a_{13}$ in Fig. 5). If $p'_x|p_x|p_y$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_{14}a_{16}|a_{13}$ in Fig. 5.).
   2. $h_z \leq h_x$: Since $p_x = p_z$, the resolved triplet $xy|z$ cannot be consistent with B (e.g., $a_{14}a_{17}|a_{20}$ in Fig. 5).

3. $p_x \neq p_y \neq p_z$: If $p_xp_y|p_z$ is consistent with C_B, then there exist two different vertices u, v in B such that $xy|z$ is rooted at u and v, thus $xy|z$ is consistent with B (e.g., $a_{12}a_{13}|a_{18}$ in Fig. 5). If $p_xp_y|p_z$ is not consistent with C_B, $xy|z$ is not rooted at any pair of vertices in B, thus $xy|z$ is not consistent with B (e.g., $a_{12}a_{18}|a_{13}$ in Fig. 5).

Similarly to fan triplets, testing if a resolved triplet is consistent with C_B translates to finding a path that starts from s in C^r_B and ends in a vertex of C^r_B defined by the leaves of the resolved triplet. Hence, every case can be handled in O(1) time. Algorithm 4 summarizes the above cases in a procedure. □

Algorithm 5 Checking if $xy|z$ is consistent with N_i.

```
procedure IsResolved(xy|z, N_i, T_i)
    if x|y|z is consistent with T_i then
        w ← lca(x, y, z)
        if w is the root of T_i then return false          ▷ e.g., a_23 a_9|a_20 in Fig. 3.1
        else let B be the block of w
            return IsResolvedInBlock(xy|z, N_i, B, C_B^r)   ▷ e.g., a_1 a_9|a_12 in Fig. 3.1
    if xy|z or xz|y or yz|x is consistent with T_i then
        Assume w.l.o.g. that xy|z is consistent with T_i
        w ← lca(x, y)
        μ ← lca(x, z)
        if μ is the parent of w in T_i then
            Let B be the block of w
            Let F be the block of μ
            if p_B(x) = p_B(y) then
                return true                                 ▷ e.g., a_2 a_3|a_4 in Fig. 3.1
            else
                if μ is the root of T_i then
                    if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^r then
                        return true                         ▷ e.g., a_11 a_13|a_15 in Fig. 3.1
                    else return false                       ▷ e.g., a_1 a_13|a_15 in Fig. 3.1
                else
                    if p_F(x) = p_F(z) then
                        if h_F(z) ≤ h_F(x) then
                            if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^r then
                                return true                 ▷ e.g., a_1 a_4|a_8 in Fig. 3.1
                            else return false               ▷ e.g., a_1 a_25|a_22 in Fig. 3.1
                        else return true                    ▷ e.g., a_1 a_4|a_21 in Fig. 3.1
                    else
                        if ∃s ⇝ (r', p_B(x), p_B(y)) in C_B^r
                                or ∃s ⇝ (p_F(z), p_F(x), p'_F(x)) in C_F^r then
                                return true                 ▷ e.g., a_1 a_4|a_12 in Fig. 3.1
                        else return false                   ▷ e.g., a_1 a_25|a_26 in Fig. 3.1
        else return true                                    ▷ e.g., a_2 a_4|a_13 in Fig. 3.1
```

Lemma 3.11 *Let N_i be a given network and T_i its block tree, and suppose that the preprocessing from Lemma 3.7 has been performed on N_i. For any $x, y, z \in \Lambda$, Algorithm 5 determines whether or not the resolved triplet $xy|z$ is consistent with N_i in* O(1) *time.*

Proof For a block B of N_i and a vertex u in B that can reach a leaf x of N_i, define $h_B(x)$ to be the height of $q_B(x)$ in N_i. In Algorithm 5 we have the procedure for testing the consistency of the resolved triplet $xy|z$. We consider the following cases, which are similar to the cases for fan triplets in Lemma 3.9:

1. $x|y|z$ is consistent with T_i: Let w be the lowest common ancestor of x, y, and z in T_i.

(a) $w = r(T_i)$: $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_{23}a_9|a_{20}$ in Fig. 4).
(b) $w \neq r(T_i)$: w corresponds to a block B in N_i, thus we use Lemma 3.10 to determine if $xy|z$ is consistent with B. If $xy|z$ is consistent with B, then it is also consistent with N_i. If $xy|z$ is not consistent with B, then it is not consistent with N_i (e.g., $a_1a_9|a_{12}$ in Fig. 4).

2. $xy|z \vee xz|y \vee yz|x$ is consistent with T_i. Assume w.l.o.g. that $xy|z$ is consistent with T_i. Let $w = lca(x, y)$ in T_i and $\mu = lca(x, z)$ in T_i, and let B be the block in N_i corresponding to w and F the block in N_i corresponding to μ:

(a) μ is not the parent of w in T_i: then there exists a vertex u in B and a vertex v in F such that $xy|z$ is rooted at v and u, thus $xy|z$ is consistent with N_i (e.g., $a_2a_4|a_{13}$ in Fig. 4).
(b) μ is the parent of w in T_i. By the definition of T_i, B is rooted at a vertex u of F that is not $r(F)$. We consider the following cases:

i. $p_B(x) = p_B(y)$: W.l.o.g., assume $h_B(x) > h_B(y)$. Then, $xy|z$ is rooted at either $r(B)$ and $q_B(x)$, or $q_F(z)$ and $q_B(x)$, or $r(F)$ and $q_B(x)$. Hence, $xy|z$ is consistent with N_i (e.g., $a_2a_3|a_4$ in Fig. 4).
ii. $(p_B(x) \neq p_B(y)) \wedge (\mu = r(T_i))$: Using Corollary 2.4, if we have that $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at $r(N_i)$ and u. Hence, $xy|z$ is consistent with N_i (e.g., $a_{11}a_{13}|a_{15}$ in Fig. 4). Otherwise, $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_1a_{13}|a_{15}$ in Fig. 4).
iii. $(p_B(x) \neq p_B(y)) \wedge (\mu \neq r(T_i))$:

A. $(p_F(x) = p_F(z)) \wedge (h_F(z) \leq h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$, thus $h_F(x) = h_F(y)$. Using Corollary 2.4, if $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at $q_F(x)$ and u. Hence, $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_8$ in Fig. 4). Otherwise, $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_1a_{25}|a_{22}$ in Fig. 4).
B. $(p_F(x) = p_F(z)) \wedge (h_F(z) > h_F(x))$: Since B is rooted at a vertex of F, we have $q_F(x) = q_F(y)$ and $h_F(x) = h_F(y)$. Then, there exists a vertex u in B such that $xy|z$ is rooted at $q_F(z)$ and u, thus $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_{21}$ in Fig. 4).
C. $p_F(x) \neq p_F(z)$: Using Corollary 2.4, if $p_B(x)p_B(y)|r'$ is consistent with C_B, where r' is the dummy leaf in C_B, then there exists a vertex u in B such that $xy|z$ is rooted at either $r(B)$ and u, or $q_F(z)$ and u, or $r(F)$ and u. If $p_F(x)p'_F(x)|p_F(z)$ is consistent with C_F, then w.l.o.g. if $h_F(x) > h_F(y)$ we have that $xy|z$ is rooted at some vertex u of F and $q_F(x)$. In both cases, $xy|z$ is consistent with N_i (e.g., $a_1a_4|a_{12}$ in Fig. 4).

If both cases are false, $xy|z$ is not rooted at any pair of vertices in N_i, thus $xy|z$ is not consistent with N_i (e.g., $a_1a_{25}|a_{26}$ in Fig. 4).

□

3.3 Triplet Distance Computation

Our second algorithm for computing the triplet distance between two given networks N_1 and N_2 is listed in Algorithm 6. It has the same basic structure as the algorithm in Sect. 2.2, but it applies the procedures presented in Sect. 3.2.1 and 3.2.2 to check triplet consistency. The main procedure is named $D()$. In the preprocessing step, for $i \in \{1, 2\}$, the algorithm builds the block tree T_i, an $n \times n$ table for T_i in order to later answer lowest common ancestor queries between pairs of leaves in T_i in O(1) time, all the contracted block networks of N_i, and finally, for every block B, the fan graph C^f_B and the resolved graph C^r_B as well as the corresponding A^f_B- and A^r_B-tables for the contracted block network C_B. The algorithm then calls the procedure $S()$ to count shared fan and resolved triplets, which is done by enumerating all possible triplets and calling IsFan and IsResolved to see which of them are consistent with both N_1 and N_2. The final answer is calculated according to Equation (1.1).

Algorithm 6 Computing $D(N_1, N_2)$ by using the data structures from Section 3.

procedure PREPROCESSING(N_1, N_2) ▷ Building the data structures
    for $i \in \{1, 2\}$ **do**
        Build T_i using Lemma 3.4
        Build an $n \times n$ table to support lca queries between pairs of leaves in T_i
        Build all the contracted block networks of N_i in accordance with Lemma 3.6
        for every block B
            Build the graphs C_B^f and C_B^r and the tables A_B^f and A_B^r as in Lemma 3.7

procedure S_f(N_1, N_2) ▷ Finding the shared fan triplets
    $sharedFan = 0$
    for $X \subseteq \Lambda$ with $|X| = 3$ **do**
        Let x, y, z be the elements of X in any order
        if ISFAN($x|y|z$, N_1) $\wedge$ ISFAN($x|y|z$, N_2) **then** $sharedF = sharedF + 1$
    return $sharedF$

procedure S_r($N_1 N_2$) ▷ Finding the shared resolved triplets
    $sharedR = 0$
    for $X \subseteq \Lambda$ with $|X| = 3$ **do**
        Let x, y, z be the elements of X in any order
        if ISRESOLVED($xy|z$, N_1) $\wedge$ ISRESOLVED($xy|z$, N_2) **then**
            $sharedR = sharedR + 1$
        if ISRESOLVED($xz|y$, N_1) $\wedge$ ISRESOLVED($xz|y$, N_2) **then**
            $sharedR = sharedR + 1$
        if ISRESOLVED($yz|x$, N_1) $\wedge$ ISRESOLVED($yz|x$, N_2) **then**
            $sharedR = sharedR + 1$
    return $sharedR$

procedure S(N_1, N_2) ▷ Finding the shared triplets
    return $S_f(N_1, N_2) + S_r(N_1, N_2)$

procedure $D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))$ ▷ Computing $D(N_1, N_2)$
    PREPROCESSING(N_1, N_2)
    return $S(N_1, N_1) + S(N_2, N_2)$ - $2S(N_1, N_2)$

From Lemma 3.4, computing T_1 and T_2 requires $O(|E_1| + |E_2|)$ time. Building the two tables for answering lowest common ancestor queries in T_1 and T_2 takes $O(n^2)$ time by bottom-up traversals. From Lemma 3.6, constructing all the contracted block networks requires $O(|E_1| + |E_2| + n^2)$ time. From Lemma 3.7, the total time required to build C_B^f, C_B^r, A_B^f, and A_B^r for every block B of N_1 and N_2 is $O(|V_1|(k_1^2 d_1^2 + 1) + |V_2|(k_2^2 d_2^2 + 1))$. Since $|V_i| = O(|E_i|)$, the preprocessing time sums up to $O(|E_1| + |E_2| + |V_1|k_1^2 d_1^2 + |V_2|k_2^2 d_2^2 + n^2)$.

Using Lemmas 3.9 and 3.11, after the preprocessing step we can determine the consistency of a triplet with N_1 or N_2 in $O(1)$ time. Since the number of triplets that need to be checked is exactly $4\binom{n}{3}$, the total running time of the algorithm is $O(|E_1| + |E_2| + |V_1|k_1^2 d_1^2 + |V_2|k_2^2 d_2^2 + n^3)$. Using the definitions of N, M, k, and d from Sect. 1, the running time can be expressed as $O(M + Nk^2d^2 + n^3)$. Hence, we obtain the following theorem:

Theorem 3.12 *The triplet distance between two networks* N_1 *and* N_2 *can be computed in* $\mathrm{O}(M + Nk^2d^2 + n^3)$ *time.*

4 Implementation and Experiments

This section presents the implementations of the two algorithms from Sects. 2 and 3, and experimental results demonstrating their practical performance. Both simulated and real datasets were used in the experiments.

4.1 Algorithm Implementation

From here on, the algorithm from Sect. 2 will be referred to as `NTDfirst` and the algorithm from Sect. 3 as `NTDsecond`. Both algorithms were implemented in the C++ programming language and the source code is publicly available at:

`https://github.com/kmampent/ntd`

Since no other implementations for computing the rooted triplet distance between two networks of arbitrary levels are available, the correctness of our program code was verified by trying a large number of pairs of input networks under varying parameters and making sure that the output of `NTDfirst` (which is simple to implement) was identical to the output of `NTDsecond` in all cases.

4.2 The Setup

The experiments were performed on a machine with 16GB RAM and Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz. The operating system was Ubuntu 16.04.2 LTS, and the compiler used was g++ 5.4 with cmake 3.11.0.

4.3 Experiment 1: Performance

The first set of experiments were designed to measure the running times and memory usage of our implementations of `NTDfirst` and `NTDsecond`. To do so systematically, we used simulated datasets. **The Input.** Given three parameters n, p, and e, where $n \geq 1$ is an integer, $0 \leq p \leq 1$, and $e \geq 0$ is an integer, an input network N' was built according to the following method:

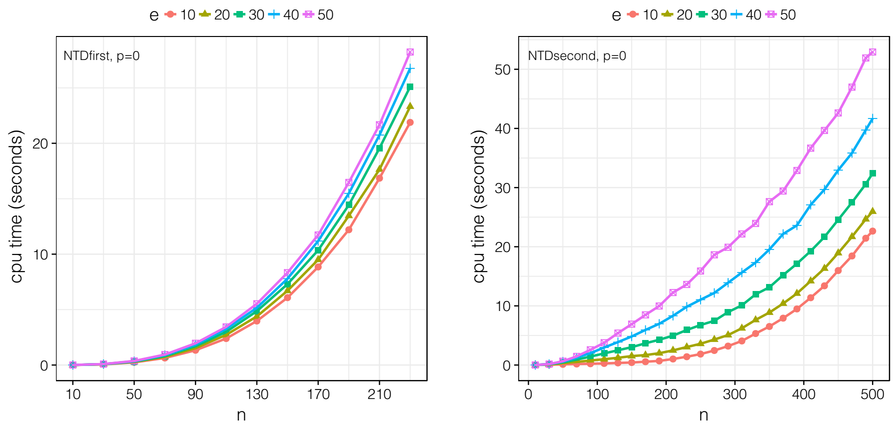
- Generate a random rooted binary tree T with n leaves in the uniform model [29].
- For each internal vertex w in T except $r(T)$, contract the edge between the parent of w and w with probability p.
- For each vertex w in T, let $d(w)$ be the number of edges on the path from $r(T)$ to w. Let $N' = T$.

- Until e edges have been added or it is impossible to add any more edges: Add an edge between two vertices in N' chosen uniformly at random, under the constraint that an edge $u \to v$ is created in N' only if $d(u) < d(v)$. (In other words, if the total number of edges that can be added is y and $y < e$, then only add those y edges.)

Experimental Results. We applied `NTDfirst` and `NTDsecond` to pairs of networks generated with the method above for varying values of n, p, and e, and measured their running times and memory usage. In the graphs shown below, every data point corresponds to the average taken over 30 runs with a set of fixed parameters. Reticulation events are typically rare in nature [30], so we used relatively small values for e, i.e., $e \leq 50$ when $n \leq 500$, to make the experiments more realistic.

The results of Experiment 1 are reported below.

1. The two algorithms' running times and memory usage increase as n increases according to the plots in Figs. 7 and 8. The first figure shows the CPU time in seconds taken when $p = 0$ and $e \in \{10, 20, 30, 40, 50\}$. For `NTDfirst` we used $10 \leq n \leq 230$, and for `NTDsecond` we used $10 \leq n \leq 500$. Space is the reason behind the restrictions on n. As can be seen in Fig. 8a, at $n = 230$ the memory usage of `NTDfirst` is getting close to the limit of the available 16GB RAM. When $n \geq 240$, the memory requirements exceed the limit, and the operating system initiates highly time-consuming communication with the disk.
2. Both algorithms take more time as the parameter e increases due to the additional edges in the generated networks, with `NTDsecond` suffering more than `NTDfirst`. Again, see Fig. 7. The explanation for this behavior is as follows. The main purpose of extending the algorithm from Sect. 2 in Sect. 3 was to avoid having



(a) **(b)**

Fig. 7 The running times of `NTDfirst` and `NTDsecond` for increasing values of n and with $p = 0$ and $e \in \{10, 20, 30, 40, 50\}$

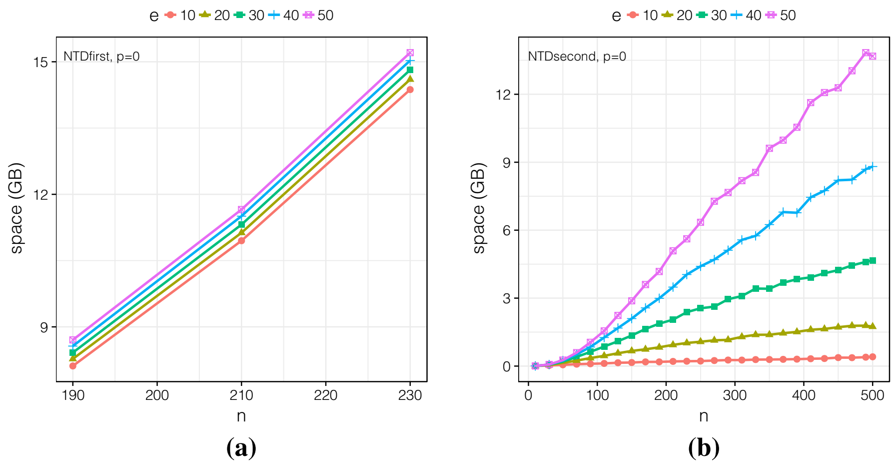


Fig. 8 The memory usage of the two algorithms for increasing values of n and with $p = 0$ and $e \in \{10, 20, 30, 40, 50\}$, as reported by the *Maximum Resident Size* parameter when calling the executable of each algorithm with `/usr/bin/time -v`

to build the highly time- and memory-consuming fan and resolved graph on the entire input network, and instead build several such graphs on smaller blocks. Figure 9 shows that a larger value of e implies a higher level k as well as fewer non-leaf blocks in N', which in turn implies more time spent by `NTDsecond` building the fan and resolved graphs. An extreme situation is when e is so large that N' has a really small number of non-leaf blocks, one of which is roughly as large as N' itself. Then, given that the preprocessing of `NTDsecond` is more complex than that of `NTDfirst`, `NTDsecond` will be slower than `NTDfirst`.

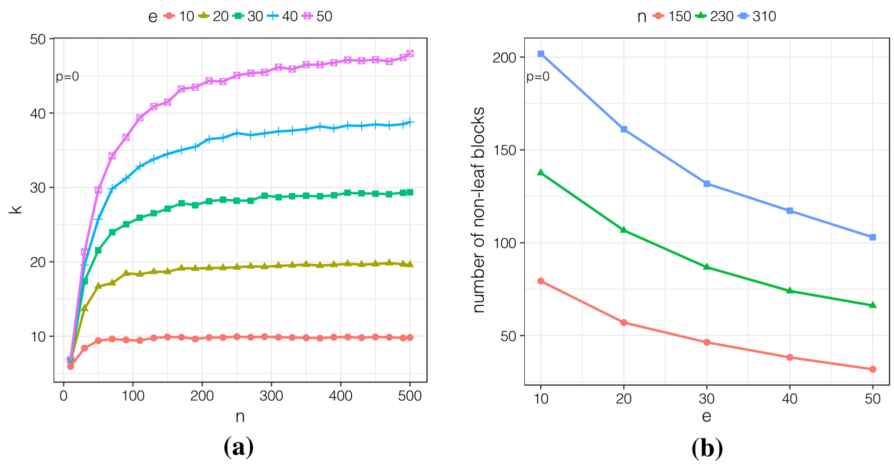
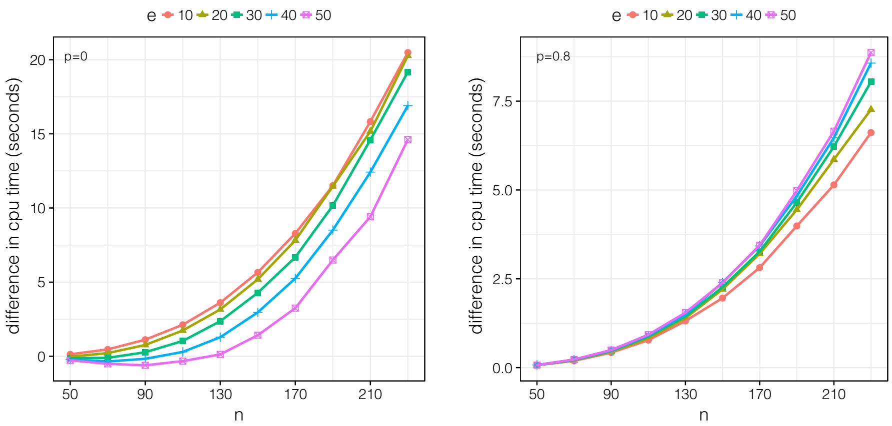


Fig. 9 The effect of e and n on k (the generated network's level) and the amount of non-leaf blocks



(a) **(b)**

Fig. 10 The running time of `NTDfirst` minus the running time of `NTDsecond` for $e \in \{10, 20, 30, 40, 50\}$ and $p \in \{0, 0.8\}$. **a** Observe that when $n = 90$, $p = 0$, and $e = 50$, the difference is negative, which means `NTDfirst` is faster than `NTDsecond`. **b** When p is large (like the case $p = 0.8$ shown here), the number of edges that can be added to the generated networks is small and the differences in running times for varying values of e less significant

An example of where this happens can be found in Fig. 10a when the parameters are $n = 90$, $p = 0$, and $e = 50$.

In contrast, when p is large, e.g., $p = 0.8$ in Fig. 10b, the effect of e on the running times is small. This holds especially for `NTDsecond`. There will be fewer internal vertices in the generated networks, which means that the number of edges that can be added decreases as well.

3. The effect of the parameter p on the relative running times of the two algorithms is shown in Fig. 11. In general, the difference in the two algorithms' running times becomes smaller as the value of p increases. For certain combinations of the parameters such as $n = 90$, $p = 0$, and $e = 50$ in Fig. 11c, `NTDfirst` is faster than `NTDsecond`, as observed earlier.

4.4 Experiment 2: Limitations of the Rooted Triplet Distance

The second set of experiments applied the algorithms to real datasets. The goal was to see how informative the current definition of the rooted triplet distance is in practice when comparing phylogenetic networks, and to investigate any potential shortcomings. **The Input.** For the real datasets, we borrowed six networks from Table S4 in [31] that describe biologically motivated alternative 'scenarios' for the evolutionary history of the *Viola* genus. They are named N_A, N_B, N_C, N_D, N_E, and N_F below. The first five networks correspond to the five scenarios A, B, C, D, and E in [31], and N_F is "Scenario E, CHAM and MELVIO resolved", which is actually the same as scenario E but with two of the subclades (overlapping subtrees) expanded.

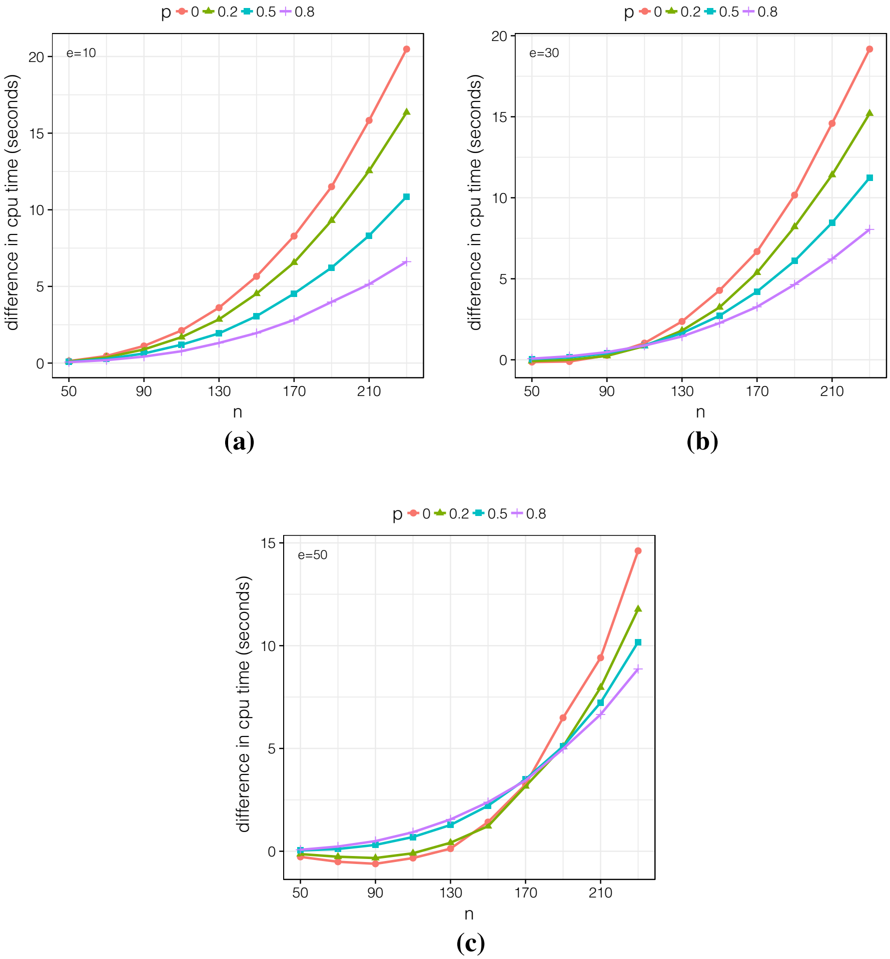
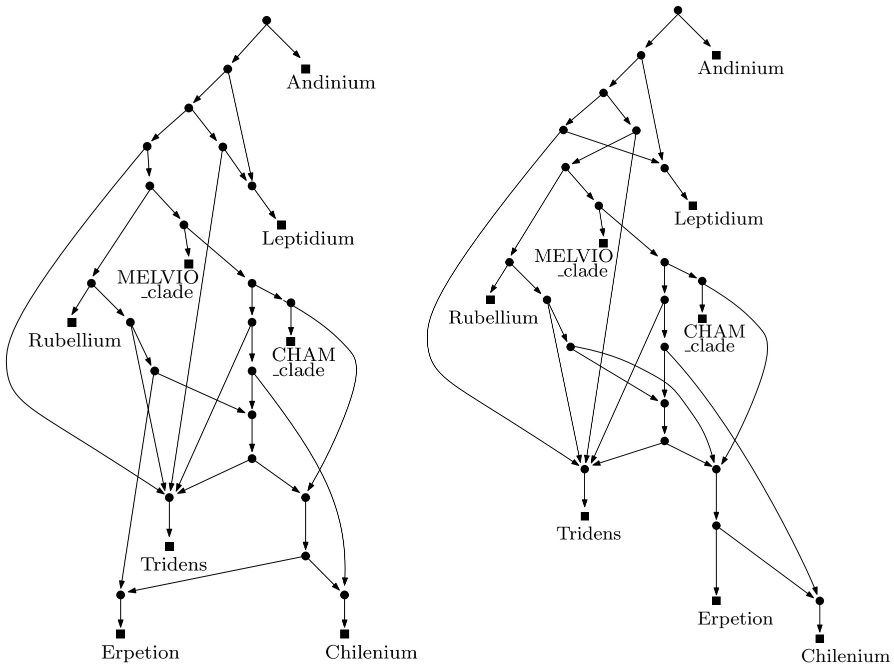


Fig. 11 The effect of different values of p on the running time of `NTDfirst` minus the running time of `NTDsecond`, for $e \in \{10, 30, 50\}$. When $n = 90$, $p = 0$, and $e = 50$, `NTDfirst` is faster than `NTD-second`

Only two of the six networks are shown here; the network N_B is displayed in Fig. 12a, and N_D in Fig. 12b. For the other four networks' branching structures, the reader is referred to Table S4 in [31].

The networks in Table S4 in [31] were inferred from a set of multilabeled trees. (A *multilabeled tree* is a generalization of a phylogenetic tree in which identical leaf labels are allowed to occur more than once.) The method that was used to construct the networks is explained in detail in Step 3 ("Inference of the Most Parsimonious Network from Multilabeled Gene Trees") in the Materials and Methods-section of [31]. Table S4 in [31] also provides these multilabeled trees. In order to represent the multilabeled trees as distinctly leaf-labeled trees as well, [31] replaced any repeated leaf label x by unique leaf labels of the form $x.1, x.2, \dots, x.i$; e.g., one



(a) The network N_B. **(b)** The network N_D.

Fig. 12 The networks N_B and N_D from [31]

occurrence of the leaf label `Tridens` was changed to `Tridens`.1, another one to `Tridens`.2, another one to `Tridens`.3, and so on. These (distinctly leaf-labeled) trees were also considered in our experiments and are referred to as T_A, T_B, T_C, T_D, T_E, and T_F.

The size of the leaf label set of T_A, T_B, T_C, T_D, T_E, and T_F is 16, 20, 21, 21, 22, and 50 leaves, respectively. For every $s \in \{A, B, C, D, E\}$, N_s contains 8 leaves, and N_F contains 16 leaves. Note that for all $s \in \{A, B, C, D, E, F\}$, the number of leaf labels in T_s is larger than than the number of leaf labels in N_s due to the leaf relabeling process just described to obtain distinctly leaf-labeled trees.

In our implementations, the input trees are represented in standard Newick format and the input networks in extended Newick format [32]. We employ the graph-theoretic standard adjacency list to store the input networks, making it easy to support different input formats at the same time.

Experimental Results. We used the trees T_s and networks N_s, where $s \in \{A, B, C, D, E, F\}$, from Table S4 in [31], as explained above. In the experiments, we computed the rooted triplet distance between each T_s and N_s and also between pairs of these networks. According to Equation (1.1), $D(T_s, N_s) = S(T_s, T_s) + S(N_s, N_s) - 2S(T_s, N_s)$. To make $\mathcal{L}(T_s) = \mathcal{L}(N_s)$ when computing $D(T_s, N_s)$, if a leaf x in N_s appeared as several leaves $x.1, \dots, x.i$ in T_s then we replaced x in N_s by leaves labeled $x.1, \dots, x.i$, attaching each of them as a child of the parent of x. The maximum time spent by any of our algorithms was when

Table 2 Experiments on the real datasets

s	$S(T_s, T_s)$	$S(N_s, N_s)$	$S(T_s, N_s)$	$D(T_s, N_s)$
A	560	716	443	390
B	1140	1870	840	1330
C	1330	2185	965	1585
D	1330	2205	964	1607
E	1540	1996	983	1570
F	19,600	43,710	16,553	30,204

The computed values of $S(T_s, T_s)$, $S(N_s, N_s)$, $S(T_s, N_s)$, and $D(T_s, N_s)$

Table 3 Experiments on the real datasets, continued

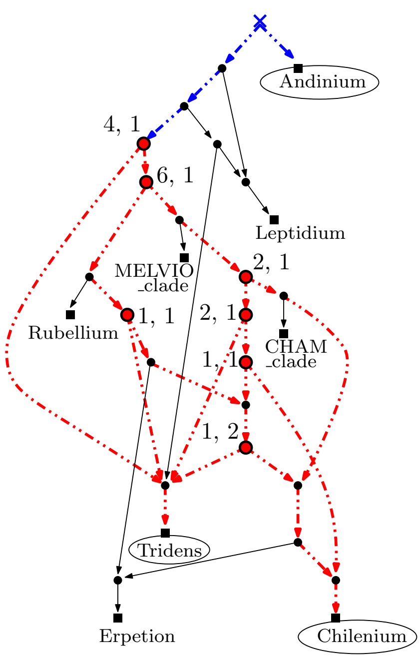
	N_A	N_B	N_C	N_D	N_E
N_A	0	20	19	20	10
N_B	20	0	1	0	10
N_C	19	1	0	1	9
N_D	20	0	1	0	10
N_E	10	10	9	10	0

The computed values of $D(N_s, N_{s'})$. In particular, observe that $D(N_B, N_D) = 0$

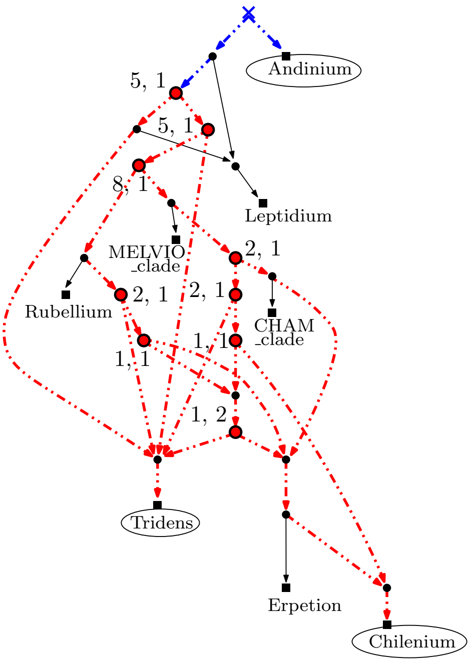
computing $D(T_F, N_F)$, with `NTDfirst` requiring only 0.18 seconds to run and `NTD-second` 0.05 seconds.

Our findings are summarized in Tables 2 and 3. By inspecting the tables, Experiment 2 reveals two ways that the current definition of the rooted triplet distance for networks could be improved:

1. Table 2 shows $S(T_s, T_s)$, $S(N_s, N_s)$, $S(T_s, N_s)$, and $D(T_s, N_s)$ for every $s \in \{A, B, C, D, E, F\}$. The values of $D(T_s, N_s)$ seem quite large compared to the number of triplets in each T_s (given by $S(T_s, T_s)$). This is because of the resolved triplets that arise when N_s is created from a multilabeled tree using the method in [31], and the fan triplets that are created whenever a leaf x is replaced by $x.1, \ldots, x.i$ in N_s. Consequently, it would be desirable to give less weights to such triplets. A more flexible definition of the rooted triplet distance that can assign different weights to different triplets could therefore be useful.
2. Next, Table 3 lists the triplet distance $D(N_s, N_{s'})$ for all pairs $s, s' \in \{A, B, C, D, E\}$. The networks $N_A, \ldots, N_E$ have identical leaf label sets, but the leaf label set of N_F is different, which is why N_F is excluded from Table 3. Interestingly, although the two networks N_B and N_D are structurally different (see Fig. 12), their triplet distance is 0. This suggests that alternative definitions of the rooted triplet distance for networks may be better in practice, as discussed in Sect. 5 below.



(a) The network N_B.



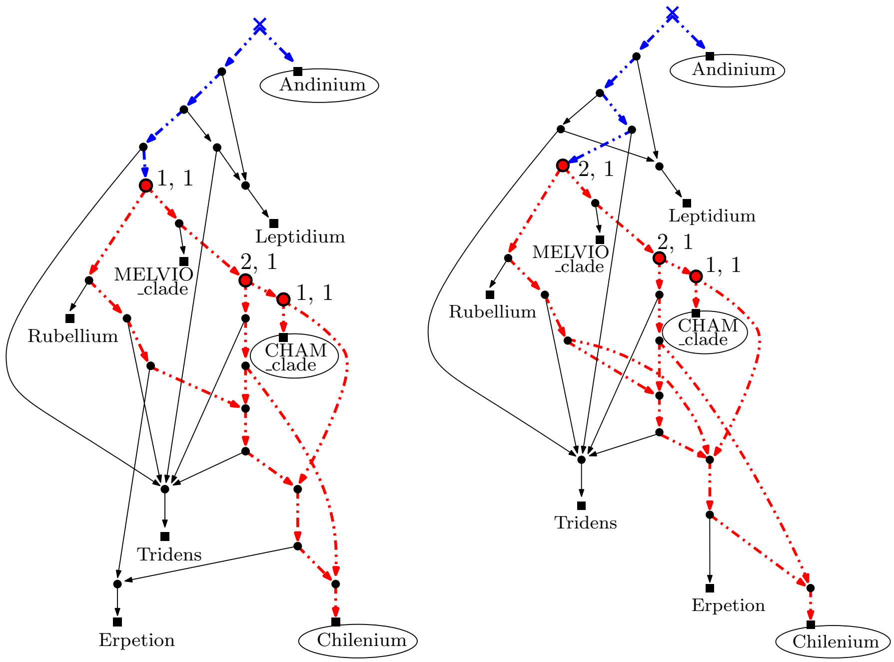
(b) The network N_D.

Fig. 13 Next to every vertex marked with a circle is the number of different pairs of disjoint paths from that vertex to the leaves with labels `Tridens` and `Chilenium`, and the number of different disjoint paths from the root to the vertex. With definition A of multiplicity for resolved triplets, the resolved triplet `Tridens Chilenium | Andinium` appears $(4 + 6 + 2 + 1 + 2 + 1) \cdot 1 + 1 \cdot 2 = 18$ times in N_B and $(5 + 5 + 8 + 2 + 2 + 2 + 1 + 1) \cdot 1 + 1 \cdot 2 = 28$ times in N_D. With definition B, this triplet appears 7 times in N_B and 9 times in N_D

5 Final Remarks

We have developed two new algorithms for computing the rooted triplet distance between two phylogenetic networks over the same leaf label set. We have also presented an implementation of the algorithms and evaluated their performance on simulated and real datasets.

Future work involves creating new algorithms that are even more efficient than the algorithms given here, as well as to research variants of the studied problem that may provide more biologically meaningful ways for comparing networks. An example of such a variant is motivated by the experiments on the real dataset in Sect. 4.4. Recall that the two networks N_B and N_D were structurally different, yet their triplet distance was 0. The reason is that, unlike in the case of trees, the same triplet can appear several times in a network, and for two networks N_1 and N_2 to be compared, if a triplet appears 1000 times in N_1 and only once in N_2, it would contribute 0 under



(a) The network N_B. **(b)** The network N_D.

Fig. 14 Next to every vertex marked with a circle is the number of different pairs of disjoint paths from that vertex to the leaves with labels `Chilenium` and `CHAM_clade`, and the number of different disjoint paths from the root to the vertex. With definition A of multiplicity for resolved triplets, the resolved triplet `Chilenium CHAM_clade | Andinium` appears $(1 + 2 + 1) \cdot 1 = 4$ times in N_B and $(2 + 2 + 1) \cdot 1 = 5$ times in N_D. With definition B, this triplet appears three times in both networks

the current definition of $D(N_1, N_2)$. However, extending the definition of the triplet distance for networks to capture information about the frequencies of triplets in the networks can be done in different ways, leading to different outcomes. For example, consider the following two alternative definitions of *multiplicity* for a resolved triplet $xy|z$, where u and v are the vertices used in the definition of the consistency of a resolved triplet with a network in Sect. 1:

A. The total number of quadruples of paths of the form $((u \rightsquigarrow v), (v \rightsquigarrow x), (v \rightsquigarrow y), (u \rightsquigarrow z))$ that are disjoint except for in u and v, and furthermore, the path from u to z does not pass through v.
B. The total number of pairs of vertices (u, v) such that there exist four paths of the form $(u \rightsquigarrow v), (v \rightsquigarrow x), (v \rightsquigarrow y), (u \rightsquigarrow z)$ that are disjoint except for in u and v, and furthermore, the path from u to z does not pass through v.

The definitions for the case of fan triplets are analogous. Now consider the two networks N_B and N_D. As shown in Fig. 13, if we follow definition A of multiplicity, the resolved triplet `Tridens Chilenium | Andinium` appears 18 times in N_B and 28 times in N_D (and we could thus let it contribute 10 to the extended rooted triplet distance). If we choose definition B instead, this resolved triplet appears 7 times in N_B and 9 times in N_D. On the other hand, according to Fig. 14, the resolved triplet `Chilenium CHAM_clade | Andinium` appears 4 times in N_B and 5 times in N_D according to definition A, but 3 times in both networks according to definition B.

In summary, definition B seems somewhat simpler to compute than definition A, but it fails to distinguish between certain cases that definition A can handle. To determine under what circumstances definition B is good enough in practice is an open problem and a future research topic.

Finally, Cardona *et al.* [33] gave an alternative generalization of the rooted triplet distance from trees to networks. While the extension proposed by Gambette and Huber [13] is closer to the definition of the widely studied rooted triplet distance for trees, efficient algorithms for Cardona *et al.*'s extension might also be useful. However, as pointed out in [13] and [33], neither one of them yields a metric for all classes of phylogenetic networks (see Corollary 1 in [13] and Figs. 19 and 20 in [33]), so another open problem is to find even more informative generalizations.

Acknowledgements JJ was partially funded by RGC/GRF project 15221420. KM acknowledges the support by the Danish National Research Foundation, grant DNRF84, via the Center for Massive Data Algorithmics (MADALGO).

References

1. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates Inc, Sunderland (2004)
2. Nakhleh, L., Sun, J., Warnow, T., Linder, C. R., Moret, B. M. E., Tholse, A.: Towards the development of computational tools for evaluating phylogenetic network reconstruction methods. In Proceedings of the 8th Pacific Symposium on Biocomputing (PSB 2003), pp. 315–326, 2003
3. Robinson, D.F., Foulds, L.R.: Comparison of phylogenetic trees. Math. Biosci. **53**(1), 131–147 (1981)
4. Dobson, A. J.: Comparing the shapes of trees. In *Combinatorial Mathematics III*, pp. 95–100. Springer, Berlin (1975)
5. Estabrook, G.F., McMorris, F.R., Meacham, C.A.: Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. Syst. Zool. **34**(2), 193–200 (1985)
6. Moore, G.W., Goodman, M., Barnabas, J.: An iterative approach from the standpoint of the additive hypothesis to the dendrogram problem posed by molecular data sets. J. Theor. Biol. **38**(3), 423–457 (1973)
7. Robinson, D.F.: Comparison of labeled trees with valency three. J. Combin. Theory B **11**(2), 105–119 (1971)

8. Penny, D., Watson, E.E., Steel, M.A.: Trees from languages and genes are very similar. Syst. Biol. **42**(3), 382–384 (1993)
9. Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. Dis. Appl. Math. **71**(1), 153–169 (1996)
10. Finden, C.R., Gordon, A.D.: Obtaining common pruned trees. J. Class. **2**(1), 255–276 (1985)
11. McVicar, M., Sach, B., Mesnage, C., Lijffijt, J., Spyropoulou, E., De Bie, T.: SuMoTED: an intuitive edit distance between rooted unordered uniquely-labelled trees. Pattern Recog. Lett. **79**, 52–59 (2016)
12. Huson, D.H., Rupp, R., Scornavacca, C.: Phylogenetic Networks: Concepts Algorithms and Applications. Cambridge University Press, Cambridge (2010)
13. Gambette, P., Huber, K.T.: On encodings of phylogenetic networks of bounded level. J. Math. Biol. **65**(1), 157–180 (2012)
14. Choy, C., Jansson, J., Sadakane, K., Sung, W.-K.: Computing the maximum agreement of phylogenetic networks. Theor. Comput. Sci. **335**(1), 93–107 (2005)
15. Gusfield, D., Eddhu, S., Langley, C.: Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. J. Bioinform. Comput. Biol. **2**(1), 173–213 (2004)
16. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM **16**(6), 372–378 (1973)
17. Jansson, J., Lingas, A.: Computing the rooted triplet distance between galled trees by counting triangles. J. Dis. Algor. **25**, 66–78 (2014)
18. Bansal, M.S., Dong, J., Fernández-Baca, D.: Comparing and aggregating partially resolved trees. Theor. Comput. Sci. **412**(48), 6634–6652 (2011)
19. Brodal, G. S., Fagerberg, R., Pedersen, C. N. S., Mailund, T., Sand, A.: Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp 1814–1832. Society for Industrial and Applied Mathematics, 2013
20. Brodal, G. S., Mampentzidis, K.: Cache oblivious algorithms for computing the triplet distance between trees. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp 21:1–21:14. Schloss Dagstuhl'Leibniz-Zentrum fuer Informatik, 2017
21. Critchlow, D.E., Pearl, D.K., Qian, C.L.: The triples distance for rooted bifurcating phylogenetic trees. Syst. Biol. **45**(3), 323–334 (1996)
22. Griebel, T., Brinkmeyer, M., Böcker, S.: EPoS: a modular software framework for phylogenetic analysis. Bioinformatics **24**(20), 2399–2400 (2008)
23. Jansson, J., Rajaby, R.: A more practical algorithm for the rooted triplet distance. J. Comput. Biol. **24**(2), 106–126 (2017)
24. Sand, A., Holt, M.K., Johansen, J., Brodal, G.S., Mailund, T., Pedersen, C.N.S.: tqDist: a library for computing the quartet and triplet distances between binary or general trees. Bioinformatics **30**(14), 2079–2080 (2014)
25. Jansson, J., Rajaby, R., Sung, W.-K.: An efficient algorithm for the rooted triplet distance between galled trees. J. Comput. Biol. **26**(9), 893–907 (2019)
26. Fortune, S., Hopcroft, J., Wyllie, J.: The directed subgraph homeomorphism problem. Theor. Comput. Sci. **10**(2), 111–121 (1980)
27. Byrka, J., Gawrychowski, P., Huber, K.T., Kelk, S.: Worst-case optimal approximation algorithms for maximizing triplet consistency within phylogenetic networks. J. Dis. Algor. **8**(1), 65–75 (2010)
28. Perl, Y., Shiloach, Y.: Finding two disjoint paths between two pairs of vertices in a graph. J. ACM **25**(1), 1–9 (1978)
29. McKenzie, A., Steel, M.: Distributions of cherries for two models of trees. Math. Biosci. **164**(1), 81–92 (2000)
30. Bordewich, M., Semple, C.: Computing the minimum number of hybridization events for a consistent evolutionary history. Dis. Appl. Math. **155**(8), 914–928 (2007)
31. Marcussen, T., Heier, L., Brysting, A.K., Oxelman, B., Jakobsen, K.S.: From gene trees to a dated allopolyploid network: insights from the angiosperm genus *Viola* (Violaceae). Syst. Biol. **64**(1), 84–101 (2015)
32. Cardona, G., Rosselló, F., Valiente, G.: Extended Newick: it is time for a standard representation of phylogenetic networks. BMC Bioinform. **9**(1), 532 (2008)

33. Cardona, G., Llabres, M., Rossello, F., Valiente, G.: Metrics for phylogenetic networks II: nodal and triplets metrics. IEEE/ACM Trans. Comput. Biol. Bioinform. **6**(3), 454–469 (2009)

Authors and Affiliations

Jesper Jansson[1] · Konstantinos Mampentzidis[2] · Ramesh Rajaby[3] · Wing-Kin Sung[3,4]

Konstantinos Mampentzidis
kmampent@cs.au.dk

Ramesh Rajaby
e0011356@u.nus.edu

Wing-Kin Sung
ksung@comp.nus.edu.sg

[1] Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

[2] Department of Computer Science, Aarhus University, Aarhus, Denmark

[3] School of Computing, National University of Singapore, 13 Computing Drive, Genome 117417, Singapore

[4] Genome Institute of Singapore, 60 Biopolis Street, Genome 138672, Singapore