Computing the Rooted Triplet Distance Between Phylogenetic Networks

Jesper Jansson[1], Konstantinos Mampentzidis[2], Ramesh Rajaby[3], and Wing-Kin Sung[3(✉)]

[1] The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
jesper.jansson@polyu.edu.hk
[2] Department of Computer Science, Aarhus University, Aarhus, Denmark
kmampent@cs.au.dk
[3] School of Computing, National University of Singapore, Singapore, Singapore
e0011356@u.nus.edu, ksung@comp.nus.edu.sg

Abstract. The *rooted triplet distance* measures the structural dissimilarity of two phylogenetic trees or networks by counting the number of rooted trees with exactly three leaf labels 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 rooted phylogenetic networks over a common leaf label set of size λ, 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. Denote $n = \max(|V_1|, |V_2|)$, $m = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$, 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 $O(\lambda \log \lambda)$ time in the special case $k \leq 1$. For $k > 1$, no efficient algorithms are known; a trivial approach leads to a running time of $\Omega(n^7\lambda^3)$ 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 nodes are allowed). In this paper, we develop two new algorithms that have no such restrictions. Their running times are $O(n^2m + \lambda^3)$ and $O(m + k^3d^3\lambda + \lambda^3)$, respectively. We also provide implementations of our algorithms and evaluate their performance in practice. This is the first publicly available software for computing the rooted triplet distance between unrestricted networks of arbitrary levels.

1 Introduction

Background. Trees are commonly used in biology to represent evolutionary relationships, with the leaves corresponding to species that exist today and internal nodes to ancestor species that existed in the past. When studying the evolution of a fixed set of species, different available data and tree construction methods [7] 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 tree distance measures in the literature, e.g., the Robinson-Foulds distance [17], the rooted triplet distance [5] for rooted trees, and the unrooted quartet distance [6] for unrooted trees.

C. J. Colbourn et al. (Eds.): IWOCA 2019, LNCS 11638, pp. 290–303, 2019.
https://doi.org/10.1007/978-3-030-25005-8_24

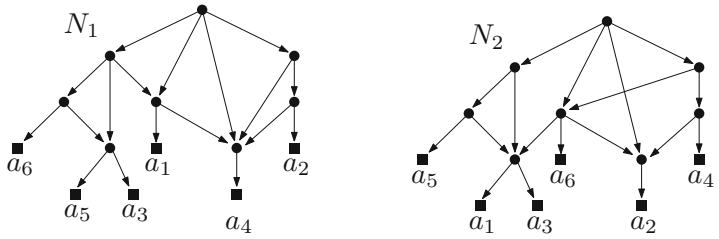

Fig. 1. N_1 is a level-2 network and N_2 is a level-3 network with $\mathcal{L}(N_1) = \mathcal{L}(N_2) = \{a_1, a_2, \ldots, a_6\}$. In this example, $D(N_1, N_2) = 33$. Some shared triplets are: $a_2|a_4|a_6$, $a_2a_4|a_6$, $a_4a_6|a_2$. Some triplets consistent with only one network are: $a_1|a_3|a_6$, $a_2a_6|a_4$.

A rooted phylogenetic network is an extension of a *rooted phylogenetic tree* (i.e., a rooted, unordered, distinctly leaf-labeled tree with no degree-1 nodes) that allows internal nodes to have more than just one parent. Such networks are designed to capture more complex evolutionary relationships when reticulation events such as horizontal gene transfer and hybridization are involved. Similarly to phylogenetic trees, it becomes useful to have distance measures for comparing phylogenetic networks. In this paper we study a natural extension, by Gambette and Huber [10], of the rooted triplet distance from the case of rooted phylogenetic trees to the case of rooted level-k phylogenetic networks.

Problem Definitions. A *rooted phylogenetic network* $N = (V, E)$ is a rooted, directed acyclic graph with one root (a node with in-degree 0), distinctly labeled leaves, and no nodes with both in-degree 1 and out-degree 1. 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 a node u in N, let $in(u)$ and $out(u)$ be the in-degree and out-degree of u. The network N can have three types of nodes. A node u is an *internal node* if $out(u) \geq 1$, a *leaf node* if $in(u) = 1$ and $out(u) = 0$, and a *reticulation node* if $out(u) \geq 1$ and $in(u) \geq 2$. By definition, N cannot have a node u with $in(u) > 1$ and $out(u) = 0$. Let $r(N)$ be the root of N and $\mathcal{L}(N)$ the set of leaves in N. A directed edge from a node u to a node v in N is denoted by $u \to v$. A path from u to v in N is denoted by $u \rightsquigarrow v$. Let the height $h(u)$ be the length (number of edges) of the longest path from u to a leaf in N. By definition, if v is a parent of u in N, we have $h(v) > h(u)$.

Let $U(N)$ be the undirected graph created by replacing every directed edge in N with an undirected edge. An undirected graph H is called *biconnected* if it has no node whose removal makes H disconnected. We call H' a *biconnected component of* $U(N)$ if H' is a maximal subgraph of $U(N)$ that is biconnected. The biconnected components of $U(N)$ are edge-disjoint but not necessarily node-disjoint. We say that N is a *level-k network*, equivalently N *has level* k, if every biconnected component of $U(N)$ contains at most k reticulation nodes. The level of a network was introduced by Choy *et al.* [4] as a parameter to measure the treelikeness of a network, with the special case of a level-0 network corresponding

to a tree and a level-1 network a *galled tree* [11]. Figure 1 shows a level-2 and a level-3 network.

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*. Following [13] and similarly to the case of trees in [1], for a network N we say that the rooted fan triplet $x|y|z$ is *consistent with* N, if there exists an internal node u in N and three directed paths of non-zero length that are node-disjoint, except for u, one going from u to x, one from u to y and one from u to z. Similarly, we say that the rooted resolved triplet $xy|z$ is *consistent with* N, if N contains two internal nodes u and v such that $u \neq v$, and there are four directed paths of non-zero length that are node-disjoint, except for u and v, one going from u to v, one from v to x, one from v to y and one from u to z. See Fig. 1 for an example. From here on, by "disjoint paths" we imply "node-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 of size 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 the two input networks [10] (see also [12, Sect. 3.2] for a discussion). Let $S(N_1, N_2)$ be the total number of triplets that are consistent with both N_1 and N_2, commonly referred to as *shared triplets*. We then have:

$$D(N_1, N_2) = S(N_1, N_1) + S(N_2, N_2) - 2S(N_1, N_2) \tag{1}$$

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 $a_2|a_4|a_6$ 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 $S(N_1, N_2)$, e.g., $a_1|a_3|a_6$ from 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 total number of resolved and fan triplets respectively that are consistent with both N_1 and N_2. We then have that $S(N_1, N_2) = S_r(N_1, N_2) + S_f(N_1, N_2)$.

We define the following notation that we use from here on. A network N_i is built on a leaf label set of size λ and is defined by the node set V_i and the edge set E_i. Moreover, N_i has level k_i and the maximum in-degree of every node in N_i is d_i. Two given networks N_1 and N_2 are built on the same leaf label set and $n = \max(|V_1|, |V_2|)$, $m = \max(|E_1|, |E_2|)$, $k = \max(k_1, k_2)$ and $d = \max(d_1, d_2)$.

Related Work. Table 1 lists the running times of different algorithms 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, with the fastest algorithm in theory and practice by Brodal *et al.* [1] running in $O(\lambda \log \lambda)$ time. For $k = 1$, an $O(\lambda^{2.687})$-time algorithm based on counting 3-cycles in an auxiliary graph was given in [12], and a faster, $O(\lambda \log \lambda)$-time algorithm that transforms the input to a constant number of instances with $k = 0$ was given in [13]. All algorithms mentioned above allow nodes of arbitrary degree in the input networks. Moreover, software packages implementing the $O(\lambda \log \lambda)$-time algorithms are available.

For $k > 1$, Byrka *et al.* [2] considered the special case of networks whose roots have out-degree 2 and whose other non-leaf nodes have in-degree 2 and

Table 1. Previous and new results for computing $D(N_1, N_2)$, where N_1 and N_2 are two level-k networks built on the same leaf label set of size λ.

Year	Reference	k	Degrees	Time complexity
1980	Fortune *et al.* [8]	Arbitrary	Arbitrary	$\Omega(n^7\lambda^3)$
2010	Byrka *et al.* [2]	Arbitrary	Binary	$O(n^3 + \lambda^3)$
2010	Byrka *et al.* [2]	Arbitrary	Binary	$O(n + k^2n + \lambda^3)$
2017	Brodal *et al.* [1]	0	Arbitrary	$O(\lambda \log \lambda)$
2017	Jansson *et al.* [13]	1	Arbitrary	$O(\lambda \log \lambda)$
2019	New	Arbitrary	Arbitrary	$O(n^2m + \lambda^3)$
2019	New	Arbitrary	Arbitrary	$O(m + k^3d^3\lambda + \lambda^3)$

out-degree 1 or in-degree 1 and out-degree 2. Given such a network $N = (V, E)$, they defined a data structure D that can be constructed in $O(|V|^3)$ time by dynamic programming and then used to determine in $O(1)$ time if any resolved triplet $xy|z$ is consistent with N. This result was then strengthened by obtaining a new data structure D' that requires $O(|V| + k^2|V|)$ construction time, where k is the level of N. If N_1 and N_2 have arbitrary levels and follow the degree constraints of N, D can be used to compute $D(N_1, N_2)$ in $O(n^3 + \lambda^3)$ time and D' can be used to compute $D(N_1, N_2)$ in $O(n + k^2n + \lambda^3)$ time.

Contribution. The data structures D and D' of Byrka *et al.* [2] can only support consistency queries for resolved triplets. However, a network with nodes of arbitrary degree may contain fan triplets. Moreover, D' exploits the fact that given the degree constraints in N, all biconnected components of $U(N)$ are node-disjoint. However, even a small change in these constraints, e.g., if we allow nodes with in-degree 2 to have an out-degree 2 instead of 1, could produce a network with biconnected components that are not node-disjoint, thus making the application of D' impossible.

Without any degree constraints in N_1 and N_2 and when k_1 and k_2 are arbitrary, an algorithm for computing $D(N_1, N_2)$ that iterates over all $4\binom{\lambda}{3}$ triplets and for each triplet applies the pattern matching algorithm in [8] to determine its consistency with N_1 and N_2, has a $\Omega(n^7\lambda^3)$ running time. In this paper we give two algorithms that improve significantly upon this approach. The running time of the first algorithm is $O(n^2m+\lambda^3)$ and the second algorithm $O(m+k^3d^3\lambda+\lambda^3)$. For networks N_1 and N_2 that satisfy the degree constraint in Byrka *et al.* [2], we prove that our algorithms can compute $D(N_1, N_2)$ using the same time complexity as that of Byrka *et al.* [2]. To determine the efficiency of the two algorithms in practice, we provide an implementation as well as extensive experiments on both simulated and real datasets. We note that this is the first publicly available software that can compute the triplet distance between two unrestricted networks of arbitrary levels.

Organization of the Article. In Sect. 2 we present the first algorithm and in Sect. 3 the second algorithm. Section 4 presents an implementation of the two

algorithms as well as experiments illustrating their practical performance. Due to space constraints, the proofs and most of the experimental results have been deferred to the journal version.

2 A First Approach

In this section we describe an algorithm that for two given networks N_1 and N_2 can compute $D(N_1, N_2)$ in $O(n^2m + \lambda^3)$ time.

Overview. The algorithm consists of a preprocessing step and a triplet distance computation step. In the preprocessing step, we extend a technique introduced by Shiloach and Perl [18] in 1978 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 $O(1)$ time. In the triplet distance computation step, we compute $D(N_1, N_2)$ by iterating over all possible $4\binom{\lambda}{3}$ 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

Fan Graph. For any network N_i, let the *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 edges:

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

Note that N_i^f contains $O(|V_i|^3)$ nodes, $O(|V_i|^2|E_i|)$ edges and also has the property described in the following lemma:

Lemma 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, node s can reach node (x, y, z) in N_i^f if and only if $x|y|z$ is a fan triplet in N_i.*

Corollary 1. *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 $r(N_i)$, if and only if s can reach (r', x, y) in N_i^f.*

Resolved Graph. For any network N_i, let the *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 edges:

1. $\{s \to (u, v) \mid u \to v \in E_i\}$

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

Note that N_i^r contains $O(|V_i|^3)$ nodes, $O(|V_i|^2|E_i|)$ edges and also has the property described in the following lemma:

Lemma 2. *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, node s can reach node (x, y, z) in N_i^r if and only if $x|yz$ is a resolved triplet in N_i.*

Corollary 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 some internal node $z \neq r(N_i)$ in N_i, to x and y that are disjoint, except for z, if and only if s can reach (r', x, y) in N_i^r.*

Given N_i^f and N_i^r, we define the $\lambda \times \lambda \times \lambda$ *fan table* A_i^f and the $\lambda \times \lambda \times \lambda$ *resolved table* A_i^r as follows. For 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. Due to Lemmas 1 and 2, both A_i^f and A_i^r can be computed by a depth first traversal (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. Finally, $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r and 0 otherwise.

2.2 Triplet Distance Computation

Algorithm 1 summarizes all the procedures needed to compute the triplet distance between two given networks N_1 and N_2. For every $i \in \{1, 2\}$ the tables A_i^f and A_i^r are built in lines 2–7. These tables are then used in lines 11–12 and 16–19 to determine in $O(1)$ time if a triplet is consistent with N_1 or N_2. Procedures $S_f()$ and $S_r()$ count the number of shared fan and resolved triplets. Both procedures enumerate over all possible triplets and use the tables A_i^f and A_i^r to determine their consistency with either network. The correctness is ensured by Lemmas 1 and 2. Procedure $S()$ reports the number of shared triplets, which is the sum of the number of shared fan triplets and shared resolved triplets. The main procedure is $D()$. It uses Eq. (1) to determine $D(N_1, N_2)$.

To analyze the running time, after the preprocessing is finished, the procedures $S_f()$ and $S_r()$ require $O(\lambda^3)$ time. For the total preprocessing time, by definition, building the data structures N_i^r and N_i^f for $i \in \{1, 2\}$ in line 3, requires $O(|V_1|^2|E_1| + |V_2|^2|E_2|)$ time. Building the auxiliary arrays A_i^r and A_i^f in lines 5–7 is performed by a depth first traversal of N_i^r and N_i^f, thus requiring $O(|V_1|^2|E_1| + |V_2|^2|E_2)$ time as well. Hence, the total time of the algorithm

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

procedure PREPROCESSING(N_1, N_2) ▷ Building the data structures
  for $i \in \{1, 2\}$ **do**
    build $N_i^f = (V_i^f, E_i^f)$ and $N_i^r = (V_i^r, E_i^r)$
    let A_i^f, A_i^r be $\lambda \times \lambda \times \lambda$ arrays initialized with 0 entries
    for three different leaves x, y and z **do**
      $A_i^f[x][y][z] = 1$ if s can reach (x, y, z) in N_i^f
      $A_i^r[x][y][z] = 1$ if s can reach (x, y, z) in N_i^r
  return $(A_1^r, A_1^f, A_2^r, A_2^f)$

procedure $S_f(A_1^f, A_2^f)$ ▷ Finding the shared fan triplets
  $sharedFan = 0$
  for three different leaves x, y and z **do**
    if $A_1^f[x][y][z] = A_2^f[x][y][z] = 1$ **then** $sharedFan = sharedFan + 1$
  return *sharedFan*

procedure $S_r(A_1^r, A_2^r)$ ▷ Finding the shared resolved triplets
  $sharedResolved = 0$
  for three different leaves x, y and z **do**
    if $A_1^r[x][y][z] = A_2^r[x][y][z] = 1$ **then** $sharedResolved = sharedResolved + 1$
    if $A_1^r[x][z][y] = A_2^r[x][z][y] = 1$ **then** $sharedResolved = sharedResolved + 1$
    if $A_1^r[y][z][x] = A_2^r[y][z][x] = 1$ **then** $sharedResolved = sharedResolved + 1$
  return *sharedResolved*

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

procedure $D(N_1 = (V_1, E_1), N_2 = (V_2, E_2))$ ▷ Computing the triplet distance
  $(A_1^r, A_1^f, A_2^r, A_2^f) =$ PREPROCESSING(N_1, N_2)
  return $S(A_1^r, A_1^f, A_1^r, A_1^f) + S(A_2^r, A_2^f, A_2^r, A_2^f) - 2S(A_1^r, A_1^f, A_2^r, A_2^f)$

becomes $O(|V_1|^2|E_1| + |V_2|^2|E_2| + \lambda^3)$. By the definition of n and m from Sect. 1, the running time becomes $O(n^2m + \lambda^3)$. Hence, we obtain the following theorem:

Theorem 1. *There exists an algorithm that computes the triplet distance between two networks N_1 and N_2 in $O(n^2m + \lambda^3)$ time.*

Let N_1 and N_2 follow the degree constraints of Byrka *et al.* [2]. We then have $n = \Theta(m)$ and the bound becomes $O(n^3 + \lambda^3)$, thus matching the bound achieved by the first data structure of Byrka *et al.* [2].

3 A Second Approach

In this section we extend the algorithm from Sect. 2 in order to exploit the information about the level of the two input networks. More specifically, we describe

an algorithm that for two given networks N_1 and N_2 can compute $D(N_1, N_2)$ in $O(m + k^3d^3\lambda + \lambda^3)$ time.

Overview. In the first approach, for a given network N_i we built the fan and resolved graph presented in Lemmas 1 and 2. In this second approach, for every biconnected component of $U(N_i)$ we define a network of approximately the same size as the biconnected component, which we call *contracted block network*. For this contracted block network we then build the corresponding fan and resolved graph. By carefully contracting every biconnected component of $U(N_i)$ into one node we obtain a tree, which we call *block tree*. We finally show how to combine the block tree and all the fan and resolved graphs of the contracted block networks of N_i to count triplets efficiently.

3.1 Preprocessing

Let N_i be a given network. From here on, we call a biconnected component of $U(N_i)$ a *block*. For simplicity, when we refer to a block of N_i, we imply a block of $U(N_i)$. We say that for a block B of N_i, node $r(B)$ is the root of B, if $r(B)$ has the largest height in N_i among all nodes in B. Note that because N_i has one root that can reach every node of N_i and B corresponds to a maximal subgraph of $U(N_i)$ that is biconnected, B can only contain one root. If B contains only one edge $u \to v$ such that $v \in \mathcal{L}(N_i)$, then B is called a *leaf block*, otherwise B is called a *non-leaf block*. Lemma 3 presents a property of all blocks of N_i.

Lemma 3. *All blocks of a given network N_i are edge-disjoint.*

Block Tree. From a high level perspective, we want to remove the cycles in $U(N_i)$ that are formed by the non-leaf blocks to obtain a directed tree on the leaf label set $\mathcal{L}(N_i)$. Let $T_i = (V', E')$ be a directed tree, from now on referred to as *block tree*, with the node set V' and edge set E' defined by the following steps:

- For every block B_j in N_i create a node b_j in T_i.
- 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 node in B_1 then create the edge $b_1 \to b_2$ in T_i.
- Create a root node ρ in T_i. For every block B_j that has $r(N_i)$ as a root, create the edge $\rho \to b_j$ in T_i.
- If B_j is a leaf block, rename b_j in T_i by the label of the leaf in B_j.

The set of all blocks of N_i and the node set $V' - r(T_i)$, i.e., the set of all nodes of T_i except the root, are bijective. 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 node $r(B_2)$ is a shared node between B_1 and B_2. Note that by the definition of a block, an edge connecting two nodes can define a block of its own. The following lemma presents some properties of T_i:

Lemma 4. *Let $T_i = (V', E')$ be the block tree of a given network N_i. The block tree T_i is a directed tree that has λ leaves, $|V'| = O(\lambda)$ and $|E'| = O(\lambda)$.*

Since the set of all blocks of N_i and the set $V' - r(T_i)$ are bijective, we obtain:

Corollary 3. *A network N_i contains $\mathrm{O}(\lambda)$ blocks.*

The following lemma presents an algorithm for constructing the block tree T_i:

Lemma 5. *Let $T_i = (V', E')$ be the block tree of a given network N_i. There exists an algorithm that builds T_i in $\mathrm{O}(|E_i|)$ time.*

Contracted Block Network. For a given network N_i, a block B in N_i and a node u in B, define L_B^u to be the set of leaves that can be reached from u without using edges in B. Let $C_B = (V', E')$ be a network, with the node set V' and edge set E' defined by the following steps:

- Let $C_B = N_i$. All operations from now on are applied on C_B.
- Remove every edge and node not in B.
- For every edge $u_1 \rightarrow u_2$ in B, if $in(u_1) = out(u_1) = in(u_2) = out(u_2) = 1$ contract the edge as follows: let $u_2 \rightarrow u_3$ be the other edge in B, then create the edge $u_1 \rightarrow u_3$, remove u_2 from B and set $L_B^{u_1} = L_B^{u_1} \cup L_B^{u_2}$.
- For every node u_1 in C_B such that $L_B^{u_1} \neq \emptyset$, we add a child leaf with label s_1 representing all leaves in $L_B^{u_1}$. We also add another child leaf s_1' as a copy leaf that will help later on to count triplets.
- Include an artificial leaf r' which is attached to the root $r(C_B)$.

Every node in C_B corresponds to a node in B and every edge between two internal nodes in C_B corresponds to a compressed path in B. We call C_B *the contracted block network of* N_i, corresponding to block B. The following lemma presents a property of C_B:

Lemma 6. *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. We then have that $|\mathcal{L}(C_B)| = \mathrm{O}(k_i d_i + 1)$, $|V'| = \mathrm{O}(k_i d_i + 1)$ and $|E'| = \mathrm{O}(k_i d_i + 1)$.*

Constructing All Contracted Block Networks Efficiently. 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 node u in B such that $u \neq r(B)$ and $x \in L_B^u$. For any leaf x associated with some block B of N_i, let $q_B(x)$ be the node in B that has a path to x without using edges in B, i.e., $x \in L_B^{q_B(x)}$, $p_B(x)$ the leaf in C_B representing x and $p'_B(x)$ the copy leaf of $p_B(x)$.

Lemma 3 implies an algorithm for constructing every block network C_B of N_i in $\mathrm{O}(|E_i|)$ time. As shown in the lemma below, by properly relabeling the leaves of N_i and with an additive $\mathrm{O}(\lambda^2)$ time, it is possible to build every block network C_B so that we can afterwards compute for every leaf $l \in \mathcal{L}(N_i)$ the functions $q_B(l)$ and $p_B(l)$ in $\mathrm{O}(1)$ time.

Lemma 7. *For a network N_i, there exists an $\mathrm{O}(|E_i| + \lambda^2)$-time algorithm that builds all the contracted block networks C_B of N_i, such that for all blocks of N_i and leaf $l \in \mathcal{L}(N_i)$ the functions $q_B(l)$ and $p_B(l)$ are computed in $\mathrm{O}(1)$ time.*

For the block network C_B, we denote C_B^f the fan graph of C_B and C_B^r the resolved graph of C_B. Moreover, we denote A_B^f the fan table of C_B and A_B^r the resolved table of C_B (see Sect. 2.1 for the definition of a fan graph & table and resolved graph & table). The following lemma shows the time required to build C_B^f, C_B^r, A_B^f and A_B^r for every block B of a given network N_i:

Lemma 8. *For a network N_i, building C_B^f, C_B^r, A_B^f and A_B^r for every block B of N_i requires $\mathrm{O}(\lambda(k_i^3 d_i^3 + 1))$ time.*

3.2 Triplet Distance Computation

Let B be a block of a network N_i. We say that $x|y|z$ is a *fan triplet consistent with B*, if there exists a node u in B that has three disjoint paths in N_i to x, y and z, except for u, one going from u to x, one from u to y and one from u to z. We also say that $x|y|z$ *is rooted at u* in B. Since u is also in N_i, this means that $x|y|z$ is rooted at u in N_i as well. Similarly, we say that $xy|z$ is a *resolved triplet consistent with B*, if there exist two nodes u and v in B such that $u \neq v$, and there are four disjoint paths in N_i, except for u and v, one going from u to v, one from v to x, one from v to y and one from u to z. Moreover, we say that $xy|z$ is *rooted at u and v* in B or N_i (similarly to the fan triplet). Note that if $x|y|z$ is a fan triplet consistent with B, then it will also be consistent with N_i. Similarly, if $xy|z$ is a resolved triplet consistent with B, it will also be consistent with N_i.

Given the data structures from the preprocessing step, Lemmas 9 and 10 together show how to determine the consistency of a fan and resolved triplet with N_i in $\mathrm{O}(1)$ time. From a high level perspective to achieve this, for three different leaves x, y and z, we consider all the possible cases for the location of the lowest common ancestor of every pair (x, y), (x, z) and (y, z) in T_i. Since every node in T_i except $r(T_i)$ corresponds to a block in N_i, we can then use the available data structures to determine efficiently if N_i has the necessary disjoint paths that would imply the consistency of the fan triplet $x|y|z$ or resolved triplet $xy|z$ with N_i. We start by showing in Lemma 9 how to determine the consistency of a triplet with a block B of N_i. Afterwards, we show in Lemma 10 how to use Lemma 9 to determine the consistency of a triplet with N_i.

Lemma 9. *Let N_i be a given network, T_i its block tree and x, y and z three different leaves. Let w be the lowest common ancestor of x, y and z in T_i, $w \neq r(T_i)$ and B the block in N_i corresponding to w. If C_B, C_B^f,C_B^r, A_B^f and A_B^r are given, there exists an algorithm that can determine in $\mathrm{O}(1)$ time if the fan triplet $x|y|z$ or resolved triplet $xy|z$ is consistent with B.*

Lemma 10. *Let N_i be a given network and x, y and z three different leaves in N_i. Given T_i, C_B^f, C_B^r, A_B^f and A_B^r for every block B in N_i, there exists an algorithm that can determine in $\mathrm{O}(1)$ time if the fan triplet $x|y|z$ or the resolved triplet $xy|z$ is consistent with N_i.*

The final algorithm is similar to Algorithm 1, the main difference is in the preprocessing step. In this step, for every $i \in \{1, 2\}$ we start by building the block tree T_i. Then, we build a $\lambda \times \lambda$ table for T_i in order to be able later to answer lowest common ancestor queries between pairs of leaves in T_i in $O(1)$ time. Afterwards, we build all the contracted block networks of N_i. Finally, for every block B in N_i and the corresponding contracted block network C_B, we build the fan graph C_B^f and the resolved graph C_B^r, as well as the corresponding A_B^f and A_B^r tables.

From Lemma 5, building T_i for every $i \in \{1, 2\}$ requires $O(|E_1| + |E_2|)$ time. Building the two tables for answering lowest common ancestor queries requires $O(\lambda^3)$ time. From Lemma 6, building all the contracted block networks requires $O(|E_1|+|E_2|+\lambda^2)$ time. From Lemma 8, the 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 $O(\lambda(k_1^3d_1^3+k_2^3d_2^3+2))$. Hence, the total preprocessing time becomes $O(|E_1| + |E_2| + \lambda(k_1^3d_1^3 + k_2^3d_2^3) + \lambda^3)$.

Using the results from Lemma 10, 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 $4\binom{\lambda}{3}$, the total running time of the algorithm remains $O(|E_1|+|E_2|+\lambda(k_1^3d_1^3+k_2^3d_2^3)+\lambda^3)$. By the definition of n, m, k and d from Sect. 1, the running time becomes $O(m + k^3d^3\lambda + \lambda^3)$. Hence, we obtain the following theorem:

Theorem 2. *There exists an algorithm that computes the triplet distance between two networks N_1 and N_2 in $O(m + k^3d^3\lambda + \lambda^3)$ time.*

Let N_i be a network that follows the degree constraints of Byrka *et al.* [2]. If for a block $B_s = (V_s, E_s)$ of N_i we define k_s to be the number of reticulation nodes in B_s, where $k_s \leq k_i$, using the same arguments as those used in the proof of Lemma 6, we get for $C_{B_s} = (V', E')$ that $|V'| = |E'| = O(k_s + 1)$. The time to build C_B^f, C_B^r, A_B^f and A_B^r for every block B of N_i then becomes $\sum_s O(k_s^3 + 1)$. Note that Lemma 8 would give a $O(\lambda k_i^3 + 1)$ time instead, because it uses λ to upper bound (from Corollary 3) the number of blocks we can have in N_i. Since $\sum_s k_s = O(|V_i|)$, the preprocessing time required by our algorithm for N_i would be $O(|V_i| + k^2|V_i|)$. Then, the time to compute $D(N_1, N_2)$ becomes $O(n + k^2n + \lambda^3)$, thus matching the time bound required by using the second data structure of Byrka *et al.* [2].

4 Implementation and Experiments

This section provides an implementation of the algorithms described in Sects. 2 and 3, referred to as `NTDfirst` and `NTDsecond` respectively, as well as experiments illustrating their practical performance.

The Setup. We implemented the two algorithms in C++ and the source code is publicly available at https://github.com/kmampent/ntd. The experiments were performed on a machine with 16 GB RAM and Intel(R) Core(TM) i5-3470 CPU @ 3.20 GHz. The operating system was Ubuntu 16.04.2 LTS. The compiler used was g++ 5.4 with cmake 3.11.0.

The Input. We consider both simulated and real datasets. For the simulated datasets, we create tree-based networks [9] as follows:

1. Build a random rooted binary tree T on λ leaves in the uniform model [16] and let $N = T$. For a node w in N, let $d(w)$ be the total number of edges on the path from $r(N)$ to N.
2. Given a parameter $e \geq 0$, add e random edges in N as follows. An edge $u \rightarrow v$ is created in N if $d(u) < d(v)$. If the total number of edges that can be added happens to be y, where $y < e$, then we only add those y edges.

For the real datasets, we consider networks that have been published in the literature and are not necessarily tree-based. More precisely, we consider the 6 trees and the corresponding networks in [15, Table S4]. The trees are given in the standard Newick format, and the networks in the extended Newick format [3].

Experiments. For the simulated datasets, in Fig. 2 we illustrate the effect of e on the CPU time in seconds of the two algorithms. Every data point in the graph is the average of 20 different runs. The effect is larger on `NTDsecond`, as larger values for e imply fewer blocks in the given networks. We note that space is the reason behind the difference restrictions on λ, i.e., for $\lambda = 230$ the memory usage of `NTDfirst` approaches the limits of the available 16 GB RAM.

For the real datasets and for every $s \in \{A, B, C, D, E, F\}$, we denote T_s the tree and N_s the corresponding network, where s is a scenario in [15, Table S4], with F corresponding to scenario "`E, CHAM and MELVIO resolved`". For the network N_F, we use its non-tree based version from [14]. From Eq. (1) we have the following: $D(T_s, N_s) = S(T_s, T_s) + S(N_s, N_s) - 2S(T_s, N_s)$. When computing $D(T_s, N_s)$ and to have $\mathcal{L}(T_s) = \mathcal{L}(N_s)$, if a leaf x in N_s appears as several leaves $x.1, \ldots, x.i$ in T_s, we replace x in N_s with the leaves $x.1, \ldots, x.i$ that we attach under the parent of x. For the size of the leaf label sets, in the trees T_A, T_B, T_C, T_D, T_E we have 16, 20, 21, 21, 22 and 50 leaves, in every network N_s where $s \in \{A, B, C, D, E\}$ we have 8 leaves and in N_F we have 16 leaves. In Table 2 we include the experimental results. Interestingly, while the two networks N_B and N_D look structurally different, $D(N_B, N_D) = 0$. This suggests that it may be useful to extend the definition of the triplet distance to take into account the number of times that each triplet occurs in a network.

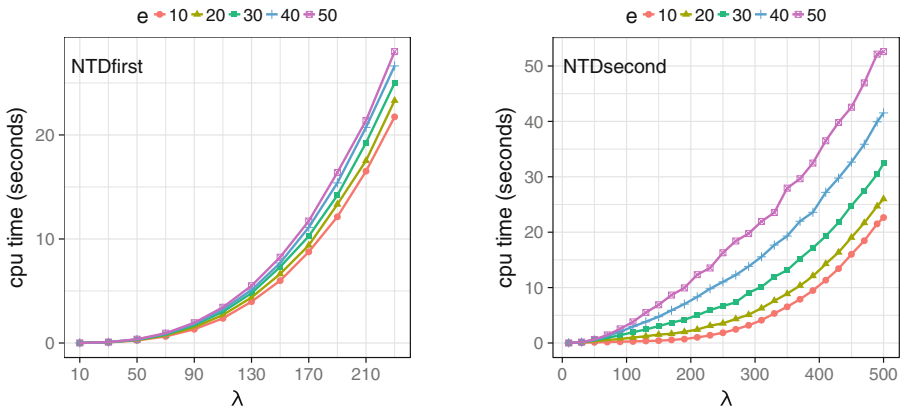

Fig. 2. Experiments on the simulated datasets: running time for different values of e.

Table 2. Experiments on the real datasets. $N_A, \ldots, N_E$ have identical leaf label sets.

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	19600	43710	16553	30204

	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

Acknowledgments. Konstantinos Mampentzidis acknowledges the support by the Danish National Research Foundation, grant DNRF84, via the Center for Massive Data Algorithmics (MADALGO).

References

1. Brodal, G.S., Mampentzidis, K.: Cache oblivious algorithms for computing the triplet distance between trees. In: Proceedings of ESA 2017, pp. 21:1–21:14 (2017)
2. Byrka, J., Gawrychowski, P., Huber, K.T., Kelk, S.: Worst-case optimal approximation algorithms for maximizing triplet consistency within phylogenetic networks. J. Discrete Algorithms **8**(1), 65–75 (2010)
3. Cardona, G., Rosselló, F., Valiente, G.: Extended Newick: it is time for a standard representation of phylogenetic networks. BMC Bioinform. **9**(1), 532 (2008)
4. Choy, C., Jansson, J., Sadakane, K., Sung, W.K.: Computing the maximum agreement of phylogenetic networks. Theor. Comput. Sci. **335**(1), 93–107 (2005)
5. Dobson, A.J.: Comparing the shapes of trees. In: Street, A.P., Wallis, W.D. (eds.) Combinatorial Mathematics III. Lecture Notes in Mathematics, vol. 452, pp. 95–100. Springer, Berlin (1975). https://doi.org/10.1007/BFb0069548
6. Estabrook, G., McMorris, F., Meacham, C.: Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. Syst. Zool. **34**(2), 193–200 (1985)
7. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates, Inc., Sunderland (2004)

8. Fortune, S., Hopcroft, J., Wyllie, J.: The directed subgraph homeomorphism problem. Theor. Comput. Sci. **10**(2), 111–121 (1980)
9. Francis, A.R., Steel, M.: Which phylogenetic networks are merely trees with additional arcs? Syst. Biol. **64**(5), 768–777 (2015)
10. Gambette, P., Huber, K.T.: On encodings of phylogenetic networks of bounded level. J. Math. Biol. **65**(1), 157–180 (2012)
11. Gusfield, D., Eddhu, S., Langley, C.: Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. J. Bioinform. Comput. Biol. **2**(1), 173–213 (2004)
12. Jansson, J., Lingas, A.: Computing the rooted triplet distance between galled trees by counting triangles. J. Discrete Algorithms **25**, 66–78 (2014)
13. Jansson, J., Rajaby, R., Sung, W.K.: An efficient algorithm for the rooted triplet distance between galled trees. In: Proceedings of AlCoB 2017, pp. 115–126 (2017)
14. Jetten, L., van Iersel, L.: Nonbinary tree-based phylogenetic networks. IEEE/ACM Trans. Comput. Biol. Bioinform. **1**(1), 205–217 (2018)
15. 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)
16. McKenzie, A., Steel, M.: Distributions of cherries for two models of trees. Math. Biosci. **164**(1), 81–92 (2000)
17. Robinson, D., Foulds, L.: Comparison of phylogenetic trees. Math. Biosci. **53**(1), 131–147 (1981)
18. Shiloach, Y., Perl, Y.: Finding two disjoint paths between two pairs of vertices in a graph. J. ACM **25**(1), 1–9 (1978)