A More Practical Algorithm for the Rooted Triplet Distance

Jesper Jansson[1(✉)] and Ramesh Rajaby[2]

[1] Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
jj@kuicr.kyoto-u.ac.jp
[2] University of Milano-Bicocca, Milano, Italy
r.rajaby@campus.unimib.it, ramesh.rajaby@gmail.com

Abstract. The *rooted triplet distance* is a measure of the dissimilarity of two phylogenetic trees with identical leaf label sets. An algorithm by Brodal *et al.* [2] that computes it in $O(n \log n)$ time, where n is the number of leaf labels, has recently been implemented in the software package tqDist [14]. In this paper, we show that replacing the hierarchical decomposition tree used in Brodal *et al.*'s algorithm by a centroid paths-based data structure yields an $O(n \log^3 n)$-time algorithm that, although slower in theory, is easier to implement and apparently faster in practice. Simulations for values of n up to $1,000,000$ support our claims experimentally.

Keywords: Bioinformatics · Phylogenetic tree comparison · Rooted triplet distance · Centroid path decomposition tree

1 Introduction

Over the years, many alternative methods for inferring phylogenetic trees have been developed; see, e.g., [7]. Due to errors in experimentally obtained data or the inherent instability of classifications, applying the same tree inference method to different datasets, applying different tree inference methods to the same dataset, or changing the assumed model of evolution may result in trees with different branching patterns. In this case, in order to identify parts of the trees that look alike or to reconcile all the trees into a single tree, methods for measuring the similarity between phylogenetic trees are needed. Measuring the similarity between two phylogenetic trees may also be useful for supporting queries in phylogenetic databases in the future [1] or for evaluating the performance of a newly proposed tree inference method by doing simulations and comparing the inferred trees to the corresponding known correct trees.

Several measures of the (dis-)similarity of two phylogenetic trees with identical leaf label sets have been suggested in the literature (see [1]). One such measure is the *rooted triplet distance* [5], which counts how many of the subtrees

J. Jansson—Funded by The Hakubi Project and KAKENHI grant number 26330014.
R. Rajaby—Funded by the EXTRA Project at the University of Milano-Bicocca.

A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 109–125, 2015.
DOI: 10.1007/978-3-319-21233-3_9

induced by cardinality-3 subsets of the leaves that differ between the two trees. Intuitively, this measure considers two phylogenetic trees that share many small embedded subtrees to be similar. This paper presents a practical algorithm for computing the rooted triplet distance, based on the framework introduced in an algorithm by Brodal *et al.* [2], along with its implementation.

1.1 Basic Definitions

In this paper, a *phylogenetic tree* is a rooted, unordered tree whose leaves are distinctly labeled and whose internal nodes have degree at least 2. From here on, phylogenetic trees are referred to as "trees" for short. The set of all nodes and the set of all leaf labels in a tree T are denoted by $V(T)$ and $\Lambda(T)$, respectively. For any $x \in V(T)$, $T(x)$ is the subtree of T rooted at x, i.e., the subgraph of T induced by the node x and all of its proper descendants in T. For any $x, y \in V(T)$, $lca^T(x, y)$ is the lowest common ancestor in T of x and y. Also, for any $x, y \in V(T)$, if x is a proper descendant of y then we write $x \prec y$.

A *rooted triplet* is a tree with exactly three leaves. Suppose t is a rooted triplet with leaf label set $\Lambda(t) = \{a, b, c\}$. There are two possibilities. If t has a single internal node then t is called a *fan triplet* and is denoted by $a|b|c$. Observe that in this case, t is a non-binary tree and $lca^t(a, b) = lca^t(a, c) = lca^t(b, c)$ holds. Otherwise, t has two internal nodes and is a binary tree; in this case, t is called a *resolved triplet* and is denoted by $xy|z$, where $\{x, y, z\} = \{a, b, c\}$ and $lca^t(x, y) \prec lca^t(x, z) = lca^t(y, z)$.

For any tree T and $\{a, b, c\} \subseteq \Lambda(T)$, the fan triplet $a|b|c$ is said to be *consistent with* T if $lca^T(a, b) = lca^T(a, c) = lca^T(b, c)$. Similarly, the resolved triplet $ab|c$ is *consistent with* T if $lca^T(a, b) \prec lca^T(a, c) = lca^T(b, c)$. Let $rt(T)$ be the set of all rooted triplets consistent with the tree T. (Thus, $|rt(T)| = \binom{|\Lambda(T)|}{3}$).

For any two trees T_1, T_2 with $\Lambda(T_1) = \Lambda(T_2)$, the *rooted triplet distance* $d_{rt}(T_1, T_2)$ is defined as $|rt(T_1) \triangle rt(T_2)|$, i.e., the number of rooted triplets that are consistent with one of the two trees but not the other. Note that dividing $d_{rt}(T_1, T_2)$ by $\binom{n}{3}$, where $n = |\Lambda(T_1)| = |\Lambda(T_2)|$, yields a dissimilarity coefficient between 0 and 1 that may be more informative than d_{rt} in some applications.

Below, we consider the problem of computing $d_{rt}(T_1, T_2)$ for two input trees T_1, T_2 with identical leaf label sets. To simplify the notation, write $L = \Lambda(T_1)$ $(= \Lambda(T_2))$ and $n = |L|$ for the given trees.

1.2 Previous Results and Related Work

The rooted triplet distance was proposed by Dobson [5] in 1975. Given two trees T_1, T_2 with identical leaf label sets, $d_{rt}(T_1, T_2)$ can be computed in $O(n^3)$ time by a straightforward algorithm. Critchlow *et al.* [4] gave an $O(n^2)$-time algorithm for the special case where T_1 and T_2 are *binary*, and Bansal *et al.* [1] showed how to compute $d_{rt}(T_1, T_2)$ in $O(n^2)$ time for two trees of *arbitrary* degrees. Recently, Brodal *et al.* [2] achieved a time complexity of $O(n \log n)$ for two trees

of arbitrary degrees. An implementation of the latter algorithm, written in C++, is available in the free software package tqDist [14].

The counterpart of the rooted triplet distance for *unrooted* trees is the *unrooted quartet distance* [6]. The currently fastest algorithm for computing the unrooted quartet distance [2] runs in $O(dn \log n)$ time, where n is the number of leaf labels and d is the maximum degree of any node in the two input trees.

An extension of the rooted triplet distance to *phylogenetic networks* has been studied in [11]. For two *galled trees* [9] (networks whose cycles are disjoint) with n leaves each, the rooted triplet distance can be computed in $o(n^{2.687})$ time [11].

1.3 Our Contributions

We present some non-trivial modifications to Brodal *et al.*'s algorithm [2] for computing $d_{rt}(T_1, T_2)$ for two trees of arbitrary degrees that make it easier to implement and more efficient in practice. The theoretical time complexity of the resulting algorithm is $O(n \log^3 n)$, which is slightly worse than that of the original version, but we show experimentally that a direct C++-implementation of the new algorithm gives a faster and more memory-efficient method than tqDist [14] (the publicly available implementation of Brodal *et al.*'s algorithm) for various types of large inputs consisting of two trees with up to $1,000,000$ leaves each.

The paper is organized as follows. Brodal *et al.*'s algorithm [2] is reviewed in Sect. 2. Section 3 describes the new algorithm, Sect. 4 discusses some implementation issues, and Sect. 5 presents the experimental results. Finally, Sect. 6 gives some concluding remarks.

2 Summary of Brodal *et al*'s Algorithm [2]

On a high level, the algorithm of Brodal *et al.* [2] works as follows. Each rooted triplet t in $rt(T_1)$ is implicitly assigned to the lowest common ancestor in T_1 of the three leaves in $\Lambda(t)$. For each internal node u in T_1, the algorithm counts how many of its assigned rooted triplets that also appear in $rt(T_2)$ by first coloring the leaves of T_2 in such a way that two leaves receive the same color if and only if they are descendants of the same child of u in T_1, and then finding the number of elements in $rt(T_2)$ compatible with this particular coloring by making a query to a special data structure called a *hierarchical decomposition tree* (HDT) that represents T_2.

To avoid unnecessary leaf recolorings, a simple, recursive recoloring scheme is used that visits all nodes of T_1 in order and generates the corresponding leaf colorings in T_2. It is reviewed in Sect. 3.2 (a) below. Constructing the HDT, augmenting it with auxiliary information to support the relevant queries, and updating this information when the leaves of T_2 are recolored are somewhat complicated; see [2] for details.

As proved in [2], the algorithm's time complexity is $O(n \log n)$.

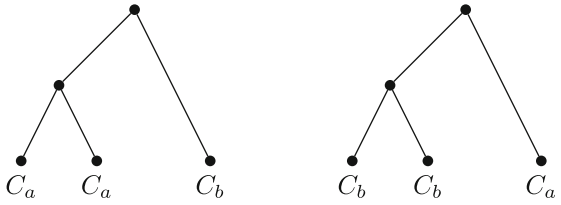


Fig. 1. Topologies induced by good triplets in T_2.

3 The New Algorithm

The new algorithm, described below, uses the same framework as Brodal *et al.*'s algorithm [2]. To be precise, we also implicitly assign each rooted triplet in $rt(T_1)$ to an internal node in T_1 and count, for each node in T_1, how many of its rooted triplets that appear in $rt(T_2)$. To handle all of T_1's nodes efficiently, we apply Brodal *et al.*'s recoloring scheme with a minor modification. The main difference between the old algorithm and the new algorithm is how the rooted triplets in $rt(T_2)$ assigned to each node in $V(T_1)$ are counted. Whereas Brodal *et al.*'s algorithm uses the (in our opinion) cumbersome HDT data structure, we use the conceptually simpler centroid path decomposition technique [3]. This makes the new algorithm a little slower in theory but easier to implement and faster in practice.

3.1 Preliminaries

For convenience, we make T_1 and T_2 into *ordered* trees by imposing the following left-to-right ordering: for every non-leaf node v in a tree, the leftmost child of v is always a child of v having the most leaf descendants (with ties broken arbitrarily), and the other children of v are ordered arbitrarily. Also, let a *triplet* be any subset of L of cardinality three; each triplet x induces a rooted triplet in T_1 (or T_2), namely the rooted triplet belonging to $rt(T_1)$ (or $rt(T_2)$) whose leaf label set equals x.

We first introduce some notation related to the leaf colorings induced by the internal nodes of T_1. Let d be the number of children of the highest degree node in T_1. Define a set of $d+1$ *colors* $\{C_0, C_1, .., C_d\}$. We usually refer to C_1 as *RED* and C_0 as *WHITE*, and sometimes call a WHITE leaf *non-colored* and a leaf that is neither RED nor WHITE *NON-RED*. Colors will be assigned to the leaf label set L, and we say that we are coloring a leaf when we are coloring its label.

Let $\{v_1, v_2, \dots\}$ be the children of an internal node $v \in V(T_1)$. Then T_1 and T_2 are *colored according to* v if and only if, for every $l \in L$, it holds that:

- l is colored by C_i if and only if $T_1(l)$ is a descendant of v_i; and
- l is WHITE if and only if $T_1(l)$ is not a descendant of v.

Suppose that T_1 and T_2 are colored according to some internal node $v \in T_1$. Let t be a triplet. We call t a *good triplet* if it induces one of the two (unordered)

topologies shown in Fig. 1 in T_2, where C_a and C_b are colors in $\{C_1, C_2, .., C_d\}$ and $a < b$. Similarly, we call t a *good fan* if its three leaves all have different colors from the set $\{C_1, C_2, .., C_d\}$. (This corresponds to the concept of a triplet being "compatible with a coloring" in [2].)

For a given color C_c, $C_c(S)$ is the number of leaves in S colored by C_c, where S can be either a single subtree or a set of subtrees (which is generally clear from the context). $C_1(S)$ will usually be referred to as $Red(S)$. Also define $C_{\bar{a}}(S) = \sum_{i=2, i\neq a}^{d} C_i(S)$ as the number of NON-RED leaves in S which are not colored by C_a, and $C_{R\bar{a}}(S) = \sum_{i=1, i\neq a}^{d} C_i(S)$ as the number of non-WHITE leaves in S which are not colored by C_a. Finally, define $Red^{(2)}(S) = \sum_{i=1}^{k} \binom{Red(S_i)}{2}$, where $\{S_1..S_k\}$ are the subtrees rooted at the root of S. We will sometimes use a node as an argument, meaning the subtree rooted at it.

Next, we recall the concept of a *centroid path* [3]. A *centroid path* starting at some node v in a tree T is a heaviest path from v to a leaf of T, i.e., a path starting at v and always choosing a child with the largest number of leaf descendants until a leaf is reached. Let *the centroid path of* T, $cp(T)$, be a centroid path starting at the root of T. In our case, the centroid path of T starts from the root and always selects the left subtree until it reaches a leaf.

The *centroid path decomposition tree* of T, denoted by $CPDT(T)$, is an ordered tree of unbounded degree defined as follows. One node u represents $cp(T)$; u is the root of $CPDT(T)$. We traverse $cp(T)$ from its lowest node to its highest; for each node r_j in T that we encounter, we add a single node v_j to the ordered set of children of u (making v_j the rightmost child so far), and then define the children of v_j as $\{CPDT(T_2^j), ..., CPDT(T_k^j)\}$, where k is the degree of r_j and T_i^j is the i-th subtree of r_j (remember that the root of T_1^j lies on the centroid path). We call u a *CP-node* (*centroid path node*), since it represents a whole centroid path in T, and v_j a *SN-node* (*single-node node*), since it represents a single node in T. If r_j is binary then v_j will have a single child.

Note that $CPDT(T)$ has height $O(\log n)$, where $n = |\Lambda(T)|$. Moreover, we immediately have:

Lemma 1. *An SN-node is always the child of a CP-node, and the only CP-node which is not the child of an SN-node is the root.*

3.2 Description of the New Algorithm

First, the algorithm constructs $CPDT(T_2)$. Then, for each internal node v of T_1 in depth-first order, the algorithm: (a) colors the trees according to v, and (b) counts the resulting number of good triplets and good fans by using $CPDT(T_2)$. Finally, it returns the value $\binom{n}{3}$ minus the total number of good triplets and good fans found.

(a) Colorings: To obtain all the colorings of the trees efficiently, the algorithm uses a slightly modified variant of Brodal *et al.*'s recursive recoloring scheme from [2]. The latter does a depth-first traversal of T_1 while maintaining two invariants:

(i) when entering a node v, all leaves in $T_1(v)$ are RED and all other leaves are WHITE;
(ii) when exiting a node v, all leaves in $T_1(v)$ are WHITE.

Initially, all leaves are colored RED. This way, invariant i) holds when the transversal starts at the root of T_1.

During the transversal, whenever an internal node $v \in V(T_1)$ is reached, the leaves in the i-th subtree of v are colored by the color C_i for $i \in [2..k]$, where k is the degree of v. At this point, the trees are colored according to v. After this, the $k-1$ subtrees that were just colored are recolored by the color WHITE, and the scheme recurses on the leftmost subtree of v, which is still RED. Observe that the first invariant holds when entering the root of this subtree. After returning from the recursive call, the leftmost subtree of v is WHITE by invariant (ii), and the other subtrees of v are treated one by one; for each such subtree, the scheme colors its leaves RED and then recurses on it. Again, invariant (i) holds at each recursive call since the subtree is colored RED and everything else is WHITE. After handling all k subtrees of v, all leaf descendants of v will be WHITE, so invariant (ii) holds when exiting from v.

The base case of the recursion is when the reached node $v \in V(T_1)$ is a leaf. In this case, the scheme colors v WHITE and exits, so that invariant (ii) holds.

Our algorithm makes the following modification to Brodal *et al.*'s recursive recoloring scheme above: We color the leaves by colors $\{C_2..C_k\}$, not by their order in T_1, but according to their left-to-right order in $CPDT(T_2)$. (Recall that by definition, the CPDT is an ordered tree).

(b) Counting Good Triplets and Good Fans: By definition, good triplets and good fans are created *only* when leaves are colored by NON-RED colors. Therefore, we let the algorithm compute the number of newly created good triplets and good fans whenever the recursive coloring scheme colors a leaf l by a NON-RED color. To do this, the algorithm traverses the leaf-to-root path starting at $CPDT(l)$, and for each node v on the path, it counts the number of good triplets and good fans that include l and whose lca in the CPDT equals v by applying Lemmas 2 and 3 below.

From here on, we denote by C_b a color different from C_a in the set $\{C_1, .., C_k\}$, but in the formulas, RED and NON-RED will usually be given separate cases. (This is the reason why sums over colors in formulas sometimes start from 2.)

Lemma 2. *Given an internal CP-node u of the CPDT and some child u_i of u, let S_i be the subtree rooted at u_i. Also, let u'_j be the j-th child of u_i and S'_j the subtree rooted at it. If there are no NON-RED leaves in $S_{>i}$ nor in $S'_{>j}$, the*

number of good triplets introduced by coloring a leaf by a color C_a in S'_j, $a \geq 2$, such that their lca is u, is:

$$\binom{Red(S_{<i})}{2} + \sum_{b=2,b\neq a}^{d} \binom{C_b(S_{<i})}{2} + \sum_{h>i} Red^{(2)}(S_h) + C_a(S_{\leq i}) \cdot Red(S_{>i}) + C_a(S'_j) \cdot Red(S_{<i}) + C_a(S'_j) \cdot C_{\bar{a}}(S_{<i})$$

while the number of good fans, whose lca is u, is:

$$C_{R\bar{a}}(S_{<i}) \cdot C_{R\bar{a}}(S'_{<j}) - \sum_{b=1,b\neq a}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) + C_{\bar{a}}(S_{<i}) \cdot RED(S'_{>j})$$

Lemma 3. *Given an SN-node v of the CPDT and some child v_i of v, let S_i be the subtree rooted at v_i. If there are no NON-RED leaves in $S_{>i}$, the number of good triplets introduced by coloring a leaf by a color C_a in S_i, $a \geq 2$, such that their lca is v, is:*

$$\sum_{j=1}^{i-1} \binom{Red(S_j)}{2} + \sum_{j>i} \binom{Red(S_j)}{2} + \sum_{b=2,b\neq a}^{d} \sum_{j=1}^{i-1} \binom{C_b(S_j)}{2} + C_a(S_i) \cdot (Red(S_{<i}) + Red(S_{>i})) + C_a(S_i) \cdot C_{\bar{a}}(S_{<i})$$

while the number of good fans, whose lca is v, is:

$$\binom{C_{R\bar{a}}(S_{<i})}{2} - \sum_{b=1,b\neq a}^{d} \binom{C_b(S_{<i})}{2} - \sum_{h=1}^{i-1} \binom{C_{R\bar{a}}(S_h)}{2} + \sum_{h=1}^{i-1} \sum_{b=1,b\neq a}^{d} \binom{C_b(S_h)}{2} + C_{\bar{a}}(S_{<i}) \cdot RED(S_{>i})$$

3.3 Time Complexity Analysis

The values in Lemmas 2 and 3 for any specified node in the CPDT can be obtained by a direct method in $O(n)$ time. This will be too slow for our purposes, so we first reduce it to $O(\log n)$ time (Lemmas 5 and 6). The solution uses the *range sum query data structure* (RSQ), a data structure for representing an array of non-negative integers $A[1..n]$ so that it is possible to:

1. given an index $i \in [1..n]$, change the value of $A[i]$;
2. given two positions $s, t \in [1..n]$, where $s \leq t$, return the value $\sum_{i=s}^{t} A[i]$.

Given an RSQ R, we refer to the array of numbers over which R supports queries as $R.A$.

We shall rely on the following result from the literature:

Lemma 4. *An RSQ supporting operations 1 and 2 in $O(\log n)$ time can be implemented in $O(n)$ space and $O(n)$ preprocessing time using a* Fenwick tree [8].

Now, for each node v in the CPDT, define and store the following set of counters, where $\{v_1, v_2, .., v_k\}$ denotes the set of children of v:

$$\begin{cases} C_c(v),\ \forall c \in 2..d, \text{as defined in Sect. 3.1} \\ C(v) = \sum_{c=2}^{d} C_c(v) \\ C_c^2(v) = \binom{C_c(v)}{2},\ \forall c \in 2..d \\ C^2(v) = \sum_{c=2}^{d} C_c^2(v) \\ C_c^{(2)}(v) = \sum_{i=1}^{k} \binom{C_c(v_i)}{2},\ \forall c \in 2..d \\ C^{(2)}(v) = \sum_{c=2}^{d} C_c^{(2)}(v) \\ SS(v) = \sum_{i=1}^{k} \binom{\sum_{b=2}^{d} C_b(v_i)}{2} + \sum_{b=2}^{d} C_b(v_i) \cdot Red(v_i) \\ SS_c(v) = \sum_{i=1}^{k} C_c(v_i) \cdot (C_{\overline{c}}(v_i) + Red(v_i)) + \binom{C_c(v_i)}{2},\ \forall c \in 2..d \end{cases}$$

Also store three RSQs, named $R_1(v)$, $R_2(v)$, and $R_3(v)$, such that $R_1(v).A[i] = Red(v_i)$, $R_2(v).A[i] = \binom{Red(v_i)}{2}$, and $R_3(v).A[i] = Red^{(2)}(v_i)$.

Lemma 5. *The values in Lemma 2 can be found in $O(\log n)$ time.*

Proof. (Refer to the notation introduced back in Lemma 2). By the definition of the coloring scheme, no NON-RED leaf is in $S_{>i}$. Therefore, $C_c(u) = C_c(S_{\leq i})$. Since $C_c(u_i) = C_c(S_i)$, it is straightforward to compute $C_c(S_{<i})$. A similar argument works for every counter: starting from its values for u and u_i, we can compute its value for $S_{<i}$ easily.

Next, when coloring leaves in S'_j, all leaves in $S_{<i}$ have already been colored. Thus, the values $C_c(S_{<i})$, $\forall c \in 2..d$, are fixed. We keep track of the current value of $C_c(S_{<i}) \cdot C_c(S'_{<j})$, $\forall c \in 2..d$, in S'_j as we color leaves in it, plus the value $\sum_{b=2}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j})$. Then: $\sum_{b=1, b\neq a}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) = \sum_{b=2}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) - C_a(S_{<i}) \cdot C_a(S'_{<j}) + Red(S_{<i}) \cdot Red(S'_{<j})$.

The other quantities can be deduced using the counters and the RSQs defined above. When making range queries on them, we apply Lemma 4, which gives a time complexity of $O(\log n)$. □

Lemma 6. *The values in Lemma 3 can be found in $O(\log n)$ time.*

Proof. The only non-trivial quantity here is $\sum_{h=1}^{i-1} \binom{\sum_{b=1, b\neq a}^{d} C_b(S_h)}{2} = SS(S_{<i}) + \sum_{h=1}^{i-1} \binom{Red(S_h)}{2} - SS_a(S_{<i})$, which we explain now. We need to compute the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$, and none of the leaves is colored by C_a. $SS(S_{<i})$ is the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$, but the two leaves are not both colored RED. Adding $\sum_{h=1}^{i-1} \binom{Red(S_h)}{2}$, we remove the

latter restriction, thus getting the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$. Finally, we remove $SS_a(S_{<i})$, which is the number of pairs of colored leaves in the same subtree such that at least one leaf is colored by C_a.

As in the proof of Lemma 5, the other quantities can be obtained in $O(\log n)$ time using Lemma 4 and the counters and RSQs above. □

During the execution of the algorithm, the number of good triplets and good fans created when coloring a leaf NON-RED can be obtained by applying Lemmas 2 and 3 to the leaf and all its ancestors in the CPDT; Lemmas 5 and 6 provide these values for any specified node of the CPDT in $O(\log n)$ time.

To ensure that Lemmas 5 and 6 can still be applied after leaves are recolored, the counters and RSQs for certain nodes need to be updated. More precisely, we extend the algorithm so that:

- Whenever a leaf is colored RED or WHITE, we traverse its leaf-to-root path in the CPDT and update every RSQ on it, taking $O(\log n)$ time per node by Lemma 4.
- Whenever a leaf is colored NON-RED, we traverse its leaf-to-root path in the CPDT and, after applying Lemmas 5 and 6 to each node on the path, we update its counters in $O(1)$ time.

In summary, each node in the CPDT that is visited after a leaf recoloring can be taken care of in $O(\log n)$ time. This gives:

Theorem 1. *The time complexity of the new algorithm is* $O(n \log^3 n)$.

Proof. Constructing $CPDT(T_2)$ in the first step takes $O(n)$ time. The construction follows directly from the definition of $CPDT$.

By Brodal *et al.*'s analysis in [2], a total of $O(n \log n)$ leaf colorings occur. Whenever a leaf is colored, we visit all nodes on its leaf-to-root path in the CPDT; its length is $O(\log n)$, leading to a total of $O(n \log^2 n)$ node visits in the CPDT. (Observe that although some nodes such as the root may be visited $\Omega(n \log n)$ times, the total number of node visits is bounded by $O(n \log^2 n)$.)

By the comments after Lemma 6, $O(\log n)$ time is used for each node visit in the CPDT. Hence, the total running time of the algorithm is $O(n \log^3 n)$. □

4 Implementation

We have implemented the new algorithm for the rooted triplet distance in two versions: a special binary trees-only optimized version, and one for general trees. The importance of the special case where both trees are binary justifies a dedicated implementation. Only plain standard C++ was used, expect for an (optional) single feature from C++11, mentioned below. The source code can be downloaded from:
http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/CPDT-dist.html

A few points and optimizations that improve the implementation's running time in practice are discussed next.

4.1 Representation of the Counters

The first issue is how to efficiently represent the counters defined in Sect. 3.3. Let d be the number of children of the highest degree node in T_1: as noted in [10], if we naively represent the counters using d-long vectors for each node in the CPDT, we may end up with quadratic memory (and thus quadratic time) when d is close to n, e.g., if all the leaves are directly attached to the root of T_1. However, we do *not* actually need d counters in each node. At any time, the number of colors used in any subtree of the CPDT cannot be larger than the number of its leaves; therefore, $O(n \log n)$ counters are needed in total.

We implemented the counters as follows. In any given node v in the CPDT, for each set of counters, we allocate an array of length $min\{d, leaves(v)\}$, where $leaves(v) = |L(CPDT(v))|$ is the number of leaves in the subtree of the CPDT rooted at v. Note that when the length of the arrays is $<d$, counters for color $k \in 1..d$ may not be stored in position k in the arrays, so we use a map int-to-int to maintain an association between the (used) colors in $1..d$ and their actual position in the arrays.

We used a hashmap to implement the maps, which allows constant-time insertion and retrieval. We tested two different implementation of the hashmap: the C++11 unordered_map [16] included in the standard library of our test system, requiring C++11 support, and the dense_hash_map class in the (former) Google project sparsehash [15]. We can choose which one to use at compile-time. Some experimental results for both libraries are reported in Sect. 5.

4.2 Two-Step Coloring

In the theoretical version of the algorithm, for simplicity, when coloring leaves by NON-RED colors in left-to-right order in the CPDT, we take each leaf in order and count the good triplets rooted at each of its ancestors up to the root; it is evident that some nodes are considered many times.

In the implementation, we do it slightly differently: First, we mark all of the nodes in the CPDT we need to consider, i.e., all nodes that are ancestors of at least one leaf being colored: for each leaf, we start at it and go up until we either end at the root or at an already marked node, marking all the nodes we traverse. Second, we visit the marked subtree in post-order; when we actually consider a given node, we already know how many leaves we are coloring for each color; modifying the formulas in Lemmas 2 and 3 to count all of the good triplets introduced by such leaves, for each color, in one go is trivial.

4.3 The Coloring Scheme

A few optimizations were made to the coloring scheme.

First, consider what happens when the coloring scheme begins. We start by coloring all the leaves RED, only to immediately recolor leaves not in the biggest subtree of the root, first WHITE and then by NON-RED colors. This happens many times, i.e., every time we color a subtree RED and immediately recurse

on it. We save some unnecessary operations by only coloring RED leaves in the biggest subtree. Since coloring a leaf RED and WHITE requires updating a number of RSQs and is a fairly expensive operation, this saves us some time.

Next, cherry nodes (i.e., internal nodes with exactly two leaves attached) do not need to be colored, since they can not yield any good triplet. If we do not consider them, we can save a lot of RED and WHITE colorings. Cherry nodes are fairly frequent, especially in binary/low branching trees.

Finally, when we color a single leaf (or even a sufficiently low number of leaves), the two-step coloring can actually be a burden. Thus, we made a few special routines that directly update the leaf-to-root path of a given leaf, without the need of marking it first.

5 Experiments

We compared the running time and memory usage in practice of the new algorithm to that of tqDist [14] by a series of experiments, as described in this section.

5.1 Experimental Setup

The experiments were performed on a computer running Ubuntu 12.04, with an Intel Xeon W3530 (quad-core, 2.8 GHz) and 16 GB of RAM. The system C++ compiler was `g++`, version 4.6.3.

We used the C++-implementations of our algorithm presented in Sect. 4: one for general trees and a special binary trees-only optimized version. As mentioned in Sect. 4, the algorithm can be compiled using two different hashmaps: C++11 unordered_map [16] (we named this CPDT) and sparsehash [15] (named CPDTg). We will improperly refer to CPDT and CPDTg as "implementations" of our algorithm, but they are actually a single implementation linked against two different hashmap libraries. tqDist [14] was built from its source code using `cmake`, as instructed by the authors. We had to disable the HDT dynamic contraction [10] of tqDist as it was making the tool run tens of times slower; built with the default parameters, it would usually take more than an hour for two random 1 million leaves trees.

Running times were measured using the `time` command, which gives the sum of system and user times and includes the time spent parsing the trees from a file; the average over 50 runs was taken. Memory usage was measured with Valgrind [13] and its heap profiling tool Massif. Due to the slowdown caused by Valgrind, we took the average over 20 runs.

5.2 Input Trees

Our implementations and tqDist were applied to pairs of trees with values of n up to $1,000,000$. Arbitrary-degree input trees were generated as follows:

> First, generate a binary tree with n leaves in the uniform model [12]. Then, for each non-root, internal node v in the tree, contract it (i.e., make the children of v become children of v's parent, and remove v) with some fixed probability p.

Below, we let p_i for $i \in \{1, 2\}$ be the chosen value of p when generating T_i. We used three values of p_1 and p_2: 0.2, 0.5, and 0.8, calling the generated trees *lowly-branching*, *moderately-branching*, and *highly-branching*, respectively. This gave 9 sets of benchmarks. In addition, we created a set where both trees are binary (equivalent to the case $p_1 = p_2 = 0$) and two extra sets where p_1 is 0.95 (resp. 0.2) and p_2 is 0.2 (resp. 0.95) in order to test the behavior of the algorithms when dealing with *extremely-branching* trees, i.e., flat trees with very high degree. Also, all sets of benchmarks were executed on pairs of *unrelated* as well as *related* trees, where unrelated trees were generated independently of each other and related trees were generated from the same binary tree.

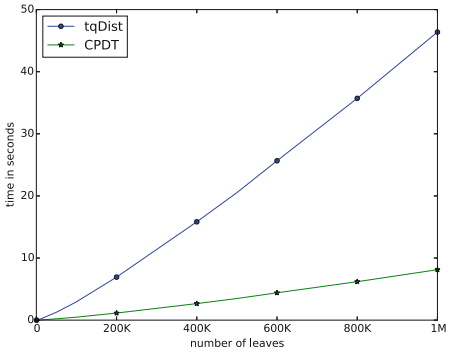
5.3 Results

Table 1 reports the average running times of the three implementations tested on pairs of trees with 1 million leaves, along with the relative speed-ups over tqDist. Figure 2 plots the average running times as a function of n for binary trees as well as for non-binary trees obtained using some representative (p_1, p_2)-values.

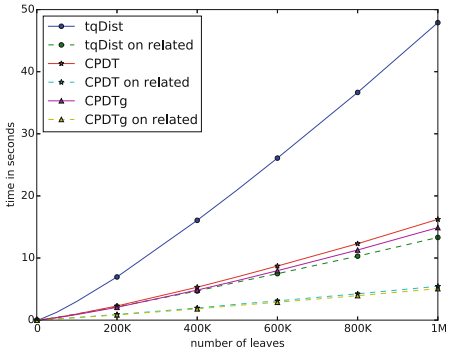
The binary case benefits greatly from having a special implementation, being faster than the more general implementation for arbitrary-degree trees, and

Table 1. Average running times, in seconds, on two 1M leaves trees, and relative speed-ups over tqDist.

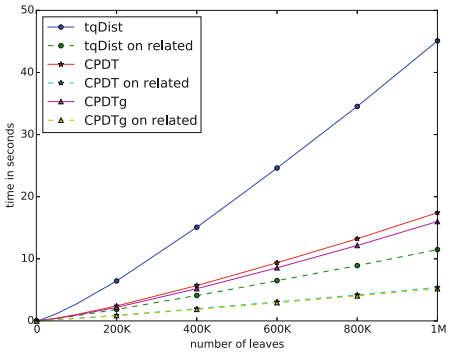
		Unrelated trees						Related trees					
p_1	p_2	tqDist		CPDT		CPDTg		tqDist		CPDT		CPDTg	
0.0	0.0	46.38	1.00x	8.12	5.71x	-	-	-	-	-	-	-	-
0.2	0.2	47.90	1.00x	16.23	2.95x	14.90	3.21x	13.31	1.00x	5.45	2.44x	5.06	2.63x
0.2	0.5	46.71	1.00x	17.89	2.61x	16.33	2.86x	12.04	1.00x	5.57	2.16x	5.08	2.37x
0.2	0.8	40.26	1.00x	15.67	2.57x	14.19	2.84x	10.29	1.00x	4.76	2.16x	4.28	2.40x
0.2	0.95	30.87	1.00x	11.26	2.74x	10.21	3.02x	9.09	1.00x	4.00	2.27x	3.59	2.53x
0.5	0.2	46.25	1.00x	15.78	2.93x	14.58	3.17x	12.75	1.00x	5.28	2.41x	5.11	2.50x
0.5	0.5	45.09	1.00x	17.42	2.59x	16.00	2.82x	11.48	1.00x	5.40	2.13x	5.17	2.22x
0.5	0.8	38.80	1.00x	15.18	2.56x	13.87	2.80x	9.79	1.00x	4.57	2.14x	4.35	2.25x
0.8	0.2	43.18	1.00x	14.67	2.94x	13.62	3.17x	11.91	1.00x	4.95	2.41x	4.70	2.53x
0.8	0.5	42.21	1.00x	16.36	2.58x	15.12	2.79x	10.61	1.00x	5.02	2.11x	4.70	2.26x
0.8	0.8	35.88	1.00x	14.08	2.55x	12.95	2.77x	8.96	1.00x	4.20	2.13x	3.86	2.32x
0.95	0.2	37.14	1.00x	12.72	2.92x	11.75	3.16x	11.15	1.00x	4.79	2.33x	4.19	2.66x



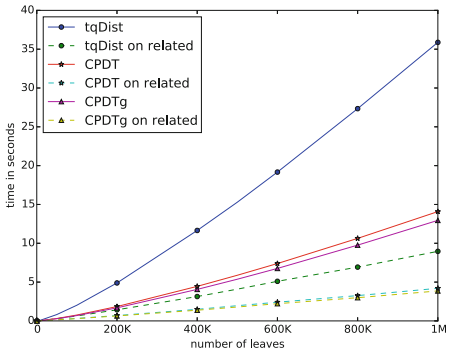
(a) Binary trees



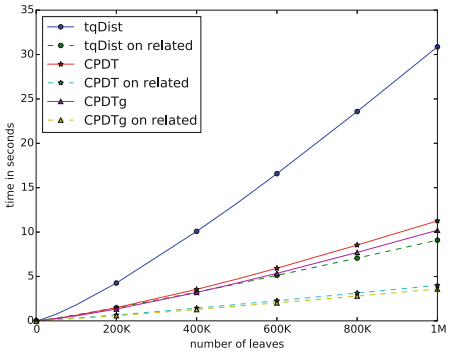
(b) $p_1 = 0.2$ $p_2 = 0.2$



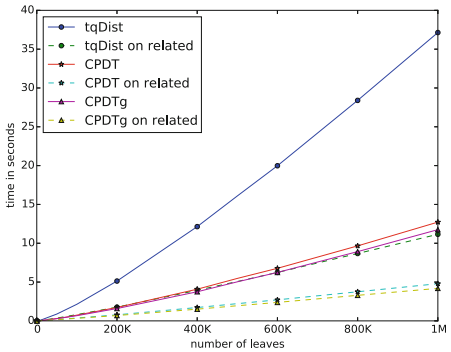
(c) $p_1 = 0.5$ $p_2 = 0.5$



(d) $p_1 = 0.8$ $p_2 = 0.8$

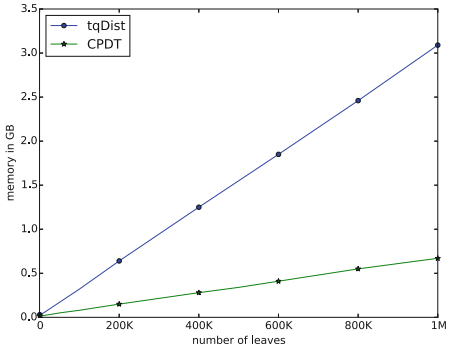


(e) $p_1 = 0.2$ $p_2 = 0.95$

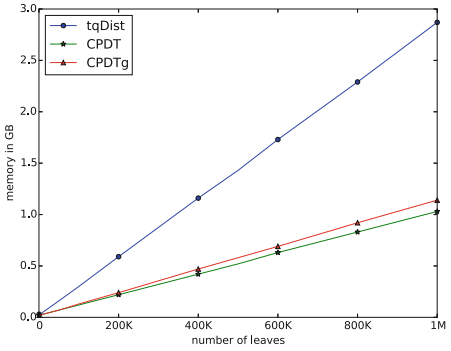


(f) $p_1 = 0.95$ $p_2 = 0.2$

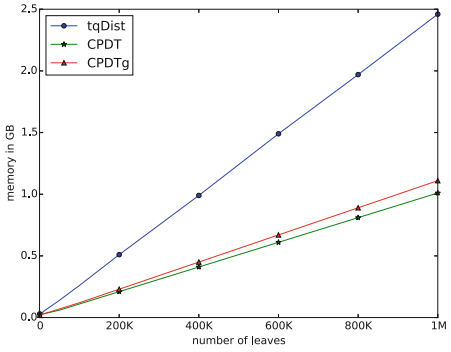
Fig. 2. Plots of the average running time in seconds (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2). Solid lines represent values on unrelated trees, while dashed lines represent values on related trees; the notion of related trees is not applicable to binary trees.



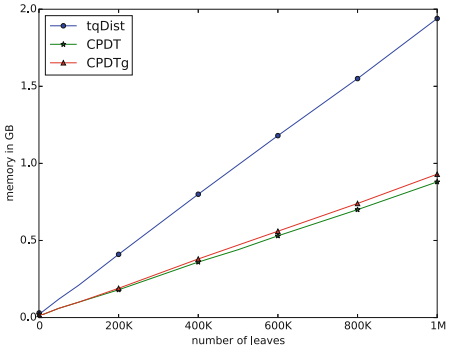
(a) Binary trees



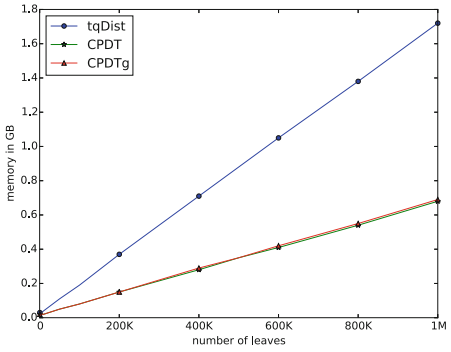
(b) $p_1 = 0.2$ $p_2 = 0.2$



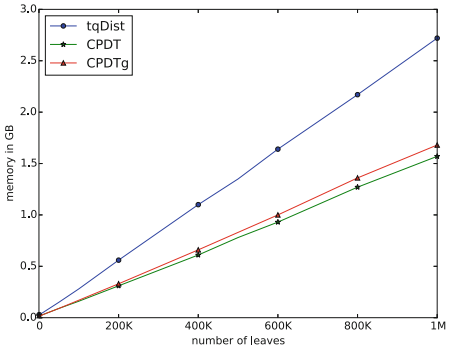
(c) $p_1 = 0.5$ $p_2 = 0.5$



(d) $p_1 = 0.8$ $p_2 = 0.8$



(e) $p_1 = 0.2$ $p_2 = 0.95$



(f) $p_1 = 0.95$ $p_2 = 0.2$

Fig. 3. Plots of the memory usage in gigabytes (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2).

Table 2. Average memory usage, in GBs, on two 1M leaves trees, and the relative memory usage decrease over tqDist.

p_1	p_2	tqDist		CPDT		CPDTg	
0.0	0.0	3.09	1.00x	0.67	4.61x	-	-
0.2	0.2	2.87	1.00x	1.03	2.79x	1.14	2.52x
0.2	0.5	2.53	1.00x	0.99	2.56x	1.08	2.34x
0.2	0.8	2.05	1.00x	0.81	2.53x	0.87	2.36x
0.2	0.95	1.72	1.00x	0.68	2.53x	0.69	2.49x
0.5	0.2	2.77	1.00x	1.04	2.66x	1.15	2.41x
0.5	0.5	2.46	1.00x	1.01	2.44x	1.11	2.22x
0.5	0.8	2.00	1.00x	0.81	2.47x	0.87	2.30x
0.8	0.2	2.73	1.00x	1.13	2.42x	1.24	2.20x
0.8	0.5	2.37	1.00x	1.14	2.08x	1.24	1.91x
0.8	0.8	1.94	1.00x	0.88	2.20x	0.93	2.09x
0.95	0.2	2.72	1.00x	1.57	1.73x	1.68	1.62x

showing nearly six-fold improvement over tqDist. Note that as it does not rely on hashmaps, we have a single implementation of the CPDT.

For unrelated arbitrary-degree trees, CPDTg is clearly the fastest implementation, consistently being around three times faster than tqDist and showing noticeable improvements over CPDT. While tqDist performs better as we increase the values of p_1 and p_2, CPDT and CPDTg only show this trend with p_1, performing worse when p_2 is 0.5 and getting better as it moves away from it. All of the implementations perform very well when T_1 is extremely-branching, proving them to be able to easily handle a huge number of colors.

For related trees, all three implementations become much faster. CPDTg still has an obvious advantage, although speed-ups here are around 2.5x. This can be at least partially explained by overheads, such as tree parsing from files, becoming more significant as the running time of the actual algorithms decrease.

The average memory usage of the three implementations for pairs of unrelated trees with 1 million leaves are reported in Table 2 and Fig. 3. CPDT is the least memory-hungry, showing an improvement over tqDist of 4.61x on binary trees and up to 2.79x on arbitrary-degree ones; CPDTg is a close second. They all benefit from increasing p_2 (meaning less internal nodes in T_2) and suffer from increasing p_1 (meaning more colors), although tqDist proves to be less sensitive to it, as the advantage of CPDT is brought down to a still very respectable 1.73x in the extreme case where T_1 is extremely-branching.

6 Concluding Remarks

Some questions for future research are: Can the theoretical or practical running times of the CPDT-based algorithm be reduced? In particular, the CPDT

respects the definition of *locally balanced* in [2], so can the analysis be refined to prove that the time complexity of the new algorithm is in fact $O(n \log^2 n)$ using the technique in Sect. 5 of [2]? To make the algorithm faster in practice, one might try to parallelize it; unfortunately, this may be difficult due to possible imbalance in the trees and the intrinsic data-dependencies of the algorithm.

Computing the quartet distance for unrooted trees seems more difficult than computing the rooted triplet distance for rooted trees. It would be interesting to see if the CPDT can be adapted to get an efficient algorithm for this variant.

We remark that an unsolved open problem is whether or not the rooted triplet distance can be computed in $O(n)$ time. A linear-time algorithm would require a set of totally different techniques than the ones used here since Brodal *et al.*'s recursive recoloring scheme already introduces $\Omega(n \log n)$ work.

References

1. Bansal, M.S., Dong, J., Fernández-Baca, D.: Comparing and aggregating partially resolved trees. Theor. Comput. Sci. **412**(48), 6634–6652 (2011)
2. Brodal, G.S., Fagerberg, R., Mailund, T., Pedersen, C.N.S., Sand, A.: Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013), pp. 1814–1832. SIAM (2013)
3. Cole, R., Farach-Colton, M., Hariharan, R., Przytycka, T., Thorup, M.: An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. SIAM J. Comput. **30**(5), 1385–1404 (2000)
4. Critchlow, D.E., Pearl, D.K., Qian, C.: The triples distance for rooted bifurcating phylogenetic trees. Syst. Biol. **45**(3), 323–334 (1996)
5. Dobson, A.J.: Comparing the shapes of trees. In: Street, A.P., Wallis, W.D. (eds.) Combinatorial Mathematics III. LNM, vol. 452, pp. 95–100. Springer-Verlag, Heidelberg (1975)
6. 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)
7. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates Inc, Sunderland (2004)
8. Fenwick, P.M.: A new data structure for cumulative frequency tables. Softw.: Pract. Experience **24**(3), 327–336 (1994)
9. Gusfield, D., Eddhu, S., Langley, C.: Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. J. Bioinform. Comput. Biol. **2**(1), 173–213 (2004)
10. Holt, M.K., Johansen, J., Brodal, G.S.: On the scalability of computing triplet and quartet distances. In: Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX 2014), pp. 9–19. SIAM (2014)
11. Jansson, J., Lingas, A.: Computing the rooted triplet distance between galled trees by counting triangles. J. Discrete Algorithms **25**, 66–78 (2014)
12. McKenzie, A., Steel, M.: Distributions of cherries for two models of trees. Math. Biosci. **164**(1), 81–92 (2000)
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM (2007)

14. 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)
15. sparsehash project webpage. https://code.google.com/p/sparsehash/
16. Documentation for unordered_map. http://www.cplusplus.com/reference/unordered_map/unordered_map/