JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 24, Number 2, 2017

Pp. 106–126
DOI: 10.1089/cmb.2016.0185

A More Practical Algorithm for the Rooted Triplet Distance

JESPER JANSSON[1] and RAMESH RAJABY[2,3]

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. that computes it in $O(n \log n)$ time and $O(n \log n)$ space, where n is the number of leaf labels, has recently been implemented in the software package tqDist. In this article, 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 and $O(n \log n)$-space algorithm that, although slower in theory, is faster in practice as well as less memory consuming. Simulations for values of n up to 4,000,000 support our claims experimentally.

Keywords: phylogenetic tree comparison, centroid path decomposition tree, implementation, rooted triplet distance.

1. INTRODUCTION

Evolutionary relationships between biological species or natural languages are often represented in treelike structures known as *phylogenetic trees* (or *phylogenies*) (Felsenstein, 2004; Nakhleh et al., 2005; Sung, 2010). Over the years, many alternative methods for inferring phylogenetic trees have been developed; see, for example, Felsenstein (2004). Owing to errors in experimentally obtained data or the inherent instability of classifications, applying the same tree inference method to different data sets for a set of species or changing the assumed model of evolution may result in trees whose leaf label sets are the same but whose branching patterns are different. In this case, 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 (Bansal et al., 2011) 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.

[1]Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Kyoto, Japan.
[2]NUS Graduate School for Integrative Sciences and Engineering, National University of Singapore, Singapore, Singapore.
[3]University of Milano-Bicocca, Milano, Italy.

A preliminary version of this article appeared in Proceedings of the 2nd International Conference on Algorithms for Computational Biology (AlCoB 2015), volume 9199 of Lecture Notes in Computer Science, pp. 109–125, Springer International Publishing, Switzerland, 2015.

Several measures of the (dis-)similarity of two phylogenetic trees with identical leaf label sets have been suggested in the literature (Bansal et al., 2011). One such measure is the *rooted triplet distance* (Dobson, 1975), which counts how many of the subtrees induced by cardinality-3 subsets of the leaves that differ between the two trees. This article presents a practical algorithm for computing the rooted triplet distance, based on the framework introduced in an algorithm by Brodal et al. (2013), along with its implementation.

1.1. Basic definitions

Let T be a rooted tree. The *degree of a node* u in T is the number of children of u, and the *degree of* T is the maximum of the degrees of all nodes in T. In this article, 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 $u \in V(T)$, $T(u)$ is the subtree of T rooted at u, that is, the subgraph of T induced by the node u and all of its proper descendants in T. For any $l \in \Lambda(T)$, $T(l)$ is the leaf in T with leaf label l. For any $u, v \in V(T)$, $lca^T(u, v)$ is the lowest common ancestor (lca) in T of u and v. Also, for any $u, v \in V(T)$, if u is a proper descendant of v, then we write $u \prec v$.

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 nonbinary 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. By definition, $|rt(T)| = \binom{|\Lambda(T)|}{3}$.

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

$$d_{rt}(T_1, T_2) = \frac{|rt(T_1) \Delta rt(T_2)|}{2},$$

where Δ stands for the symmetric difference between two sets. In other words, $d_{rt}(T_1, T_2)$ equals the number of cardinality-3 subsets of the leaf label set that give rise to a conflict in the branching patterns of T_1 and T_2. See Figure 1 for an example. Intuitively, the rooted triplet distance considers two trees that share many small embedded subtrees to be similar. 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.

Henceforth, 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, we 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 (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. (1996) gave an $O(n^2)$-time algorithm for the special case where T_1 and T_2 are *binary*, and Bansal et al. (2011) 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. (2013) achieved a time complexity of $O(n \log n)$ for two trees of arbitrary degrees.

An implementation of Brodal et al.'s (2013) fast algorithm, written in C++, is available in the free software package tqDist (Sand et al., 2014). The rooted triplet distance can also be computed in an older

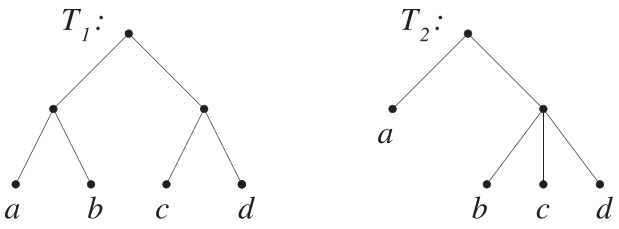


FIG. 1. An example. Here, $rt(T_1) = \{ab|c, ab|d, cd|a, cd|b\}$ and $rt(T_2) = \{bc|a, bd|a, cd|a, b|c|d\}$, so $d_{rt}(T_1, T_2) = 3$.

software package named EPoS (Griebel et al., 2008), but because EPoS is much slower than tqDist and typically runs out of memory for trees with $n > 10,000$, it will not be included in the experimental evaluation hereunder.

As for other related work, the counterpart of the rooted triplet distance for *unrooted* trees is the *unrooted quartet distance* (Estabrook et al., 1985). The currently fastest algorithm for computing the unrooted quartet distance (Brodal et al., 2013) runs in $O(dn \log n)$ time, where n is the number of leaf labels and d is the maximum number of neighbors of any node in the two input trees. An extension of the rooted triplet distance to *phylogenetic networks* was introduced in Gambette and Huber (2012) and further studied in Jansson and Lingas (2014); for two *galled trees* (Gusfield et al., 2004) (phylogenetic networks whose cycles are disjoint) with n leaves each, the rooted triplet distance can be computed in $o(n^{2.687})$ time (Jansson and Lingas, 2014).

1.3. Our contributions

We present some nontrivial modifications to Brodal et al.'s (2013) algorithm for computing $d_{rt}(T_1, T_2)$ for two trees of arbitrary degrees that make it 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 (Sand et al., 2014), the publicly available implementation of Brodal et al.'s algorithm, for various types of large inputs consisting of two trees with up to 4 million leaves each.

The article is organized as follows. Brodal et al.'s (2013) algorithm is reviewed in Section 2. Section 3 describes the new algorithm, Section 4 discusses implementation issues, and Section 5 presents the experimental results. Finally, Section 6 contains some concluding remarks.

2. SUMMARY OF BRODAL ET AL.'S ALGORITHM

On a high level, the algorithm of Brodal et al. (2013) works as follows. Each rooted triplet t in $rt(T_1)$ is implicitly assigned to the lca 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 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 their corresponding leaves in T_1 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 coloring.

A naive implementation of this idea leads to a quadratic time complexity. As demonstrated in Brodal et al. (2013), the time complexity can be reduced to $O(n \log n)$ by employing two special tools. The first one is a data structure called a *hierarchical decomposition tree* (HDT) that represents T_2 so that the number of rooted triplets in $rt(T_2)$ compatible with a given leaf coloring of T_2 can be retrieved quickly. How to construct the HDT, augment it with auxiliary information to support the relevant queries, and update this information when the leaves of T_2 are recolored are somewhat complicated and will not be described here; see Brodal et al. (2013) for details. The second tool is a recursive recoloring scheme that visits the nodes of T_1 systematically to generate all the required leaf colorings of T_2 while avoiding unnecessary recoloring operations. The recoloring scheme will be needed in Section 3 hereunder, so the rest of this section explains how it works.

Let $\{C_0, C_1, \ldots, C_d\}$ be a set of colors, where d is the degree of T_1. In the recoloring scheme, whenever a leaf in T_1 is assigned a color, the leaf in T_2 with that leaf label is automatically assigned the same color. The scheme does a depth-first traversal of T_1 while maintaining two invariants:

(i) When entering a node u, all leaf descendants of u have the color C_1 and all other leaves have the color C_0.
(ii) When exiting a node u, all leaves in the entire tree have the color C_0.

Initially, all leaves in T_1 are colored by C_1 so that invariant (i) holds when the traversal starts at the root of T_1. During the traversal, whenever an internal node $u \in V(T_1)$ is reached, the color C_i is assigned to all leaves in the subtree rooted at u_i for $i \in \{2, 3, \ldots, k\}$, where k is the degree of u, u_1 is any child of u with the largest number of leaf descendants, and $u_2, u_3, \ldots, u_k$ are the other children of u in any order. (At this point, the trees are said to be *colored according to* u and the main algorithm can check the number of

elements in $rt(T_2)$ that are compatible with this coloring by querying the HDT.) Next, the leaves in the $k-1$ subtrees that were just colored are recolored by the color C_0 and the scheme recurses on the subtree rooted at u_1, whose leaves still have the color C_1. Observe that the first invariant holds when entering the root of this subtree. After returning from the recursive call, all leaves in the subtree rooted at u_1 have color C_0 by invariant (ii), and the subtrees rooted at the other children of u are treated one by one; for each such subtree, the scheme colors all of its leaves by color C_1 and then recurses on it. Again, invariant (i) holds at each recursive call because the leaves in the subtree are colored by C_1 and everything else has color C_0. After handling all k subtrees of u, all leaf descendants of u will have the color C_0, so invariant (ii) holds when exiting from u. The base case of the recursion is when the reached node $u \in V(T_1)$ is a leaf; in this case, the scheme colors u by C_0 and exits so that invariant (ii) holds.

As shown by Brodal et al., a leaf coloring of T_2 for every internal node of T_1 is generated by the scheme and the total number of leaf recolorings is $O(n \log n)$.

3. THE NEW ALGORITHM

The new algorithm is described in this section. It uses the same framework as Brodal et al.'s (2013) algorithm. To be precise, it also implicitly assigns each rooted triplet in $rt(T_1)$ to an internal node in T_1 and determines, for each node in T_1, how many of its assigned rooted triplets appear in $rt(T_2)$. The significant 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 HDT data structure, the new algorithm uses the (in our opinion) conceptually simpler centroid path decomposition technique (Cole et al., 2000).

3.1. Preliminaries

For convenience, we make T_1 and T_2 into *ordered* trees by imposing the following left-to-right ordering on the children of each node: for every nonleaf node v in a tree, the leftmost child of v is a child of v having the most leaf descendants (with ties broken arbitrarily) and the other children of v are ordered arbitrarily. Rooted triplets are always unordered trees, so the definitions of $rt(T_1)$, $rt(T_2)$, and $d_{rt}(T_1, T_2)$ are unaffected by the left-to-right ordering of T_1 and T_2.

We first introduce some notation related to leaf colorings corresponding to the internal nodes of T_1. Let d be the degree of T_1. Define a set of $d+1$ *colors* $\{C_0, C_1, \ldots, C_d\}$. We usually refer to C_1 as *RED* and C_0 as *WHITE*, and a leaf that is neither RED nor WHITE as *NON-RED*. The 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 $\{v_1, v_2, \ldots, v_x\}$ be the children of an internal node $v \in V(T_1)$ with degree x. 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 for $i \in \{1, 2, \ldots, x\}$ and
- l is WHITE if and only if $T_1(l)$ is not a descendant of v.

Define a *triplet* as any subset of L of cardinality-3; 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. 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 Figure 2 in T_2, where C_a and C_b are colors in $\{C_1, C_2, \ldots, C_d\}$ and $a < b$. Similarly, t is called a *good fan* if all its three leaves have different colors from the set $\{C_1, C_2, \ldots, C_d\}$. (This corresponds to the concept of a triplet being ''compatible with a coloring'' in Brodal et al. (2013).)

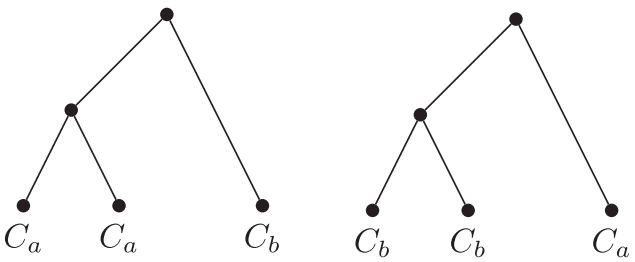


FIG. 2. Topologies induced by good triplets in T_2.

For a given color C_c and S either a single subtree or a set of subtrees, let $C_c(S)$ be the number of leaves in S colored by C_c. $C_1(S)$ will 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 that 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 that are not colored by C_a. Finally, when S is a single tree, define $st(S)$ as the set of subtrees rooted at the children of the root of S, and $Red^{(2)}(S) = \sum_{S_i \in st(S)} \binom{Red(S_i)}{2}$. We will sometimes use a node as an argument, meaning the subtree rooted at it.

Next, recall the concept of a *centroid path* (Cole et al., 2000). A *centroid path* starting at any node v in a tree T is a path from v to some leaf of T that, at each internal node, always chooses a child having the largest number of leaf descendants.

For any tree T that has been ordered according to the rule introduced at the beginning of this subsection, let $cp(T)$ (''the centroid path of T'') be the centroid path that starts at the root of T and always selects the leftmost child until it reaches a leaf. The *centroid path decomposition tree* (CPDT) 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 $CPDT(T)$. Traverse $cp(T)$ from its lowest node to its highest, and for each node r_j in T encountered, add a single node u_j to the ordered set of children of u (making u_j the rightmost child so far), and then define the children of u_j as $\{CPDT(T_2^j), \ldots, CPDT(T_k^j)\}$, where k is the degree of r_j and T_i^j is the subtree of T rooted at the ith child of r_j in the left-to-right ordering (Figure 3). We call u a *CP-node* (*centroid path node*) because it represents a whole centroid path in T, and u_j an *SN-node* (*single-node node*) because it represents a single node in T. If r_j in T has degree 2, then the corresponding SN-node u_j in $CPDT(T)$ will have a single child. We immediately have the following lemma.

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

Moreover, for any $v_A, v_B \in V(T)$ with $v_A \prec v_B$ that are roots of centroid paths represented by CP-nodes in $CPDT(T)$, $|\Lambda(T(v_A))| \leq |\Lambda(T(v_B))|/2$ holds by the definition of a centroid path, so every leaf in T belongs to at most $O(\log n)$ centroid paths represented by CP-nodes, where $n = |\Lambda(T)|$. Thus, $CPDT(T)$ has height $O(\log n)$.

3.2. Description of the new algorithm

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

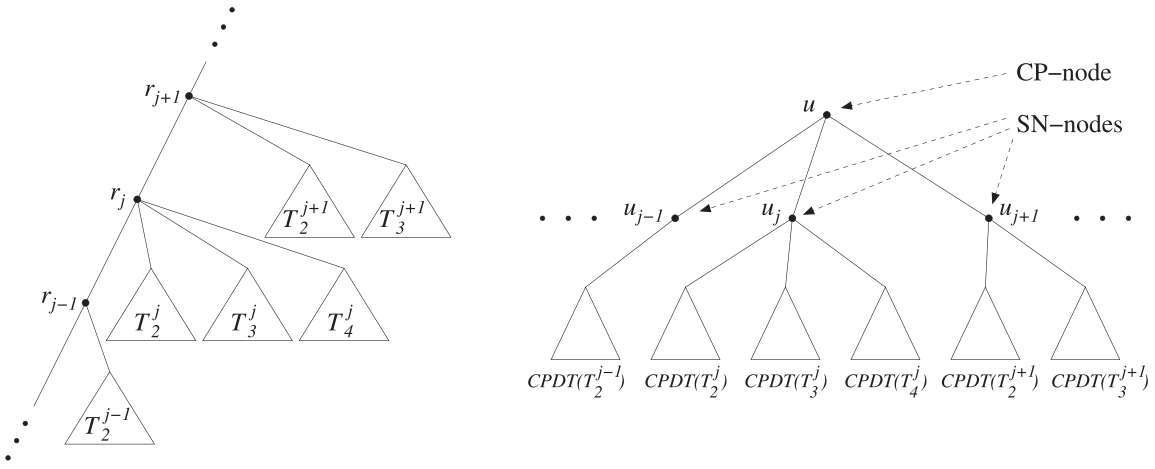


FIG. 3. Illustrating the definition of the centroid path decomposition tree (CPDT). Suppose the ordered tree T has the structure on the left. Then $CPDT(T)$ has the structure on the right. In $CPDT(T)$, the node u (representing the centroid path $cp(T)$) is a CP-node, and the children of u (representing nodes on $cp(T)$) are SN-nodes.

To do part (a), the algorithm uses Brodal et al.'s recursive recoloring scheme (summarized in Section 2) to generate all the leaf colorings, with the following two important modifications:

1. Whenever an internal node $v \in V(T_1)$ is reached, all leaves in $\Lambda(T_1(v)) \setminus \Lambda(T_1(v_1))$ are first recolored WHITE.
2. After all leaves in $\Lambda(T_1(v)) \setminus \Lambda(T_1(v_1))$ have been recolored WHITE, every leaf in the subtree rooted at v_i for $i \in \{2, 3, \ldots, k\}$ is assigned the color C_i just like in Brodal et al.'s recoloring scheme, but the sequence in which the leaves are assigned colors follows their left-to-right ordering in $CPDT(T_2)$. (Recall that by definition, the CPDT is an ordered tree.)

Section 3.4 shows that these two modifications allow one to keep track of coloring information efficiently using the CPDT. The pseudocode for part (a) is presented as procedure ColorTree in Figure 4.

For part (b) (counting good triplets and good fans), note that good triplets and good fans are created *only* when leaves are colored by NON-RED colors. Also, every good triplet in T_2 corresponds to a good triplet in $CPDT(T_2)$ whose lca is either a CP-node or an SN-node, and similarly for every good fan. The algorithm therefore computes the number of newly created good triplets and good fans whenever the recursive coloring scheme colors a leaf l by a NON-RED color as follows: the algorithm traverses the leaf-to-root path starting at leaf l in $CPDT(T_2)$, and for each node u on the path, it counts the number of good triplets and good fans that include l and whose lca in $CPDT(T_2)$ equals u. By adhering to the left-to-right ordering in $CPDT(T_2)$ when coloring the leaves, the algorithm is able to do the counting by applying either Lemmas 2 and 3 or Lemmas 4 and 5 from Section 3.3 hereunder, depending on whether u is a CP-node or an SN-node. The pseudocode for part (b) is presented as procedure ColorLeaf in Figure 5.

The main algorithm makes one call to ColorTree with the root of T_1 as the argument to obtain the total number of good triplets and good fans, and ColorTree subsequently makes multiple calls to itself and to ColorLeaf.

```
Procedure ColorTree(v)
/* Requires: v ∈ V(T1), all leaves belonging to T1(v) are RED, and all other leaves in T1 are WHITE. */
 1: if v is a leaf then
 2:     ColorLeaf(v, WHITE)
 3: else
 4:     Let v1, v2, ..., v_dv be the children of v in left-to-right order
 5:     Let M be a list of leaves in T1(v2), ..., T1(v_dv), ordered by their left-to-right order in CPDT(T2)
 6:     for all l in M do
 7:         ColorLeaf(l, WHITE)
 8:     end for
 9:     for i = 2 to d_v do
10:         for all l in Λ(T1(v_i)) do
11:             color(l) := C_i
12:         end for
13:     end for
14:     for all l in M do
15:         ColorLeaf(l, color(l))
16:     end for
17:     for all l in M do
18:         ColorLeaf(l, WHITE)
19:     end for
20:     ColorTree(v1)
21:     for i = 2 to d_v do
22:         for all l in Λ(T1(v_i)) do
23:             ColorLeaf(l, RED)
24:         end for
25:         ColorTree(v_i)
26:     end for
27: end if
```

FIG. 4. Procedure ColorTree for generating leaf colorings, based on Brodal et al.'s recoloring scheme.

```
Procedure ColorLeaf(v, color)
 1: Let u be the leaf of the CPDT corresponding to v
 2: while u is not NULL do
 3:     if color is neither RED nor WHITE then
 4:         if u is a CP-node then
 5:             Use Lemma 2 to count the number of good triplets in T_2 whose lca is u in the CPDT
 6:             Use Lemma 3 to count the number of good fans in T_2 whose lca is u in the CPDT
 7:         else if u is an SN-node then
 8:             Use Lemma 4 to count the number of good triplets in T_2 whose lca is u in the CPDT
 9:             Use Lemma 5 to count the number of good fans in T_2 whose lca is u in the CPDT
10:         end if
11:     end if
12:     Update the coloring info of u as in Section 3.4
13:     u := parent of u
14: end while
```

FIG. 5. Procedure ColorLeaf for counting good triplets and good fans.

3.3. Algorithm details and correctness

This subsection first states and proves the four key lemmas (Lemmas 2–5) that the procedure ColorLeaf in Section 3.2 uses to count newly created good triplets and good fans. The setting is as follows. Suppose that u is an internal CP-node of the CPDT, u_i is a child of u, and u'_j is a child of u_i, and that the algorithm is assigning some NON-RED color to a leaf in the subtree rooted at u'_j. Furthermore, suppose that the other leaves have previously been colored so that all leaves in the subtrees rooted at the right siblings (if any) of u_i and u'_j are RED or WHITE, whereas the other leaves can have any color (Fig. 6). The formulas in Lemmas 2 and 3 count how many good triplets and good fans whose lca equals u are introduced when this occurs, and Lemmas 4 and 5 do the same for SN-nodes.

In the formulas, RED and NON-RED will usually be given separate cases, which is why certain sums over colors in formulas start from 2. For any internal node u in the CPDT, its ordered children are denoted (from left to right) by $u_1, \ldots, u_d$, where d is the degree of u and S_i is the subtree rooted at u_i for each $i \in \{1, \ldots, d\}$. Define

$$S_{<i} = \{S : root(S) = u_j,\ j < i\}$$
$$S_{>i} = \{S : root(S) = u_j,\ j > i\}$$
$$S_{\leq i} = \{S : root(S) = u_j,\ j \leq i\}$$
$$S_{\geq i} = \{S : root(S) = u_j,\ j \geq i\}.$$

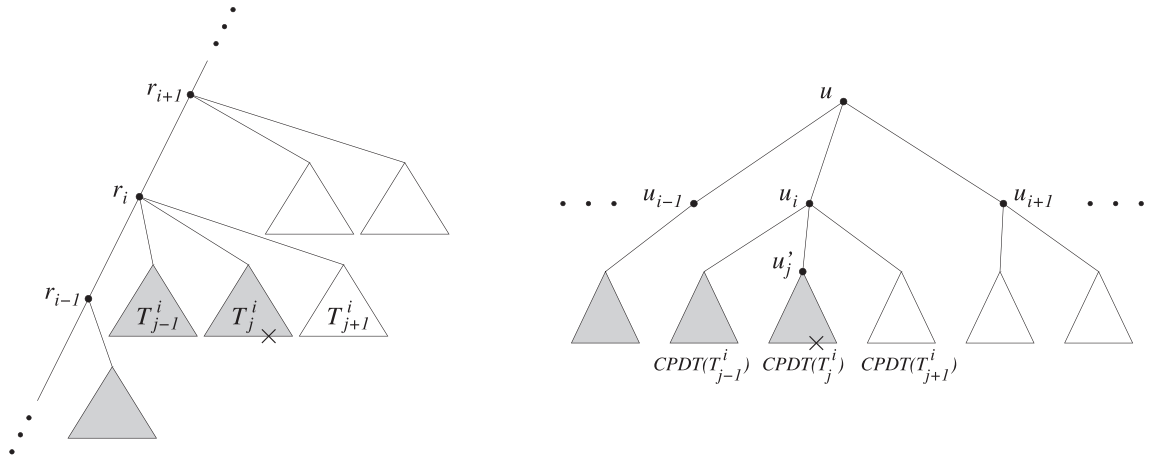


FIG. 6. The tree T_2 is shown on the left and its CPDT on the right. When the algorithm assigns a NON-RED color to the leaf marked by an "X," a number of good triplets and good fans whose lowest common ancestor in the CPDT is u will be created according to Lemmas 2 and 3. A leaf in a shaded subtree can have any color, whereas a leaf in a nonshaded subtree is either RED or WHITE.

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 in the CPDT is u, is*

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

Proof. Let l be the leaf that is being assigned the color C_a. A good triplet whose lca in $CPDT(T_2)$ is u is created if (i) there are two leaves x, y with the color C_b for some $b \notin \{0, a\}$ such that $xy|l \in r(T_2)$ and u is the lca of x, y, and l or (ii) there are two leaves x, y, where x has the color C_a and y has the color C_b with $b \notin \{0, a\}$, $xl|y \in r(T_2)$, and u is the lca of x, y, and l. As shown in Table 1, there are three possibilities for each of (i) and (ii), referred to as cases 1–3 and cases 4–6, respectively.

By basic combinatorics, the number of good triplets falling in case 1 is

$$\binom{Red(S_{<i})}{2}.$$

For case 2, the number is

$$\sum_{b=2,\, b\neq a}^{d} \binom{C_b(S_{<i})}{2}.$$

For case 3, the number is

$$\sum_{h>i} \sum_{S_t \in st(S_h)} \binom{Red(S_t)}{2} = \sum_{h>i} Red^{(2)}(S_h).$$

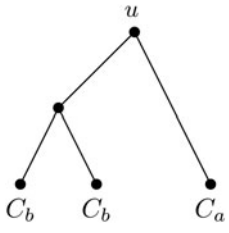
For case 4, the number is

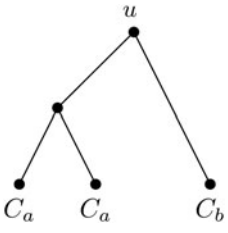
$$C_a(S_{\leq i}) \cdot Red(S_{>i}).$$

For case 5, the number is

$$C_a(S'_j) \cdot Red(S_{<i}).$$

Table 1. The Different Cases in the Proof of Lemma 2





Case 1 $R, R \in S_{<i}$	Case 4 $C_a \in S_{\leq i}$, $R \in S_{>i}$
$\binom{Red(S_{<i})}{2}$	$C_a(S_{\leq i}) \cdot Red(S_{>i})$
Case 2 $C_b, C_b \in S_{<i}$	Case 5 $C_a \in S_i$, $R \in S_{<i}$
$\binom{C_b(S_{<i})}{2}$	$C_a(S'_j) \cdot Red(S_{<i})$
Case 3 $R, R \in S_t$, $S_t \in st(S_h)$, $h > i$ $\sum_{h>i} \sum_{S_t \in st(S_h)} \binom{Red(S_t)}{2}$	Case 6 $C_a \in S_i$, $C_b \in S_{<i}$ $C_a(S'_j) \cdot C_b(S_{<i})$

Coloring a leaf in a subtree S'_j of a subtree S_i of a centroid path node u by the color C_a may introduce the mentioned types of good triplets, whose lowest common ancestor in $CPDT(T_2)$ is u.

The left and right columns correspond to conditions (i) and (ii) in the proof of Lemma 2. "$R \in S_i$" for some subtree S_i means that a leaf in S_i is currently RED (this corresponds to $C_b = C_1$).

Finally, for case 6, the number is

$$C_a(S'_j) \cdot \sum_{b=2,\, b\neq a}^{d} C_b(S_{<i}).$$

In case 6, C_a is only permitted to be in an S'_j because $C_a \in S'_{<j}$ would induce a fan of the form $C_a|C_a|C_b$ in T_2, which is not to be counted. ■

Lemma 3. *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 fans introduced by coloring a leaf by a color C_a in S'_j, $a \geq 2$, such that their lca in the CPDT is u, is*

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

Proof. Consider a triplet $\{x, y, z\}$, introduced by coloring a leaf in S'_j by C_a, inducing a good fan with u as lca. Without loss of generality, assume the leaf in S'_j is x. Then

(a) $y \in S_{<i}$ and either $z \in S'_{<j}$ or $z \in S'_{>j}$; furthermore, neither y nor z is colored by color C_a;
(b) y is colored by a color C_b and z by a color C_c, where $C_b, C_c \in \{C_1, C_2, \ldots, C_d\}$ such that $C_b \neq C_c$.

First, consider triplets such that $z \in S'_{<j}$. The number of triplets satisfying condition (a) is $\sum_{b=1,\, b\neq a}^{d} C_b(S_{<i}) \cdot \sum_{b=1,\, b\neq a}^{d} C_b(S'_{<j})$. We subtract the number of triplets satisfying condition (a) but not (b), that is $\sum_{b=1,\, b\neq a}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j})$, which gives the first part of the formula. Second, consider $z \in S'_{>j}$. By the assumptions, z can only be RED, and the number of such triplets is $\sum_{b=2,\, b\neq a}^{d} C_b(S_{<i}) \cdot Red(S'_{>j})$, which gives the second part of the formula. ■

Lemma 4. *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 in the CPDT is v, is*

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

Proof. The proof is similar to the proof of Lemma 2. We divide the triplets that are induced into two cases:

(a) one C_a-colored leaf (the one being colored) in S_i, and two C_b-colored leaves in S_j, $j \neq i$, where C_b is a color different from C_a;
(b) two C_a-colored leaves (one of which is being colored) in S_i, and another leaf, of a different color C_b, in S_j, $j \neq i$.

We also differentiate between $C_b = C_1$ (RED) and $C_b \notin \{C_1, C_a\}$, which gives two subcases for each of (a) and (b). Table 2 summarizes all the subcases.

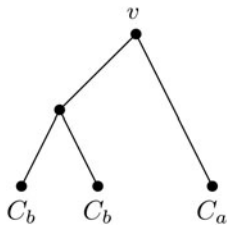
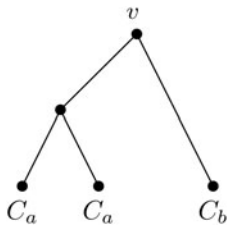
The number of triplets in case 1 of Table 2 is

$$\sum_{j=1}^{i-1} \binom{Red(S_j)}{2} + \sum_{j>i} \binom{Red(S_j)}{2}.$$

For case 2, the number is

$$\sum_{b=2,\, b\neq a}^{d} \sum_{j=1}^{i-1} \binom{C_b(S_j)}{2}.$$

TABLE 2. THE DIFFERENT CASES IN THE PROOF OF LEMMA 4

	
(1) $R, R \in S_j$, $j \neq i$ $\sum_{j=1}^{i-1} \binom{Red(S_j)}{2} + \sum_{j>i} \binom{Red(S_j)}{2}$	(3) $C_a \in S_i$, $R \in S_j$, $j \neq i$ $C_a(S_i) \cdot (Red(S_{<i}) + Red(S_{>i}))$
(2) $C_b, C_b \in S_j$, $j < i$ $\sum_{j=1}^{i-1} \binom{C_b(S_j)}{2}$	(4) $C_a \in S_i$, $C_b \in S_{<i}$ $C_a(S_i) \cdot C_b(S_{<i})$

When coloring a leaf in a subtree S_i of an SN-node v by C_a, the mentioned different types of good triplets, such that their lowest common ancestor is v, may be introduced.

For case 3, the number is

$$C_a(S_i) \cdot (Red(S_{<i}) + Red(S_{>i})).$$

Finally, for case 4, the number is

$$C_a(S_i) \cdot \sum_{b=2,\, b\neq a}^{d} C_b(S_{<i}).$$

■

Lemma 5. *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 fans introduced by coloring a leaf by a color C_a in S_i, $a \geq 2$, such that their lca in the CPDT is v, is*

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

Proof. Consider a triplet $\{x, y, z\}$, introduced by coloring a leaf in S_i by C_a, inducing a good fan with v as lca. Without loss of generality, assume that the leaf in S_i is x and that y is to the left of z in the CPDT. There are two possible situations:

(1) $y, z \in S_{<i}$.
(2) $y \in S_{<i}$, $z \in S_{>i}$.

($y, z \in S_{>i}$ is not possible as it would imply two RED leaves, which cannot induce a good fan.)
In case (1), the following additional conditions must be satisfied:

(a) neither y nor z is colored by the color C_a;
(b) y is colored by a color C_b and z by a color C_c, where $C_b, C_c \in \{C_1, C_2, \ldots, C_d\}$ such that $C_b \neq C_c$; and
(c) $y \in S_{h_1}$, $z \in S_{h_2}$, $h_1 \neq h_2$.

The number of triplets satisfying condition (a) is

$$\binom{\sum_{b=1,\, b\neq a}^{d} C_b(S_{<i})}{2} = \binom{C_{R\bar{a}}(S_{<i})}{2}.$$

We subtract the number of triplets satisfying (a) but not (b), that is,

$$\sum_{b=1,\, b\neq a}^{d} \binom{C_b(S_{<i})}{2},$$

and (a) but not (c), that is,

$$\sum_{h=1}^{i-1} \binom{\sum_{b=1,\, b\neq a} C_b(S_h)}{2} = \sum_{h=1}^{i-1} \binom{C_{R\bar{a}}(S_h)}{2}.$$

Finally, we add back the number of triplets satisfying (a) but neither (b) nor (c) because we removed it twice:

$$\sum_{h=1}^{i-1} \sum_{b=1,\, b\neq a}^{d} \binom{C_b(S_h)}{2}.$$

Case (2), is simple; by the assumption, z can only be RED and y can be any color except for RED or C_a, giving the term

$$\sum_{b=2,\, b\neq a}^{d} C_b(S_{<i}) \cdot Red(S_{>i}).$$ ■

Observe that whenever ColorTree calls ColorLeaf (l, C_a) with $a \geq 2$, all leaves belonging to $T_1(v_1)$ will be RED, whereas all leaves in $T_1(v_2), \ldots, T_1(v_{d_v})$ that come after l in the left-to-right ordering in $CPDT(T_2)$ will still be WHITE. This means that the conditions ''if there are no NON-RED leaves in $S_{>i}$ nor in $S'_{>j}$'' in Lemmas 2 and 3 and ''if there are no NON-RED leaves in $S_{>i}$'' in Lemmas 4 and 5 are always satisfied when the algorithm invokes the lemmas.

We now prove that for each given coloring according to a node v in $V(T_1)$, the algorithm indeed counts the number of unique good triplets and good fans, which then yields the algorithm's correctness.

Theorem 1. *Suppose T_1 and T_2 are colored according to $v \in V(T_1)$. Then every good triplet and every good fan is counted exactly once by the algorithm.*

Proof. For any leaf label l in T_2, let $\mathrm{Anc}_{T_2}(l)$ be the set of ancestors of $T_2(l)$. Also let $\mathrm{Anc}_{CPDT}(l) = \{a_0, a_1, a_2, \ldots, a_r\}$ be the ordered set of the ancestors of l in $CPDT(T_2)$; a_0 is the leaf l itself, whereas a_r is the root. Given an arbitrary node a_j in $\mathrm{Anc}_{CPDT}(l)$, $j \geq 1$, let $\mathrm{cp}_{T_2}(a_j)$ be the set of nodes on the centroid path in T_2 represented by a_j; when a_j is an SN-node, $\mathrm{cp}_{T_2}(a_j)$ is a single node.

Lemma 2, when applied to a CP-node v in the CPDT, counts the number of unique good triplets rooted at it, thus at any node in $\mathrm{cp}_{T_2}(v)$. Lemma 4 does the same to an SN-node. When we color l by a NON-RED color, the good triplets we are introducing must clearly have their lca in T_2 in $\mathrm{Anc}_{T_2}(l)$, so their lca in $CPDT(T_2)$ must be in $\mathrm{Anc}_{CPDT}(l)$; therefore, the algorithm counts every good triplet introduced by l.

Let us consider two arbitrary nodes $a_i, a_j \in \mathrm{Anc}_{CPDT}(l)$, $i > j$: $cp_{T_2}(a_i) \cap cp_{T_2}(a_j) \neq \emptyset$ iff a_i is a CP-node and $parent(a_j) = a_i$, in which case $cp_{T_2}(a_i) \cap cp_{T_2}(a_j) = cp_{T_2}(a_j)$. We only need to prove that we are not double-counting triplets rooted at node $v = cp_{T_2}(a_j)$. Let $\{v_1, v_2, \ldots, v_k\}$ be the set of children of v. Given a triplet, its lca in CPDT is a_j iff its lca in T_2 is v and it has no leaves in v_1, whereas its lca in CPDT is a_i iff either its lca in T_2 is v and it has at least one leaf in v_1 or its lca is an ancestor of v. Therefore, no triplet is counted twice. To sum up, when we color a leaf label l, we count every newly introduced good triplet exactly once.

Finally, the proof that no good triplet is counted when coloring two different leaves is trivial, because every triplet will only be counted when its rightmost NON-RED leaf is colored.

An analogous argument works for the good fans. ■

3.4. Time and space complexity analysis

In this subsection, we analyze the complexity of the new algorithm.

The values in Lemmas 2–5 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 7 and 8).

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 sum $\sum_{i=s}^{t} A[i]$.

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 classical result from the literature [see, e.g., (Fredman, 1982)]:

Lemma 6. *An RSQ supporting operations (1) and (2) in $O(\log n)$ time can be implemented in $O(n)$ space and $O(n)$ preprocessing time.*

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

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

Also store three RSQs, named $R_1(v)$, $R_2(v)$, and $R_3(v)$, such that

$$R_1(v).A[i] = Red(v_i), R_2(v).A[i] = \binom{Red(v_i)}{2}, \text{ and } R_3(v).A[i] = Red^{(2)}(v_i).$$

Lemma 7. *The values in Lemmas 2 and 3 can be found in $O(\log n)$ time.*

Proof. By the assumptions in Lemmas 2 and 3, no NON-RED leaf is in $S_{>i}$. Therefore, $C_c(u) = C_c(S_{\leq i})$. As $C_c(u_i) = C_c(S_i)$, it is easy 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 $S_{<i}$.

Next, when coloring leaves in S'_j, all leaves in $S_{<i}$ have already been colored. Thus, the values $C_c(S_{<i})$ for all $c \in \{2, \ldots, d\}$ are fixed. We keep track of the current value of $C_c(S_{<i}) \cdot C_c(S'_{<j})$ for all $c \in \{2, \ldots, d\}$ 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 already defined. When making range queries on them, we apply Lemma 6, which gives a time complexity of $O(\log n)$. ■

Lemma 8. *The values in Lemmas 4 and 5 can be found in $O(\log n)$ time.*

Proof. The only nontrivial quantity 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 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 7, the other quantities can be obtained in $O(\log n)$ time using Lemma 6 and the counters and RSQs already mentioned. ■

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 2–5 to the leaf and all its ancestors in the CPDT; Lemmas 7 and 8 provide these values for any specified node of the CPDT in $O(\log n)$ time.

To ensure that Lemmas 7 and 8 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 6 (Lemmas 7 and 8 are not applied here because no good triplets or good fans can be created when coloring a leaf RED or WHITE) and
- whenever a leaf is colored NON-RED, we traverse its leaf-to-root path in the CPDT and, after applying Lemmas 7 and 8 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 the following theorem.

Theorem 2. *The time complexity of the new algorithm is $O(n \log^3 n)$ and the space complexity is $O(n \log n)$.*

Proof. Constructing $CPDT(T_2)$ in the first step takes $O(n)$ time according to the definition of *CPDT*. By Brodal et al.'s (2013) analysis, 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 8, $O(\log n)$ time is used for each node visit in the CPDT. Thus, the time complexity of the algorithm is $O(n \log^3 n)$.

To analyze the space complexity, first consider how to represent the counters efficiently. (When d is large, e.g., if all leaves of T_1 are directly attached to the root, representing the counters naively leads to quadratic space.) At any time, a subtree of the CPDT needs to use at most as many colors as it has leaves. As the height of the CPDT is $O(\log n)$, every leaf in the CPDT is a descendant of $O(\log n)$ nodes and may, therefore, affect $O(\log n)$ different counters. Hence, the algorithm only needs to use $O(n \log n)$ counters, and each one takes one word of memory. Next, by Lemma 6, the space needed by the three RSQs for any node v in the CPDT is $O(d(v))$, where $d(v)$ is the degree of v in $CPDT(T_2)$. Summing over all nodes gives $\sum_{v \in V(CPDT(T_2))} d(v) = O(n)$. In total, the space complexity is $O(n \log n) + O(n) = O(n \log n)$ words. ■

4. IMPLEMENTATION

We have implemented the algorithm in Section 3 in two versions: a special binary trees-only optimized version and one for general trees. The importance of the special case in which both trees are binary justifies a dedicated implementation. The binary trees-only version is more efficient and much easier to implement because of the following:

- The number of colors is constant, which means that the representation of the counters defined in Section 3.4 becomes more compact.
- The formulas in Lemmas 2–5 become simpler.
- All SN-nodes will now always have a single child, so we can omit them; consequently, the CPDT only needs one type of node (CP-node), which makes the data structure smaller and eliminates the need for SN-node operations.

Only plain standard C++ was used, except for an (optional) single feature from C++11, mentioned hereunder. The source code can be downloaded from

(http://sunflower.kuicr.kyoto-u.ac.jp/∼jj/Software/CPDT-dist.html).

A few optimizations that improve the running time in practice are discussed next.

4.1. Representation of the counters and RSQs

We implemented the counters as follows. Let d be the number of children of the highest degree node in T_1. In any given node v in the CPDT, for each set of counters, we allocate an array of length $\min\{d, leaves(v)\}$, where $leaves(v)$ 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 $k \in \{1, \ldots, d\}$ 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 $\{1, \ldots, d\}$ 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 implementations of the hashmap: the C++11 unordered_map (unordered_map) 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 (sparsehash). We can choose which one to use at compile time. Some experimental results for both libraries are reported in Section 5.

To implement the RSQs in Section 3.4, we used a data structure by Fenwick (1994), which has a good practical performance while achieving the time and space complexity bounds stated in Lemma 6.

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 the nodes in the CPDT that we need to consider, that is, all nodes that are ancestors of at least one leaf being colored; for each leaf, we start at it and go up until we reach either the root or an already marked node, marking all nodes along the way. Second, we traverse the marked subtree in postorder and when we visit a given node, we already know how many leaves we are coloring for each color; modifying the formulas in Lemmas 2–5 to count all the good triplets introduced by such leaves, for each color, in one go is straightforward.

We illustrate this using Lemma 2. Suppose we are coloring k leaves in S'_j at once, and currently $C_a(S'_j) = 0$. Good triplets falling into cases 1–3 will be introduced once for each leaf colored C_a; hence the first part of the equation becomes

$$k\left(\binom{(S_{<i})}{2} + \sum_{b=2,\, b\neq a}^{d} \binom{C_b(S_{<i})}{2} + \sum_{h>i} Red^{(2)}(S_h)\right).$$

Next, by coloring all k leaves at once (instead of one by one), $\binom{k}{2} Red(S_{<i})$ triplets are introduced for case 5, and $\binom{k}{2} C_b(S_{<i})$ for case 6, because each possible pair among the k leaves may form a good triplet. Case 4 is a bit more complex, as it has two subcases: either both C_a-colored leaves are in S'_j, or only one is and the second one is in $C_a(S_{<i})$ or $C_a(S'_{<j})$. Therefore, the second part of the equation becomes

$$\left(k\left(C_a(S_{<i}) + C_a(S'_{<j})\right) + \binom{k}{2}\right) Red(S_{>i}) + \binom{k}{2}(Red(S_{<i}) + C_b(S_{<i})).$$

Newly introduced good fans and good triplets of the form $C_bC_b|C_a$ can be naively counted k times. In contrast, good triplets of the form $C_aC_a|C_b$ fall into two cases: either (a) only one C_a-colored leaf is recolored or (b) both C_a-colored leaves are recolored, and the formulas are adapted accordingly.

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, that is, 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. As coloring a leaf RED and WHITE requires updating a number of RSQs and is a fairly expensive operation, this saves us a lot of time.

Next, *cherry nodes* (degree-2 nodes whose two children are leaves) are not colored because they cannot yield any good triplets. By not considering them, we eliminate some RED and WHITE recolorings.

Finally, when we color a single leaf (or even a sufficiently low number of leaves), the two-step coloring can actually be a burden. In such a case, we skip the marking step and apply Lemmas 2–5 directly on every node along the leaf-to-root path.

5. EXPERIMENTS

We compared the running time and memory usage in practice of the new algorithm with that of tqDist (Sand et al., 2014) 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 12 GB of RAM. The system C++ compiler was g++, version 4.6.3.

We used the C++ implementations of our algorithm presented in Section 4: one for general trees and a special binary trees-only optimized version. As mentioned in Section 4, the algorithm can be compiled using two different hashmaps: C++11 unordered_map (unordered_map) (we named this CPDT) and sparsehash (sparsehash) (named CPDTg). We 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 (Sand et al., 2014) was built from its source code using cmake, as instructed by the authors. We had to disable the HDT dynamic contraction (Holt et al., 2014) of tqDist as it was making the tool run tens of times slower; built with the default parameters, it would usually take more than 1 hour for two random 1 million leaves trees. We also modified it to use unsigned 64 bits instead of signed bits so that it would correctly compute the rooted triplet distance for very large values of n.

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 using Valgrind (Nethercote and Seward, 2007) and its heap profiling tool Massif. Owing 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 4,000,000. Arbitrary-degree input trees were generated as follows.

First, generate a binary tree with n leaves in the *uniform model* (McKenzie and Steel, 2000; Semple and Steel, 2003; Blum et al., 2006), that is, the probabilistic model of a phylogenetic tree in which all binary trees on n leaves are equally likely. Next, for each nonroot, 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.

The mathematical properties of the uniform model have previously been well studied (see, e.g., (McKenzie and Steel, 2000; Semple and Steel, 2003; Blum et al., 2006) and the references therein). For example, it is known that the expected average depth of a leaf in a random binary tree with n leaves generated in the uniform model is $\Theta(n^{1/2})$ (Blum et al., 2006). The uniform model is also called the *PDA model* in the literature (Semple and Steel, 2003).

Hereunder, 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. In addition, we created a set in which both trees are binary by setting $p_1 = p_2 = 0$,

and two sets in which p_1 is 0.95 (respectively, 0.2) and p_2 is 0.2 (respectively, 0.95) to test the algorithms when dealing with *extremely branching* trees, that is, flat trees with very high degree. Also, the benchmarks were executed on pairs of *unrelated* as well as *related* trees, where two unrelated trees were generated independently of each other and two related trees were generated by starting from the same binary tree. Finally, to test the algorithms on trees of height $\Theta(n)$, we applied them to pairs of randomly generated *caterpillars* (binary trees in which every internal node has at least one child that is a leaf), constructed by taking random permutations of the set $\{1, 2, \ldots, n\}$.

5.3. Results

Table 3 reports the average running times of the three implementations tested on pairs of trees with 4 million leaves, along with the relative speedups over tqDist. Figure 7 shows the plots of the average running times as a function of n for binary trees as well as for nonbinary trees obtained using some representative (p_1, p_2) values.

The binary case ($p_1 = p_2 = 0$)) benefits greatly from having a special implementation, being about 40% faster than the general implementation for arbitrary-degree trees and showing a more than sixfold improvement over tqDist when $n = 4,000,000$. As it does not rely on hashmaps, we have a single implementation of the CPDT.

For unrelated arbitrary-degree trees with $n = 4,000,000$, CPDTg is clearly the fastest implementation, consistently being at least three times faster than tqDist and showing noticeable improvements over CPDT. All three implementations seem to prefer instances in which either p_1 or p_2 is large. While the latter is obvious, as a large value of p_2 implies that the number of internal nodes in T_2 will be small, the former is interesting as it indicates that they are able to handle a huge number of colors more easily. (We remark here that additional experiments have shown that for smaller n such as $n = 10,000$, CPDTg is only 2.17 times faster than tqDist on average, and for $n = 100,000$, CPDTg is 2.65 times faster than tqDist.)

For related trees, all three implementations become much faster. CPDTg still has an obvious advantage, although speedups here are around 2.0–2.5×. This can be at least partially explained by overheads, such as tree parsing from files, becoming more significant as the running times of the actual algorithms decrease. The notion of ''related trees'' is not applicable when $p_1 = p_2 = 0$ because without contractions, the generated T_1 and T_2 will always be equal to each other.

The average memory usage of the three implementations for pairs of unrelated trees is reported in Table 4 and Figure 8. CPDT is the least memory hungry, showing an improvement over tqDist of 3.78× on binary trees and up to 2.35× on arbitrary-degree trees, with CPDTg being a close second. They all benefit from increasing p_2 (meaning fewer internal nodes in T_2, so that the CPDT needs less memory). In contrast, only CPDT and CPDTg seem to suffer from increasing p_1 (meaning more colors and hence more counters), and in the extremely branching case, the advantages of CPDT and CPDTg decrease to 1.33× and 1.24×, respectively, over tqDist.

For pairs of caterpillars, the average running times and average memory usage are plotted in Figure 9. Both tqDist and CPDT are much faster in this case than for randomly generated binary trees, that is, the case shown in Figure 7a, but CPDT is still faster than tqDist. The memory usage is about the same as in Figure 8a. This shows that even when the input trees have large heights, CPDT performs well.

Table 3. Average Running Times (in Seconds) on Two 4 Million Leaves Trees and Relative Speedups Over tqDist

		Unrelated trees						*Related trees*					
p_1	p_2	*tqDist*		*CPDT*		*CPDTg*		*tqDist*		*CPDT*		*CPDTg*	
0.0	0.0	282.49	1.00×	42.92	6.58×	—	—	—	—	—	—	—	—
0.2	0.2	293.10	1.00×	83.98	3.49×	78.04	3.76×	55.05	1.00×	24.20	2.27×	22.61	2.43×
0.2	0.95	230.42	1.00×	63.16	3.65×	58.38	3.95×	42.45	1.00×	18.01	2.36×	16.40	2.59×
0.5	0.5	281.72	1.00×	91.76	3.07×	85.51	3.29×	49.18	1.00×	24.10	2.04×	23.62	2.08×
0.8	0.8	237.14	1.00×	77.07	3.08×	72.43	3.27×	40.65	1.00×	19.02	2.14×	17.71	2.30×
0.95	0.2	213.50	1.00×	68.14	3.13×	63.97	3.34×	47.20	1.00×	21.76	2.17×	18.98	2.49×

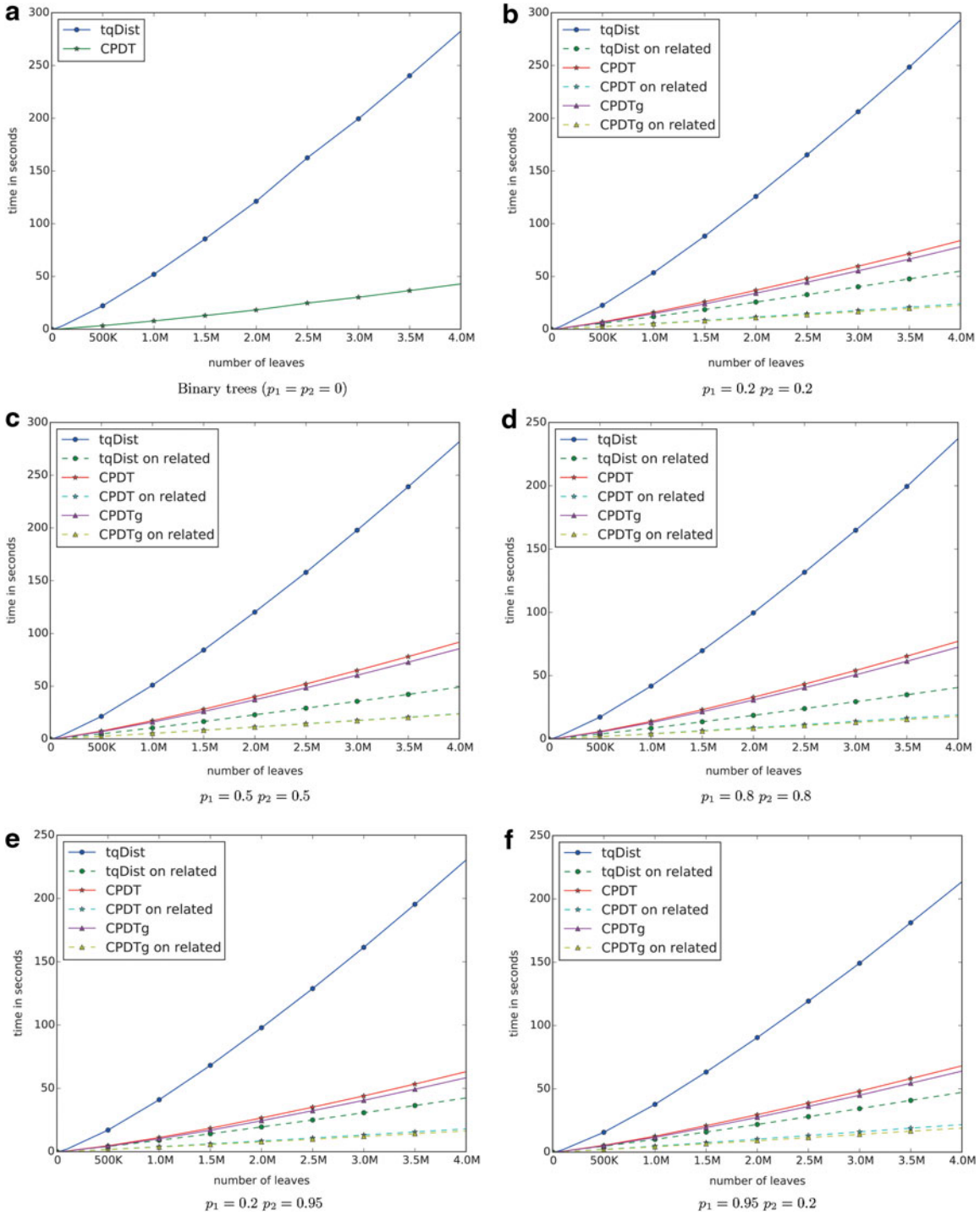


FIG. 7. Plots of the average running time in seconds (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2). Solid lines represent values on unrelated trees, and dashed lines represent values on related trees.

TABLE 4. AVERAGE MEMORY USAGE (IN GB) ON TWO 4 MILLION LEAVES TREES AND THE RELATIVE MEMORY USAGE DECREASE OVER TQDIST

p_1	p_2	*tqDist*		*CPDT*		*CPDTg*	
0.0	0.0	9.97	1.00×	2.64	3.78×	—	—
0.2	0.2	9.10	1.00×	4.10	2.22×	4.54	2.00×
0.2	0.95	6.27	1.00×	2.67	2.35×	2.73	2.30×
0.5	0.5	7.76	1.00×	4.01	1.94×	4.40	1.76×
0.8	0.8	6.38	1.00×	3.51	1.82×	3.72	1.72×
0.95	0.2	8.53	1.00×	6.42	1.33×	6.86	1.24×

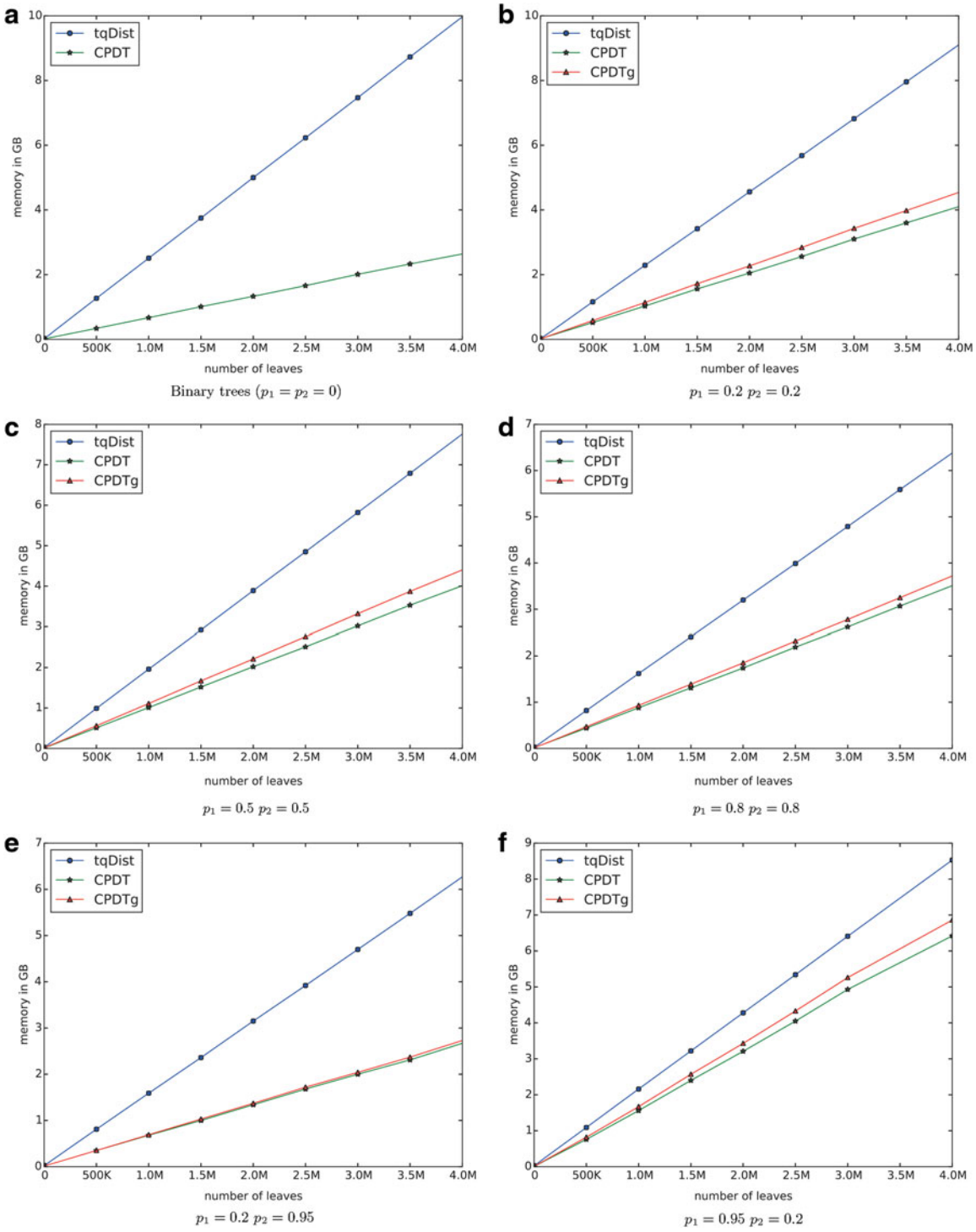


FIG. 8. Plots of the memory usage in gigabytes (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2).

Finally, we experimentally confirmed that the time complexity of our current implementation is $O(n \log^3 n)$ by dividing the running times by the value of $n \ln^3 n$ for increasing n (Table 5). In the table, $a(n)$ is the average running time for randomly generated inputs with n leaves divided by $n \ln^3 n$ and multiplied by a large constant to make the results easily readable. According to the table, $a(n)$ seems to converge to a constant as $n \to \infty$.

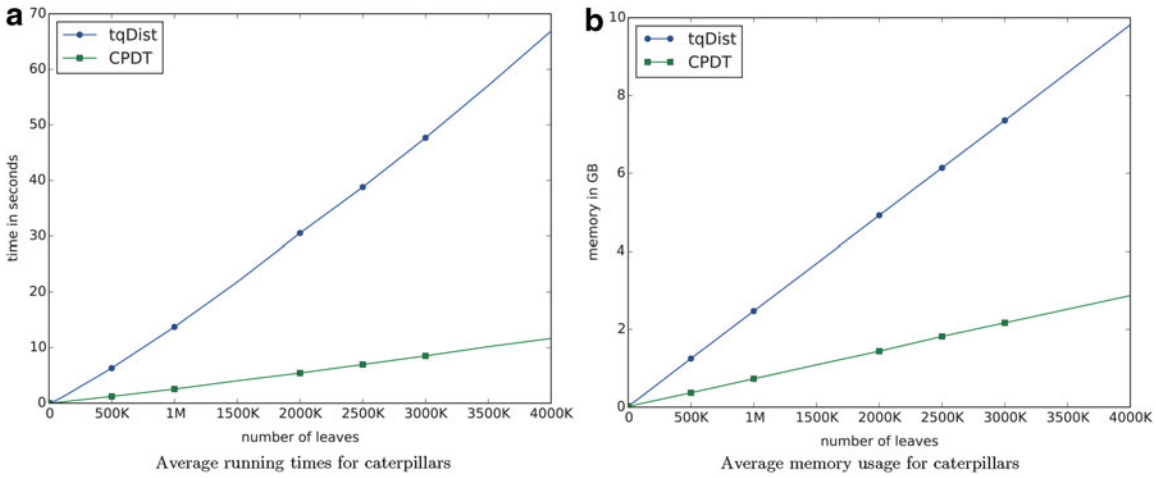


FIG. 9. Plots for caterpillar trees, corresponding to those in Figure 7a and Figure 8a.

6. CONCLUDING REMARKS

Bansal et al.'s (2011) *parametric rooted triplet distance* is a generalization of the rooted triplet distance that allows a conflict between a fan triplet in one input tree and a resolved triplet in the other input tree to be penalized less severely than a conflict between two resolved triplets. It is defined for any real number p with $0 \leq p \leq 1$ as $d_{rt}^{(p)}(T_1, T_2) = |\mathcal{D}(T_1, T_2)| + p \cdot (|\mathcal{R}(T_1, T_2)| + |\mathcal{R}(T_2, T_1)|)$, where $|\mathcal{D}(T_1, T_2)|$ is the number of cardinality-3 subsets of the leaf label set that induce two different resolved triplets in T_1 and T_2 and $|\mathcal{R}(T_i, T_j)|$ is the number of cardinality-3 subsets of the leaf label set that induce a resolved triplet in T_j and a fan triplet in T_j. (Selecting $p = 1$ yields the original rooted triplet distance.) The algorithms in Bansal et al. (2011) and Brodal et al. (2013) can be modified to compute the parametric rooted triplet distance without increasing the asymptotic time complexity, and we now show that the algorithm in Section 3 is easily extendable in a similar way. Let $f(T_i, T_j)$ for any $i, j \in \{1, 2\}$ be the total number of good fans found by the algorithm in Section 3 when computing $d_{rt}(T_i, T_j)$. Then $f(T_i, T_i)$ equals the number of fan triplets in $rt(T_i)$, so $|\mathcal{R}(T_1, T_2)| = f(T_2, T_2) - f(T_1, T_2)$, and analogously, $|\mathcal{R}(T_2, T_1)| = f(T_1, T_1) - f(T_1, T_2)$. Thus, the term $p \cdot (|\mathcal{R}(T_1, T_2)| + |\mathcal{R}(T_2, T_1)|)$ in the formula for $d_{rt}^{(p)}(T_1, T_2)$ can be computed without increasing the asymptotic time complexity. (We remark that in practice, it may be faster to use some special-purpose $O(n)$-time method to compute $f(T_1, T_1)$ and $f(T_2, T_2)$, such as those presented in Bansal et al. (2011) and Brodal et al. (2013).) The other term, $|\mathcal{D}(T_1, T_2)|$, is obtained at no extra cost by taking $\binom{n}{3} - (|\mathcal{S}(T_1, T_2)| + |\mathcal{R}(T_1, T_2)| + |\mathcal{R}(T_2, T_1)| + f(T_1, T_2))$, where $|\mathcal{S}(T_1, T_2)|$ is the total number of good triplets found by the algorithm in Section 3 when computing $d_{rt}(T_1, T_2)$.

A question for future research is as follows: Can the theoretical or practical running times of the CPDT-based algorithm be reduced? As noted at the end of Section 5, dividing the running times of our current

TABLE 5. EXPERIMENTAL VALIDATION OF THE ASYMTOMATIC RUNNING TIME

n	$a(n)$
1,000,000	0.991215
1,500,000	0.999724
2,000,000	0.995861
2,500,000	1.03432
3,000,000	1.01342
3,500,000	1.01844
4,000,000	1.01718

$a(n)$ seems to converge to a constant as $n \to \infty$, which suggests that the time complexity is indeed $O(n \log^3 n)$.

implementation by $n \ln^3 n$ seems to converge to a constant as $n \to \infty$, which confirms the theoretical analysis in Section 3.4. However, the CPDT respects the definition of *locally balanced* in Brodal et al. (2013), so perhaps the algorithm can be refined in a way such that the analysis technique in Section 5 of Brodal et al. (2013) can be applied? To achieve an improved bound of $O(n \log^2 n)$ on the time complexity, one needs to prove that when the procedure ColorTree makes a recursive call to any $v \in V(T_1)$ and recolors the leaves in each $T_1(v_i)$, the algorithm can charge an average of $O((1 + \log \frac{|\Lambda(T_1(v))|}{|\Lambda(T_1(v_i))|}) \cdot \log n)$ work to every leaf in $T_1(v_i)$. For this purpose, it may be necessary to modify ColorTree so that any recursive call to $v_i \in V(T_1)$ first compresses the CPDT to $O(|\Lambda(T_1(v_i))|)$ size, that is, by restricting $CPDT(T_2)$ to the leaves in $\Lambda(T_1(v_i))$, without spending too much additional time to update all the auxiliary information.

To make the algorithm faster in practice, one might try to parallelize it. Unfortunately, this is difficult because of 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 (Brodal et al., 2013). It would be interesting to see whether the CPDT can be adapted to get an efficient algorithm for this variant.

A fundamental 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 those used here because Brodal et al.'s recursive recoloring scheme already introduces $\Omega(n \log n)$ work.

ACKNOWLEDGMENTS

J.J. was funded by The Hakubi Project at Kyoto University and KAKENHI grant number 26330014. R.R. was funded by National University of Singapore and the EXTRA Project at the University of Milano-Bicocca. Computational power for the experiments in Section 5 was provided by the MIUR PRIN 2010–2011 grant ''Automi e Linguaggi Formali: Aspetti Matematici e Applicativi,'' code 2010LYA9RH. The authors thank Professor Tatsuya Akutsu and Professor Gianluca Della Vedova for their support.

AUTHOR DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Bansal, M.S., Dong, J., and Fernández-Baca, D. 2011. Comparing and aggregating partially resolved trees. *Theor. Comput. Sci.* 412, 6634–6652.

Blum, M.G.B., François, O., and Janson, S., 2006. The mean, variance and limiting distribution of two statistics sensitive to phylogenetic tree balance. *Ann. Appl. Probab.* 16, 2195–2214.

Brodal, G.S., Fagerberg, R., Mailund, T., et al. 2013. 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), 1814–1832. SIAM. New Orleans, LA, USA.

Cole, R., Farach-Colton, M., Hariharan, R., et al. 2000. An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. *SIAM J. Comput.* 30, 1385–1404.

Critchlow, D.E., Pearl, D.K., and Qian, C. 1996. The triples distance for rooted bifurcating phylogenetic trees. *Syst. Biol.* 45, 323–334.

Dobson, A.J. 1975. Comparing the shapes of trees. In *Proceedings of Combinatorial Mathematics III*, *Lecture Notes in Mathematics*, volume 452, 95–100. Springer-Verlag. Queensland, Australia.

Estabrook, G.F., McMorris, F.R., and Meacham, C.A. 1985. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Syst. Zool.* 34, 193–200.

Felsenstein, J. 2004. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, MA.

Fenwick, P.M. 1994. A new data structure for cumulative frequency tables. *Softw. Pract. Exp.* 24, 327–336.

Fredman, M.L. 1982. The complexity of maintaining an array and computing its partial sums. *J. ACM* 29, 250–260.

Gambette, P., and Huber, K.T. 2012. On encodings of phylogenetic networks of bounded level. *J. Math. Biol.* 65, 157–180.

Griebel, T., Brinkmeyer, M., and Böcker, S. 2008. EPoS: A modular software framework for phylogenetic analysis. *Bioinformatics* 24, 2399–2400.

Gusfield, D., Eddhu, S., and Langley, C. 2004. Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *J. Bioinform. Comput. Biol.* 2, 173–213.

Holt, M.K., Johansen, J., and Brodal, G.S. 2014. On the scalability of computing triplet and quartet distances. In *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments* (ALENEX 2014), 9–19. SIAM. Portland, OR, USA.

Jansson, J., and Lingas, A. 2014. Computing the rooted triplet distance between galled trees by counting triangles. *J. Discrete Algorithms* 25, 66–78.

McKenzie, A., and Steel, M. 2000. Distributions of cherries for two models of trees. *Math. Biosci.* 164, 81–92.

Nakhleh, L., Warnow, T., Ringe, D., et al. 2005. A comparison of phylogenetic reconstruction methods on an Indo-European dataset. *Trans. Philol. Soc.* 103, 171–192.

Nethercote, N., and Seward, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (PLDI 2007), 89–100. ACM. San Diego, CA, USA.

Sand, A., Holt, M.K., Johansen, J., et al. 2014. tqDist: A library for computing the quartet and triplet distances between binary or general trees. *Bioinformatics* 30, 2079–2080.

Semple, C., and Steel, M. 2003. *Phylogenetics*, *Oxford Lecture Series in Mathematics and its Applications*, volume 24. Oxford University Press. Oxford, UK.

Sung, W.-K. 2010. *Algorithms in Bioinformatics: A Practical Introduction*. Chapman & Hall/CRC. Boca Raton, FL, USA.

sparsehash, sparsehash project webpage. https://code.google.com/p/sparsehash/ Last viewed: October 14, 2016.

unordered_map, Documentation for unordered_map. www.cplusplus.com/reference/unordered_map/unordered_map/ Last viewed: October 14, 2016.

Address correspondence to:
Dr. Jesper Jansson
Laboratory of Mathematical Bioinformatics
Institute for Chemical Research
Kyoto University
Gokasho
Uji
Kyoto 611-0011
Japan

E-mail: jj@kuicr.kyoto-u.ac.jp