Faster Exact Algorithms for Hybridization Number and rSPR Distance

Zhi-Zhong Chen* Lusheng Wang†

Abstract

Due to hybridization events in evolution, studying two different genes of a set of species may yield two related but different phylogenetic trees for the set of species. In this case, we want to measure the dissimilarity of the two trees. The rooted subtree prune and regraft (rSPR) distance and the hybridization number of the two trees have been used for this purpose, and many algorithms and software tools have been developed for computing the rSPR distance and the hybridization number of two given phylogenetic trees. The previously fastest exact algorithms for rSPR distance and hybridization number run in time $O\left(2.415^d n\right)$ and $O\left(3^d n + 3^d (h-1)^{h-d+2}\right)$ respectively, where d is the rSPR distance between the two given trees T_1 and T_2, n is the number of leaves in T_1 (or equivalently, in T_2), and h is the hybridization number of T_1 and T_2. In this paper, we present faster exact algorithms for rSPR distance and hybridization number which run in time $O\left(2.344^d n\right)$ and $O\left(2.415^d n + 2.415^d (h-1)^{h-d+2}\right)$, respectively.

Keywords: Phylogenetic tree, rSPR distance, hybridization number, fixed-parameter algorithm.

1 Introduction

When studying the evolutionary history of a set of existing species, one can obtain a phylogenetic tree of the set of species with high confidence by looking at a segment of sequences or a set of genes. When looking at another segment of sequences, a different phylogenetic tree can be obtained with high confidence, too. In this case, we want to measure the dissimilarity of the two trees. The rooted subtree prune and regraft (rSPR) distance and the hybridization number of the two trees have been used for this purpose [12].

Unfortunately, it is NP-hard to compute the rSPR distance of two given phylogenetic trees [4, 12], and so is to compute the hybridization number of two given phylogenetic trees [4, 5, 12]. So, it is challenging to design algorithms that can compute the rSPR distance or the hybridization number of two given trees of large rSPR distance or hybridization number. Indeed, this has motivated researchers to design approximation algorithms [2, 3, 12, 15] or exact algorithms [4, 19, 18, 17], as well as heuristic algorithms [1, 11, 13, 14, 16], for computing the rSPR distance of two given trees. Similarly, several software packages have been developed for computing the hybridization number of two given trees [10, 7, 8, 9, 19, 21]. The previously fastest exact algorithm for rSPR distance is due to Whidden *et al.* [17] and runs in $O\left(2.415^d n\right)$ time, where n and d are the number of leaves

*Corresponding author. Division of Information System Design, Tokyo Denki University, Ishizaka, Hatoyama, Hiki, Saitama, 359-0394, Japan. Email: zzchen@mail.dendai.ac.jp. Phone: +81-49-296-5249. Fax: +81-49-296-7072.

†Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong SAR. Email: lwang@cs.cityu.edu.hk. Phone: +852-2788-9820. Fax: +852-2788-8614.

and the rSPR distance of the input trees, respectively. On the other hand, the previously fastest exact algorithm for hybridization number is given in [8] and runs in $O\left(3^d n + 3^d(h-1)^{h-d+2}\right)$ time, where h is the hybridization number of the input trees.

In this paper, we present faster exact algorithms for rSPR distance and hybridization number which run in time $O\left(2.344^d n\right)$ and $O\left(2.415^d n + 2.415^d(h-1)^{h-d+2}\right)$, respectively. Our algorithm for rSPR distance builds on Whidden *et al.*'s but requires a number of new ideas. The main idea is to repeatedly find a pair of sibling leaves in one of the input trees more carefully. On the other hand, our algorithm for hybridization number builds on the algorithm in [8] but heavily relies on a new lemma which enables us to avoid the worst case of the algorithm in [8].

The remainder of this paper is organized as follows. In Section 2, we give the basic definitions that will be used throughout the paper. In Section 3, we present and analyze our algorithm for hybridization number. In Section 4, we sketch Whidden *et al.*'s algorithm for rSPR distance, because our new algorithm for rSPR distance will build on theirs. In Section 5, we outline our algorithm for rSPR distance. In Section 6, we detail a main step in our algorithm for rSPR distance. In Section 7, we analyze the time complexity of our algorithm for rSPR distance.

2 Preliminaries

Throughout this paper, a *rooted forest* always means a directed acyclic graph in which every node has in-degree at most 1 and out-degree at most 2.

Let F be a rooted forest. The *roots* (respectively, *leaves*) of F are those nodes whose in-degrees (respectively, out-degrees) are 0. The *size* of F, denoted by $|F|$, is the number of roots in F minus 1. A node v of F is *unifurcate* if it has only one child in F. If a root v of F is unifurcate, then *contracting v in F* is the operation that modifies F by deleting v. If a non-root node v of F is unifurcate, then *contracting v in F* is the operation that modifies F by first adding an edge from the parent of v to the child of v and then deleting v.

For convenience, we view each node u of F as an ancestor and descendant of u itself. For a node v of F, the *subtree F_v of F rooted at v* is the subgraph of F whose nodes are the descendants of v in F and whose edges are those edges connecting two descendants of v in F. If v is a non-root node of F, then F_v is called a *pendant subtree* of F. On the other hand, if v is a root of F, then F_v is a *component tree* of F. F is a *rooted tree* if it has only one root. If u and v are two leaves in the same component tree of F, then the *pendant subtrees of F between u and v* are the pendant subtrees whose roots w satisfy that the (undirected) path between u and v in F does not contain w but contains the parent of w. If U is a set of leaves in a component tree of F, then $\mathrm{LCA}_F U$ denotes the lowest common ancestor (LCA) of the leaves in U.

A *rooted binary forest* is a rooted forest in which the out-degree of every non-leaf node is 2. Let F be a rooted binary forest. F is a *rooted binary tree* if it has only one root. If v is a non-root node of F with parent p and sibling u, then *detaching the pendant subtree with root v* is the operation that modifies F by first deleting the edge (p, v) and then contracting p. A *detaching operation* on F is the operation of detaching a pendant subtree of F. If v is a root leaf of F, then *eliminating v* from F is the operation that modifies F by simply deleting v. On the other hand, if v is a non-root leaf of F, then *eliminating v* from F is the operation that modifies F by first detaching the subtree rooted at v and then deleting v.

A *phylogenetic tree* on a set X of species is a rooted binary tree whose leaf set is X. Let T_1 and T_2 be two phylogenetic trees on the same set X of species. If we can apply a sequence of detaching operations on each of T_1 and T_2 so that they become the same forest F, then we refer to F as an *agreement forest* (AF) of T_1 and T_2. A *maximum agreement forest* (MAF) of T_1 and T_2 is an

agreement forest of T_1 and T_2 whose size is minimized over all agreement forests of T_1 and T_2. The size of an MAF of T_1 and T_2 is called the *rSPR distance* between T_1 and T_2.

Let F be an agreement forest of T_1 and T_2. Obviously, for each $i \in \{1, 2\}$, the leaves of T_i one-to-one correspond to the leaves of F. For convenience, we hereafter identify each leaf v of F with the leaf of T_i corresponding to v. Similarly, for each $i \in \{1, 2\}$, the non-leaf nodes of F correspond to distinct non-leaf nodes of T_i. More precisely, a non-leaf node u of F corresponds to $\mathrm{LCA}_{T_i}\{v_1, \ldots, v_\ell\}$, where v_1, ..., v_ℓ are the leaf descendants of u in F. Again for convenience, we hereafter identify each non-leaf node u of F with the non-leaf node of T_i corresponding to u. With these correspondences, we can use F, T_1, and T_2 to construct a directed graph G_F as follows:

- The nodes of G_F are the roots of F.
- For every two roots r_1 and r_2 of F, there is an edge from r_1 to r_2 in G_F if and only if r_1 is an ancestor of r_2 in T_1 or T_2.

We refer to G_F as the *decision graph associated with* F. If G_F is acyclic, then F is an *acyclic agreement forest* (AAF) of T_1 and T_2. If F is an AAF of T_1 and T_2 and its size is minimized over all AAFs of T_1 and T_2, then F is a *maximum acyclic agreement forest* (MAAF) of T_1 and T_2. Note that our definition of an AAF is the same as those in [6, 20] but is different from that in [17] [1]. Moreover, the *hybridization number* of T_1 and T_2 is the size of an MAAF of T_1 and T_2.

The following lemma has been shown in [8]:

Lemma 2.1 *Suppose that C is a cycle of G_F and r_1, ..., r_ℓ are the nodes of C. Then, each $r_j \in \{r_1, \ldots, r_\ell\}$ has two children u_j and u'_j in F. Moreover, for every non-root node v of F not contained in $\{u_1, u'_1, \ldots, u_\ell, u'_\ell\}$, C remains a cycle in G_F after F is modified by detaching the subtree of F rooted at v.*

3 New Algorithm for Hybridization Number

We first sketch the algorithm in [8] for computing the hybridization number of two given phylogenetic trees, and then describe how to speed it up. The algorithm in [8] indeed solves the following problem (denoted by HNC, for convenience):

Input: (k, T'_1, F_2), where k is an integer, T'_1 is a rooted tree obtained from some phylogenetic tree T_1 on a set X of species by eliminating zero or more leaves, and F_2 is a rooted forest obtained from some phylogenetic tree T_2 on X by performing zero or more detaching operations.

Output: "Yes" if performing k more detaching operations on F_2 leads to an AAF of T_1 and T_2; "no" otherwise.

Obviously, to compute the hybridization number of two given phylogenetic trees T_1 and T_2, it suffices to solve HNC on input (k, T_1, T_2) for $k = 0, 1, 2, \ldots$ (in this order), until a "yes" is outputted. To invoke the algorithm in [8] for HNC on input (k, T_1, T_2), we need to initially associate a *label set* $\{x\}$ to each leaf x of T_1. During its execution, the algorithm will then merge the label sets of two sibling leaves in T'_1 when necessary, and will always maintain the following invariant:

Invariant 1: The label sets of every pair of leaves in T'_1 are disjoint and the label set of each leaf in T'_1 is also the leaf set of some pendant subtree in F_2. (*Comment:* For convenience, for each leaf u in T'_1, we use $F_2(u)$ to denote the root of the pendant subtree in F_2 whose leaf set is the same as the label set of u.)

[1] It is known in the literature that the definition of AAF in [17] is incorrect.

The algorithm in [8] for HNC is recursive and proceeds as follows. In the base case, k is negative and we output "no" and return. So, suppose that $k \geq 0$. Then, whenever T_1' has two sibling leaves u and v such that $F_2(u)$ and $F_2(v)$ are also siblings in F_2, we modify T_1' by merging u and v into a single leaf (say, u) whose label set is the union of the label sets of the two merged leaves. Moreover, whenever T_1' has a non-root leaf u such that $F_2(u)$ is a root of F_2, we modify T_1' by eliminating u. We repeat these two types of modifications of T_1' until none of them is possible. After that, one of the following two cases occurs.

Case 1: T_1' is a single node. In this case, F_2 is an AF of T_1 and T_2. To test if F_2 is an AAF, we construct the decision graph G_{F_2} associated with F_2 and test if it is acyclic or not. If G_{F_2} is acyclic, then we can output "yes" and stop. Otherwise, we check if $k \geq 1$ or not. If $k \leq 0$, then we can output "no" and return. On the other hand, if $k \geq 1$, then we find a cycle C in G_{F_2} (say, by a depth-first search in G_{F_2}). By Lemma 2.1, in order to make F_2 acyclic, we have to select one node r of C and modify F_2 by detaching the subtree of F_2 rooted at an (arbitrary) child of r. Note that since r is a root of F_2, detaching the subtree of F_2 rooted at a child of r is achieved by simply deleting r from F_2 and is hence independent of the choice of the child. So, we have at most $|C| \leq k$ ways to break C, where $|C|$ denotes the number of nodes in C. After modifying F_2 in this way, we again test if F_2 is an AAF, as before. We repeat this process until we can output a "yes" or "no".

Case 2: T_1' is not a single node. In this case, T_1' has at least two leaves and hence contains at least one pair of sibling leaves. So, we select two *arbitrary* sibling leaves u and v in T_1' and use them to distinguish two cases as follows.

Case 2.1: $F_2(u)$ and $F_2(v)$ are in different component trees of F_2. In this case, in order to transform F_2 into an AF of T_1 and T_2, it suffices to try two choices to modify F_2, namely, by either detaching the subtree rooted at $F_2(u)$ or detaching the subtree rooted at $F_2(v)$. For the former (respectively, latter) choice, we also modify T_1' by eliminating u (respectively, v) and further recursively solve HNC on input $(k-1, T_1', F_2)$. So, we make two recursive calls here.

Case 2.2: $F_2(u)$ and $F_2(v)$ are in the same component tree of F_2. In this case, in order to transform F_2 into an AF of T_1 and T_2, it suffices to try three choices to modify F_2. The first two choices are the same as those in *Case 2.1*. In the third choice, we count the number b of pendant subtrees of F_2 between $F_2(u)$ and $F_2(v)$, modify F_2 by detaching the pendant subtrees between u and v, and recursively solve HNC on input $(k-b, T_1', F_2)$. Note that $b \geq 2$ and we make three recursive calls here.

Chen and Wang [8] show that their algorithm for HNC runs in $O\left(3^d n + 3^d (h-1)^{h-d+2}\right)$ time, where d is the rSPR distance between the two given trees T_1 and T_2, n is the number of leaves in T_1 (or equivalently, in T_2), and h is the hybridization number of T_1 and T_2. We note that h and d are usually quite close in practice.

The next lemma is the key to improving the algorithm in [8] for HNC.

Lemma 3.1 *Assume that the output of HNC on input $(k-1, T_1', F_2)$ is "no". Also suppose that u and v are sibling leaves in T_1' such that $F_2(v)$ and the parent p of $F_2(u)$ are siblings in F_2. Let q be the parent of p and $F_2(v)$ in F_2. Further assume that we can obtain an AAF F of T_1 and T_2 from F_2 by first detaching $F_2(u)$ and then performing $k-1$ more detaching operations on F_2, where we never detach a pendant subtree whose root is a child of q. Then, we can obtain another AAF of T_1 and T_2 from F_2 by first detaching $F_2(v)$ and then performing $k-1$ more detaching operations on F_2.*

PROOF. Obviously, p is not a node in F but q is. Moreover, since the output of HNC on input $(k-1, T_1', F_2)$ is "no", one child of q in F is $F_2(v)$ by Lemma 2.1. Let w be the other child of q in F. Note that w may be the sibling of $F_2(u)$ in F_2 or not. Furthermore, $F_2(u)$ may remain in F or

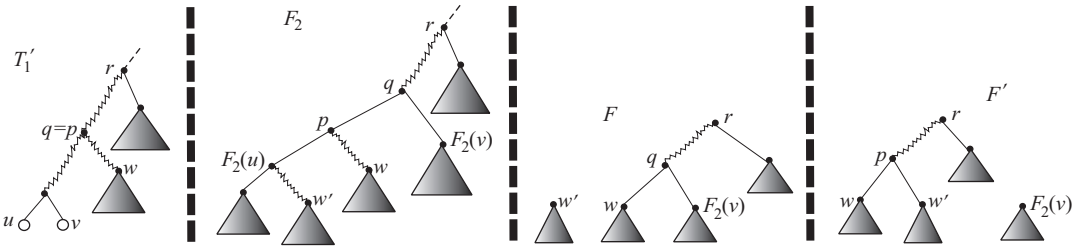

Figure 1: A portion of T_1', F_2, F, and F' in the proof of Lemma 3.1, where each zig-zag line indicates a path containing zero or more edges and each black triangle indicates a pendant subtree.

not. If $F_2(u)$ does not remain in F, then let w' be an arbitrary root of F that is a descendant of $F_2(u)$ in F_2; otherwise, let w' be $F_2(u)$.

Let F' be the forest obtained from F by first replacing the edge (q, w) with the three edges (q, p), (p, w), (p, w') and further detaching the subtree rooted at $F_2(v)$. Obviously, F' is an AF of T_1 and T_2. It remains to show that F' is an AAF of T_1 and T_2. For a contradiction, assume that F' is not an AAF of T_1 and T_2. Then, $G_{F'}$ contains a cycle. Consider a cycle C in $G_{F'}$ whose length is minimized over all cycles in $G_{F'}$. Then, C is an *alternating* cycle, i.e., there do not exist two consecutive edges (a, b) and (b, c) in C such that for some $i \in \{1, 2\}$, a is an ancestor of b in T_i and b is an ancestor of c in T_i. We next distinguish two cases as follows.

Case 1: q is not a root of F. In this case, $F_2(v)$ is a node of $G_{F'}$ but is not a node of G_F. In contrast, w' is a node of G_F but is not a node of $G_{F'}$. Other than $F_2(v)$ and w', each node of G_F is a node of $G_{F'}$ and vice versa. Thus, if $F_2(v)$ is not a node of C, then C is also a cycle in G_F and we are done because we have found a contradiction. So, we assume that $F_2(v)$ is a node of C. Let $(z_1, F_2(v))$ and $(F_2(v), z_2)$ be the directed edge entering and leaving $F_2(v)$ in C, respectively. Let r be the root of the component tree in F that contains q. Note that r is also a root of F'. Also recall that $F_2(v)$ is a node of F. So, both r and $F_2(v)$ are also nodes in both T_1 and T_2. One crucial point is that r is an ancestor of $F_2(v)$ in both T_1 and T_2. By this point and the minimality of C, r cannot be a node of C and (r, z_2) is an edge of G_F. To see that (z_1, r) is also an edge of G_F, first observe that z_1 is a proper ancestor of $F_2(v)$ in T_1 or T_2 because $(z_1, F_2(v))$ is an edge of $G_{F'}$. If z_1 is a proper ancestor of $F_2(v)$ in T_1 (respectively, T_2), then z_1 is a proper ancestor of r in T_1 (respectively, T_2) because the component tree of F with root r contains $F_2(v)$ and z_1 is a root of F. Hence, (z_1, r) is an edge of G_F. Therefore, if we modify C by replacing $F_2(v)$ with r, we obtain a cycle in G_F and hence a contradiction.

Case 2: q is a root of F. In this case, q is not a root of F' and p becomes a new root of F'. Moreover, the next two statements hold:

1. The location of p in T_1 is the same as that of q in T_1.
2. p is a proper descendant of q in T_2.

So, C must contain p or $F_2(v)$, because otherwise C would be a cycle in G_F. If C contains $F_2(v)$, then as in Case 1, we can prove that modifying C by replacing $F_2(v)$ with q yields a cycle in G_F. So, we may assume that C contains p but does not contain $F_2(v)$. Let (z_1, p) and (p, z_2) be the directed edge entering and leaving p in C, respectively. By Statements 1 and 2, (q, z_2) is an edge of the decision graph G_F associated with F. To see that (z_1, q) is also an edge of G_F, first observe that z_1 is a proper ancestor of $F_2(v)$ in T_1 or T_2 because $(z_1, F_2(v))$ is an edge of $G_{F'}$. If z_1 is a proper ancestor of $F_2(v)$ in T_1 (respectively, T_2), then z_1 is a proper ancestor of q in T_1 (respectively, T_2) because the component tree of F with root q contains $F_2(v)$ and z_1 is a root of F. Hence, (z_1, q) is

an edge of G_F. Therefore, if we modify C by replacing $F_2(v)$ with q, we obtain a cycle in G_F and hence a contradiction. □

Based on Lemma 3.1, we improve the algorithm in [8] for HNC by replacing Case 2.2 with the following two cases:

Case 2.2': Either $F_2(v)$ and the parent p of $F_2(u)$ are siblings in F_2 or $F_2(u)$ and the parent p of $F_2(v)$ are siblings in F_2. We assume that $F_2(v)$ and the parent p of $F_2(u)$ are siblings in F_2; the other case is similar. Then, Lemma 3.1 implies that in order to transform F_2 into an AF of T_1 and T_2, it suffices to try two choices to modify F_2, namely, by either detaching the subtree rooted at $F_2(v)$ or detaching the subtree rooted at the sibling of $F_2(u)$. For the latter case, we then recursively solve HNC on input $(k-1, T_1', F_2)$. On the other hand, for the former choice, we also modify T_1' by eliminating v and further recursively solve HNC on input $(k-1, T_1', F_2)$. So, we make two recursive calls here.

Case 2.3': $F_2(u)$ and $F_2(v)$ are in the same component tree of F_2 but Case 2.2' does not happen. In this case, we proceed as in the old Case 2.2.

Theorem 3.2 *Given two phylogenetic trees T_1 and T_2 with the same leaf set, we can compute the hybridization number of T_1 and T_2 in $O\left(\left(1+\sqrt{2}\right)^d n + \left(1+\sqrt{2}\right)^d (h-1)^{h-d+2}\right)$ time, where h and d are the hybridization number and the rSPR distance of T_1 and T_2, respectively.*

Proof. The analysis of the time complexity of the improved algorithm is almost the same as that of the original algorithm given in [8]. So, we only sketch the analysis here. First, as in [17], we can prove that for each $d \leq i \leq h$, we enumerate at most $(1+\sqrt{2})^i$ AFs F of T_1 and T_2 with $|F| = i$. Moreover, for each enumerated AF F with $|F| = i$, it takes $O\left((h-1)^{h-i+2}\right)$ time to try all possibilities to obtain an AAF of T_1 and T_2 from F by performing $h-i$ more detaching operations on F. So, the total time is $O\left(\left(1+\sqrt{2}\right)^d n + \sum_{i=d}^{h}\left(1+\sqrt{2}\right)^i (h-1)^{h-i+2}\right) = O\left(\left(1+\sqrt{2}\right)^d n + \left(1+\sqrt{2}\right)^d (h-1)^{h-d+2}\right)$. □

4 Sketch of Whidden et al.'s Algorithm for rSPR Distance

In this section, we sketch the previously fastest algorithm (due to Whidden et al. [17]) for computing the rSPR distance of two given phylogenetic trees. Whidden *et al.*'s algorithm indeed solves the following problem (denoted by rSPRDC, for convenience):

Input: (k, T_1, F_2), where k is a nonnegative integer, T_1 is a phylogenetic tree on a set X of species, and F_2 is a rooted forest obtained from some phylogenetic tree T_2 on X by performing zero or more detaching operations.

Output: "Yes" if performing k more detaching operations on F_2 leads to an AF of T_1 and T_2; "no" otherwise.

Obviously, to compute the rSPR distance between two given phylogenetic trees T_1 and T_2, it suffices to solve rSPRDC on input (k, T_1, T_2) for $k = 0, 1, 2, \ldots$ (in this order), until a "yes" is outputted.

Note that the input integer k to rSPRDC must be nonnegative. So, every time before we call an algorithm A for rSPRDC on an input (k, T_1, F_2), we need to check if $k \geq 0$. If $k > 0$, we proceed to call A on input (k, T_1, F_2); otherwise, we do not make the call. However, in order to keep the description of A simpler, we do not explicitly mention this checking process when we describe A.

Whidden *et al.*'s algorithm for rSPRDC is recursive and proceeds as follows. In the base case, $k = 0$ and it suffices to check if each component tree of F_2 is a pendant subtree of T_1. If each component tree of F_2 is a pendant subtree of T_1, then we output "yes" and stop. Otherwise, we output "no" and return. So, suppose that $k > 0$. Then, whenever T_1 has two sibling leaves u and v such that u and v are also sibling leaves in F_2, we modify T_1 and F_2 by merging u and v into a single leaf (say, u). Moreover, whenever F_2 has a root u that is also a leaf, we modify T_1 and F_2 by eliminating u from them. We repeat these two types of modifications of T_1 and F_2 until none of them is possible. After that, if F_2 becomes empty, then we can output "yes" and stop. Otherwise, we select two *arbitrary* sibling leaves u and v in T_1 and use them to distinguish three cases as follows.

Case 1: u and v are in different component trees of F_2. In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try two choices to modify them, namely, by either eliminating u or eliminating v. For each choice, we further recursively solve rSPRDC on input $(k-1, T_1, F_2)$. So, we make two recursive calls here.

Case 2: u and v are in the same component tree of F_2 and either (I) u and the parent of v are siblings in F_2 or (II) v and the parent of u are siblings in F_2. See Figure 2. In this case, if (I) (respectively, (II)) holds, then in order to transform T_1 and F_2 into identical forests with the minimum number of detaching operations, it suffices to modify F_2 by detaching the subtree rooted at the sibling w of v (respectively, u), and further recursively solve rSPRDC on input $(k-1, T_1, F_2)$ [17]. So, we make only one recursive call here.

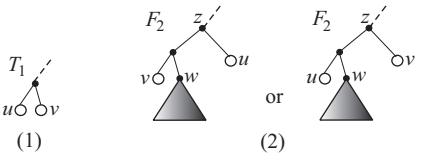

Figure 2: The best case in Whidden *et al.*'s algorithm: (1) The subtree of T_1 rooted at the parent of u and v, (2) The subtree of F_2 rooted at the LCA z of u and v, where each black triangle indicates a pendant subtree of F_2.

Case 3: u and v are in the same component tree of F_2 and neither (I) nor (II) in *Case 2* holds. In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try three choices to modify them. The first two choices are the same as those in *Case 1*. In the third choice, we count the number b of pendant subtrees of F_2 between u and v, modify F_2 by detaching the pendant subtrees between u and v, and recursively solve rSPRDC on input $(k-b, T_1, F_2)$. Note that $b \geq 2$ and we make three recursive calls here.

Let $t(k)$ be the time needed to solve rSPRDC on input (k, T_1, F_2). Whidden *et al.* [17] show that $t(k) = x^k n$, where n is the number of leaves in T_1 and x is the smallest real number satisfying the inequality $x^2 \geq 2x+1$. Intuitively speaking, this inequality originates from *Case 3* above where $b = 2$ (see Figure 3). Since $x = 1 + \sqrt{2}$, their algorithm runs in $O\left(2.415^k n\right)$ time.

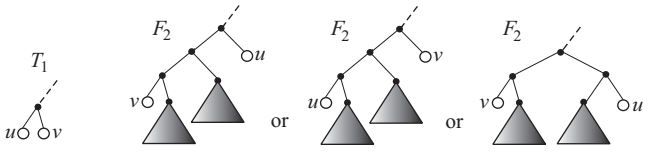

Figure 3: The worst case in Whidden *et al.*'s algorithm, where each black triangle indicates a pendant subtree of F_2.

5 Ideas for Improving Whidden et al.'s Algorithm

To improve Whidden *et al.*'s algorithm, our idea is to find two sibling leaves u and v in T_1 more carefully as follows.

Step 1. *Case 2* in Section 4 is *the best case* in Whidden *et al.*'s algorithm because we make only one recursive call in this case. So, we first try to find two sibling leaves u and v (in T_1) satisfying the condition in the best case (cf. Figure 2). If such two sibling leaves u and v exist in T_1, then we use them to modify F_2 and further recursively solve rSPRDC as in Case 2 of Whidden *et al.*'s algorithm. So, we hereafter assume that such two sibling leaves u and v do not exist in T_1.

Step 2. We then try to find two sibling leaves u and v (in T_1) such that the sibling q of the parent of u and v in T_1 is also a leaf of T_1 and either (1) u and q are sibling leaves in F_2 or (2) v and q are sibling leaves in F_2 (see Figure 4). If such two sibling leaves u and v exist in T_1, then we say that *the best case* in our algorithm occurs because this case is essentially symmetric to the best case in Whidden *et al.*'s algorithm [2]. So, if such two sibling leaves u and v exist in T_1 and (1) (respectively, (2)) holds, then we can modify T_1 and F_2 by eliminating v (respectively, u) from them, and further recursively solve rSPRDC on input $(k-1, T_1, F_2)$. So, we hereafter assume that the best case in our algorithm does not occur.

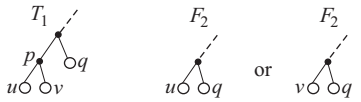

Figure 4: The best case in our algorithm.

Step 3. We try to find two sibling leaves u and v (in T_1) satisfying the condition in Case 1 of Whidden *et al.*'s algorithm. If such two sibling leaves u and v exist in T_1, then we use them to modify T_1 and F_2 and further recursively solve rSPRDC as in Case 1 of Whidden *et al.*'s algorithm. So, we hereafter assume that such two sibling leaves u and v do not exist in T_1.

Step 4. We try to find two sibling leaves u and v (in T_1) such that there are at least three pendant subtrees of F_2 between u and v. If such two sibling leaves u and v exist in T_1, then we use them to modify F_2 and further recursively solve rSPRDC as in Case 3 of Whidden *et al.*'s algorithm. So, we hereafter assume that such two sibling leaves u and v do not exist in T_1.

Step 5. We select two sibling leaves u and v (in T_1) whose distance from the root of T_1 is maximized. Let q be the sibling of the parent of u and v in T_1. By our choice of u and v, it follows that either q is a leaf or both children u' and v' of q in T_1 are leaves (see Figure 5). So, we have two cases to consider. Since both cases are complicated, we detail them in the next section.

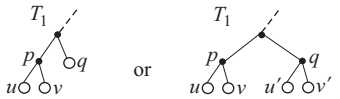

Figure 5: The two possible cases for two sibling leaves u and v farthest from the root.

6 Details of Step 5 in Section 5

In this section, we detail Step 5 in Section 5. By the assumptions in Steps 1 through 4 in Section 5, we know that for two arbitrary sibling leaves u and v in T_1, $z = \mathrm{LCA}_{F_2}\{u, v\}$ satisfies exactly one

[2] In other words, we can use a similar argument of [17] to prove the correctness of the processing of T_1 and F_2 in this case.

of the following conditions:

C1. z is the grandparent of both u and v in F_2 (see Figure 6(1)).

C2. z is the parent of one of u and v and is the great-grandparent of the other in F_2 (see Figure 6(2)).

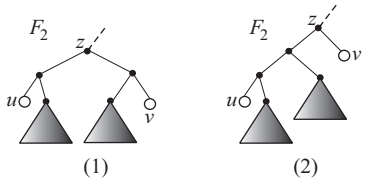

Figure 6: The two possible cases of F_2 for two sibling leaves in T_1, where each black triangle indicates a pendant subtree of F_2.

Recall that in Step 5, we first select two sibling leaves u and v (in T_1) whose distance from the root of T_1 is maximized. Let p be the parent of u and v in T_1 and q be the sibling of p in T_1. By our choice of u and v, it follows that either q is a leaf or both children of q in T_1 are leaves (see Figure 5). So, we have two cases to consider. The difficulty is how to use u and v to modify T_1 and/or F_2 and further recursively solve rSPRDC. In Section 6.1, we detail how to do this in the case where q is a leaf. The other case is detailed in Section 6.2.

6.1 The Case Where q Is a Leaf

Since u and v are sibling leaves in T_1, the assumptions in Steps 1 through 4 in Section 5 imply that u and q are not sibling leaves in F_2 and neither are v and q in F_2. Since u (respectively, v) and q are not sibling leaves in F_2, the (undirected) path between u (respectively, v) and q contains at least three edges if q belongs to the component tree of F_2 that contains u and v. We can also assume that the distance from z to u is not shorter than the distance from z to v, where $z = \mathrm{LCA}_{F_2}\{u, v\}$. So, one of the following cases occurs.

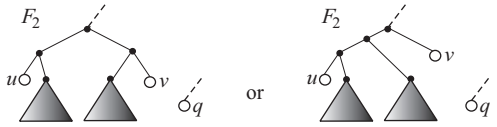

Figure 7: The subtrees of F_2 rooted at $\mathrm{LCA}_{F_2}\{u, v\}$ and q in Case 1.1, where each black triangle indicates a pendant subtree.

Case 1.1: q does not belong to the same component tree of F_2 as u and v (see Figure 7). In this case, our idea for speeding up Whidden *et al.*'s algorithm is as follows. Basically, we emulate Whidden *et al.*'s algorithm by using the sibling pair (u, v) to try three different choices of modifying T_1 and F_2. The first choice is to eliminate u from T_1 and F_2. A crucial point is that after the elimination of u, we know that v and q become a new sibling pair of T_1 and they belong to different component trees of F_2, implying that we need to use the pair (v, q) to further try two different choices of modifying T_1 and F_2 (namely, eliminating either v or q from T_1 and F_2). So, the first choice leads to two ways of modifying T_1 and F_2 one of which is to eliminate u and v from T_1 and F_2 and the other is to eliminate u and q from T_1 and F_2. Similarly, the second choice is to eliminate v from T_1 and F_2 and it leads to two ways of modifying T_1 and F_2 one of which is to eliminate v and u from T_1 and F_2 and the other is to eliminate v and q from T_1 and F_2. Note that the first and the second choices lead to four ways of modifying T_1 and F_2 and hence lead to four recursive calls.

The really crucial point here is that the four recursive calls can be reduced to three, because two of them are identical. Finally, the third choice is to modify F_2 by detaching the pendant subtrees between u and v. In summary, in order to transform T_1 and F_2 into identical forests, it suffices to try the following four different choices to modify them and further make recursive calls:

1. We eliminate u and v from T_1 and F_2, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.
2. We eliminate u and q from T_1 and F_2, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.
3. We eliminate v and q from T_1 and F_2, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.
4. We detach the two pendant subtrees of F_2 between u and v, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

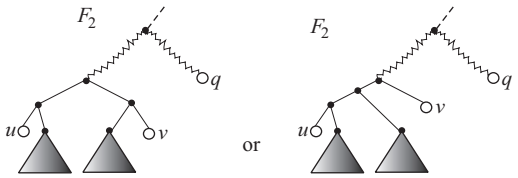

Figure 8: A portion of F_2 in Case 1.2, where each zig-zag line indicates a path containing at least one edge and each black triangle indicates a pendant subtree.

Case 1.2: u, v, and q belong to the same component tree of F_2 and $\mathrm{LCA}_{F_2}\{u, v\}$ is not an ancestor of q in F_2 (see Figure 8). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try four different choices to modify them and further make recursive calls:

1. We eliminate u from T_1 and F_2, and then recursively solve rSPRDC on input $(k-1, T_1, F_2)$.
2. We eliminate v and q from T_1 and F_2, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.
3. We first eliminate v from T_1 and F_2. Let b be the number of pendant subtrees of F_2 between u and q. Note that $b \geq 2$. We then detach the b pendant subtrees of F_2 between u and q. After that, we recursively solve rSPRDC on input $(k-1-b, T_1, F_2)$.
4. We detach the two pendant subtrees of F_2 between u and v, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

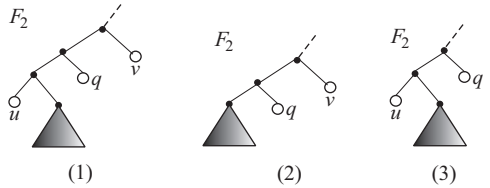

Figure 9: (1) The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u, v\}$ in Case 1.3, (2) the situation in Case 1.3 immediately after eliminating u from F_2, and (3) the situation in Case 1.3 immediately after eliminating v from F_2, where each black triangle indicates a pendant subtree.

Case 1.3: u, v, and q belong to the same component tree of F_2, $\mathrm{LCA}_{F_2}\{u, v\}$ is an ancestor of q in F_2, the (undirected) path between u and q in F_2 contains exactly three edges and so does the (undirected) path between v and q in F_2 (see Figure 9(1)). In order to transform T_1 and F_2 into identical forests, it suffices to try three difference choices to modify them and further make recursive calls:

1. We first eliminate u from T_1 and F_2 (see Figure 9(2)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further detach the subtree of F_2 rooted at the sibling of q. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

2. We first eliminate v from T_1 and F_2 (see Figure 9(3)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further detach the subtree of F_2 rooted at the sibling of u. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

3. We detach the two pendant subtrees of F_2 between u and v, and further recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

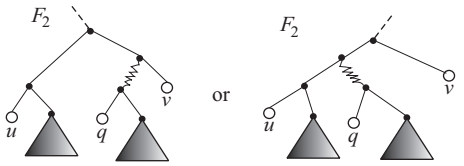

Figure 10: A portion of F_2 in Case 1.4, where each zig-zag line indicates a path containing at least one edge and each black triangle indicates a pendant subtree.

Case 1.4: u, v, and q belong to the same component tree of F_2, $\mathrm{LCA}_{F_2}\{u, v\}$ is an ancestor of q in F_2, and the (undirected) path between u and q in F_2 contains at least four edges and is not shorter than the (undirected) path between v and q in F_2 (see Figure 10). In this case, we proceed as in Case 1.2.

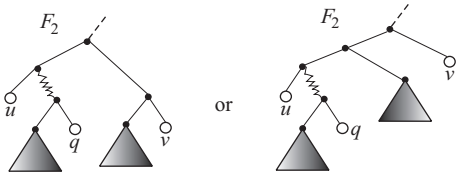

Figure 11: A portion of F_2 in Case 1.5, where each zig-zag line indicates a path containing at least one edge and each black triangle indicates a pendant subtree.

Case 1.5: u, v, and q belong to the same component tree of F_2, $\mathrm{LCA}_{F_2}\{u, v\}$ is an ancestor of q in F_2, and the (undirected) path between u and q in F_2 is shorter than the (undirected) path between v and q in F_2 (see Figure 11). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try four different choices to modify them and further make recursive calls:

1. We eliminate v from T_1 and F_2, and then recursively solve rSPRDC on input $(k-1, T_1, F_2)$.

2. We eliminate u and q from T_1 and F_2, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

3. We first eliminate u from T_1 and F_2. Let b be the number of pendant subtrees of F_2 between v and q. Note that $b \geq 2$. We then detach the b pendant subtrees of F_2 between v and q. After that, we recursively solve rSPRDC on input $(k-1-b, T_1, F_2)$.

4. We detach the two pendant subtrees of F_2 between u and v, and then recursively solving rSPRDC on input $(k-2, T_1, F_2)$.

6.2 The Case Where q Is a Non-leaf

Let u' and v' be the children of q in T_1. Let $z = \mathrm{LCA}_{F_2}\{u, v\}$ and $z' = \mathrm{LCA}_{F_2}\{u', v'\}$. By the assumptions in Steps 1 through 4 in Section 5, we know that Condition C1 or C2 holds for u and v and also for u' and v'. We can also assume that the distance from z to u in F_2 is not shorter than the distance from z to v in F_2. Similarly, we can further assume that the distance from z' to u' in F_2 is not shorter than the distance from z' to v' in F_2. So, one of the following cases occurs.

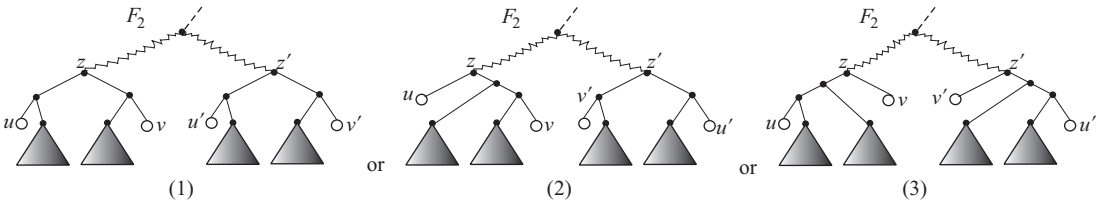

Figure 12: A portion of T_1 and F_2 in Case 2.1, where each black triangle indicates a pendant subtree and each zig-zag line indicates a path containing at least one edge.

Case 2.1: z and z' belong to the same component tree of F_2 and z is neither an ancestor nor a descendant of z' in F_2. In this case, we can further assume that whenever Condition C2 holds for u' and v', it holds for u and v as well. See Figure 12. We next distinguish two subcases as follows.

Case 2.1.1: z and z' are not siblings in F_2 or Condition C1 holds for u and v. In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2 (see Figure 13). Then, v and v' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between v and v'. The crucial point is that $b \geq 3$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-2-b, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate u and v' from T_1 and F_2 (so that v and u' become siblings in T_1 and the number b of pendant

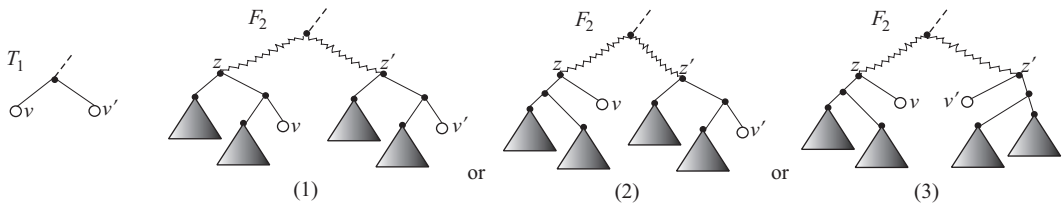

Figure 13: A portion of T_1 and F_2 in Case 2.1 immediately after eliminating u and u' from T_1 and F_2, where each black triangle indicates a pendant subtree and each zig-zag line indicates a path containing at least one edge.

subtrees of F_2 between v and u' is at least 4) and then proceed to use the sibling leaves v and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

3. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2. Then, we detach the two pendant subtrees of F_2 between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 (so that u and v' become siblings in T_1 and the number b of pendant subtrees of F_2 between u and v' is at least 4) and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and v' from T_1 and F_2 (so that u and u' become siblings in T_1 and the number b of pendant subtrees of F_2 between u and u' is at least 4) and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify T_1 and F_2 as follows. First, we eliminate v from T_1 and F_2. Then, we detach the two pendant subtrees of F_2 between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify F_2 by detaching the two pendant subtrees of F_2 between u and v. After these modifications, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

Case 2.1.2: z and z' are siblings in F_2 and Condition C2 holds for u and v. In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 similarly as in Item 1 in Case 2.1.1. The only difference is that after eliminating u and u' from T_1 and F_2, the number b of pendant subtrees of F_2 between v and v' is 2.

2. We modify T_1 and F_2 similarly as in Item 2 in Case 2.1.2. The only difference is that after eliminating u and v' from T_1 and F_2, the number b of pendant subtrees of F_2 between v and u' is 3.

3. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2. Then, we detach the two pendant subtrees of F_2 between u' and v'. Now, u' and v' are siblings in both T_1 and F_2. So, we modify T_1 and F_2 by merging u' and v' into a single leaf (say, u'). As can be seen from Figure 14, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further detach the subtree of F_2 rooted at the sibling of v. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 4 in Case 2.1.1. The only difference is that after eliminating u and v' from T_1 and F_2, the number b of pendant subtrees of F_2 between v and u' is 3.

5. Same as Item 5 in Case 2.1.1.

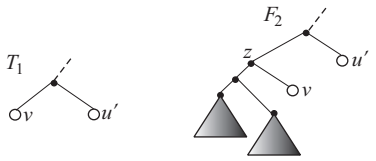

Figure 14: A portion of T_1 and F_2 in Case 2.1.2, where each black triangle indicates a pendant subtree.

6. Same as Item 6 in Case 2.1.1.

7. Same as Item 7 in Case 2.1.1.

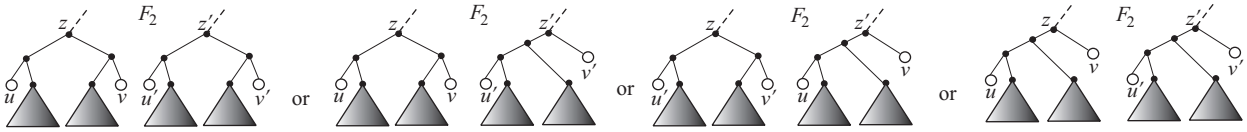

Figure 15: The subtrees of F_2 rooted at z and z' in Case 2.2, where each black triangle indicates a pendant subtree.

Case 2.2: z and z' belong to different component trees of F_2 (see Figure 15). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try eleven different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that v and v' belong to different component trees in F_2. We proceed to further modify T_1 and F_2 in two different ways and accordingly make two recursive calls as in Case 1 in Whidden *et al.*'s algorithm. In more detail, the two recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$ and $(k-3, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate u and v' from T_1 and F_2 and then proceed to use the sibling leaves v and u' to further modify T_1 and F_2 in two different ways and accordingly make two recursive calls as in Case 1 in Whidden *et al.*'s algorithm.

3. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2. Then, we detach the two pendant subtrees of F_2 between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in two different ways and accordingly make two recursive calls as in Case 1 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in two different ways and accordingly make two recursive calls as in Case 1 in Whidden *et al.*'s algorithm.

6. We modify T_1 and F_2 as follows. First, we eliminate v from T_1 and F_2. Then, we detach the two pendant subtrees of F_2 between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify F_2 by detaching the two pendant subtrees of F_2 between u and v. After these modifications, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

Case 2.3: $z = z'$ (see Figure 16). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try two different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2 (see Figure 17). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further modify T_1 and F_2 by eliminating v. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$. Note that eliminating u first and v next is equivalent to eliminating v first and u next.

2. We modify F_2 by detaching the two pendant subtrees of F_2 between u and v. After these modifications, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

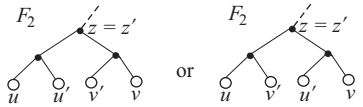

Figure 16: The subtree of F_2 rooted at $z = z'$ in Case 2.3.

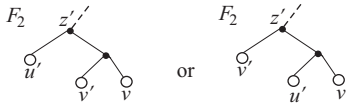

Figure 17: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.3. immediately after eliminating u.

Case 2.4: z is a child of z' in F_2 and the sibling of u in F_2 is u' (see Figure 18). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try five different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2 (see Figure 19(1)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further modify F_2 by detaching the subtree rooted at the parent of v. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

2. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2 (see Figure 19(2)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further modify F_2 by detaching the subtree rooted at the sibling of u. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

3. We modify T_1 and F_2 by eliminating v and v' from T_1 and F_2. We then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

4. We modify T_1 and F_2 by eliminating v from T_1 and F_2. We then modify F_2 by detaching the two pendant subtrees of F_2 between u' and v'. After the modification, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

5. We modify F_2 by detaching the two pendant subtrees of F_2 between u and v. After the modification, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

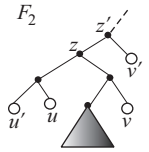

Figure 18: The subtree of F_2 rooted at z' in Case 2.4.

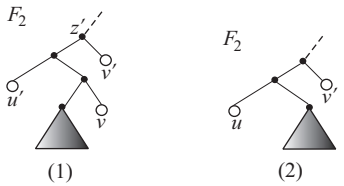

Figure 19: (1) The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.4 immediately after eliminating u from T_1 and F_2. (2) The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u, v'\}$ in Case 2.4 immediately after eliminating v and u'.

Case 2.5: z' is a child of z in F_2 and the sibling of u' in F_2 is u. This case is similar to Case 2.4.

Case 2.6: z is a child of z' in F_2 and the sibling of u in F_2 is not u' (see Figure 20). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try five different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate v from T_1 and F_2 (see Figure 21(1)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further modify F_2 by detaching the subtree rooted at the parent of u. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

2. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2 (see Figure 21(2)). Then, the best case in Whidden *et al.*'s algorithm occurs. More precisely, we can further modify F_2 by detaching the subtree rooted at the sibling of v. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

3. We modify T_1 and F_2 by eliminating u and v' from T_1 and F_2. We then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

4. We modify T_1 and F_2 by eliminating u from T_1 and F_2. We then modify F_2 by detaching the two pendant subtrees of F_2 between u' and v'. After the modification, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

5. We modify F_2 by detaching the two pendant subtrees of F_2 between u and v. After the modification, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

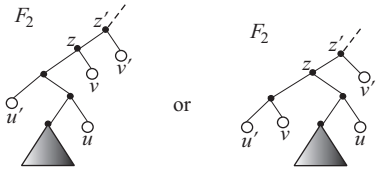

Figure 20: A portion of F_2 in Case 2.6.

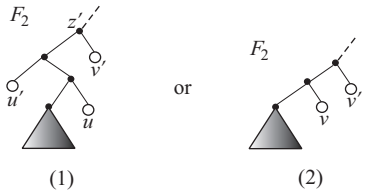

Figure 21: (1) The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.6 immediately after eliminating v from T_1 and F_2, and (2) the subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{v, v'\}$ in Case 2.6 immediately after eliminating u and u'.

Case 2.7: z' is a child of z in F_2 and the sibling of u' in F_2 is not u. This case is similar to Case 2.6.

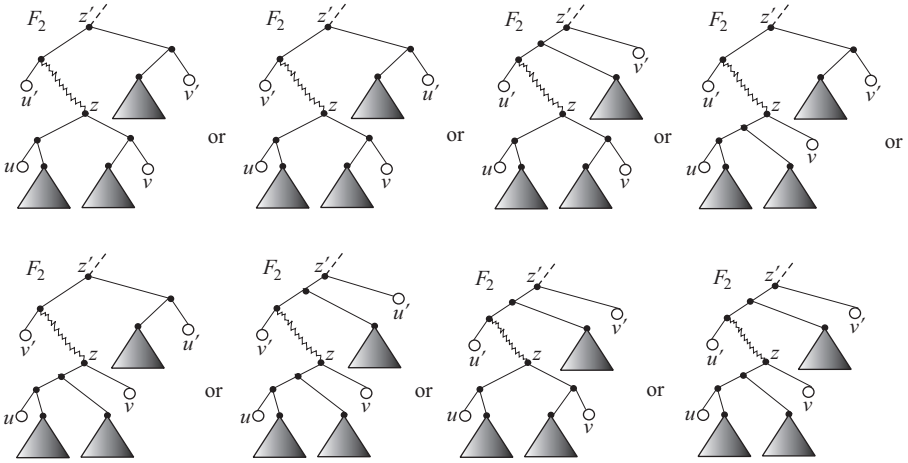

Figure 22: All possibilities of the relative locations of u, v, u', v', z, and z' in F_2 in Case 2.8, where each black triangle indicates a pendant subtree and each zig-zag line indicates a path containing at least one edge.

Case 2.8: There is a directed path P from z' to z in F_2 and P contains at least two edges. Then, Figure 22 shows all possibilities of the relative locations of u, v, u', v', z, and z' in F_2. Note that each zig-zag line in the figure indicates a path containing at least one edge. Depending on whether each such path contains one, two, or at least three edges, we distinguish several subcases as follows.

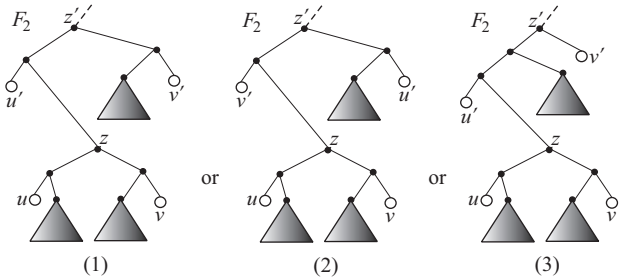

Figure 23: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.1.

Case 2.8.1: The sibling of z in F_2 is u' or v' and Condition C1 holds for u and v (see Figure 23). Obviously, if the situation in Figure 23(2) occurs, then by switching u' and v', we come to the situation in Figure 23(1). So, without loss of generality, we may assume that the sibling of z in F_2 is always u'. Now, in order to transform T_1 and F_2 into identical forests, it suffices to try eleven different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between v and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 by eliminating u and v' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 by eliminating v and v' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

6. We modify T_1 and F_2 by first eliminating v from them and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 24. As can be seen from Figure 24, the best case in our algorithm occurs and so we can further modify T_1 and F_2 by eliminating v'. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

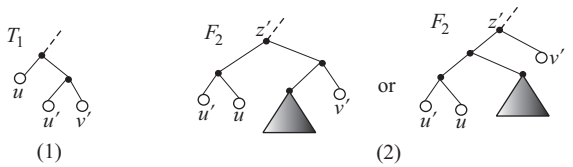

Figure 24: (1) The subtree of T_1 rooted at $\{u, u'\}$ in Case 2.8.1 immediately after merging the subtree rooted at the paren of u and v into a single leaf u, and (2) the subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.1 immediately after detaching the pendant subtrees of F_2 between u and v and further merging the subtree rooted at the parent of u and v into a single leaf u.

Case 2.8.2: The sibling of z in F_2 is u' or v', and Condition C2 holds for u and v (see Figure 25). Obviously, if the situation in Figure 25(2) occurs, then by switching u' and v', we come to the

situation in Figure 25(1). So, without loss of generality, we may assume that the sibling of z in F_2 is always u'. Now, in order to transform T_1 and F_2 into identical forests, it suffices to try nine different choices to modify them and make recursive calls as follows.

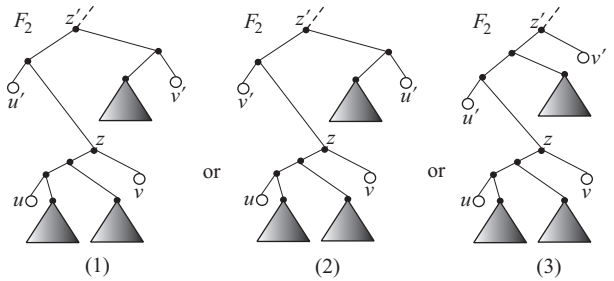

Figure 25: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.2.

1. We modify T_1 and F_2 by eliminating u and u' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

2. We modify T_1 and F_2 as follows. First, we eliminate u and v' from them. Then, the situation is as shown in Figure 26. As can be seen from Figure 26, the best case in Whidden *et al.*'s algorithm occurs and so we can further modify F_2 by detaching the subtree rooted at the sibling of v. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2. Then, u and v' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between u and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and v'.

5. We modify T_1 and F_2 by eliminating v and v' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

6. We modify F_2 by first eliminating v and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 24. As can be seen from Figure 24, the best case in our algorithm occurs and so we can further modify T_1 and F_2 by eliminating v'. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

Case 2.8.3: z is a grandchild of z' in F_2, the sibling of z in F_2 is neither u' nor v', and Condition C1 holds for u and v (see Figure 27). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try seventeen different choices to modify them and make recursive calls as follows.

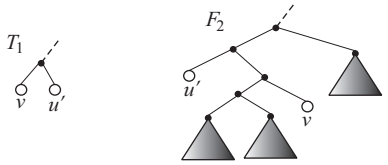

Figure 26: The situation in Case 2.8.2 immediately after eliminating u and v' from T_1 and F_2.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between v and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate u and v' from T_1 and F_2 and then proceed to use the sibling leaves v and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify F_2 by first eliminating v and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 28. We further try three different choices to modify T_1 and F_2 as follows.

   (a) We eliminate u' from T_1 and F_2. Then, the situation is as shown in Figure 29(1). As can be seen from Figure 29(1), the best case in Whidden *et al.*'s algorithm occurs and so we can further modify F_2 by detaching the subtree rooted at the sibling of u. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

   (b) We eliminate v' from T_1 and F_2. Then, the situation is as shown in Figure 29(2). As can be seen from Figure 29(2), the best case in Whidden *et al.*'s algorithm occurs and

so we can further modify F_2 by detaching the subtree rooted at the sibling of u'. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

(c) We modify F_2 by detaching the two pendant subtrees between u' and v'. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

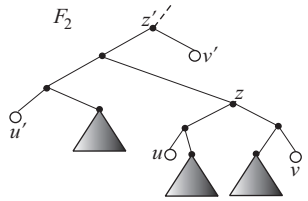

Figure 27: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.3.

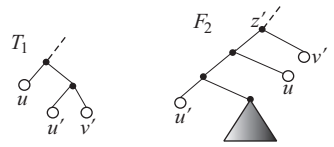

Figure 28: The situation in Case 2.8.3 immediately after detaching the pendant subtrees of F_2 between u and v and further merging the identical subtree of T_1 and F_2 rooted at the parent of u and v into a single leaf u.

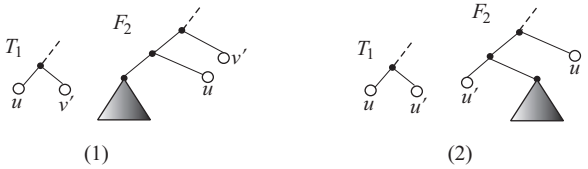

Figure 29: (1) The situation in Item 7a of Case 2.8.3 immediately after eliminating u' from T_1 and F_2. (2) The situation in Item 7b of Case 2.8.3 immediately after eliminating v' from T_1 and F_2.

Case 2.8.4: z is a grandchild of z' in F_2, the sibling of z in F_2 is neither u' nor v', and Condition C2 holds for u and v (see Figure 30). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try thirteen different choices to modify them and make recursive calls as follows.

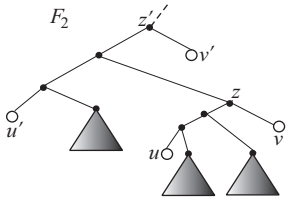

Figure 30: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.4.

1. We modify T_1 and F_2 by eliminating u and u' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

2. We modify T_1 and F_2 by eliminating u and v' from them, and then recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2. Then, u and v' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between u and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and v'.

5. We modify T_1 and F_2 similarly as in Item 4. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify F_2 by first eliminating v and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 28. Thus, we can proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Items 7a through 7c in Case 2.8.3.

Case 2.8.5: The sibling of the parent y of z in F_2 is u' or v', y is not an ancestor of u' or v', and Condition C1 holds for u and v (see Figure 31). Obviously, if the situation in Figure 31(2) occurs, then by switching u' and v', we come to the situation in Figure 31(1). So, without loss of generality, we may assume that the sibling of the parent of z in F_2 is always u'. Now, in order to transform T_1 and F_2 into identical forests, it suffices to try seventeen different choices to modify them and make recursive calls as follows.

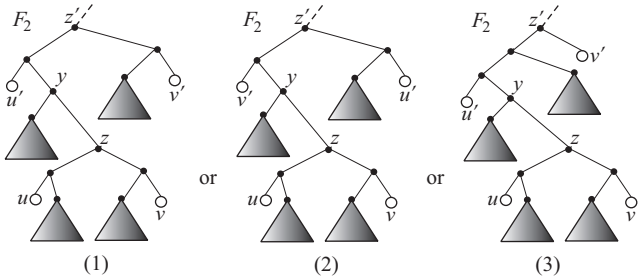

Figure 31: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.5.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that there are four pendant subtrees of F_2 between v and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-6, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 as follows. First, we eliminate u and v' from T_1 and F_2. Then, v and u' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between v and u'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and u'.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 similarly as in Item 2. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify T_1 and F_2 by first eliminating v from them and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 32. We further try three different choices to modify T_1 and F_2 as follows.

   (a) We eliminate u' from T_1 and F_2. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

   (b) We eliminate v' from T_1 and F_2. Then, the situation is as shown in Figure 33. As can be seen from Figure 33, the best case in Whidden *et al.*'s algorithm occurs and so we can further modify F_2 by detaching the subtree rooted at the sibling of u. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

   (c) We modify F_2 by detaching the two pendant subtrees between u' and v'. After that, we recursively solve rSPRDC on input $(k-4, T_1, F_2)$.

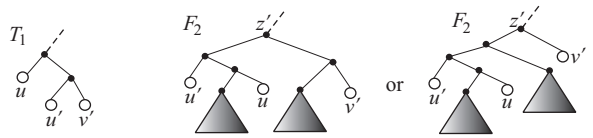

Figure 32: The situation in Case 2.8.5 immediately after detaching the pendant subtrees of F_2 between u and v and further merging the identical subtree of T_1 and F_2 rooted at the parent of u and v into a single leaf u.

Case 2.8.6: The sibling of the parent y of z in F_2 is u' or v', y is not an ancestor of u' or v', and Condition C2 holds for u and v (see Figure 34). Obviously, if the situation in Figure 34(2)

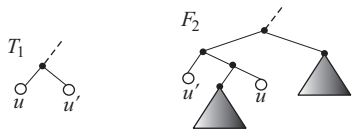

Figure 33: The situation in Item 7b of Case 2.8.5 immediately after eliminating u' from T_1 and F_2.

occurs, then by switching u' and v', we come to the situation in Figure 34(1). So, without loss of generality, we may assume that the sibling of the parent of z in F_2 is always u'. Now, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

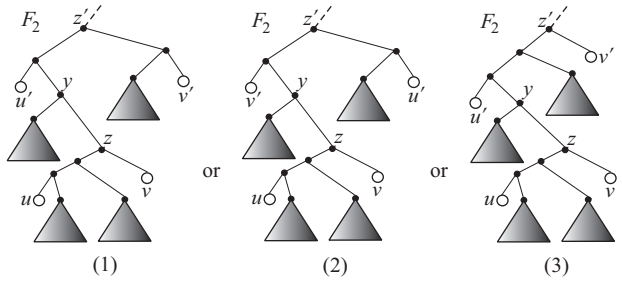

Figure 34: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.6.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees of F_2 between v and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 by eliminating u and v' from them. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2. Then, u and v' become two sibling leaves in T_1. The crucial point is that there are four pendant subtrees of F_2 between u and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-6, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and v'.

5. We modify T_1 and F_2 as follows. First, we eliminate v and v' from T_1 and F_2. Then, u and u' become two sibling leaves in T_1. The crucial point is that there are three pendant subtrees

of F_2 between u and u'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and u'.

6. We modify F_2 by first eliminating v and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 as follows. First, we detach the two pendant subtrees of F_2 between u and v. Then, the subtree of T_1 rooted at the parent of u and v is identical to the subtree of F_2 rooted at the parent of u and v. So, we can modify T_1 and F_2 by merging the subtree into a single leaf (say, u). Now, the situation is as shown in Figure 32. Thus, as in Item 7 of Case 2.8.5, we further try three different choices to modify T_1 and F_2 and then make three recursively calls.

Case 2.8.7: z is a great-grandchild of z' in F_2, the sibling of the parent y of z in F_2 is neither u' nor v', y is not an ancestor of u' or v', and Condition C1 holds for u and v (see Figure 35). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

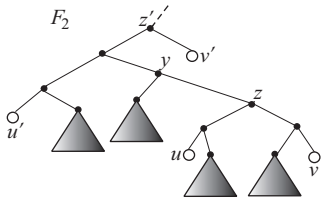

Figure 35: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.7.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. The crucial point is that there are four pendant subtrees of F_2 between v and v'. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-6, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate u and v' from T_1 and F_2 and then proceed to use the sibling leaves v and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

3. We modify F_2 by first eliminating u and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and u' from T_1 and F_2 and then proceed to use the sibling leaves u and v' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

5. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify F_2 by first eliminating v and then detaching the two pendant subtrees of F_2 between u' and v'. We further recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 by detaching the two pendant subtrees of F_2 between u and v. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

Case 2.8.8: z is a great-grandchild of z' in F_2, the sibling of the parent y of z in F_2 is neither u' nor v', y is not an ancestor of u' or v', and Condition C2 holds for u and v (see Figure 36). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

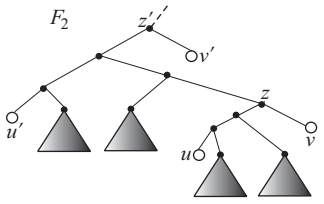

Figure 36: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.8.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2 (see Figure 37). Then, v and v' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between v and v'. The crucial point is that $b = 3$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-5, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

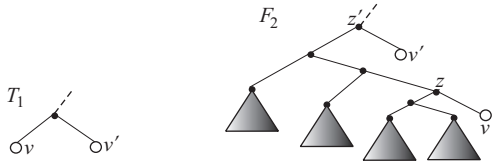

Figure 37: The situation after eliminating u and u' from T_1 and F_2 in Figure 36.

2. We modify T_1 and F_2 similarly as in Item 1. The only difference is that we first eliminate u and v' from T_1 and F_2 and then proceed to use the sibling leaves v and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

3. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2. Then, we modify F_2 by detaching the pendant subtrees between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

4. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2 (see Figure 38). Then, u and v' become two sibling leaves in T_1. Let b be the number of pendant subtrees

of F_2 between u and v'. The crucial point is that $b = 4$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-6, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and v'.

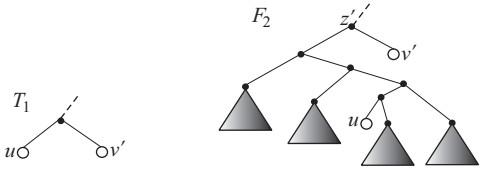

Figure 38: The situation after eliminating v and u' from T_1 and F_2 in Figure 36.

5. We modify T_1 and F_2 similarly as in Item 4. The only difference is that we first eliminate v and v' from T_1 and F_2 and then proceed to use the sibling leaves u and u' to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm.

6. We modify T_1 and F_2 as follows. First, we eliminate v from T_1 and F_2. Then, we modify F_2 by detaching the pendant subtrees between u' and v'. After these modifications, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify F_2 by detaching the pendant subtrees between u and v. After these modifications, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

Case 2.8.9: None of Cases 2.3.1 through 2.3.8 occurs (see Figure 39). In this case, in order to transform T_1 and F_2 into identical forests, it suffices to try fifteen different choices to modify them and make recursive calls as follows.

1. We modify T_1 and F_2 as follows. First, we eliminate u and u' from T_1 and F_2. Then, v and v' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between v and v'. The crucial point is that $b \geq 4$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-2-b, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and v'.

2. We modify T_1 and F_2 as follows. First, we eliminate u and v' from T_1 and F_2. Then, v and u' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between v and u'. The crucial point is that $b \geq 3$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-2-b, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves v and u'.

3. We modify T_1 and F_2 as follows. First, we eliminate u from T_1 and F_2. Then, we modify F_2 by detaching the two subtrees between u' and v'. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

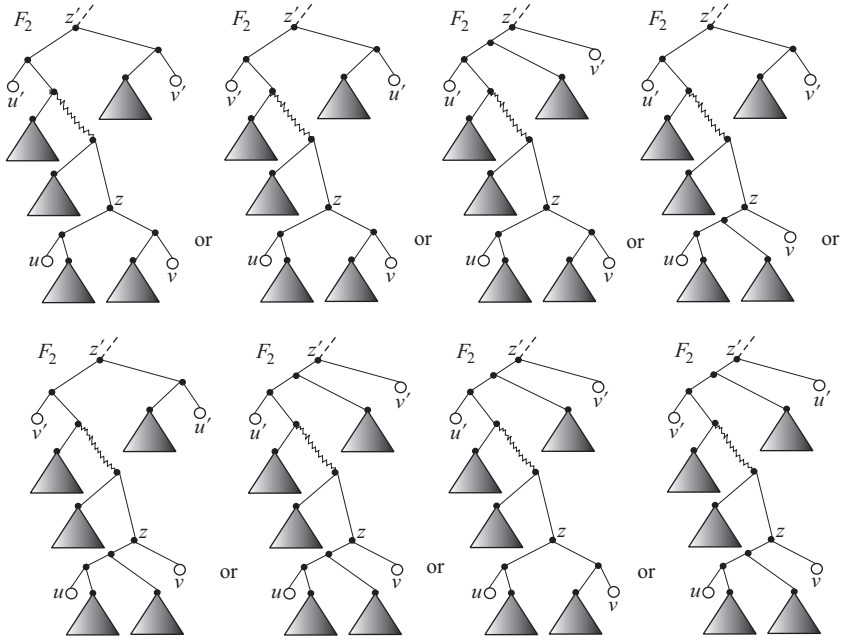

Figure 39: The subtree of F_2 rooted at $\mathrm{LCA}_{F_2}\{u', v'\}$ in Case 2.8.9, where each black triangle indicates a pendant subtree and each zig-zag line indicates a path containing at least one edge.

4. We modify T_1 and F_2 as follows. First, we eliminate v and u' from T_1 and F_2. Then, u and v' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between u and v'. The crucial point is that $b \geq 5$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-2-b, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and v'.

5. We modify T_1 and F_2 as follows. First, we eliminate v and v' from T_1 and F_2. Then, u and u' become two sibling leaves in T_1. Let b be the number of pendant subtrees of F_2 between u and u'. The crucial point is that $b \geq 4$. We proceed to further modify T_1 and F_2 in three different ways and accordingly make three recursive calls as in Case 3 in Whidden *et al.*'s algorithm. In more detail, the three recursive calls solve rSPRDC on input $(k-3, T_1, F_2)$, $(k-3, T_1, F_2)$, and $(k-2-b, T_1, F_2)$, respectively. Intuitively speaking, we are able to avoid the worst case in Whidden *et al.*'s algorithm when processing the sibling leaves u and u'.

6. We modify T_1 and F_2 as follows. First, we eliminate v from T_1 and F_2. Then, we modify F_2 by detaching the two subtrees between u' and v'. After that, we recursively solve rSPRDC on input $(k-3, T_1, F_2)$.

7. We modify T_1 and F_2 by detaching the pendant subtrees of F_2 between u and v. After that, we recursively solve rSPRDC on input $(k-2, T_1, F_2)$.

Case 2.9: There is a directed path P from z to z' in F_2 and P contains at least two edges. This case is similar to Case 2.8.

7 Time Complexity of the New Algorithm for rSPR Distance

In this section, we analyze the time complexity of our algorithm for rSPRDC.

Recall that our algorithm for rSPRDC is recursive and its base case is when the input integer k is 0. For technical reasons, we modify our algorithm so that its base case is when the input integer k is 1. In other words, in the modified version, we need to solve rSPRDC without making a recursive call for those inputs (k, T_1, F_2) with $k \leq 1$. Fortunately, it is easy to see that we can solve rSPRDC on an input (k, T_1, F_2) with $k \leq 1$ in linear time (say, by calling Whidden *et al.*'s algorithm). So, in the analysis below, we assume that our algorithm has been modified so that its base case is when the input integer k is 1.

The execution of the algorithm on input (k, T_1, F_2) can be modeled by a tree Γ in which the root corresponds to (k, T_1, F_2), each other node corresponds to a recursive call, and a recursive call A is a child of another call B if and only if B calls A directly. We call Γ the *search tree* on input (k, T_1, F_2). For convenience, we define the *size* of Γ to be the number of its nodes. The *depth* of a node u in Γ is the distance between the root and u in Γ. In particular, the depth of the root is 0. The *depth* of Γ is the maximum depth of a node in Γ.

Let $s(k)$ denote the maximum size of a search tree on input (k, T_1, F_2), where the maximum is taken over all T_1 and F_2. Clearly, $s(1) = 1$. Let k be an arbitrary positive integer. Obviously, if two sibling leaves are found in Step 1 (respectively, 2, 3, or 4) of our algorithm (cf. Section 5), then Inequality 1 (respectively, 1, 2, or 3) holds:

$$s(k) \leq s(k-1) + 1 \tag{7.1}$$
$$s(k) \leq 2s(k-1) + 1 \tag{7.2}$$
$$s(k) \leq 2s(k-1) + s(k-3) + 1 \tag{7.3}$$

Now, suppose that two sibling leaves are found in Step 5 of our algorithm. Then, one of Cases 1.1 through 2.9 in Section 6 happens. The inequalities for these cases are listed below.

$$s(k) \leq 4s(k-2) + 1 \quad \text{(Case 1.1)} \tag{7.4}$$
$$s(k) \leq s(k-1) + 2s(k-2) + s(k-3) + 1 \quad \text{(Cases 1.2, 1.5)} \tag{7.5}$$
$$s(k) \leq 3s(k-2) + 1 \quad \text{(Cases 1.3, 1.4)} \tag{7.6}$$
$$s(k) \leq s(k-2) + 10s(k-3) + s(k-5) + 3s(k-6) + 1 \quad \text{(Case 2.1.1)} \tag{7.7}$$
$$s(k) \leq s(k-2) + 9s(k-3) + 2s(k-4) + 2s(k-5) + s(k-6) + 1 \quad \text{(Case 2.1.2)} \tag{7.8}$$
$$s(k) \leq s(k-2) + 10s(k-3) + 1 \quad \text{(Case 2.2)} \tag{7.9}$$
$$s(k) \leq 2s(k-2) + 1 \quad \text{(Case 2.3)} \tag{7.10}$$
$$s(k) \leq 3s(k-2) + 2s(k-3) + 1 \quad \text{(Cases 2.4} \sim \text{2.7)} \tag{7.11}$$
$$s(k) \leq 2s(k-2) + 7s(k-3) + 2s(k-5) + 1 \quad \text{(Cases 2.8.1, 2.9.1)} \tag{7.12}$$
$$s(k) \leq 2s(k-2) + 6s(k-3) + s(k-5) + 1 \quad \text{(Cases 2.8.2, 2.9.2)} \tag{7.13}$$
$$s(k) \leq 10s(k-3) + 3s(k-4) + 4s(k-5) + 1 \quad \text{(Cases 2.8.3, 2.9.3)} \tag{7.14}$$
$$s(k) \leq 2s(k-2) + 6s(k-3) + 3s(k-4) + 2s(k-5) + 1 \quad \text{(Cases 2.8.4, 2.9.4)} \tag{7.15}$$
$$s(k) \leq 11s(k-3) + 2s(k-4) + 2s(k-5) + 2s(k-6) + 1 \quad \text{(Cases 2.8.5, 2.9.5)} \tag{7.16}$$
$$s(k) \leq s(k-2) + 9s(k-3) + 2s(k-4) + 2s(k-5) + s(k-6) + 1 \quad \text{(Cases 2.8.6, 2.9.6)} \tag{7.17}$$
$$s(k) \leq s(k-4) + 10s(k-3) + 4s(k-6) + 1 \quad \text{(Cases 2.8.7, 2.9.7)} \tag{7.18}$$
$$s(k) \leq s(k-2) + 10s(k-3) + s(k-5) + 2s(k-6)$$

$$+s(k-7)+1 \qquad \text{(Cases 2.8.8, 2.8.9, 2.9.8, 2.9.9)} \tag{7.19}$$

We claim that for all $k \geq 1$, $s(k) \leq 2.344^k - 1$. We show the claim by induction on k. When $k = 1$, we have $s(1) = 1 < 2.344^1 - 1$. So, suppose that $k \geq 2$. Then, by replacing $s(k-1), \ldots, s(k-7)$ in Inequalities 1 through 18 with $2.344^{k-1} - 1, \ldots, 2.344^{k-7} - 1$ respectively, we can verify that $s(k) \leq 2.344^k - 1$.

Obviously, excluding the recursive calls, the total time needed by the steps of the algorithm is linear. So, the time complexity of the algorithm is $O\left(2.344^k n\right)$, where n is the number of leaves in the input trees.

In summary, we have the following theorem:

Theorem 7.1 *Given two phylogenetic trees T_1 and T_2 with the same leaf set, we can compute the rSPR distance of T_1 and T_2 in $O\left(2.344^k n\right)$ time, where n is the number of leaves in the input trees.*

8 Concluding Remarks

We have presented faster algorithms for rSPR distance and hybridization number. Not only the algorithms have better theoretical time complexities than the previous bests, but also our preliminary experimental results show that the new algorithms run much faster for practical and simulated datasets. Since there are a lot of details in the implementation of the algorithms, we do not include the experimental results in this paper and will instead include them in a future work.

Acknowledgments

Zhi-Zhong Chen was supported in part by the Grant-in-Aid for Scientific Research of the Ministry of Education, Science, Sports and Culture of Japan, under Grant No. 24500023. Lusheng Wang was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 121608].

References

[1] Beiko, R.G. and Hamilton, N. (2006) Phylogenetic identification of lateral genetic transfer events, *BMC Evol. Biol.*, **6**, 159-169.

[2] Bonet, M.L., John, K. St., Mahindru, R., and Amenta, N. (2006) Approximating subtree distances between phylogenies, *Journal of Computational Biology*, **13**, 14191434.

[3] Bordewich, M., McCartin, C., and Semple, C. (2008) A 3-approximation algorithm for the subtree distance between phylogenies. *Journal of Discrete Algorithms*, **6**, 458471.

[4] Bordewich, M. and Semple, C. (2005) On the computational complexity of the rooted subtree prune and regraft distance, *Annals of Combinatorics*, **8**, 409-423.

[5] Bordewich, M. and Semple, C. (2007) Computing the minimum number of hybridization events for a consistent evolutionary history, *Discrete Applied Mathematics*, **155**, 914-928.

[6] Bordewich, M. and Semple, C. (2007) Computing the hybridization number of two phylogenetic trees is fixed-parameter tractable, *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, **4**, 458-466.

[7] Chen, Z.-Z. and Wang, L. (2010) HybridNet: A tool for constructing hybridization networks, *Bioinformatics*, **26**, 2912-2913.

[8] Chen, Z.-Z. and Wang, L. (2012a) Algorithms for reticulate networks of multiple phylogenetic trees, *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, **9**, 372-384.

[9] Chen, Z.-Z., Wang, L., and Yamanaka, S. (2012) A Fast Tool for Minimum Hybridization Networks, *BMC Bioinformatics*, 13:155.

[10] Collins, L., Linz, S., and Semple, C. (2011) Quantifying hybridization in realistic time. *Journal of Computational Biology*, **18**, 1305-1318.

[11] Goloboff, P.A. (2007) Calculating SPR distances between trees, *Cladistics*, **24**(4), 591-597.

[12] Hein, J., Jing, T., Wang, L., and Zhang, K. (1996) On the complexity of comparing evolutionary trees. *Discrete Appl. Math.*, **71**, 153-169.

[13] Hill, T., Nordström, K.J., Thollesson, M., Säfström, T.M., Vernersson, A.K., Fredriksson, R., and Schiöth, H.B. (2010) SPRIT: Identifying horizontal gene transfer in rooted phylogenetic trees, *BMC Evolutionary Biology* 10:42.

[14] MacLeod, D., Charlebois, R.L., Doolittle, F., and Bapteste, E. (2005) Deduction of probable events of lateral gene transfer through comparison of phylogenetic trees by recursive consolidation and rearrangement, *BMC Evolutionary Biology*, **5**(1):27.

[15] Rodrigues, E.M, Sagot, M.-F., and Wakabayashi, Y. (2007) The maximum agreement forest problem: Approximation algorithms and computational experiments. *Theoretical Computer Science*, **374**, 91110.

[16] Than, C., Ruths, D., Nakhleh, L. (2008) PhyloNet: a software package for analyzing and reconstructing reticulate evolutionary relationships, *BMC Bioinformatics*, **9**:322.

[17] Whidden, C., Beiko, R. G., and Zeh, N. (2010) Fast FPT algorithms for computing rooted agreement forest: theory and experiments, *LNCS*, **6049**, 141-153.

[18] Whidden, C. and Zeh, N. (2009) A unifying view on approximation and FPT of agreement forests. *LNCS*, **5724**, 390-401.

[19] Wu, Y. (2009) A practical method for exact computation of subtree prune and regraft distance. *Bioinformatics*, **25**(2), 190-196.

[20] Wu, Y. (2010) Close lower and upper bounds for the minimum reticulate network of multiple phylogenetic trees, *Bioinformatics [ISMB]*, **26**, 140-148.

[21] Wang, J. and Wu, Y. (2010) Fast computation of the exact hybridization number of two phylogenetic trees, *Proceedings of ISBRA 2010*, 203-214.