Better Practical Algorithms for rSPR Distance and Hybridization Number

Kohei Yamada
Division of Information System Design, Tokyo Denki University, Japan
19rmd38@ms.dendai.ac.jp

Zhi-Zhong Chen
Division of Information System Design, Tokyo Denki University, Japan
zzchen@mail.dendai.ac.jp

Lusheng Wang
Department of Computer Science, City University of Hong Kong, China
cswangl@cityu.edu.hk

Abstract

The problem of computing the rSPR distance of two phylogenetic trees (denoted by RDC) is NP-hard and so is the problem of computing the hybridization number of two phylogenetic trees (denoted by HNC). Since they are important problems in phylogenetics, they have been studied extensively in the literature. Indeed, quite a number of exact or approximation algorithms have been designed and implemented for them. In this paper, we design and implement one exact algorithm for HNC and several approximation algorithms for RDC and HNC. Our experimental results show that the resulting exact program is much faster (namely, more than **80 times faster** for the easiest dataset used in the experiments) than the previous best and its superiority in speed becomes even more significant for more difficult instances. Moreover, the resulting approximation programs output much better results than the previous bests; indeed, the outputs are always nearly optimal and often optimal. Of particular interest is the usage of the Monte Carlo tree search (MCTS) method in the design of our approximation algorithms. Our experimental results show that with MCTS, we can often solve HNC exactly within short time.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases phylogenetic tree, fixed-parameter algorithms, approximation algorithms, Monte Carlo tree search

Digital Object Identifier 10.4230/LIPIcs.WABI.2019.5

Supplement Material Our programs are available at `http://rnc.r.dendai.ac.jp/rsprHN.html`.

1 Introduction

Constructing the evolutionary history of a set of species is an important problem in the study of biological evolution. Phylogenetic trees are used in biology to represent the ancestral history of a collection of existing species. This is appropriate for many groups of species. However, due to reticulation events such as hybridization, recombination, and lateral gene transfer, there are certain groups for which the ancestral history cannot be represented by a tree. For this kind of groups of species, it is more appropriate to represent their ancestral history by rooted acyclic digraphs, where vertices of in-degree at least two represent reticulation events.

More specifically, by looking at two different segments of sequences or two different sets of genes of a set of extant species, we may obtain two different phylogenetic trees T_1 and T_2 of the same extant species with high confidence. Given T_1 and T_2, we want to construct a reticulate network N with the smallest number of reticulation events needed to explain the

evolution of the species under consideration [13]. Roughly speaking, N is the smallest rooted acyclic digraph such that each of T_1 and T_2 is homeomorphic to a subgraph of N. The number of vertices of in-degree larger than 1 in N is called the *hybridization number* of T_1 and T_2. The problem of computing the hybridization number of two given phylogenetic trees, denoted by HNC, is NP-hard [12, 4]. For this reason, quite a number of approximation algorithms and fixed-parameter algorithms have been designed and implemented for HNC [1, 9, 11, 14, 16, 20]. To the best of our knowledge, the software in [10] for solving HNC exactly achieves the previously best speed in practice, while the software in [16] for solving HNC approximately achieves the previously best approximation-ratio in practice.

A problem closely related to HNC is the problem of computing the rSPR distance of two given phylogenetic trees T_1 and T_2 of the same extant species. The *rSPR distance* between T_1 and T_2 can be defined as the minimum number of edges that should be deleted from each of T_1 and T_2 in order to transform them into homeomorphic rooted forests F_1 and F_2. The problem of computing the rSPR distance of two trees, denoted by RDC, is NP-hard [4, 12]. This has motivated researchers to design and implement either exact or approximation algorithms for RDC [4, 6, 7, 8, 10, 12, 15, 17, 20, 19]. To the best of our knowledge, the software in [7] for solving RDC exactly (respectively, approximately) achieves the previously best speed (respectively, approximation-ratio) in practice (`http://rnc.r.dendai.ac.jp/rspr.html`).

In this paper, we first improve Chen *et al.*'s exact algorithm [10] for HNC. Since rSPR distance is a lower bound on hybridization number, the main idea is to use the lower bound on rSPR distance outputted by Chen *et al.*'s algorithm [7] to cut unnecessary branches of the search tree. Another main idea is to arrange the child recursive calls of each recursive call carefully. Our experimental results show that the resulting algorithm can be implemented into a program that runs more than **80 times faster** than Chen *et al.*'s UltraNet [10] for the easiest dataset used in the experiments. Moreover, its superiority in speed becomes even more significant for more difficult instances.

We then present a new approximation algorithm for RDC. Although this algorithm does not necessarily always output a better result than the algorithm in [7], we can obtain a new algorithm which calls the two algorithms and outputs the better result returned by them. Our experimental results show that the resulting algorithm can be implemented into a program that often outputs better results than Chen *et al.*'s program [7]. We further propose to use the so-called Monte Carlo tree search (MCTS) method [5] to improve *any* approximation algorithm A for RDC. In our application of MCTS, instead of performing a number of random play-outs [1] in the simulation phase of each round, we make a single call of A and then in the backpropagation phase, use its returned result to update information in the sequence of nodes selected for this round. Our experimental results show that the MCTS-based algorithm (denoted by MCTS_A) can be implemented into a program that outputs much better and indeed always nearly optimal results. It is worth mentioning that even if A has a theoretical performance guarantee (say, a 2-approximation ratio), we are unable to prove any theoretical performance guarantee for MCTS_A. Nevertheless, our experimental results show that MCTS_A almost always outputs better results than A. Of course, if we want MCTS_A to have the same theoretical performance guarantee as A, then we can modify MCTS_A so that it calls A and uses A's output instead if its own is worse.

[1] Roughly speaking, a random play-out here means computing an approximate solution by always making a random choice whenever a decision has to be made among a set of multiple choices.

We further combine our MCTS-based approximation algorithm for RDC with the integer-linear programming (ILP) approach in [16] to obtain a new approximation algorithm for HNC. Our experimental results show that the new algorithm can be implemented into a program that outputs much better (indeed always nearly optimal and often optimal) results than the previous best in [16].

Our programs are available at `http://rnc.r.dendai.ac.jp/rsprHN.html`.

2 Preliminaries

Throughout this paper, a *rooted forest* always means a directed acyclic graph in which every vertex 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 vertices 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 vertex 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 vertex 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 a vertex v of F, the *subtree of* F *rooted at* v, denoted by F^v, is the subgraph of F whose vertices are the descendants of v in F and whose edges are those edges connecting two descendants of v in F. If v is a root of F, then F^v is a *component tree* of F; otherwise, it is a *pendant subtree* of F. For convenience, we view each vertex u of F as an ancestor and descendant of u itself. A vertex u is *lower than* another vertex $v \neq u$ in F if u is a descendant of v in F. The *lowest common ancestor* (LCA) of a set U of vertices in F, denoted by $\ell_F(U)$, is the lowest vertex v in F such that for every vertex $u \in U$, v is an ancestor of u in F. Note that if no component tree of F contains all vertices of U, then $\ell_F(U)$ does not exist. Two vertices u and v of F are *incomparable* if neither of them is an ancestor of the other in F. For two incomparable vertices u and v appearing in the same component tree of F, $D_F(u, v)$ denotes the set of all vertices w such that w is not a vertex of the (undirected) path $P_{u,v}$ between u and v in F but is the child of some inner vertex of $P_{u,v}$. For each pendant subtree T of F that has at least two leaves, the leaf-label set of T is a *cluster* of F.

A *rooted binary forest* is a rooted forest in which the out-degree of every non-leaf vertex 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 vertex of F with parent p, then *detaching* F^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.

2.1 Phylogenetic Trees and Forests

Let X be a set of existing species. A *phylogenetic tree* on X is a rooted binary tree whose leaf set is X. A *phylogenetic forest* is the graph obtained by applying a sequence of zero or more detaching operations on a phylogenetic tree. In other words, a phylogenetic forest is a graph whose connected components are phylogenetic trees on different sets of species.

An *FF pair* is a pair (F_1, F_2), where F_1 and F_2 are two phylogenetic forests on the same set X of species. A *TT pair* is an FF pair (F_1, F_2) such that both F_1 and F_2 are trees.

For an FF pair (F_1, F_2), the labeled leaves of F_1 naturally one-to-one correspond to those of F_2 (i.e., each pair of corresponding leaves have the same label). We extend the correspondence between the labeled leaves of F_1 and F_2 to (some of) their ancestors recursively as follows. Suppose that v_1 is a non-leaf vertex of F_1, v_2 is a non-leaf vertex of F_2, and the children of v_1 in F_1 one-to-one correspond to those of v_2 in F_2. Then, v_1 corresponds to v_2.

An FF pair (F_1, F_2) is *proper* if every root of F_1, except at most one, corresponds to a root in F_2. Obviously, a TT pair is also a proper FF pair. *Simplifying* a proper FF pair (F_1, F_2) is to repeatedly perform the following operation on F_1 and F_2 until it is not applicable:

- If some non-root vertex v of F_1 corresponds to a root of F_2, then modify F_1 by detaching F_1^v.

Obviously, if (F_1, F_2) is proper, then it remains proper after being simplified.

Throughout the remainder of this paper, *an FF pair always means a proper FF pair*. A *sub-FF pair* of a TT pair (T_1, T_2) is an FF pair (F_1, F_2) such that for each $i \in \{1, 2\}$, F_i is obtained from T_i by performing zero or more detaching operations.

For an FF-pair (F_1, F_2), if a vertex v_1 of F_1 and a vertex v_2 of F_2 correspond to each other, then both v_1 and v_2 are *matched* and they are the *mates* of each other. For brevity, if v is a matched vertex of F_i for some $i \in \{1, 2\}$, then we will also use v to denote its mate in F_{3-i}.

2.2 Agreement Forests

Let (F_1, F_2) be a sub-FF pair of a TT pair (T_1, T_2). If we can apply a sequence of detaching operations on each of F_1 and F_2 so that they become the same forest F, then we refer to F as an *agreement forest* (AF) of (F_1, F_2). A *maximum agreement forest* (MAF) of (F_1, F_2) is an AF of (F_1, F_2) whose size is minimized over all AFs of (F_1, F_2). The size of an MAF of (F_1, F_2) minus $|F_2|$ is called the *rSPR distance* of (F_1, F_2), and is denoted by $d(F_1, F_2)$. Obviously, an AF F of (F_1, F_2) is also an AF of (T_1, T_2). The following lemma is shown in [7].

▶ **Lemma 1** ([7]). *Given an FF-pair (F_1, F_2), we can compute a lower bound b_ℓ and an upper bound b_u on the rSPR distance of (F_1, F_2) in cubic time such that $b_u \leq 2b_\ell$.*

Suppose that F is an AF of (T_1, T_2). For each $i \in \{1, 2\}$, we can define an injective mapping f_i from the vertex set of F to that of T_i as follows. For each leaf u of F, $f_i(u)$ is the leaf of T_i with the same label. For each non-leaf vertex u of F, $f_i(u)$ is $\ell_{T_i}(f_i(v_1), \ldots, f_i(v_q))$, where v_1, …, v_q are the leaf descendants of u in F. For convenience, we hereafter also use each vertex u of F to denote $f_i(u)$ in T_i. We can now use F, T_1, and T_2 to construct a directed graph G_F as follows:

- The vertices 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 $(T_1.T_2)$; otherwise, F is a *cyclic agreement forest* (CAF) of (T_1, T_2). If F is an AAF of (T_1, T_2) and its size is minimized over all AAFs of (T_1, T_2), then F is a *maximum acyclic agreement forest* (MAAF) of (T_1, T_2). The *hybridization number* of (T_1, T_2) is the size of an MAAF of (T_1, T_2), and is denoted by $h(T_1, T_2)$.

We are now ready to define the problems studied in this paper:

Hybridization Number Computation (HNC):

Input: A TT-pair (T_1, T_2).

Output: The hybridization number of (T_1, T_2).

rSPR Distance Computation (RDC):

Input: A TT-pair (T_1, T_2).

Output: The rSPR distance of (T_1, T_2).

2.3 Transforming a CAF to an AAF

Suppose that F is a CAF of a TT-pair (T_1, T_2). We construct a directed graph D as follows. For every non-leaf vertex of F, we create a vertex in D. There is an edge in D from a vertex u to a vertex v precisely if in either F_1 or F_2 (or in both), there is a directed path from u to v. A *minimum directed feedback vertex set* (MDFVS) of D is a minimum-sized set U of vertices in D such that modifying D by removing the vertices in U yields a directed acyclic graph.

▶ **Lemma 2** ([14]). *Let U be an MDFVS of D. Then, to transform F to an AAF of (F_1, F_2) by performing a minimum number of detaching operations on F, it suffices to modify F by removing the vertices corresponding to those in U and further contracting unifurcate vertices.*

Let V be the set of vertices in D. By Lemma 2, to compute an MDFS U of D, it suffices to solve the following integer linear programming (ILP) model [16]:

$$\text{Minimize} : \sum_{v \in V} x_v \tag{1}$$

$$\text{s.t.} \quad 0 \leq \ell_v \leq |V| - 1 \qquad \textit{for all } v \in V \tag{2}$$

$$\ell_v \geq \ell_u + 1 - |V| x_u - |V| x_v \qquad \textit{for all } e = (u, v) \in E \tag{3}$$

$$\ell_v \in \mathbb{Z} \qquad \textit{for all } v \in V \tag{4}$$

$$x_v \in \{0, 1\} \qquad \textit{for all } v \in V \tag{5}$$

Fortunately, in our application, we will have an integer k and only want to know whether the optimal value of the objective function is bounded by k from above. So, we modify the model by replacing the objective function with any constant (say, 0) and adding the new constraint $\sum_{v \in V} x_v \leq k$. We refer to this modified model as *the ILP model associated with* (T_1, T_2, F, k).

3 Solving HNC Exactly

Our algorithm for solving HNC exactly will use a subroutine for the following parameterized version of HNC.

Parameterized HNC (PHNC):

Input: (T_1, T_2, F_1, F_2, k), where (T_1, T_2) is a TT pair, (F_1, F_2) is a sub-FF pair of (T_1, T_2), and k is an integer.

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

Several definitions are in order. Let (F_1, F_2) be an FF-pair, and $i \in \{1, 2\}$. A vertex v of F_i is *active* if v is a matched non-root vertex of F_i and its parent in F_i is not matched. Since (F_1, F_2) is an FF-pair, all active vertices of F_1 fall into the same component tree of F_1. An *active sibling-pair* of F_i is a pair (u, v) of active vertices in F_i such that u and v are siblings in F_i.

3.1 Key Ideas

Basically, our algorithm is a significantly refined version of the algorithm for HNC implemented in Chen and Wang's UltraNet [10]. In this subsection, we list the key new ideas behind our new algorithm for HNC.

First, the new algorithm builds on a recent 2-approximation algorithm for RDC [7]. When we compute the hybridization number, we use the lower bound outputted by the approximation algorithm to bound the search of the hybridization number. Since the lower bound is often nearly optimal, this bounding idea makes it possible for our algorithm to find the hybridization number in short time. Since the exact algorithm for RDC in [7] is also fast, we can use it to bound the search of the hybridization number instead of using the 2-approximation algorithm for RDC.

Secondly, the new algorithm is recursive and we make child recursive-calls in a careful order. More precisely, child recursive-calls that appear to finish in shorter time are made earlier than those that look likely to finish in longer time.

Thirdly, when we make a recursive call, we may know certain vertices v such that the subtree rooted at v should not be detached, and so we lock these vertices so that the subtrees rooted at them will never be detached in subsequent recursive calls. Moreover, the locked vertices help us make fewer child recursive-calls.

Finally, when our algorithm needs to transform a CAF F of a TT-pair (T_1, T_2) to an AAF of (T_1, T_2), we use the ILP-based method outlined in Section 2.3. However, we modify the ILP model in Section 2.3 as follows.

- Let D be the digraph constructed from F and (T_1, T_2) as in Section 2.3. Since F is a CAF, D has a cycle and we need to remove at least one vertex from D to make D acyclic. Once D becomes acyclic, its number of vertices has decreased by at least 1. So, it is safe to modify the ILP model by changing the upper bound on the value of ℓ_v from $|V| - 1$ to $|V| - 2$.
- Some vertices of F may have been locked. So, for each locked vertex v of F, we can modify the model by fixing $x_v = 0$.
- By Lemma 4 in [9], we know that for each edge (p, c) of F, if removing a set U of vertices from D with $\{p, c\} \subseteq U$ makes D acyclic, then removing the vertices of $U \setminus \{c\}$ also makes D acyclic. Thus, for each edge (p, c) of F, we can add the constraint $x_c \leq x_p$ to the model.

3.2 The Algorithm

Throughout this subsection, fix an instance (T_1, T_2) of HNC.

Our algorithm for computing $h(T_1, T_2)$ exactly first repeatedly performs a *cluster reduction* on T_1 and T_2 until no such reduction is applicable. For the detail of *cluster reductions*, the reader is referred to [2]. As the result of zero or more cluster reductions on T_1 and T_2, we obtain a sequence $(T_{1,1}, T_{2,1})$, ..., $(T_{1,q}, T_{2,q})$ of instances of HNC such that $q \geq 1$ and $h(T_1, T_2) = \sum_{i=1}^{q} h(T_{1,i}, T_{2,i})$. Hence, it suffices to compute $h(T_{1,i}, T_{2,i})$ for each $i \in \{1, \ldots, q\}$. Therefore, for simplicity, we hereafter assume that $q = 1$ and in turn $(T_1, T_2) = (T_{1,1}, T_{2,1})$.

Our algorithm then uses the program in [7] for RDC to compute $d(T_1, T_2)$. The program can also output an AF F of (T_1, T_2) with size $d(T_1, T_2)$. So, our algorithm checks whether F is indeed an AAF of (T_1, T_2) (by constructing the decision graph G_F associated with F and testing if G_F is acyclic or not). If it is, then $d(T_1, T_2)$ is also $h(T_1, T_2)$ and so the algorithm outputs $d(T_1, T_2)$ and stops. Thus, we hereafter assume that F is not an AAF of (T_1, T_2).

To compute $h(T_1, T_2)$, it suffices to solve PHNC on input (T_1, T_2, T_1, T_2, k) for $k = d(T_1, T_2)$, $d(T_1, T_2) + 1$, ... (in this order) until a "yes" is returned. So, it remains to detail our algorithm for PHNC. During its execution, our algorithm will lock certain non-root vertices v of F_2 at certain time points so that F_2^v will never be detached thereafter; it will always maintain the following invariant:

Invariant 1: Whenever a non-root vertex is locked by the algorithm, it knows that it will return "yes" with the locking if and only if it will return "yes" without the locking.

Our algorithm for PHNC is recursive and proceeds as follows. It starts by checking whether $k \geq 0$. If $k < 0$, then this is **Base Case 1** and it returns "no". So, we hereafter assume $k \geq 0$. Then, it simplifies (F_1, F_2) and further checks the following base case:

Base Case 2: *All roots of F_1 are matched.* In this case, F_1 and F_2 are the same forest and hence F_2 is an AF of (T_1, T_2). To test if F_2 is an AAF, our algorithm constructs the decision graph G_{F_2} associated with F_2 and tests if it is acyclic or not. If G_{F_2} is acyclic, then it returns "yes". Otherwise, it checks if $k \geq 1$ or not. If $k \leq 0$, then it returns "no". On the other hand, if $k \geq 1$, then it constructs the ILP model associated with (T_1, T_2, F_2, k) and solves the ILP model by an ILP solver (say, CPLEX or GUROBI); it returns "yes" if and only if the model is feasible.

We hereafter assume that one or more roots of F_1 are still not matched. Our algorithm then uses the program in [7] to compute a lower bound b_ℓ and an upper bound b_u on $d(F_1, F_2)$. The program will also return an AF F of (F_1, F_2) with size b_u as a witness for b_u. If $k < b_\ell$, then this is **Base Case 3**, and the algorithm returns "no". Otherwise, the algorithm checks if the ILP model associated with (T_1, T_2, F, k) is feasible or not. If it is feasible, then this is **Base Case 4**, and the algorithm returns "yes".

We hereafter assume that $k \geq b_\ell$ and the ILP model associated with (T_1, T_2, F, k) is infeasible. Clearly, both F_1 and F_2 must have at least one active sibling-pair. Our algorithm now distinguishes several cases *in the following order*.

Case 1: There is an active sibling-pair (u, v) in F_1 such that $|D_{F_2}(u, v)| = 1$. In this case, we clearly know that to transform F_2 into an AF of (F_1, F_2), we need to select at least one $x \in \{u, v, w\}$ and detach F_2^x, where $D_{F_2}(u, v) = \{w\}$. So, if all vertices of $\{u, v, w\}$ are locked, then this is **Base Case 5**, and the algorithm returns "no". Thus, we may assume that at least one vertex of $\{u, v, w\}$ is not locked. As observed in [18], selecting $x = u$ is the same as selecting $x = v$ (which means that the former selection leads to a "yes"-output if and only if so does the latter). Hence, if u or v is not locked, then our algorithm chooses an arbitrary unlocked $x \in \{u, v\}$ and makes a recursive call on input $(T_1, T_2, F_1, F_2', k-1)$, where F_2' is obtained from F_2 by detaching F_2^x. In addition, if w is also not locked, then our algorithm makes a recursive call on input $(T_1, T_2, F_1, F_2'', k-1)$, where F_2'' is obtained from F_2 by detaching F_2^w and further locking x in case the recursive call on input $(T_1, T_2, F_1, F_2', k-1)$ has been made. So, we make one or two recursive calls. If at least one call returns "yes", the algorithm returns "yes"; otherwise, it returns "no".

Case 2: There is an active sibling-pair (u, v) in F_2 such that $|D_{F_1}(u, v)| = 1$ and the unique vertex w in $D_{F_1}(u, v)$ is active. This case is symmetric to Case 1; so, the algorithm proceeds as in Case 1 except that each of u, v, and w is replaced by its mate.

Case 3: Neither Case 1 nor 2 occurs. In this case, our algorithm searches F_1 for an active sibling-pair (u, v) *in the following order*:

Type 1: Both u and v are locked in F_2.

Type 2: u and v belong to different connected components of F_2.

Type 3: Either u or v is locked in F_2.

Type 4: The sibling s of the parent of u and v in F_1 satisfies that either s is active or both children of s in F_1 are active.

Type 5: (u, v) is of none of the above types.

We emphasize that the smaller type of an active sibling-pair in F_1 is, the more our algorithm prioritizes it. Intuitively speaking, choosing an active sibling-pair of a smaller type in F_1 will likely lead to fewer recursive calls.

Suppose that our algorithm has selected an active sibling-pair (u, v) in F_1 as above. Our algorithm constructs a family $\mathcal{F}$ of sets as follows. Initially, $\mathcal{F}$ is empty. For each $y \in \{u, v\}$ such that y is not locked in F_2, we add the set $\{y\}$ to $\mathcal{F}$. Moreover, if no vertex in $D_{F_2}(u, v)$ is locked in F_2, then we add $D_{F_2}(u, v)$ to $\mathcal{F}$. Since Case 1 does not occur, $|D_{F_2}(u, v)| \geq 2$. Clearly, to transform F_2 into an AF of (F_1, F_2), we need to select a set $S \in \mathcal{F}$ and detach F_2^w for all $w \in S$. Thus, if $\mathcal{F}$ is empty, then our algorithm returns "no". Otherwise, it sorts the sets in $\mathcal{F}$ so that larger sets precede smaller sets. Let $S_1, \ldots, S_t$ be the sets in $\mathcal{F}$. For each $i \in \{1, \ldots, t\}$, let $F_{2,i}$ be the phylogenetic forest obtained from F_2 by first detaching F_2^y for all $y \in S_i$ and further distinguishing two cases as follows:

1. If $|S_i| \geq 2$, then lock both u and v in F_2.
2. If $i \geq 2$ and $|S_{i-1}| = |S_i| = 1$, then lock the vertex of S_{i-1} in F_2.

Now, our algorithm makes t recursive calls on input $(T_1, T_2, F_1, F_{2,1}, k-|S_1|), \ldots, (T_1, T_2, F_1, F_{2,t}, k - |S_t|)$. If at least one call returns "yes", the algorithm returns "yes"; otherwise, it returns "no".

4 Solving RDC Approximately

Basically, we want an approximate algorithm that outputs better results than the algorithm in [7]. Although the algorithm in [8] has a worse theoretical-guarantee than the algorithm in [7], it does not necessarily mean that the former always outputs worse results. So, we obtain a new approximation algorithm for RDC which simply runs the algorithms in [7, 8] and outputs the better result returned by them.

Our new idea is to use MCTS to improve the performance of *any* approximation algorithm for RDC. MCTS has a number of variants, but we here use the basic one (namely, the UCT algorithm) for its simplicity.

4.1 Outline of the Algorithm

In the remainder of this section, fix an FF-pair (F_1, F_2). Our algorithm for computing $d(F_1, F_2)$ approximately is recursive and starts by simplifying (F_1, F_2) and further checking whether F_2 is already an AF of (F_1, F_2). If it is, then this is **Base Case 1** and it returns 0. So, assume that F_2 is not an AF of (F_1, F_2). Then, F_1 has a unique non-matched root r. If r has at most 6 leaf descendants in F_1, then this is **Base Case 2** and our algorithm computes $d(F_1, F_2)$ in $O(1)$ time by brute force. Thus, we further assume that r has more than 6 leaf descendants in F_1. Now, our algorithm finds a *promising* vertex z in F_2, next detaches F_2^z, further makes a recursive call on the modified (F_1, F_2), and finally returns $c + 1$, where c is the value returned by the recursive call.

It remains to consider how to find a promising z. In the following two cases, we know an optimal choice of z, i.e., we know that the choice of z will lead to an optimal solution [6]:

- **Optimal Case 1:** *(u, v) is an active sibling-pair in F_1 with $|D_{F_2}(u, v)| = 1$.* In this case, z is the unique vertex in $D_{F_2}(u, v)$.
- **Optimal Case 2:** *(u, v) is an active sibling-pair in F_2 with $|D_{F_1}(u, v)| = 1$ and the unique vertex in $D_{F_1}(u, v)$ is a leaf.* In this case, z is the mate of the unique vertex in $D_{F_1}(u, v)$.

We hereafter assume that none of the above optimal cases occurs. Next, we outline how to find a promising z with MCTS. The idea behind MCTS is to build a small-sized search tree Γ. We will always use ρ to denote the root of Γ. In our case, each node α of Γ holds the following information:

- $f(\alpha)$: A sub-FF pair of (F_1, F_2). (*Comment:* We use $f(\alpha)_1$ and $f(\alpha)_2$ to denote the first and the second forest in $f(\alpha)$, respectively.)
- $t(\alpha)$: The *number of times* α has been visited so far.
- $s(\alpha)$: The *score* of α.
- $Q(\alpha)$: the *reward* α has received so far.

When creating a node α, we are always given a sub-FF pair $(\hat{F}_1, \hat{F}_2)$ of (F_1, F_2) and initialize $f(\alpha) = (\hat{F}_1, \hat{F}_2)$, $t(\alpha) = 0$, $s(\alpha) = 0$, and $Q(\alpha) = 0$. To evaluate a child α of a node β of Γ, we use the *UCT value* of α, which is computed as follows:

$$\frac{Q(\alpha)}{t(\alpha)} + C \cdot \sqrt{\frac{2 \ln t(\beta)}{t(\alpha)}},$$

where C is a constant (called the *balance constant* and fixed to be 0.2 in our experiments). The *best child* of a node β in Γ is the child of β in Γ whose UCT value is maximized over all children of β in Γ.

Initially, Γ has a unique node, namely, the root ρ created with (F_1, F_2). We then grow Γ by repeatedly performing the following steps (in this order) for a predetermined number (fixed to be 60 in our experiments) of repetitions:

1. Select a leaf-node α in Γ by starting at ρ and repeatedly descending to the best child of the current node until reaching a leaf. (*Comment:* Ties are broken arbitrarily.)
2. Expand α. (*Comment:* See Section 4.2.)
3. Perform a simulation for α by calling an approximation algorithm (say, the algorithm in [7]) on input $f(\alpha)$, and then update $s(\alpha)$ to $App(f(\alpha)) + |f(\alpha)_2| - |f(\rho)_2|$, where $App(f(\alpha))$ means the approximate rSPR distance of $f(\alpha)$ returned by the approximation algorithm. (*Comment:* We refer to this step as the *simulation step*.)
4. Compute the reward $Q(\alpha) = \begin{cases} 1 & \text{if } s(\alpha) \leq \text{the average score of the nodes in } \Gamma \\ 0 & \text{otherwise} \end{cases}$.
5. Backpropagate the reward $Q(\alpha)$ from α all the way to the root ρ by performing the following step for all ancestors β of α in Γ:
   - Increase $t(\beta)$ by 1 and increase $Q(\beta)$ by $Q(\alpha)$,

Once finishing growing Γ as above, we select the best child γ of ρ. As will be detailed in Section 4.2, $f(\gamma)_2$ is obtained from $f(\rho)_2$ by detaching the subtrees rooted at the vertices of a set S. Finally, we set z to be an arbitrary vertex in S.

4.2 Expanding a Node α

Suppose that we have selected a leaf node α to expand. We first simplify $f(\alpha)$ and then search $f(\alpha)_1$ and $f(\alpha)_2$ for an active sibling-pair (u, v) *in the following order*:

Type 1: (u, v) is an active sibling-pair in $f(\alpha)_1$ with $|D_{f(\alpha)_2}(u, v)| = 1$.

Type 2: (u, v) is an active sibling-pair in $f(\alpha)_2$ such that $|D_{f(\alpha)_1}(u, v)| = 1$ and the unique vertex in $D_{f(\alpha)_1}(u, v)$ is a leaf

Type 3: (u, v) is an active sibling-pair in $f(\alpha)_1$ such that u and v belong to different connected components of $f(\alpha)_2$.

Type 4: (u, v) is an active sibling-pair in $f(\alpha)_1$ such that u and v belong to the same connected component of $f(\alpha)_2$ and $\ell_{f(\alpha)_2}(u, v)$ is a root of $f(\alpha)_2$.

Type 5: (u, v) is of none of the above types.

We emphasize that the smaller type of an active sibling-pair is, the more our algorithm prioritizes it.

If (u, v) is not found, we know that $f(\alpha)_2$ is an AF of $f(\alpha)$ and hence we have nothing to do with expanding α. Thus, we hereafter assume that (u, v) has been found. Then, we construct a family $\mathcal{F}$ of sets as follows.

- If (u, v) is of Type 1 (respectively, 2), then $\mathcal{F}$ consists of only $D_{f(\alpha)_2}(u, v)$ (respectively, $D_{f(\alpha)_1}(u, v)$).
- If (u, v) is of Type 3 or 4, then $\mathcal{F}$ consists of $\{u\}$ and $\{v\}$.
- If (u, v) is of Type 5, then $\mathcal{F}$ consists of $\{u\}$, $\{v\}$, and $D_{f(\alpha)_2}(u, v)$.

We now use $\mathcal{F}$ to create the children of α as follows. For each set $S \in \mathcal{F}$, we create a child β_S, where $f(\beta_S)_1 = f(\alpha)_1$ and $f(\beta_S)_2$ is obtained from $f(\alpha)_2$ by detaching the subtrees rooted at the vertices in S.

5 Solving HNC Approximately

We say that an approximation algorithm A for RDC is *useful* if given a TT-pair (T_1, T_2), A can not only output an approximate value d' of $d(T_1, T_2)$ but also output an AF F of (T_1, T_2) with $|F| = d'$. Our approximation algorithm given in Section 4 is useful and so are all known approximation algorithms for RDC. Using a useful approximation A for RDC, we can design an approximation algorithm for HNC, denoted by A_{hn}, as follows. Given a TT-pair (T_1, T_2), A_{hn} calls A to obtain an approximate value d' of $d(T_1, T_2)$ and an AF F of (T_1, T_2) with $|F| = d'$. If F is an AAF of (T_1, T_2), then d' is also an approximate value of $h(T_1, T_2)$ and hence A_{hn} returns d'. So, assume that F is a CAF of (T_1, T_2). Then, as in Section 2.3, we can transform F into an AAF of (T_1, T_2) by solving an ILP model. Thus, d' plus the optimal value of the objective function of the model gives us an approximate value of $h(T_1, T_2)$ and so A_{hn} returns it.

6 Experimental Results

To compare our new algorithms against the previous bests, we have implemented them in Java. In this section, we compare the real performance of our programs against that of the previous bests. In our experiments, we use a Linux (x64) desktop PC with Intel i7-4790 CPU (4.00GHz, 8 threads) and 32GB RAM. As the ILP solver, we use the IBM CPLEX which is freely available for academic research.

We define the *average approximation ratio* (AAR) of an approximation algorithm A (for RDC or HNC) as follows. For a given instance I, we use $A(I)$ (respectively, $B(I)$) to denote the value outputted by A (respectively, an exact algorithm) on input I; the *approximation ratio* of A for I, denoted by $r_A(I)$, is $\frac{A(I)}{B(I)}$. The AAR of A for a set $\mathcal{I}$ of instances is $\frac{\sum_{I \in \mathcal{I}} r_A(I)}{|I|}$.

As in previous studies [1, 3, 7, 10, 16, 17], we here generate simulated datasets randomly. More specifically, for a given pair (n, m) of parameters, we use the program of [3] to generate a dataset consisting of 120 TT-pairs, where each TT-pair is generated by first generating a random phylogenetic tree T_1 with n leaves and then obtaining another phylogenetic tree T_2 by applying m random rSPR operations on T_1. So, the rSPR distance of each pair (T_1, T_2) in the dataset is at most m, but the hybridization number of (T_1, T_2) may be larger than m. In our experiments stated below, we choose (n, m) from $\{(100, 50), (200, 80), (200, 100)\}$ and generate a dataset $\mathcal{I}(n, m)$ for each (n, m) in this set.

Table 1 Comparing the AARs of Approximation Algorithms for RDC.

Svv	CMW	CHN	CombApp	MCTS_CMW	MCTS_CHN	CombMCTS
1.41	1.133	1.135	1.104	1.03	1.03	1.019
1.391	1.141	1.127	1.108	1.048	1.044	1.031

The first and the second rows show the results for $\mathcal{I}(100, 50)$ and $\mathcal{I}(200, 100)$, respectively; Svv, CMW, and CHN mean the algorithm in [15], [8], and [7], respectively; MCTS_CMW and MCTS_CHN mean our MCTS algorithm with CMW and CHN used in the simulation step, respectively; CombApp (respectively, CombMCTS) means the algorithm which runs CMW and CHN (respectively, MCTS_CMW and MCTS_CHN) and outputs the better solution returned by them.

Table 2 Comparing the AARs of Approximation Algorithms for HNC.

Svv_{hn}	CMW_{hn}	CHN_{hn}	CombApp_{hn}	MCTS_CMW_{hn}	MCTS_CHN_{hn}	CombMCTS_{hn}
1.397	1.146	1.134	1.1	1.032	1.031	1.02
1.419	1.087	1.083	1.062	1.021	1.02	1.015

The first and the second rows show the results for $\mathcal{I}(100, 50)$ and $\mathcal{I}(200, 80)$, respectively.

6.1 Results on Approximating RDC

Since all programs used in our experiments for approximating RDC are fast, it is meaningless to compare them in terms of running time. So, we compare them in terms of their AARs. We use $\mathcal{I}(100, 50)$ and $\mathcal{I}(200, 100)$ in the experiment. Our experimental results are summarized in Table 1. From the table, we can see that MCTS is very helpful in improving the performance of approximation algorithms for RDC. In particular, our best algorithm (namely, CombMCTS) achieves a significantly better AAR than the previous best (namely, CHN). We did not test MCTS_Svv because the source code of Svv has not been made public.

6.2 Results on Approximating HNC

Since we want the exact hybridization number to be known, we use the two easiest datasets (namely, $\mathcal{I}(100, 50)$ and $\mathcal{I}(200, 80)$) in this experiment to compare the AARs of our approximation algorithms for HNC against the previous bests. Our experimental results are summarized in Table 1. From the table, we can see that MCTS is very helpful in improving the performance of approximation algorithms for HNC as well. In particular, our best algorithm (namely, CombMCTS_{hn}) achieves a much better AAR than the previous best (namely, Svv_{hn}). Indeed, our experimental results show that for about half the tested instances, CombMCTS_{hn} found optimal solutions.

6.3 Results on Computing HNC Exactly

To compare the speed of our new exact algorithm for HNC against the previous best (namely, UltraNet in [10]), we use $\mathcal{I}(100, 50)$ and $\mathcal{I}(200, 80)$ again. For each tested instance, we set a 1-hour time limit on the running time of each program. As the result, UltraNet fails to solve **1** (respectively, **16**) instances of $\mathcal{I}(100, 50)$ (respectively, $\mathcal{I}(200, 80)$), while our new program fails to solve none. With the failed instances excluded, the average running time of UltraNet is **54.46** (respectively, **323.86**) seconds for the first (respectively, second) dataset, while that

of our new program is only **0.66** (respectively, **0.86**) seconds. So, our new program is more than **82 times faster** than UltraNet and its superiority in speed over UltraNet becomes more significant for larger instances.

References

1. B. Albrecht, C. Scornavacca, A. Cenci, and D.H. Huson. Fast computation of minimum hybridization networks. *Bioinformatics*, 28(2):191–197, 2012.
2. M. Baroni, C. Semple, and M. Steel. Hybrids in real time. *Systematic Biology*, 55(1):46–56, 2006.
3. R.G. Beiko and N. Hamilton. Phylogenetic identification of lateral genetic transfer events. *BMC Evolutionary Biology*, 6(15):159–169, 2006.
4. M. Bordewich and C. Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics*, 8(4):409–423, 2005.
5. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.
6. Z.-Z. Chen, Y. Fan, and L. Wang. Faster exact computation of rSPR distance. *Journal of Combinatorial Optimization*, 29(3):605–635, 2015.
7. Z.-Z. Chen, Y. Harada, Y. Nakamura, and L. Wang. Faster exact computation of rSPR distance via better approximation. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, to appear.
8. Z.-Z. Chen, E. Machida, and L. Wang. An Approximation Algorithm for rSPR Distance. In *22nd International Computing and Combinatorics Conference, Ho Chi Minh City, Vietnam, August 2-4, 2016*, pages 468–479, 2016.
9. Z.-Z. Chen and L. Wang. Algorithms for reticulate networks of multiple phylogenetic trees. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 9(2):372–384, 2012.
10. Z.-Z. Chen and L. Wang. An ultrafast tool for minimum reticulate networks. *Journal of Computational Biology*, 20(1):38–41, 2013.
11. L. Collins, S. Linz, and C. Semple. Quantifying hybridization in realistic time. *J. of Comput. Biol.*, 18(10):1305–1318, 2011.
12. J. Hein, T. Jing, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Disc. Appl. Math.*, 71(1-3):153–169, 1996.
13. D.H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, 2010.
14. S. Kelk, L. van Iersel, N. Lekic, S. Linz, C. Scornavacca, and L. Stougie. Cycle killer...qu'est-ce que c'est? On the comparative approximability of hybridization number and directed feedback vertex set. *SIAM J. Discrete Math.*, 26(4):1635–1656, 2012.
15. F. Schalekamp, A. van Zuylen, and S. van der Ster. A Duality Based 2-Approximation Algorithm for Maximum Agreement Forest. In *43rd International Colloquium on Automata, Languages and Programming, Rome, Italy, July 11-15, 2016*, pages 70:1–70:14, 2016.
16. L. van Iersel, S. Kelk, N. Lekic, and C. Scornavacca. A practical approximation algorithm for solving massive instances of hybridization number for binary and nonbinary trees. *BMC Bioinformatics*, 15(127), 2014.
17. C. Whidden, R.G. Beiko, and N. Zeh. Fast FPT algorithms for computing rooted agreement forest: theory and experiments. In *International Symposium on Experimental Algorithms, Naples, Italy, May 20-22, 2010*, pages 141–153, 2010.
18. C. Whidden, R.G. Beiko, and N. Zeh. Fixed-parameter algorithms for maximum agreement forests. *SIAM J. Comput.*, 42(4):1431–1466, 2013.
19. C. Whidden and N. Zeh. A unifying view on approximation and FPT of agreement forests. In *9th International Workshop on Algorithms in Bioinformatics, Philadelphia, PA, USA, September 12-13, 2009*, pages 390–401, 2009.
20. Y. Wu. A practical method for exact computation of subtree prune and regraft distance. *Bioinformatics*, 25(2):190–196, 2009.