An $\Omega(n^2/\log n)$ Speed-Up of TBR Heuristics for the Gene-Duplication Problem

Mukul S. Bansal and Oliver Eulenstein

Abstract—The gene-duplication problem is to infer a species supertree from gene trees that are confounded by complex histories of gene duplications. This problem is NP-complete and thus requires efficient and effective heuristics. Existing heuristics perform a stepwise search of the tree space, where each step is guided by an exact solution to an instance of a local search problem. We improve on the time complexity of the local search problem by a factor of $n^2/\log n$, where n is the size of the resulting species supertree. Typically, several thousand instances of the local search problem are solved throughout a stepwise heuristic search. Hence, our improvement makes the gene-duplication problem much more tractable for large-scale phylogenetic analyses.

Index Terms—Computational phylogenetics, gene duplication, supertrees, local search.

1 Introduction

An abundance of potential information for phylogenetic analyses is provided by the rapidly increasing amount of available genomic sequence information. Most phylogenetic analyses combine genomic sequences, from presumably orthologous loci or loci whose homology is the result of speciation, into gene trees. These analyses largely have to neglect the vast amounts of sequence information in which gene duplication generates gene trees that differ from the actual species tree. Phylogenetic information from such gene trees can be utilized through a species tree obtained by solving the gene-duplication problem [1]. This problem is a *supertree problem*, that is, assembling from a set of gene trees a supertree that contains all species found in at least one of the input trees. The decision version of the gene-duplication problem is NP-complete [2]. Existing heuristics aimed at solving the gene-duplication problem search the space of all possible supertrees guided by a series of exact solutions to instances of a local search problem [3]. The gene-duplication problem has shown much potential for building phylogenetic species trees for snakes [4], vertebrates [5], [6], *Drosophia* [7], and plants [8]. Yet, the computation time of local search problems that are solved by existing heuristics has largely limited the size of such studies.

Throughout the current section, n denotes the number of leaves in the resulting species tree, and for brevity in stating time complexities, gene trees and the resulting species tree are assumed to have similar sizes.

We improve on the best existing (naive) solution for a particular local search problem, the Tree Bisection and Reconnection (TBR)[1] local search problem, by a factor of $n^2/\log n$. Heuristics solving the TBR local search problem, TBR heuristics, were rarely applied in practice due to inefficient runtimes. Our method greatly improves the speed of TBR-based heuristics for the gene-duplication problem and makes it possible to infer larger supertrees that were previously difficult, if not impossible, to compute.

For convenience, we use the term "tree" to refer to a rooted and fully binary tree. The terms "leaf-gene" and "leaf-species" refer to a gene or species that is represented by a leaf of a gene or species tree, respectively, throughout this work unless otherwise stated.

Previous results. The gene-duplication problem is based on the Gene-Duplication (GD) model of Goodman et al. [12]. In the following, we 1) describe the GD model, 2) formulate the gene-duplication problem, and 3) describe a heuristic approach of choice [3] to solve the gene-duplication problem.

GD model. The GD model [1], [13], [14], [15], [16], [17], [18], [19] explains incompatibilities between a pair of "comparable" gene and species trees through gene duplications. A gene tree is *comparable* with a species tree if a sample mapping, called *leaf-mapping*, exists that maps every leaf-gene to the leaf-species from which it was sampled. Fig. 1a depicts an example. Gene tree G is inferred from the leaf-genes that were sampled from the leaf-species of the species tree described by the leaf-mapping. However, both trees describe incompatible evolutionary histories. The GD model explains such incompatibilities by reconciling the gene tree with postulated gene duplications. For example, in Fig. 1b, a reconciled gene tree R can be theoretically inferred from the species tree S by duplicating a gene x in species X into the copies x' and x'' and letting both copies speciate according to the topology of S. In this case, the gene tree can be embedded into the reconciled tree. Thus, the gene tree can be reconciled by using the duplication of gene x to explain the incompatibility. The gene duplications that are necessary under the GD model to reconcile the gene tree can be described by the mapping $\mathcal{M}$, which is an extension of the given leaf-mapping. The mapping $\mathcal{M}$ maps every gene in the gene tree to the most recent species in the species tree that could have contained the gene. For example, in Fig. 1, the most recent species that could have

1. TBR is a standard tree edit operation; see, for example, [9], [10], and [11].

• The authors are with the Department of Computer Science, Iowa State University, Ames, IA 50011. E-mail: {bansal, oeulenst}@cs.iastate.edu.

Manuscript received 9 Dec. 2007; revised 16 May 2008; accepted 25 June 2008; published online 7 July 2008.
For information on obtaining reprints of this article, please send e-mail to: tcbb@computer.org, and reference IEEECS Log Number TCBBSI-2007-12-0169.
Digital Object Identifier no. 10.1109/TCBB.2008.69.

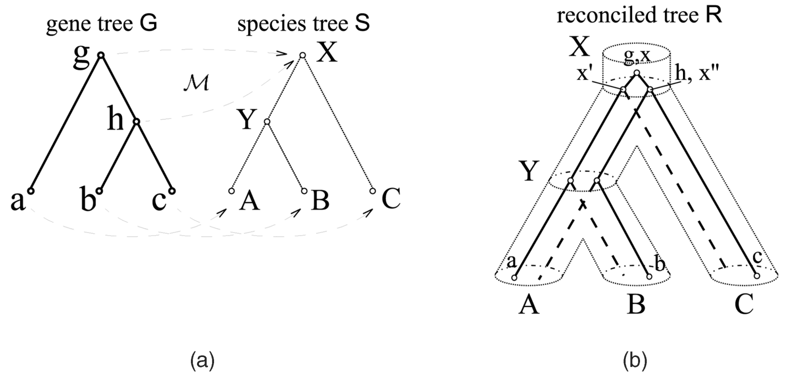

Fig. 1. (a) Gene tree G and species tree S are comparable, as the mapping from the leaf-genes to the leaf-species indicates. $\mathcal{M}$ is the LCA mapping from G to S. (b) R is the reconciled tree for G and S. In species X of R, gene x duplicates into the genes x' and x''. The solid lines in R represent the embedding of G into R.

contained the ancestral gene h is the ancestral species X. More precisely, $\mathcal{M}$ maps each gene to the least common ancestor (LCA) of the species from which the leaves (genes) of the subtree rooted at the gene were sampled (given by the leaf-mapping). A gene in the gene tree is a *(gene) duplication* if it has a child with the same mapping under $\mathcal{M}$. In Fig. 1, gene g and its child h map under the mapping $\mathcal{M}$ to the same species X. The *reconciliation cost* for a gene tree and a comparable species tree is measured in the number of duplications in the gene tree induced by the species tree. The *reconciliation cost* for a given set of gene trees and a species tree is the sum of the reconciliation costs for every gene tree in the set and the species tree. The reconciliation cost is linear-time computable [16], [20], [21].

Gene-duplication problem and heuristic. The *gene-duplication problem* is to find, for a given set of gene trees, a comparable species tree with the minimum reconciliation cost. The decision variant of this problem and some of its characterizations are NP-complete [2], [22], while some parameterizations are fixed-parameter tractable [23], [24]. However, GeneTree [3], an implementation of a standard local search heuristic for the gene-duplication problem, was used to show that the gene-duplication problem can be an effective approach for phylogenetic inference [5], [6]. Therefore, in practice, heuristics are commonly applied to solve the gene-duplication problem, even if they are unable to guarantee an optimal solution. While the local search heuristic for the gene-duplication problem performs reasonably well in computing smaller sized instances, it does not allow the computation of larger species supertrees. In this heuristic, a graph is defined for the given set of gene trees and some, typically symmetric, tree edit operation. The nodes in this graph are the species trees, which are comparable with every given gene tree. An edge adjoins two nodes exactly if the corresponding trees can be transformed into each other by the tree edit operation. The *reconciliation cost of a node* in the graph is the reconciliation cost of the species tree represented by that node and the given gene trees. Given a starting node in the graph, the heuristic's task is to find a maximal-length path of steepest descent in the reconciliation cost of its nodes and to return the last node on such a path. This path is found by solving the local search problem for every node along the path. The *local search* problem is to find a node with the minimum reconciliation cost in the neighborhood (all adjacent nodes) of a given node. The neighborhood searched depends on the edit operation. Edit operations of interest are rooted subtree pruning and regrafting (SPR) [9], [10], [25] and rooted TBR [9], [10], [11]. We defer the definition of these operations to Section 2. The best known runtimes for the SPR and TBR local search problems are $O(kn^2)$ [26] and $O(kn^4)$ (naive solution), respectively, where k is the number of gene trees.

Our contribution. The efficient solution for the SPR local search problem makes SPR-based heuristics suitable for large-scale phylogenetic analyses. Currently, TBR-based heuristics are not applicable for phylogenetic analyses because no usably efficient solution is known for the TBR local search problem. However, TBR-based heuristics are more desirable because they significantly extend the search space explored at each local search step. In particular, TBR heuristics search a neighborhood of $\Theta(n^3)$ nodes, including the $\Theta(n^2)$ nodes of the SPR neighborhood, at each local search step. Our contribution is an $O(kn^2 \log n)$ algorithm for the TBR local search problem. This makes TBR heuristics almost as efficient as SPR heuristics for large-scale phylogenetic analyses.

2 Basic Definitions, Notation, and Preliminaries

In this section, we first introduce basic definitions and notation and then define preliminaries required for this work.

2.1 Basic Definitions and Notation

A *tree* T is a connected graph with no cycles, consisting of a node set $V(T)$ and an edge set $E(T)$. The nodes in $V(T)$ of degree at most one are called *leaves* and denoted by $\mathrm{Le}(T)$. A node in $V(T)$ that is not a leaf is called an *internal* node. T is *rooted* if it has exactly one distinguished node called the *root*, which we denote by $\mathrm{Ro}(T)$. Let T be a rooted tree. For any pair of nodes x, $y \in V(T)$, where y is on a path from $\mathrm{Ro}(T)$ to x, we call 1) y an *ancestor* of x and 2) x a *descendant* of y. If $\{y, x\} \in E(T)$, then we call y the *parent* of x denoted by $\mathrm{Pa}(x)$, and we call x a *child* of y. We write (y, x) to denote the edge $\{y, x\}$, where $y = \mathrm{Pa}(x)$. The set of all children of y is denoted by $\mathrm{Ch}(y)$. If two nodes in T have the same parent, they are called *siblings*. T is (fully) *binary* if every internal node has exactly two children. A *subtree* of T rooted at node $x \in V(T)$, denoted by T_x, is the tree induced by x

and all its descendants. The *depth* of a node $x \in V(T)$ is the number of edges on the path from $\mathrm{Ro}(T)$ to x. The *LCA* of a nonempty subset $L \subseteq \mathrm{V}(T)$, denoted as $\mathrm{lca}(L)$, is the common ancestor of all nodes in L with maximum depth.

2.2 The Gene-Duplication Problem

We now introduce the necessary definitions to state the gene-duplication problem. A *species tree* is a tree that depicts the evolutionary relationships of a set of species. Given a gene family[2] for a set of species, a *gene tree* is a tree that depicts the evolutionary relationships among the sequences encoding only that gene family in the given species. Thus, the nodes in a gene tree represent genes. In order to compare a gene tree G with a species tree S, a mapping from each gene $g \in V(G)$ to the most recent species in S that could have contained g is required.

Definition 2.1 (mapping). *The* leaf-mapping $\mathcal{L}_{G,S} : \mathrm{Le}(G) \to \mathrm{Le}(S)$ *specifies the species $\mathcal{M}_{G,S}(g)$ from which gene g was sampled. An extension of $\mathcal{L}_{G,S}$ to $\mathcal{M}_{G,S} : V(G) \to V(S)$ is the* mapping *defined by $\mathcal{M}_{G,S}(g) = \mathrm{lca}(\mathcal{L}_{G,S}(\mathrm{Le}(G_g)))$.*

Note that related to the definition above, we write $\mathcal{M}^{-1}_{G,S}(g)$ to denote the set of nodes in G that map to node $g \in V(S)$ under the mapping $\mathcal{M}_{G,S}$.

Definition 2.2 (comparability). *Given trees G and S, we say that G is* comparable *to S if there exists a leaf-mapping $\mathcal{L}_{G,S}$.*[3] *A set of gene trees $\mathcal{G}$ is* comparable *to S if each gene tree in $\mathcal{G}$ is comparable with S.*

Let G be comparable to S for the remainder of this section.

Definition 2.3 (duplication). *A node $v \in V(G)$ is a* (gene) duplication *if $\mathcal{M}_{G,S}(v) = \mathcal{M}_{G,S}(u)$ for some $u \in \mathrm{Ch}(v)$, and we define $\mathrm{Dup}(G,S) = \{g \in V(G) : g \text{ is a duplication}\}$.*

Definition 2.4 (reconciliation cost). *We define reconciliation costs for gene and species trees as follows:*

1. $\Delta(G,S) = |\mathrm{Dup}(G,S)|$ *is the* reconciliation cost from G to S.
2. $\Delta(\mathcal{G},S) = \sum_{G\in\mathcal{G}} \Delta(G,S)$ *is the* reconciliation cost *from $\mathcal{G}$ to S.*
3. *Let $\mathcal{T} = \{S : \mathcal{G} \text{ is comparable with } S\}$. We define $\Delta(\mathcal{G}) = \min_{S\in\mathcal{T}} \Delta(\mathcal{G},S)$ to be the* reconciliation cost *of $\mathcal{G}$.*

Problem 1 (duplication).

Instance: A set $\mathcal{G}$ of gene trees.

Find: A species tree S^* such that $\Delta(\mathcal{G},S^*) = \Delta(\mathcal{G})$.

2.3 Local Search Problems

Here, we first provide definitions for the rerooting operation (denoted RR) and the TBR [11] and SPR [25] edit operations and then formulate the related local search problems that were motivated in the Introduction.

Definition 2.5 (RR operation). *Let T be a tree and $x \in V(T)$. $\mathrm{RR}(T,x)$ is defined to be the tree T if $x = \mathrm{Ro}(T)$. Otherwise, $\mathrm{RR}(T,x)$ is the tree that is obtained from T by 1) suppressing $\mathrm{Ro}(T)$ and 2) subdividing the edge $\{\mathrm{Pa}(x), x\}$ by a new root node. We define the following extension: $\mathrm{RR}(T) = \bigcup_{x\in V(T)}\{\mathrm{RR}(T,x)\}$.*

Definition 2.6 (TBR operation). *(See Fig. 2.) For technical reasons, we first define for a tree T the* planted tree *$\Phi(T)$, that is, the tree obtained by adding an additional edge, called* root edge, *$\{u, \mathrm{Ro}(T)\}$ to T.*

Let T be a tree, $e = (u,v) \in E(T)$, and X and Y be the connected components that are obtained by removing edge e from T, where $v \in X$, and $u \in Y$. We define $\mathrm{TBR}_T(v,x,y)$ for $x \in X$ and $y \in Y$ to be the tree that is obtained from $\Phi(T)$ by first removing edge e, then replacing the component X by $\mathrm{RR}(X,x)$, and then adjoining a new edge f between $x' = \mathrm{Ro}(\mathrm{RR}(X,x))$ and Y as follows:

1. *Create a new node y' that subdivides the edge $(\mathrm{Pa}(y), y)$.*
2. *Adjoin the edge f between nodes x' and y'.*
3. *Suppress the node u and rename x' as v and y' as u.*

We say that the tree $\mathrm{TBR}_T(v,x,y)$ is obtained from T by a **TBR** *operation that* bisects *the tree T into the components X and Y and* reconnects *them above the nodes x and y.*

We define the following notation:

1. $\mathrm{TBR}_T(v,x) = \bigcup_{y\in Y}\{\mathrm{TBR}_T(v,x,y)\}$.
2. $\mathrm{TBR}_T(v) = \bigcup_{x\in X} \mathrm{TBR}_T(v,x)$.
3. $\mathrm{TBR}_T = \bigcup_{(u,v)\in E(T)} \mathrm{TBR}_T(v)$.

An SPR operation for a given tree T can be briefly described through the following three steps: 1) prune some subtree P from T, 2) add a root edge to the remaining tree S, and 3) regraft P into an edge of the remaining tree S. For our purposes, we define the SPR operation as a special case of the TBR operation.

Definition 2.7 (SPR operation). *Let T be a tree, $e = (u,v) \in E(T)$, and X and Y be the connected components that are obtained by removing edge e from T, where $v \in X$ and $u \in Y$. We define $\mathrm{SPR}_T(v,y)$ for $y \in Y$ to be the tree $\mathrm{TBR}_T(v,v,y)$. We say that the tree $\mathrm{SPR}_T(v,y)$ is obtained from T by an* **SPR** *operation that* prunes *subtree T_v and* regrafts *it above node y.*

We define the following notation:

1. $\mathrm{SPR}_T(v) = \bigcup_{y\in Y}\{\mathrm{SPR}_T(v,y)\}$.
2. $\mathrm{SPR}_T = \bigcup_{(u,v)\in E(T)} \mathrm{SPR}_T(v)$.

Problem 2 (TBR-Scoring (TBR-S)).

Instance: A gene tree set $\mathcal{G}$ and a comparable species tree S.

Find: A tree $T^* \in \mathrm{TBR}_S$ such that

$$\Delta(\mathcal{G},T^*) = \min_{T\in\mathrm{TBR}_S} \Delta(\mathcal{G},T).$$

Problem 3 (TBR-Restricted Scoring (TBR-RS)).

Instance: A triple $(\mathcal{G},S,v)$, where $\mathcal{G}$ is a set of gene trees, S is a comparable species tree, and $(u,v) \in E(S)$.

Find: A tree $T^* \in \mathrm{TBR}_S(v)$ such that

$$\Delta(\mathcal{G},T^*) = \min_{T\in\mathrm{TBR}_S(v)} \Delta(\mathcal{G},T).$$

2. A *gene family* is a set of homologous genes assumed to have shared ancestry.

3. Note that mathematically speaking, such a leaf-mapping always exists. However, in the current context, we are only concerned with biologically relevant leaf-mappings.

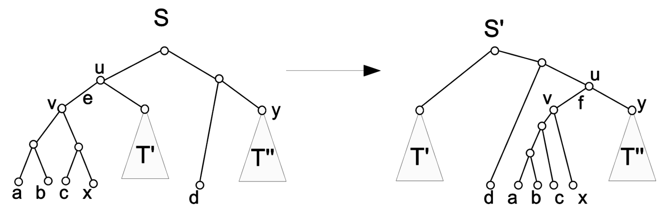

Fig. 2. Example depicting a TBR operation that transforms tree S into tree $S' = \mathrm{TBR}_S(v, x, y)$.

The problems SPR-*Scoring* (SPR-*S*) and SPR-*Restricted Scoring* (SPR-*RS*) are defined analogously as follows:

Problem 4 (SPR-Scoring (SPR-S)).

Instance: A gene tree set $\mathcal{G}$ and a comparable species tree S.

Find: A tree $T^* \in \mathrm{SPR}_S$ such that

$$\Delta(\mathcal{G}, T^*) = \min_{T \in \mathrm{SPR}_S} \Delta(\mathcal{G}, T).$$

Problem 5 (SPR-Restricted Scoring (SPR-RS)).

Instance: A triple $(\mathcal{G}, S, v)$, where $\mathcal{G}$ is a set of gene trees, S is a comparable species tree, and $(u, v) \in E(S)$.

Find: A tree $T^* \in \mathrm{SPR}_S(v)$ such that

$$\Delta(\mathcal{G}, T^*) = \min_{T \in \mathrm{SPR}_S(v)} \Delta(\mathcal{G}, T).$$

Throughout the rest of this paper, we use the following terminology:

1. $\mathcal{G}$ is a set of gene trees,
2. S denotes a compatible species tree,
3. $r = \mathrm{Ro}(S)$,
4. P denotes a proper (pruned) subtree of S, and
5. $v = \mathrm{Ro}(P)$.

3 Solving the TBR-S Problem

In this section, we study the TBR-S problem in more detail. First, we show how the algorithm developed by Bansal et al. [26] to solve the SPR-RS problem can be slightly modified to solve the TBR-S problem. This already improves the runtime of the existing naive solution considerably. Second, we show how the inherent structure of the TBR-S problem can be used to further improve the runtime. To do this, we define the "BestRooting" (BR) problem and show how an efficient solution for this problem leads to an efficient solution for the TBR-S problem.

Recall that in essence, a TBR operation involves pruning a subtree, say P, from S, rerooting P to form a tree P', and then regrafting P' onto S. We therefore introduce the following notation.

Notation. Given S and P, we define $\mathrm{S}(P')$ to be the tree obtained by 1) pruning P from S, 2) rerooting P to obtain $P' \in \mathrm{RR}(P)$, and 3) regrafting P' above node r.

3.1 Relating Scores of TBR and SPR Neighborhoods

The following algorithm Alg-SPR-RS is a brief restatement of the algorithm presented in [26] to solve the SPR-RS instance $(\mathcal{G}, S, v)$ efficiently.

Algorithm Alg-SPR-RS

1. Prune P from S and regraft P above node r to obtain $S(P)$. Compute the reconciliation cost of $S(P)$.
2. Compute the difference between the reconciliation cost of each tree in $\mathrm{SPR}_S(v)$ and $S(P)$. This gives the reconciliation cost of each tree in $\mathrm{SPR}_S(v)$.

Observe that $\mathrm{SPR}_S(v) = \mathrm{TBR}_S(v, v)$. In fact, Alg-SPR-RS can be modified to efficiently compute the reconciliation costs of all trees in $\mathrm{TBR}_S(v, x)$ for any node $x \in V(P)$. To do this, we simply modify step 1 of Alg-SPR-RS as follows:

1. Prune P from S, reroot P to obtain $P' = \mathrm{RR}(P, x)$, and regraft P' above node r to obtain $S(P')$. Compute the reconciliation cost of $S(P')$.

Note that this modification does not change the algorithms's complexity.

Observation 1. *The TBR-RS problem on $(\mathcal{G}, S, v)$ can be solved by computing the reconciliation cost of each tree in $\mathrm{TBR}_S(v, x)$, for all $x \in V(P)$. The TBR-S problem in turn can be solved by solving the TBR-RS problem $|V(S)| - 1$ times.*

Let us assume, for convenience, similar gene tree and species tree sizes. It is known that the SPR-RS problem is solvable in $O(kn)$ time [26], where $k = |\mathcal{G}|$. Based on Observation 1 and the modification described above, the TBR-S problem can then be solved in $O(kn^3)$ time. This already gives us a speed-up of $\Theta(n)$ over the best known (naive) solution for this problem. We will show how to solve the TBR-S problem in $O(kn^2 \log n)$ time. This gives a speed-up of $\Theta(n^2/\log n)$ over existing algorithms. Also, it should be noted that the correctness or efficiency of our algorithm does not depend on the simplifying assumption of similar gene and species tree sizes.

It is interesting to note that the size of the set TBR_S is $\Theta(n^3)$. Thus, for one gene tree, the time complexity of computing and enumerating the reconciliation costs of all trees in TBR_S is $\Omega(n^3)$.

However, to solve the TBR-S problem, one is only interested in finding a tree with the minimum reconciliation cost. This lets us solve the TBR-S problem in time that is sublinear in the size of TBR_S and obtain a time complexity of $O(n^2 \log n)$ for the TBR-S problem. In fact, after the initial $O(n^2 \log n)$ preprocessing step, our algorithm can output the reconciliation cost of any tree in TBR_S in $O(1)$ time.

3.2 Relating TBR-RS with SPR-RS

To obtain our speed-up, we concentrate on improving the complexity of solving the TBR-RS problem. To do this, we take a closer look at step 2 of Alg-SPR-RS. This part of the algorithm computes the difference in reconciliation cost of each tree in $\mathrm{SPR}_S(v)$ and the tree $S(P)$. To compute this difference, the algorithm considers only the leaf set of P and not its topology. This means that the difference values would be the same if P was replaced by any tree $P' \in \mathrm{RR}(P)$. Based on this observation, we have the following theorem.

Theorem 3.1. *Let $x', x'' \in V(P)$ and $y', y'' \in V(S) \setminus (V(P) \cup \{r\})$. Let*

$$T_1 = TBR_S(v, x', y'),\ T_2 = \mathrm{TBR}_S(v, x', y''),$$
$$T_3 = \mathrm{TBR}_S(v, x'', y'),\ \text{and } T_4 = \mathrm{TBR}(v, x'', y'').$$

Then, $\Delta(\mathcal{G}, T_1) - \Delta(\mathcal{G}, T_2) = \Delta(\mathcal{G}, T_3) - \Delta(\mathcal{G}, T_4)$.

Proof. Let $\delta'(y')$ and $\delta'(y'')$ represent the difference values computed by algorithm Alg-SPR-RS at nodes y' and y'', respectively, for regrafting subtree $P' = \mathrm{RR}(P, x')$. Similarly, let $\delta''(y')$ and $\delta''(y'')$ represent the difference values computed by Alg-SPR-RS at nodes y' and y'', respectively, for regrafting subtree $P' = \mathrm{RR}(P, x'')$. Then, by definition

$$\begin{aligned}
\delta'(y') &= \Delta(\mathcal{G}, \mathrm{TBR}_S(v, x', y')) - \Delta(\mathcal{G}, S(\mathrm{RR}(P, x'))),\\
\delta'(y'') &= \Delta(\mathcal{G}, \mathrm{TBR}_S(v, x', y'')) - \Delta(\mathcal{G}, S(\mathrm{RR}(P, x'))),\\
\delta''(y') &= \Delta(\mathcal{G}, \mathrm{TBR}_S(v, x'', y')) - \Delta(\mathcal{G}, S(\mathrm{RR}(P, x''))),\\
\delta''(y'') &= \Delta(\mathcal{G}, \mathrm{TBR}_S(v, x'', y'')) - \Delta(\mathcal{G}, S(\mathrm{RR}(P, x''))).
\end{aligned}$$

We also know that $\delta'(y') = \delta''(y')$ and $\delta'(y'') = \delta''(y'')$, which implies that $\delta'(y') - \delta'(y'') = \delta''(y') - \delta''(y'')$. Rewriting this using the equations above gives us the theorem. □

Based on this theorem and algorithm Alg-SPR-RS, we have the following corollary.

Corollary 3.1. *To obtain the reconciliation cost of each tree in $\mathrm{TBR}_S(v)$, it is sufficient to compute the reconciliation cost of $S(P')$ for each $P' \in \mathrm{RR}(P)$ and then perform step 2 of Alg-**SPR**-RS starting with any $S(P')$, where $P' \in \mathrm{RR}(P)$.*

This is because the output of step 2 of Alg-SPR-RS will be the same for all $S(P')$, where $P' \in \mathrm{RR}(P)$.

To solve the TBR-RS problem, it is sufficient to find one tree in $\mathrm{TBR}_S(v)$ with the minimum reconciliation cost. Based on Alg-SPR-RS and Corollary 3.1, we have the following theorem.

Theorem 3.2. *Let $T_1 = \mathrm{TBR}_S(v, x, y)$ be a tree with the minimum reconciliation cost in $\mathrm{TBR}_S(v)$. If $P' = \mathrm{RR}(P, x)$, then $S(P')$ must have the minimum reconciliation cost among all trees in $\bigcup_{P'' \in \mathrm{RR}(P)} S(P'')$.*

Proof. Let us suppose, for the sake of contradiction, that $S(P')$ does not have the minimum reconciliation cost among the trees in $\bigcup_{P'' \in \mathrm{RR}(P)} S(P'')$. Instead, let $x' \in V(P)$ be such that $S(\mathrm{RR}(P, x'))$ has the lowest reconciliation cost in $\bigcup_{P'' \in \mathrm{RR}(P)} S(P'')$. Then, Theorem 3.1 implies that the tree $\mathrm{TBR}(v, x', y)$ must have a lower reconciliation cost than tree T_1, which is a contradiction. Thus, $S(P')$ must have the minimum reconciliation cost among all trees in $\bigcup_{P'' \in \mathrm{RR}(P)} S(P'')$. □

In other words, to obtain a solution for the TBR-RS problem, for instance, $(\mathcal{G}, S, v)$, it is sufficient to obtain the reconciliation costs of only the trees in $\mathrm{TBR}_S(v, x)$, where $P' = \mathrm{RR}(P, x)$ such that $S(P')$ has the minimum reconciliation cost. Based on Corollary 3.1 and Theorem 3.2, we have the following corollary.

Corollary 3.2. *The minimum reconciliation cost of a tree in $\mathrm{TBR}_S(v)$ can be obtained by performing step 2 of Alg-**SPR**-RS starting with $S(P')$, where $P' \in \mathrm{RR}(P)$ such that $S(P')$ has the minimum reconciliation cost.*

Problem 6 (BR).

- *Instance: A set of gene trees $\mathcal{G}$, a compatible species tree S, and a proper subtree P of S.*
- *Find: A tree $P' \in \mathrm{RR}(P)$ for which $\Delta(\mathcal{G}, S(P'))$ is minimum.*

Thus, based on Observation 1, Theorems 3.1 and 3.2, and Corollaries 3.1 and 3.2, an efficient solution to the BR problem leads naturally to an efficient solution for the TBR-S problem. The remainder of this paper deals mostly with our solution to solve the BR problem efficiently. In the next section, we take a closer look at the BR problem and study some of its structural properties.

4 Structural Properties of the BR Problem

Our solution to solve the BR problem for a set of input gene trees involves computing the reconciliation cost of $S(P')$, where $P' \in \mathrm{RR}(P)$, for each gene tree separately and then combining the results to obtain the final solution. The solution for the BR problem is easily obtained by selecting that $P' \in \mathrm{RR}(P)$ for which the sum of the reconciliation costs from each gene tree is minimum. Therefore, in the remainder of this section, we assume that there is only one input gene tree G for the BR problem. Thus, the problem to be solved is the following:

Problem 7 (ROOTING).

- *Instance: A triple (G, S, P), where G is a gene tree, S is a compatible species tree, and P is a proper subtree of S.*
- *Find: The reconciliation cost $\Delta(G, S(P'))$ for each $P' \in \mathrm{RR}(P)$.*

Notation. In order to avoid redundancy and facilitate comprehension, throughout the remainder of this paper, we abbreviate the mapping $\mathcal{M}_{G,T}$ simply to $\mathcal{M}_T$, for any species tree T. In addition, if a node $g \in V(G)$ is a gene duplication under mapping $\mathcal{M}_T$ for some species tree T, then we say that $\mathrm{d}_T(g) = t$, and $\mathrm{d}_T(g) = f$ otherwise.

To solve the ROOTING problem, we first calculate the reconciliation cost of $S(P)$. As P is rerooted to form P', the duplication status of some of the nodes from G may change, which changes the reconciliation cost. We show how to efficiently compute this difference between the reconciliation cost of $S(P)$ and the reconciliation cost of $S(P')$ for each $P' \in \mathrm{RR}(P)$.

To realize this strategy, it is imperative to study the change in the duplication status of nodes in the gene tree as P is rerooted step by step. In particular, consider the mapping $\mathcal{M}_{S(P)}$. Under this mapping, some of the nodes from $V(G)$ are duplications, and some are not. Lemma 4.1 helps us identify those nodes in $V(G)$ that maintain their duplication status under mapping $\mathcal{M}_{S(P')}$ for every $P' \in \mathrm{RR}(P)$. For any other node, say, g, in $V(G)$, Lemmas 4.2-4.5 allow us to identify those $P' \in \mathrm{RR}(P)$ for which (under mapping $\mathcal{M}_{S(P')}$) 1) $\mathrm{d}_{S(P')}(g) = \mathrm{d}_{S(P)}(g)$ (Lemmas 4.2-4.5), 2) $\mathrm{d}_{S(P)}(g) = f$ but $\mathrm{d}_{S(P')}(g) = t$

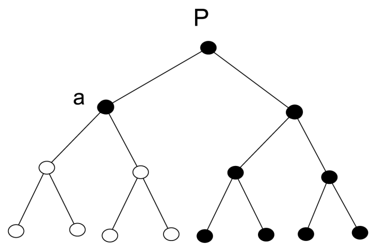

Fig. 3. Here, $a = \mathcal{M}_{S(P)}(g)$. According to Lemma 4.2, if x is one of the filled-in nodes of P and $P' = \mathrm{RR}(P, x)$, then $\mathrm{d}_{S(P')}(g) = \mathrm{d}_{S(P)}(g)$.

(Lemma 4.3), and 3) $\mathrm{d}_{S(P)}(g) = t$ but $\mathrm{d}_{S(P')}(g) = f$ (Lemma 4.5).

Lemma 4.1. *If $g \in V(G)$ such that $\mathcal{M}_S(g) \notin V(P)$, then $\mathrm{d}_{S(P')}(g) = \mathrm{d}_S(g)$ for any $P' \in \mathrm{RR}(P)$*

Proof. Observe that for any node $g \in G$ for which $\mathcal{M}_S(g) \notin V(P)$, the mapping $\mathcal{M}_{S(P')}(g)$ is the same for each $P' \in \mathrm{RR}(P)$. The lemma follows. □

Thus, under our strategy, we only need to consider those nodes in G that map to a node in $V(P)$ under $\mathcal{M}_S$. These are the nodes that are responsible for any difference in the reconciliation costs of $S(P)$ and $S(P')$, where $P' \in \mathrm{RR}(P)$.

Definition 4.1. *A node $g \in V(G)$ is* relevant *if $\mathcal{M}_S(g) \in V(P)$.*

Notation. For the remainder of this section, let $g \in V(G)$ be a relevant internal node and $\mathrm{Ch}(g) = \{g', g''\}$. In order to aid intuition and simplify presentation, we define A, B, and C to be the sets $V(P_a) \setminus \{a\}$, $V(P_b) \setminus \{b\}$, and $V(P_c) \setminus \{c\}$, respectively, where a, b, $c \in V(P)$.

Observation 2. *g' and g'' must be relevant.*

This implies that the duplication status of a relevant node depends only on the mappings of other relevant nodes. Based on this crucial observation, we have the following lemmas.

Lemma 4.2. *Let $a = \mathcal{M}_{S(P)}(g)$. If $P' = \mathrm{RR}(P, x)$ for $x \in V(P) \setminus A$, then $\mathrm{d}_{S(P')}(g) = \mathrm{d}_{S(P)}(g)$. (See Fig. 3 for an example.)*

Proof. If $P' = \mathrm{RR}(P, x)$ for $x \in V(P) \setminus A$, then the subtree of $S(P')$ rooted at a must be identical for each P'. This means that the node g and all its descendants would have the same mapping under $\mathcal{M}_{S(P')}$, for all $P' = \mathrm{RR}(P, x)$, where $x \in V(P) \setminus A$. Thus, the duplication status of g is preserved. □

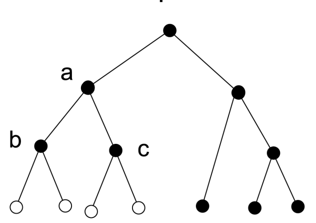

Fig. 4. Here, $a = \mathcal{M}_{S(P)}(g)$, $b = \mathcal{M}_{S(P)}(g')$, and $c = \mathcal{M}_{S(P)}(g'')$. If $P' = \mathrm{RR}(P, x)$, then according to Lemma 4.3, $\mathrm{d}_{S(P')}(g) = t$ if and only if x is not a filled-in node of P.

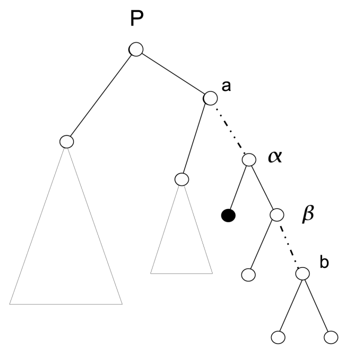

Fig. 5. Here, $a = \mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g')$, and $b = \mathcal{M}_{S(P)}(g'')$. α is the node nearest to b along the path between a and b in $S(P)$ such that there exists a node $v \in V(G_{g'})$ with $\mathcal{M}_{S(P)}(v) \in V(P_\alpha)$. The filled-in node is the node $\mathcal{M}_{S(P)}(v)$. In this example, according to Lemma 4.5, if $P' = \mathrm{RR}(P, x)$, then $\mathrm{d}_{S(P')}(g) = f$ if and only if $x \in V(P_\beta) \setminus B$.

Lemma 4.3. *Suppose $\mathrm{d}_{S(P)}(g) = f$ and let $b = \mathcal{M}_{S(P)}(g')$ and $c = \mathcal{M}_{S(P)}(g'')$. If $P' = \mathrm{RR}(P, x)$, then $\mathrm{d}_{S(P')}(g) = t$ if and only if $x \in B \cup C$. (See Fig. 4 for an example.)*

Proof. Let $a = \mathcal{M}_{S(P)}(g)$. We break up the problem into four distinct cases, which cover all possible values for x.

Case 1: $x \in V(P) \setminus A$. In this case, by Lemma 4.2, $\mathrm{d}_{S(P')}(g) = \mathrm{d}_{S(P)}(g)$. Thus, $\mathrm{d}_{S(P')}(g) = f$.

Case 2: $x \in A \setminus (B \cup C)$. The node g' must keep mapping to node b, and the node g'' must keep mapping to node c for all values of x in this case. Let $d = \mathcal{M}_{S(P')}(g)$, where $P' = \mathrm{RR}(P, x)$. Observe that d must be the LCA of nodes b and c in the tree $S(P')$. However, the subtrees rooted at b and c in $S(P')$ must be disjoint, i.e., they do not share any nodes. Thus, $d \neq b \neq c$, and hence, $\mathrm{d}_{S(P')}(g) = f$ in this case.

Case 3: $x \in B$. In this case, let $d = \mathcal{M}_{S(P')}(g')$, where $P' = \mathrm{RR}(P, x)$. Then, in the tree $S(P')$, the node d must be an ancestor of node b, and the node b is an ancestor of node c. Also, the node g'' must keep mapping to node c for all values of $x \in B$. This implies that $\mathcal{M}_{S(P')}(g) = d$, and hence, $\mathrm{d}_{S(P')}(g) = t$ in this case.

Case 4: $x \in C$. This case is symmetric to case 3 above. The lemma follows: □

Lemma 4.4. *If $\mathcal{M}_{S(P')}(g) = \mathcal{M}_{S(P')}(g') = \mathcal{M}_{S(P')}(g'')$ for some $P' \in \mathrm{RR}(P)$, then $\mathrm{d}_{S(P'')}(g) = t$ for every $P'' \in \mathrm{RR}(P)$.*

Proof. We will show that under every mapping $\mathcal{M}_{S(P'')}$ for every $P'' \in \mathrm{RR}(P)$, at least one of g' or g'' maps to $\mathcal{M}_{S(P'')}(g)$. Suppose, for contradiction, that there exists a $P''' \in \mathrm{RR}(P)$ such that $\mathrm{d}_{S(P''')}(g) = f$. Let $b = \mathcal{M}_{S(P''')}(g')$ and $c = \mathcal{M}_{S(P''')}(g'')$. Then, by Lemma 4.3, we know that $\mathrm{d}_{S(P'')}(g) = t$ if and only if $P'' = \mathrm{RR}(P''', x)$ for $x \in B \cup C$.

However, if $x \in B$, then for each $P'' = \mathrm{RR}(P''', x)$, we must have $c = \mathcal{M}_{S(P'')}(g'') \neq \mathcal{M}_{S(P'')}(g)$. Similarly, if $x \in C$, then for each $P'' = \mathrm{RR}(P''', x)$, we must have $\mathcal{M}_{S(P'')}(g') = b \neq \mathcal{M}_{S(P'')}(g)$. This implies that there could not be any $P' = \mathrm{RR}(P)$ (since $\mathrm{RR}(P) = \mathrm{RR}(P''')$) such that $\mathcal{M}_{S(P')}(g) = \mathcal{M}_{S(P')}(g') = \mathcal{M}_{S(P')}(g'')$. This is a contradiction. Thus, $\mathrm{d}_{S(P'')}(g) = t$ for every $P'' \in \mathrm{RR}(P)$. □

Lemma 4.5. *Let $a = \mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g')$ and $b = \mathcal{M}_{S(P)}(g'')$. Let α denote the node nearest to b along the path between a and b in $S(P)$ such that there exists a node $v \in V(G_{g'})$ with $\mathcal{M}_{S(P)}(v) \in V(P_\alpha)$. If $\alpha \neq b$, then let β be the* *child of α that lies along the path from α to b. Then, we have the following:*

1. *If $\alpha = b$, then $\mathrm{d}_{S(P')}(g) = t$ for each $P' \in \mathrm{RR}(P)$.*
2. *Otherwise, given that $P' = \mathrm{RR}(P, x)$, $\mathrm{d}_{S(P')}(g) = f$ if and only if $x \in V(P_\beta) \setminus B$.*

(See Fig. 5 for an example.)

Proof. If $\alpha = b$, then there is a child b' of b such that there exists a node $v \in V(G_{g'})$ with $\mathcal{M}_{S(P)}(v) \in V(P_{b'})$. Consider the tree $S(P'')$, where $P'' = \mathrm{RR}(P, b')$. Under mapping $\mathcal{M}_{S(P'')}$, the nodes g' and g'' must both map to the root node of $S(P'')$. Hence, node g must map to the root node as well. By Lemma 4.4, $\mathrm{d}_{S(P')}(g) = t$ for each $P' \in \mathrm{RR}(P)$.

If $\alpha \neq b$ and $P' = \mathrm{RR}(P, x)$, we break up the problem into three distinct cases, which cover all possible values for x.

Case 1. $x \in B$. Let $d = \mathcal{M}_{S(P')}(g'')$. Observe that for $x \in V(P_b)$, $\mathcal{M}_{S(P')}(g')$ must be the node α. Now, in the tree $S(P')$, the node d must be an ancestor of node b, which in turn is a proper ancestor of node α. This implies that $\mathcal{M}_{S(P')}(g) = d$, and hence, $\mathrm{d}_{S(P')}(g) = t$ in this case.

Case 2. $x \in V(P_\beta) \setminus B$. The node g' must map to node α, and the node g'' will keep mapping to node b, for all values of x in this case. Let $d = \mathcal{M}_{S(P')}(g)$. Observe that d must be the LCA of nodes α and b in the tree $S(P')$. However, the subtrees rooted at b and c in $S(P')$ must be disjoint, i.e., they do not share any nodes. Thus, in $S(P')$, $d \neq \alpha \neq b$, and hence, $\mathrm{d}_{S(P')}(g) = f$ in this case.

Case 3. $x \in V(P) \setminus V(P_\beta)$. Let $d = \mathcal{M}_{S(P')}(g')$. Observe that for $x \in V(P) \setminus V(P_\beta)$, $\mathcal{M}_{S(P')}(g'') = b$. In the tree $S(P')$, the node d must be an ancestor of node α, which in turn is a proper ancestor of node b. This implies that $\mathcal{M}_{S(P')}(g) = d$, and hence, $\mathrm{d}_{S(P')}(g) = t$ in this case.

The lemma follows. □

Based on these lemmas, we now give an efficient algorithm to solve the ROOTING problem.

5 Description of the Algorithm

We first design an efficient algorithm, called Alg-RCT (see Algorithm 1), which solves the ROOTING problem. Based on the lemmas seen in Section 3, we then show how this algorithm fits into our algorithm for solving the TBR-S problem. Finally, we prove the correctness of our algorithm and analyze its complexity.

In our algorithm to solve the ROOTING problem, we compute two values, *gain* and *loss*, at each node in P. For a node $s \in V(P)$, the value of *gain* at s, denoted by *gain*(s), represents the number of additional nodes from G that will become duplications when $P' = \mathrm{RR}(P, s)$ is rerooted to form $P'' = \mathrm{RR}(P, t)$, for both $t \in \mathrm{Ch}(s)$. Similarly, the value *loss*(s) represents the number of nodes from G that will lose their duplication status when $P' = \mathrm{RR}(S, \mathrm{Pa}(s))$ is rerooted to form $P'' = \mathrm{RR}(S, s)$.

The overall strategy of this algorithm is to consider the nodes in G one at a time and update the relevant *gain* and *loss* values on P for that node. For those nodes in G that satisfy the preconditions of one of Lemmas 4.1-4.4, this is relatively straightforward. But the nodes in G satisfying the precondition of Lemma 4.5 are harder to handle efficiently. Updating the relevant *gain* and *loss* values for such nodes makes use of the procedure IntervalTree, described in Algorithm 2.

A fairly detailed overview of algorithm Alg-RCT (which makes use of algorithm IntervalTree) now follows. For a more detailed description of this algorithm, the reader is encouraged to refer to Algorithms 1 and 2.

Algorithm 1. Alg-RCT

```
1: Input: A gene tree $G$, species tree $S$, and the pruned
   subtree $P$.
2: Output: The reconciliation cost $\Delta(G, S(P'))$ for each
   $P' \in \mathrm{RR}(P)$
3: Create two counters gain and loss at each node in $P$.
   They are all set to 0 initially.
4: Construct the mapping $\mathcal{M}_{S(P)}$ and let $\delta \leftarrow \Delta(G, S(P))$.
5: Traverse through $G$ and "mark" all those nodes that map
   to a node in $P$ under mapping $\mathcal{M}_{S(P)}$. Also record the
   gene duplication status of each marked node.
6: for each marked internal node $g \in V(G)$ do
7:     Let $\mathrm{Ch}(g) = \{g', g''\}$.
8:     if $\mathrm{d}_{S(P)}(g) == f$ then
9:         $gain(\mathcal{M}_{S(P)}(g')) \leftarrow gain(\mathcal{M}_{S(P)}(g')) + 1$.
10:        $gain(\mathcal{M}_{S(P)}(g'')) \leftarrow gain(\mathcal{M}_{S(P)}(g'')) + 1$.
11:    else
12:        if $\mathcal{M}_{S(P)}(g) == \mathcal{M}_{S(P)}(g')$ AND $\mathcal{M}_{S(P)}(g) ==$
           $\mathcal{M}_{S(P)}(g'')$ then
13:            Continue with the next iteration of the "for"
               loop at Line 6.
14:    else
15:        Let $\mathcal{M}_{S(P)}(g) == \mathcal{M}_{S(P)}(g')$. (Note: In this else
           statement, only one of $g'$ or $g''$ may map to the
           same node as $g$ in $P$. W.l.o.g., we assume that
           this node is $g'$.)
16:        Let $a = \mathcal{M}_{S(P)}(g)$, and $\mathrm{Ch}_P(a) = \{b, c\}$.
17:        W.l.o.g., assume that $\mathcal{M}_{S(P)}(g'') \in V(P_c)$.
18:        Mark node $g'$ as "Special."
19: Call INTERVALTREE$(G, P)$.
20: Set a weight $W$ initialized to 0 for each node in $P$.
21: for each node $s$ in a preorder traversal of $P$ do
22:     if If $s = \mathrm{Ro}(P)$ then
23:         $W(s) \leftarrow 0$.
24:     else
25:         if $s \in \mathrm{Ch}(\mathrm{Ro}(P))$ then
26:             $W(s) \leftarrow \delta$.
27:     else
28:         $W(s) \leftarrow W(\mathrm{Pa}(s)) + gain(\mathrm{Pa}(s)) - loss(s)$
```

Algorithm 2. INTERVALTREE

```
1: Input: $G$, $P$ (We use the same notation as in Alg-RCT).
2: Label $G$ according to an in-order tour.
3: Create an empty set called "End" at each node in $P$.
4: for each Special node $u$ of $G$ do
5:     Let $v$ denote the sibling of $u$ in $G$. Add $u$ to the End
       set of $\mathcal{M}_{S(P)}(v)$.
6: Create an empty Interval tree.
7: for each node $x$ in an Euler tour of $P$ do
8:     During the tree traversal, if we are moving into
```

```
        subtree $P_x$, then add the nodes in the End set of $x$ to
        the interval tree, and if we are moving out of subtree
        $P_x$, then remove the nodes in the End set of $x$ from
        the interval tree.
9:      if $x$ is a leaf node then
10:         for each node $y \in \mathcal{M}^{-1}_{S(P)}(x)$ do
11:             Perform a stabbing query on the interval tree
                with $y$. Store the result in a set called
                “TempSet.”
12:             for each node $u$ in TempSet do
13:                 Remove the Special status for node $u$ and
                    delete it from the interval tree.
14: Delete all the End sets associated with nodes in $P$. Also,
    delete the interval tree.
15: Create two empty sets called “Start” and “End” at each
    node in $P$.
16: for each Special node $u$ of $G$ do
17:     Add $u$ to the Start set of $a$ in $P$. And add $u$ to the End
        set of $\mathcal{M}_{S(P)}(v)$ in $P$).
18: Create an empty Interval tree.
19: for each node $x$ in a left-child-first postorder traversal
    of $P$ do
20:     Add the nodes in the End set of $x$ to the interval tree.
        Remove the nodes in the Start set of $x$ from the
        interval tree.
21:     if $x$ is a leaf node then
22:         for each node $y \in \mathcal{M}^{-1}_{S(P)}(x)$ do
23:             Perform a stabbing query on the interval tree
                with $y$. Store the result in a set called
                “TempSet.”
24:             for each node $u$ in Tempset do
25:                 Let $\alpha(u) \leftarrow lca(\{x, \mathcal{M}_{S(P)}(v)\})$, where $v$ is
                    the sibling of the node $u$.
26:                 Delete the interval for node $u$ from the
                    interval tree.
27: for each node $x$ in a right-child-first postorder traversal
    of $P$ do
28:     Add the nodes in the End set of $x$ to the interval tree.
        Remove the nodes in the Start set of $x$ from the
        interval tree.
29:     if $x$ is a leaf node then
30:         for each node $y \in \mathcal{M}^{-1}_{S(P)}(x)$ do
31:             Perform a stabbing query on the interval tree
                with $y$. Store the result in a set called
                “TempSet.”
32:             for each node $u$ in Tempset do
33:                 Let $\beta(u) \leftarrow lca(\{x, \mathcal{M}_{S(P)}(v)\})$, where $v$ is
                    the sibling of the node $u$.
34:                 Delete the interval for node $u$ from the
                    interval tree.
35: for each Special node $u$ of $G$ do
36:     Let $\gamma$ be that node among $\alpha(u)$ or $\beta(u)$ that has the
        greatest depth.
37:     if $\gamma \neq a$ AND $\gamma \neq gain(\mathcal{M}_{S(P)}(v))$ then
38:         Let $\gamma'$ be the child of $\gamma$ in $P$ that lies along the path
            from $a$ to $\mathcal{M}_{S(P)}(v)$). Set $loss(\gamma') \leftarrow loss(\gamma') + 1$,
            and $gain(\mathcal{M}_{S(P)}(v)) \leftarrow gain(\mathcal{M}_{S(P)}(v)) + 1$.
```

5.1 Algorithm Alg-RCT

The input for Alg-RCT is the instance (G, S, P) of the ROOTING problem.

The output is a $W : V(P) \rightarrow \mathbb{N}_0$ node weighted version of tree P, where $W(s) = \Delta(G, S(P'))$ for $P' = \mathrm{RR}(P, s)$.

1. *Initialization.* Construct $S(P)$ and initialize two counters $gain(s)$ and $loss(s)$ with 0 for each node $s \in V(P)$. Then, compute $\mathcal{M}_{S(P)}$. Create two empty sets “Start” and “End” at each node in P.
2. *Partially updating the values for gain and loss.* For each relevant node $g \in V(G)$ do the following: If $\mathrm{d}_{S(P)}(g) = f$, then $gain(\mathcal{M}_{S(P)}(x)) \leftarrow gain(\mathcal{M}_{S(P)}(x)) + 1$ for each $x \in \mathrm{Ch}(g)$. If $\mathrm{d}_{S(P)}(g) = t$, where $a = \mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g')$, and $\mathcal{M}_{S(P)}(g'') \neq a$, for $\mathrm{Ch}(g) = \{g', g''\}$, add g' to the Start set of node a and the End set of node $\mathcal{M}_{S(P)}(g'')$.
3. *Fully updating the values for gain and loss.* We now update the loss and gain values for those nodes that satisfy the condition of Lemma 4.5. These nodes are marked as “Special.” Following the notation from Lemma 4.5, the goal is to find node $\alpha \in P$ for each Special node from G. For convenience, here, we only give a high-level description of the algorithm to be followed for this step, and the reader is referred to Algorithm 2 for a detailed description. An in-order labeling of G lets us store the subtree G_g for any Special node $g \in V(G)$ as an interval. These intervals can be stored in an interval tree so that stabbing queries can be performed efficiently. We first traverse the tree P in an Euler tour order and locate those nodes in G that satisfy case 1 of Lemma 4.5. Such nodes need not be considered any further, and hence, their Special status is removed. Next, we traverse P in postorder, and for each node, say, x, we keep track of those nodes from the gene tree that might have a descendant mapping to x and for which α can be deduced from x. This is done by making use of the Start and End sets established in the previous step. This “currently active” set of nodes (intervals) is maintained dynamically in the interval tree. Performing stabbing queries on the interval tree allows us to obtain those Special nodes for which the α nodes can be deduced easily from x. This postorder traversal is actually broken into two separate traversals: a left-child-first postorder traversal and a right-child-first postorder traversal. This is required in order to be able to find the node α for each Special node.
4. *Computing the reconciliation costs.* The tree P is initialized to be P, and its node weights are set to 0. Set $\delta \leftarrow \Delta(G, S(P))$. For each node s in a preorder traversal on the tree P, we calculate the weight of that node as follows: If $s \in \mathrm{Ch}(\mathrm{Ro}(P))$, then $W(s) \leftarrow \delta$. Otherwise, set $W(s) \leftarrow W(\mathrm{Pa}(s)) + gain(\mathrm{Pa}(s)) - loss(s)$.

Next, we prove the correctness of algorithm Alg-RCT and analyze its complexity.

Lemma 5.1. *Consider a node $g \in V(G)$ and a node $s \in V(P)$. Let $\mathrm{Ch}_P(s) = \{s', s''\}$. If $\mathrm{d}_{S(P')}(g) = f$ and $\mathrm{d}_{S(P'')}(g) = t$, where $P' = \mathrm{RR}(P, s)$ and $P'' = \mathrm{RR}(P, s')$, then we must have $\mathrm{d}_{S(P''')}(g) = t$, where $P''' = \mathrm{RR}(P, s'')$.*

Proof. This follows from the statement of Lemma 4.3. □

Recall that the output of algorithm Alg-RCT is a $W : V(P) \rightarrow \mathbb{N}_0$ node weighted version of tree P, where $W(s) = \Delta(G, S(P'))$ for $P' = \mathrm{RR}(P, s)$.

Given a node $s \in V(P)$ and a child t of s, let x be the number of nodes from G that will become duplications when $P' = \mathrm{RR}(P, s)$ is rerooted to form $P'' = \mathrm{RR}(P, t)$. Then, Lemma 5.1 implies that $gain(s) = x$. Now, Observation 3 follows easily based on the definitions of *gain* and *loss* and the computation of the value $W(s)$ at each node $s \in V(P)$ (see steps 20-28 of Algorithm 1).

Observation 3. *If the values of $gain(s)$ and $loss(s)$ are correctly computed at each $s \in V(P)$ of $W(s)$, then the value of $W(s)$ is also correctly computed for each $s \in V(P)$.*

Recall that the overall strategy of algorithm Alg-RCT is to consider all the nodes in G one at a time and update the relevant *gain* and *loss* values on P for that node. Consider any node $g \in V(G)$. We will say that *g correctly updates* the values *gain* and *loss* on P if and only if 1) for each node $s \in V(P)$ for which $\mathrm{d}_{S(P')}(g) = f$ and $\mathrm{d}_{S(P'')}(g) = t$, where $P' = \mathrm{RR}(P, s)$ and $P'' = \mathrm{RR}(P, t)$, $t \in \mathrm{Ch}(s)$, we set $gain(s) \leftarrow gain(s) + 1$, and 2) for each node $s \in V(P)$ for which $\mathrm{d}_{S(P')}(g) = t$ and $\mathrm{d}_{S(P'')}(g) = f$, where $P' = \mathrm{RR}(S, \mathrm{Pa}(s))$ and $P'' = \mathrm{RR}(S, s)$, we set $loss(s) \leftarrow loss(s) + 1$.

Now, Observation 4 follows directly based on the definition of *gain* and *loss*.

Observation 4. *If each node in G correctly updates the value of the relevant* gain *and* loss *counters on P, then the values of $gain(s)$ and $loss(s)$ are correctly computed at each $s \in V(P)$.*

Theorem 5.1. *Algorithm Alg-RCT solves the ROOTING problem.*

Proof. Observations 3 and 4 directly imply that in order to prove the correctness of algorithm Alg-RCT, it is sufficient to prove that each node in G correctly updates the *gain* and *loss* values on P. Therefore, let g be some node in $V(G)$. We will show that according to algorithm Alg-RCT, g correctly updates the values *gain* and *loss* on P. Let $\mathrm{Ch}(g) = \{g', g''\}$; then, depending on which node is picked, there are four distinct possibilities, exactly one of which must hold true:

1. g is not a relevant node.
2. g is relevant, and $\mathrm{d}_{S(P)}(g) = f$.
3. g is relevant, $d_{S(P)}(g) = t$, and $\mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g') = \mathcal{M}_{S(P)}(g'')$.
4. g is relevant, $d_{S(P)}(g) = t$, and $\mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g') \neq \mathcal{M}_{S(P)}(g'')$.

Let us now study how algorithm Alg-RCT behaves in each of these cases:

1. g is not a relevant node. In this case, by Lemma 4.1, no changes should be made to any of the *gain* and *loss* values on P. As can be seen in steps 5 and 6 of Algorithm 1, the algorithm ignores all nonrelevant nodes. Hence, in this case, algorithm Alg-RCT correctly updates the values of *gain* and *loss*.
2. g is relevant, and $\mathrm{d}_{S(P)}(g) = f$. In this case, by Lemma 4.3, we must only increment the value *gain* at the nodes $\mathcal{M}_{S(P)}(g')$ and $\mathcal{M}_{S(P)}(g'')$ by 1. As can be seen in steps 8-10 of Algorithm 1, this is exactly what algorithm Alg-RCT does.
3. g is relevant, $d_{S(P)}(g) = t$, and $\mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g') = \mathcal{M}_{S(P)}(g'')$. In this case, by Lemma 4.4, no changes should be made to any of the *gain* and *loss* values on P. A look at steps 12 and 13 of Algorithm 1 confirms that algorithm Alg-RCT works correctly in this case.
4. g is relevant, $d_{S(P)}(g) = t$, and $\mathcal{M}_{S(P)}(g) = \mathcal{M}_{S(P)}(g') \neq \mathcal{M}_{S(P)}(g'')$. In this case, g' is marked as a Special node, and execution is transferred to Algorithm 2 (see steps 18 and 19). The first step in Algorithm 2 is to label G according to an in-order tour. This lets us store any subtree G_z, $z \in V(G)$, as an interval with the following property: a node is a descendant of G_z if and only if its label lies within the interval associated with z. This is used to determine in output-sensitive time (by making use of stabbing queries on an interval tree) those nodes in some subset of $V(G)$ that contain a given node as their descendant.
According to Lemma 4.5, before we can update any g and *loss* values, we must first locate the node α (see Lemma 4.5 for the definition of α). Next, we show how Algorithm 2 locates this node α. Let b denote the node $\mathcal{M}_{S(P)}(g'')$. Following Lemma 4.5, we subdivide our analysis into two parts:

   a. $\alpha = b$. By the definition of α, we have this case if and only if there exists a node $v \in V(G_{g'})$ with $\mathcal{M}_{S(P)}(v) \in \mathrm{Le}(P_b)$. According to part 1 of Lemma 4.5, in this case, no changes should be made to any of the *gain* and *loss* values on P.

      This case is handled in steps 7-13 of Algorithm 2. Here, the node g' is first added to the End set at b. This means that the interval tree contains the interval for g' when we enter the subtree P_b during the Euler tour of P. Next, the algorithm essentially does the following: at each leaf node, x, in P_b, it asks if there exists a node v in $G_{g'}$ for which $\mathcal{M}_{S(P)}(v) = x$. If yes, then we remove the Special status of node g'. This ensures that the node g does not affect any *gain* and *loss* values.

   b. $\alpha \neq b$. This case is handled in steps 15-38 of Algorithm 2. Observe that if g falls under this case, then it is unaffected by steps 7-13 of Algorithm 2. Handling this case involves traversing an arbitrarily ordered version of P in postorder twice, once in a left-child-first manner and again in a right-child-first manner. Each of these traversals produces a possible candidate for the node α for g. The actual α for g is the node closest to b among these two candidates.

      Suppose the node α for g was such that b lies in the left subtree of P_α. Then, let us consider steps 19-26 of Algorithm 2, i.e., the part where we perform the left-child-first postorder traversal of P. In this case, when we enter the right subtree of P_α during the postorder traversal, the interval tree must contain the interval for g'. Next, the algorithm essentially does the following: at each leaf node, x, in the right subtree of P_α, it asks if there exists a node v in $G_{g'}$ for which $\mathcal{M}_{S(P)}(v) = x$. If yes, then the algorithm sets $lca(\{b, x\})$ as the candidate for α for g and removes the interval for node g' from the interval tree. This removal ensures that the selected candidate is not supplanted later by some other candidate that is further away from b. Because we traverse P in postorder, in this case, we must have $\alpha = lca(\{b, x\})$.

Symmetrically, if the node α for g was such that b lies in the right subtree of P_α, then in this case, the correct candidate for α would be found during the right-child-first postorder traversal (steps 27-34).

Once α is located for g, the algorithm proceeds to update the relevant *gain* and *loss* values, in accordance with Lemma 4.5 (see steps 35-38).

Thus, algorithm Alg-RCT correctly solves the ROOTING problem. □

We now study the time complexity of algorithm Alg-RCT. The input for the ROOTING problem is a gene tree G, a species tree S, and the pruned subtree P of S. Let $n = |\mathrm{Le}(S)|$ and $m = |\mathrm{Le}(S)| + |\mathrm{Le}(G)|$.

Theorem 5.2. *The time complexity of algorithm Alg-RCT is* $O(m \log m)$.

Proof. First, we study the time complexity of Algorithm 2. Steps 2-6 take $O(m)$ time. Consider the *for* loop in steps 7-13. The complexity of this part is dominated by the complexity of the *for* loop in steps 10-13. This loop is executed at most $n + |\cup_{x \in V(P)} \mathcal{M}^{-1}_{S(P)}(x)|$ times, which is $O(m)$. During each of these iterations, we 1) perform a stabbing query, which takes time $O(\log |V(G)|) + c$, where c is the number of nodes returned by the stabbing query (see [27]), and 2) spend $O(\log |V(G)|)$ time on each of the c nodes returned (see [27]). A crucial observation here is that each node of G is stabbed at most once. This implies that throughout the $O(|V(G)|)$ iterations, the total number of nodes returned by the stabbing query is $O(|V(G)|)$. Thus, the total complexity of steps 7-13 is $O(m) \times O(\log |V(G)|)$, which is $O(m \log m)$.

Steps 14-18 require $O(m)$ time. Consider now the *for* loop in steps 19-26 and the *for* loop in steps 27-34. The analysis of both these loops is analogous to the analysis of the *for* loop in steps 7 through 13 shown above.

The *for* loop of steps 35 through 38 traverses through the nodes of G at most once and spends $O(1)$ time on each node. This gives a time complexity of $O(m)$ for this part of the algorithm. Hence, the total time complexity of Algorithm 2 is $O(m \log m)$.

We now look at Algorithm 1. Steps 3-5 take $O(m)$ time. Next, the *for* loop in steps 6-18 involves traversing the nodes of G at most once each, and at each node, we spend $O(1)$ time. This gives the *for* loop a total time complexity of $O(|V(G)|)$, which is $O(m)$. As shown, step 19 has a time complexity of $O(m \log m)$. Steps 20-28 involve traversing the tree P twice and spending $O(1)$ at each node during both traversals, yielding a time complexity of $O(n)$ for these steps. The total time complexity of Algorithm 1 is therefore $O(m \log m)$.

Thus, we get a total time complexity of $O(m \log m)$ for algorithm Alg-RCT. □

5.2 Algorithm Alg-RCT($\mathcal{G}, S, P$)

This algorithm solves the TBR-S problem. The algorithm is described as follows: We first use Algorithm Alg-RCT to solve the BR problem, as shown in Section 4. As shown in Section 3, a solution to the BR problem leads naturally to a solution for the TBR-S problem.

We now study the correctness and time complexity of algorithm Alg-TBR. In order to simplify our analysis, we assume that all $G \in \mathcal{G}$ have approximately the same size. We point out that this is purely a simplifying assumption and does not affect the complexity of the algorithm. Recall that $n = |\mathrm{Le}(S)|$, and $m = |\mathrm{Le}(S)| + |\mathrm{Le}(G)|$. In addition, let $k = |\mathcal{G}|$.

We have the following theorem.

Theorem 5.3. *Algorithm Alg-TBR solves the TBR-S problem in* $O(knm \log m)$ *time.*

Proof. *Correctness.* To establish the correctness of our algorithm for the TBR-S problem, it is sufficient to show that the ROOTING problem is correctly solved by Algorithm Alg-RCT. Therefore, Theorem 5.1 immediately implies the correctness of Alg-TBR.

Complexity. As seen in Theorem 5.2, the time complexity of Alg-RCT(G, S, P) is $O(m \log m)$. This implies that the complexity of the BR problem is $O(km \log m)$. Therefore, by Corollary 3.2, the time complexity of the TBR-RS problem is $O(km) + O(km \log m)$, which is $O(km \log m)$. The time complexity of Alg-TBR is thus $O(n) \times O(km \log m)$, which is $O(knm \log m)$. □

The time complexity of the existing naive solution for the TBR-S problem is $O(kn^3m)$. Thus, our algorithm improves on the current solution by a factor of $n^2/\log m$.

6 Outlook and Conclusion

Despite the inherent complexity of the duplication problem, it has been an effective approach for incorporating data from gene families into a phylogenetic inference [4], [5], [6], [7]. The duplication problem is typically approached by using local search heuristics. Among these, TBR heuristics are especially desirable for large-scale phylogenetic analyses, but current solutions have prohibitively large runtimes. Our algorithm offers a vast reduction in runtime, which makes TBR heuristics applicable for such large-scale analyses.

The ideas developed in this paper could possibly be applied to other problems related to the reconciliation of gene and species trees. For example, our solution for the ROOTING problem can be used to efficiently find an optimal rooting for any species tree, with respect to the given gene trees.

Acknowledgments

The authors wish to thank the anonymous referees for their invaluable comments. This work was supported in part by NSF Grant EF-0334832.

References

[1] R. Guigó, I. Muchnik, and T.F. Smith, "Reconstruction of Ancient Molecular Phylogeny," *Molecular Phylogenetics and Evolution,* vol. 6, no. 2, pp. 189-213, 1996.
[2] B. Ma, M. Li, and L. Zhang, "On Reconstructing Species Trees from Gene Trees in Term of Duplications and Losses," *Proc. Second Ann. Int'l Conf. Research in Computational Molecular Biology (RECOMB '98),* S. Istrail, P. Pevzner, and M. Waterman, eds., pp. 182-191, 1998.
[3] R.D.M. Page, "GeneTree: Comparing Gene and Species Phylogenies Using Reconciled Trees," *Bioinformatics,* vol. 14, no. 9, pp. 819-820, 1998.
[4] J.B. Slowinski, A. Knight, and A.P. Rooney, "Inferring Species Trees from Gene Trees: A Phylogenetic Analysis of the Elapidae (Serpentes) Based on the Amino Acid Sequences of Venom Proteins," *Molecular Phylogenetics and Evolution,* vol. 8, no. 3, pp. 349-362, 1997.
[5] R.D.M. Page, "Extracting Species Trees from Complex Gene Trees: Reconciled Trees and Vertebrate Phylogeny," *Molecular Phylogenetics and Evolution,* vol. 14, no. 1, pp. 89-106, 2000.
[6] R.D.M. Page and J. Cotton, "Vertebrate Phylogenomics: Reconciled Trees and Gene Duplications," *Proc. Seventh Pacific Symp. Biocomputing (PSB '02),* R.B.A. et al., eds., pp. 536-547, Jan. 2002.
[7] J.A. Cotton and R.D.M. Page, "Tangled Tales from Multiple Markers: Reconciling Conflict between Phylogenies to Build Molecular Supertrees," *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life,* O.R.P. Bininda-Emonds, ed., pp. 107-125, Springer, 2004.
[8] M.J. Sanderson and M.M. McMahon, "Inferring Angiosperm Phylogeny from EST Data with Widespread Gene Duplication," *BMC Evolutionary Biology,* vol. 7, supplementary 1:S3, 2007.
[9] D.L. Swofford, G.J. Olsen, P.J. Waddel, and D.M. Hillis, "Phylogenetic Inference," *Molecular Systematics,* D.M. Hillis, C. Moritz, and B.K. Mable, eds., chapter 11, pp. 407-509, Sinauer Assoc., 1996.
[10] B.L. Allen and M. Steel, "Subtree Transfer Operations and Their Induced Metrics on Evolutionary Trees," *Annals of Combinatorics,* vol. 5, pp. 1-13, 2001.
[11] D. Chen, O. Eulenstein, D. Fernández-Baca, and J.G. Burleigh, "Improved Heuristics for Minimum-Flip Supertree Construction," *Evolutionary Bioinformatics,* vol. 2, 2006.
[12] M. Goodman, J. Czelusniak, G.W. Moore, A.E. Romero-Herrera, and G. Matsuda, "Fitting the Gene Lineage into Its Species Lineage. A Parsimony Strategy Illustrated by Cladograms Constructed from Globin Sequences," *Systematic Zoology,* vol. 28, pp. 132-163, 1979.
[13] R.D.M. Page, "Maps between Trees and Cladistic Analysis of Historical Associations among Genes, Organisms, and Areas," *Systematic Biology,* vol. 43, no. 1, pp. 58-77, 1994.
[14] B. Mirkin, I. Muchnik, and T.F. Smith, "A Biologically Consistent Model for Comparing Molecular Phylogenies," *J. Computational Biology,* vol. 2, no. 4, pp. 493-507, 1995.
[15] O. Eulenstein, "Predictions of Gene-Duplications and Their Phylogenetic Development," PhD dissertation, Univ. of Bonn, Germany. gMD Research Series No. 20/1998, ISSN: 1435-2699, 1998.
[16] L. Zhang, "On a Mirkin-Muchnik-Smith Conjecture for Comparing Molecular Phylogenies," *J. Computational Biology,* vol. 4, no. 2, pp. 177-187, 1997.
[17] K. Chen, D. Durand, and M. Farach-Colton, "NOTUNG: A Program for Dating Gene Duplications and Optimizing Gene Family Trees," *J. Computational Biology,* vol. 7, no. 3/4, 2000.
[18] P. Bonizzoni, G.D. Vedova, and R. Dondi, "Reconciling a Gene Tree to a Species Tree under the Duplication Cost Model," *Theoretical Computer Science,* vol. 347, nos. 1-2, pp. 36-53, 2005.
[19] P. Górecki and J. Tiuryn, "On the Structure of Reconciliations," *Proc. Second RECOMB Comparative Genomics Satellite Workshop,* J. Lagergren, ed., pp. 42-54, 2004.
[20] M.A. Bender and M. Farach-Colton, "The LCA Problem Revisited," *Proc. Fourth Latin Am. Symp. Theoretical Informatics (LATIN '00),* G.H. Gonnet, D. Panario, and A. Viola, eds., pp. 88-94, 2000.
[21] D. Harel and R.E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM J. Computing,* vol. 13, no. 2, pp. 338-355, 1984.
[22] M.R. Fellows, M.T. Hallett, C. Korostensky, and U. Stege, "Analogs and Duals of the MAST Problem for Sequences and Trees," *Proc. Sixth Ann. European Symp. Algorithms (ESA '98),* G. Bilardi, G.F. Italiano, A. Pietracaprina, and G. Pucci, eds., pp. 103-114, 1998.
[23] U. Stege, "Gene Trees and Species Trees: The Gene-Duplication Problem Is Fixed-Parameter Tractable," *Proc. Sixth Int'l Workshop Algorithms and Data Structures (WADS '99),* F.K.H.A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, eds., pp. 288-293, 1999.
[24] M.T. Hallett and J. Lagergren, "New Algorithms for the Duplication-Loss Model," *Proc. Fourth Ann. Int'l Conf. Research in Computational Molecular Biology (RECOMB '00),* R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, eds., pp. 138-146, 2000.
[25] M. Bordewich and C. Semple, "On the Computational Complexity of the Rooted Subtree Prune and Regraft Distance," *Annals of Combinatorics,* vol. 8, pp. 409-423, 2004.
[26] M.S. Bansal, J.G. Burleigh, O. Eulenstein, and A. Wehe, "Heuristics for the Gene-Duplication Problem: A $\Theta(n)$ Speed-Up for the Local Search," *Proc. 11th Ann. Int'l Conf. Research in Computational Molecular Biology (RECOMB '07),* T.P. Speed and H. Huang, eds., pp. 238-252, 2007.
[27] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer, 2000.

Mukul S. Bansal received the BTech degree in computer science and engineering from the International Institute of Information Technology, India, in July 2004 and the MS degree in computer science from Iowa State University (ISU), Ames, USA, in December 2006. He is currently a PhD candidate in the Department of Computer Science, ISU. His research interests include computational biology and phylogenetics, graph theory, combinatorial optimization, approximation algorithms, and algorithms in general.

Oliver Eulenstein received the PhD degree in computational biology from the University of Bonn with Thomas Lengauer in 1998, and was a postdoctoral fellow with Dan Gusfield at the University of California at Davis. He is an associate professor of computer science in the Department of Computer Science, Iowa State University, Ames, USA. His research focuses on computational biology, particularly on computational phylogenetics.