Discovering Frequent Agreement Subtrees from Phylogenetic Data

Sen Zhang, *Member*, *IEEE*, and Jason T.L. Wang, *Member*, *IEEE*

Abstract—We study a new data mining problem concerning the discovery of frequent agreement subtrees (FASTs) from a set of phylogenetic trees. A phylogenetic tree, or phylogeny, is an unordered tree in which the order among siblings is unimportant. Furthermore, each leaf in the tree has a label representing a taxon (species or organism) name, whereas internal nodes are unlabeled. The tree may have a root, representing the common ancestor of all species in the tree, or may be unrooted. An unrooted phylogeny arises due to the lack of sufficient evidence to infer a common ancestor of the taxa in the tree. The FAST problem addressed here is a natural extension of the maximum agreement subtree (MAST) problem widely studied in the computational phylogenetics community. The paper establishes a framework for tackling the FAST problem for both rooted and unrooted phylogenetic trees using data mining techniques. We first develop a novel canonical form for rooted trees together with a phylogeny-aware tree expansion scheme for generating candidate subtrees level by level. Then, we present an efficient algorithm to find all FASTs in a given set of rooted trees, through an Apriori-like approach. We show the correctness and completeness of the proposed method. Finally, we discuss the extensions of the techniques to unrooted trees. Experimental results demonstrate that the proposed methods work well, and are capable of finding interesting patterns in both synthetic data and real phylogenetic trees.

Index Terms—Data mining, evolutionary bioinformatics, computational phylogenetics, algorithmic design, pattern discovery.

1 Introduction

Scientists model phylogenetic relations using unordered leaf-labeled trees and develop methods for constructing these trees [27]. Different theories concerning the evolutionary history of the same set of species often result in different phylogenetic trees. Even the same phylogenetic theory may yield different trees for different orthologous genes. This leads to a fundamental research problem in phylogenetics: how to determine what two different hypothetical phylogenetic trees regarding the same set of taxa have in common. This problem can be partially answered by computing a maximum agreement subtree (MAST) of the two phylogenetic trees. An agreement subtree between two trees t_1 and t_2 is a substructure that occurs in both trees [2], [10], [11], [12]. A MAST between t_1 and t_2 is an agreement subtree of t_1 and t_2; furthermore, there is no other agreement subtree of t_1 and t_2 that has more leaves than the MAST.

The MAST problem was first studied by Finden and Gordon [11]. The authors developed a heuristic algorithm for finding the MAST of two binary rooted trees, which runs in time $O(n^5)$, where n is the number of nodes in the trees. Ganeshkumar and Warnow [12] later gave an $O(n^2)$ algorithm, and Farach et al. [10] presented an $O(n^{1.5}logn)$ algorithm with different constraint assumptions on tree topologies. When the MAST problem is generalized from two trees to multiple trees, the problem was shown to be polynomial-time solvable for trees with bounded degrees [2], [10]. For trees with unbounded degrees, this problem is NP-hard [2]. More recently, Berry and Nicolas [5] developed a linear-time parameterized algorithm to solve the MAST problem. An observation is that a MAST of multiple trees is usually of small size and thus uninformative, especially when a large number of phylogenetic trees are under consideration [12]. Furthermore, if there is an incorrectly inferred phylogeny in the trees, the MAST would provide wrong information too.

For example, a study *S497* [17] in TreeBASE [23] shows that biologists built a set of five rooted phylogenetic trees for six Hamamelis-related species. Each of the five trees depicts a hypothesis about the evolutionary history of the six species. The five phylogenetic trees are shown in the first two rows in Fig. 1. Three subtree patterns st_1, st_2, and st_3 are shown in the last row in Fig. 1. Here, st_1 and st_2 are MASTs of the five trees, since they are subtrees of all the five trees, and no other subtrees occurring in all the five trees have more leaves than st_1 and st_2. The pattern st_3 is a subtree of three trees only, namely, t_1, t_3, and t_5 and therefore not a MAST of the five trees. Nevertheless, in phylogenetics, st_3 is not necessarily less informative than st_1 or st_2 for two reasons: 1) The number of leaves of st_3 is prominently greater than that of the two MAST patterns st_1 and st_2, and 2) st_3 occurs in a majority of the trees. Motivated by this observation, we develop a new tree mining algorithm, called Phylominer [36], to find all frequent agreement subtrees (FASTs) from a given set of rooted phylogenetic trees, that is, our algorithm will find not only st_1 and st_2 but also st_3 when applied to the above example.

- *S. Zhang is with the Department of Mathematics, Computer Science, and Statistics, The State University of New York, College at Oneonta, Ravine Parkway, Oneonta, NY 13820. E-mail: zhangs@oneonta.edu.*
- *J.T.L. Wang is with the Bioinformatics Program and the Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102. E-mail: wangj@njit.edu.*

Manuscript received 2 Dec. 2006; revised 31 July 2007; accepted 10 Sept. 2007; published online 13 Sept. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0542-1206.
Digital Object Identifier no. 10.1109/TKDE.2007.190676.

1041-4347/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

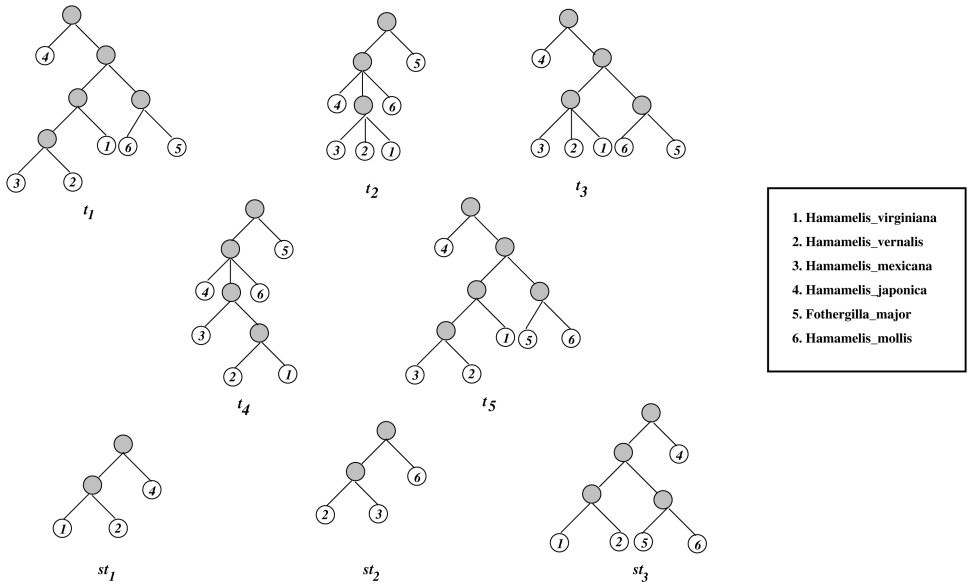

Fig. 1. Five rooted trees for six Hamamelis-related species are shown in the first two rows. Three subtrees are displayed in the last row, where st_1 and st_2 are MASTs, but st_3 is not. The names of the six species are Hamamelis_virginiana, Hamamelis_vernalis, Hamamelis_mexicana, Hamamelis_japonica, Fothergilla_major, and Hamamelis_mollis, respectively, which are represented by node labels 1, 2, 3, 4, 5, and 6, respectively, in the trees.

From a biological viewpoint, the FASTs indicate which species are evolutionarily related according to the majority of phylogenies under analysis where the phylogenies could be inferred from different phylogenetic tree reconstruction algorithms. For example, consider Fig. 1 again. Based on the majority of the five different phylogenies in the figure, Fothergilla_major (species 5) is more closely related to Hamamelis_mollis (species 6) than Hamamelis_japonica (species 4), as suggested in [17]. This information is revealed in the FAST pattern st_3 but not in the MAST patterns st_1 and st_2, suggesting that FAST patterns are more important than MAST patterns in phylogenetics.

1.1 Related Work

Ordered tree mining problems have been studied by several researchers. Asai et al. [3] proposed a rightmost expansion algorithm to find induced subtrees in rooted ordered trees. Contemporarily, Zaki [34] developed similar techniques capable of finding frequent embedded subtrees in a forest of rooted ordered trees. Yang et al. [33] studied the ordered tree mining problem in the context of XML management by adapting the rightmost expansion scheme to solving a frequent XML query pattern discovery problem. Wang et al. [28] presented a dynamic programming algorithm for finding the consensus of two general ordered trees, which was applied to motif finding in RNA secondary structures.

In the area of unordered tree mining, Xiao et al. [31] proposed an efficient frequent subtree discovery algorithm through path joining operations. Asai et al. [4] and Nijssen and Kok [21] independently discussed an essentially identical tree enumeration technique for unordered tree mining. More recently, Chi et al. [7], [8] presented a suite of algorithms to find frequent induced subtrees in both rooted and unrooted unordered trees. Shasha et al. [25] developed methods to find cousin pairs in unordered trees with applications to phylogeny. For a comprehensive survey of tree mining methods and applications, please refer to [6].

In parallel with the tree mining research, graph mining is a closely related field that also has been intensely studied during the past decade. Kuramochi and Karypis [16] extended traditional frequent itemset algorithms to find frequent patterns in graph data. Yan and Han [32] proposed a novel canonical graph form to find closed frequent subgraphs. Huan et al. [15] devised a different canonical form to efficiently discover frequent subgraphs in the presence of graph isomorphism. For the readers who are interested in the state of the art of graph mining, please refer to [30]. Chi et al. [6], [8] also gave an excellent survey on acyclic graph mining.

Here, we present a new algorithm to tackle the FAST problem arising in data mining and computational phylogenetics. Our work differs from the above approaches in two ways. First, in contrast to the general trees studied by previous researchers [3], [7], [8], [21], [34], we focus on leaf-labeled phylogenetic trees, which are commonly used to model evolutionary histories of related species. Second, our work was directly motivated by the MAST problem studied in computational phylogenetics. This makes our algorithms unique, because the subtrees we mine for are application oriented and different from the patterns found in all the previous tree mining papers. Specifically, Chi et al.'s [7] work is a recent breakthrough in unordered tree mining; however, their algorithms find induced subtrees from unordered trees defined in the general tree context, rather than embedded subtrees from leaf-labeled trees considered here. Zaki's [34] Treeminer is a powerful algorithm to mine for embedded subtrees from ordered trees, but his embedded subtree definition is rather tolerant. By contrast, an agreement subtree in the phylogenetics context is unordered and demands strict topological restrictions on valid embeddings. The problem tackled here requires that

the agreement subtrees to be mined for should be both unordered and embedded. This makes the problem drastically different from the tree mining problems published in the literature. Neither [34] nor [7] can find exactly the FASTs in multiple phylogenies as our algorithms do. Furthermore, there is no straightforward way to efficiently adapt the previous methods to the problem addressed here, which the proposed Phylominer is designed for. Our tree mining method thus joins the many others already developed [3], [4], [7], [8], [21], [25], [28], [31], [33], [34].

In [35], we formalized the FAST problem for rooted phylogenetic trees, sketched the ideas behind Phylominer, and reported its implementation status [36]. Here, we extend the work in [35], [36] by 1) presenting the algorithmic details and theoretical foundation of Phylominer, analyzing its correctness and complexity, 2) extending the techniques of Phylominer to handle unrooted phylogenetic trees, and 3) conducting a complete experimental study to evaluate the performance of the tree mining algorithms for both rooted and unrooted phylogenies.

The rest of the paper is organized as follows: Section 2 presents basic concepts and terminologies. Section 3 describes in detail the Phylominer algorithm for rooted trees and shows the correctness and completeness of the algorithm. We also extend the techniques of Phylominer to handle unrooted trees. Section 4 presents experimental results. Section 5 concludes the paper.

2 Preliminaries

Let L denote a set of labels, with each label representing an evolutionary unit. An evolutionary unit can be a taxon, organism, species, protein, gene, etc. Let the cardinality of L, denoted by $|L|$, be k. Without loss of generality, L can be considered as a set of k positive integers $\{n_1, n_2, \ldots, n_{k-1}, n_k\}$.

Phylogenetic tree. A phylogenetic tree, or phylogeny, t on L is a rooted leaf-labeled unordered tree in which 1) there are $|L|$ leaves, and each leaf is associated with a distinct unique label drawn from L, 2) all internal nodes have no labels, and 3) a special node, denoted $r(t)$, is designated as the root of the tree. Furthermore, the fanout, that is, the number of children, of each internal node in t is at least 2. (A phylogeny differs from general trees, in which internal nodes may have labels, and the fanout of an internal node can be 1.) The *depth* of the phylogeny is the number of edges in the longest root-to-leaf path. The *size* of the phylogeny is the number of its leaves, which equals the cardinality of L. For convenience, a phylogeny with k leaves is called a k-leaf tree; a node label will be used to represent the corresponding node, and vice versa, when the context is clear.

Subtree. Let N_t (N_{st}, respectively) represent the set of nodes in tree t (tree st, respectively). We say st is a subtree of t, if there exists a mapping f from the nodes in N_{st} to the nodes in N_t such that the mapping is an injective function f: $N_{st} \rightarrow N_t$, satisfying the following properties for all nodes u, $v \in N_{st}$:

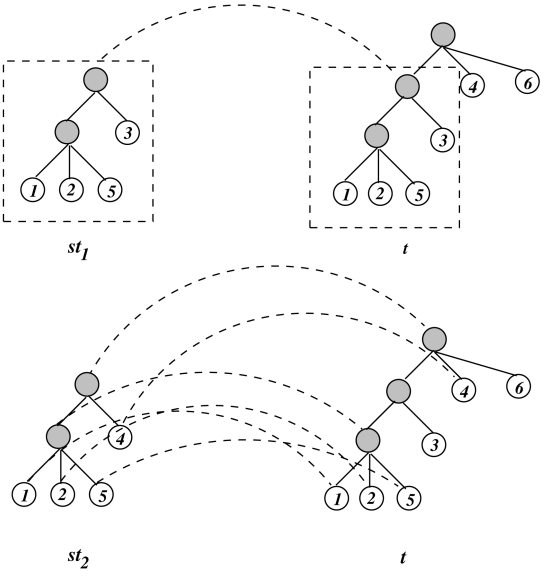

Fig. 2. The subtree st_1 is an induced subtree of tree t with no edge contraction, and the subtree st_2 is an embedded subtree of t with an edge contraction. Shaded nodes are internal nodes that do not have labels; only leaf nodes have labels.

- $label(f(u)) = label(u)$, where $label(u)$ represents the label of u if u is a leaf or is undefined if u is an internal node (label preservation).
- $f(u) \in desc(f(v))$ if and only if $u \in desc(v)$, where $desc(v)$ is the set of descendants of node v (ancestor-descendant preservation).
- $LCA(f(u), f(v)) = f(LCA(u, v))$, where $LCA(u, v)$ is the least common ancestor of u, v (least common ancestor preservation).

A phylogeny st on SL is a subtree of phylogeny t on L, if $SL \subset L$, st is a subtree of t, and st can be obtained by restricting t to the leaf set SL through pruning all leaves $l \in L - SL$. This definition is represented by $st \equiv t|_{SL}$, where $t|_{SL}$ denotes the operation of restricting t to SL through leaf pruning, and $\equiv$ denotes the isomorphism relationship between two unordered trees. Notice that pruning a leaf from a phylogenetic tree may trigger an edge contraction [29] in meeting the requirement that the fanout of any internal node must be at least 2. Specifically, the edge contraction works as follows: After a leaf l is pruned or removed, l's parent p may have a single child c only. Thus, p is removed, making c become a child of p's parent. In general, multiple edge contractions can be triggered if multiple leaves are pruned.

Fig. 2 shows two different injective mappings from two subtrees to tree t. From the mapping lines, it can be seen that st_1 is an induced subtree of tree t [7], whereas st_2 is an embedded subtree of t [34] due to the edge contraction triggered by pruning the leaf labeled 3. Both the embedded subtree and the induced subtree, which is a special case of the embedded subtree without causing edge contractions in leaf-labeled unordered phylogenetic trees can be handled by our Phylominer algorithm. By contrast, Chi et al.'s [7] work deals with only induced subtrees in general unordered trees, whereas Zaki's [34] Treeminer deals with embedded subtrees in general ordered trees (in which the

order among siblings is important). The embedded subtrees considered by Zaki in [34] are more tolerant than the subtrees tackled here, which can be considered as strictly embedded subtrees, in that the leaves of the embedded subtrees in [34] can be internal nodes of the original trees, and the fanout of an internal node in the embedded subtrees in [34] can be 1. By contrast, the leaves of the subtrees here must be leaves of the original phylogenetic trees, and the fanout of each internal node in the subtrees here must be at least 2. The embedded subtree definition in [34] allows patterns that would be invalid in phylogenetic applications.

Total agreement subtree. Let $DT = \{t_1, t_2, \ldots, t_m\}$ be a set of phylogenies on the leaf set L, and let SL be a subset of L. A phylogeny st on SL is a *total agreement subtree* (or AST) for DT, if st is a subtree of every tree in DT, that is, $st \equiv t_1|_{SL} \equiv t_2|_{SL} \ldots \equiv t_m|_{SL}$.

MAST. If st has the maximum number of leaves among all ASTs for DT, then st is a *MAST* for DT. In Fig. 1, both st_1 and st_2 are MASTs of the five phylogenetic trees in the figure.

FAST. A subtree or pattern p is said to be supported by a phylogeny t, or p occurs in t, if p is a subtree of t. We define $supp_{p,i}$ to be 1 if $t_i \in DT$ supports p; otherwise, $supp_{p,i}$ is 0. The *support* of the subtree $st = p$ with respect to DT is defined as $(\sum_{1 \le i \le m} supp_{p,i}/|DT|) \times 100$ percent. A phylogeny is a *FAST* for DT if its support is greater than or equal to a user-specified minimum support value, $minsup$. Our goal is to find all FASTs in a given set of rooted phylogenies where the support of the subtrees is greater than or equal to $minsup$.

Maximum FAST (MFAST). If st has the maximum number of leaves among all FASTs for DT, then st is an MFAST for DT. Obviously, an MFAST might have more leaves than a MAST. In Fig. 1, st_3 is a MFAST of the five phylogenetic trees in the figure when $minsup$ is set to 60 percent.

It should be pointed out that the set of FAST patterns is a superset of the set of MAST patterns. The algorithm for the FAST problem can find FASTs occurring in a small portion of a given data set (for example, with $minsup = 10$ percent), whereas the algorithm for the MAST problem always finds the MASTs occurring in all trees in the data set.

3 Phylogenetic Tree Mining

The same rooted leaf-labeled unordered tree can be present in different ordered representations. This is the so-called tree isomorphism problem, which is generally believed to make pattern mining in unordered trees more sophisticated than that in ordered trees. To solve this problem, we propose a new canonical form for rooted phylogenetic trees. This canonical form will allow us to represent each phylogenetic tree by using one unique ordered tree. We then introduce the concept of equivalence classes, based on which our efficient candidate subtree generation strategy is developed.

3.1 Canonical Form

The proposed canonical form for rooted phylogenetic trees is a total ordering scheme among leaf labels in L, which conforms to the integer comparison property, that is, the ordering of L is $1 < 2 < \ldots < n < n+1 < \ldots$. Based on this leaf label ordering scheme, the canonical form of a rooted phylogenetic tree t requires the assignment of virtual labels to all originally unlabeled internal nodes in t. Specifically, the virtual label of an internal node in t is the smallest integer label among all the integer labels of t's child nodes. Once the virtual labels are assigned to all internal nodes of t, we define the canonical form of t to be a specially designed ordered tree, in which all sibling nodes (including both leaf nodes and internal nodes) are arranged such that for every sibling pair (v, u) in t, node v always appears before node u in the depth-first traversal (DFT) order of t if $label(v) < label(u)$.

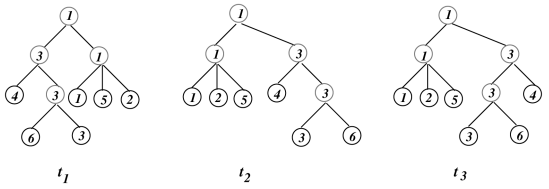

Fig. 3. Three ordered representations of the same unordered tree.

Fig. 3 shows an example of three different ordered representations of the same unordered tree. It can be seen that t_3 is in canonical form, whereas t_1 and t_2 are not. However, the isomorphism between the three rooted trees becomes obvious once t_1 and t_2 are transformed to their canonical forms.

According to the definition in Section 2, it is not difficult to see that any rooted phylogenetic tree can be transformed to its canonical form by a straightforward algorithm, which visits every node of the tree in a bottom-up manner, and during the visit of each internal node, the algorithm reorders all its child nodes. This procedure to transform a tree t to its canonical form is referred to as *canonicalizing* the tree t.

Lemma 3.1. *Canonicalizing a tree t takes $O(k)$ time, where k is the number of leaves in t.*

Proof. Let I represent the set of internal nodes in t. Let $g(i)$, $i \in I$, denote the number of children of the internal node i in t. To canonicalize the subtree rooted at each internal node i requires sorting all its $g(i)$ children, which can be done in $O(g(i))$ time by using the count sort algorithm. To canonicalize t, the above sorting operation is performed at all internal nodes and, hence, the time complexity for canonicalizing t is $O(\sum_i g(i)) = O(k)$. □

Property 1. *A direct pruning of the last leaf, based on the DFT order, of a rooted phylogenetic tree in canonical form results in a subtree that is still in canonical form.*

Property 2. *A direct pruning of the second last leaf, based on the DFT order, of a rooted phylogenetic tree in canonical form results in a subtree that is still in canonical form.*

Here, a direct pruning means pruning a leaf without further canonicalizing the resulting tree. In Section 3.5, we will show that the direct pruning properties suggest an efficient joining scheme regarding how to arrange the last leaves of two k-leaf trees to obtain $(k+1)$-leaf trees in their canonical forms without going through further

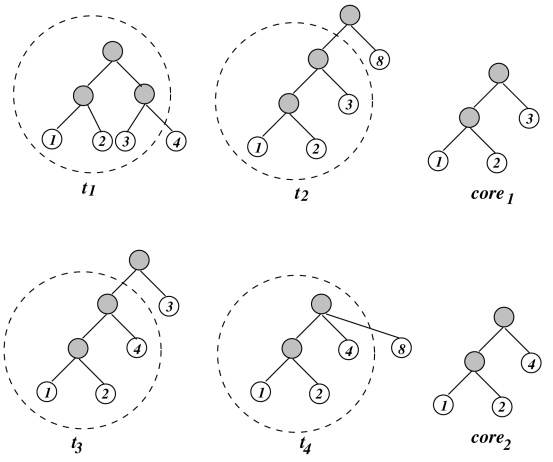

Fig. 4. Four trees are grouped into two equivalence classes. Trees t_1 and t_2 are in the same equivalence class, whereas trees t_3 and t_4 are in another equivalence class.

canonicalization. We next introduce some terms needed in explaining our Phylominer algorithm.

Weight scheme. After all internal nodes are labeled, each leaf i can be associated with a weight, denoted $w(i)$, which is an ordered label list obtained by concatenating the labels of all nodes along the path from the root to the leaf i. For example, the weights of the leaves of tree t_3 in Fig. 3 are the following: $w(1)$ is "1, 1, 1," $w(5)$ is "1, 1, 5," $w(4)$ is "1, 3, 4," and so on. The weights of leaf nodes can be compared from the most significant (leftmost) element down to the least significant (rightmost) element. For example, the weight order of the leaves in tree t_3 in Fig. 3 is $w(4) > w(6) > w(3) > w(5) > w(2) > w(1)$. We introduce the weight scheme here to facilitate the discussion of canonicalization of unordered phylogenetic trees. It is also necessary for understanding the concept of the heaviest leaf defined below.

Heaviest leaf. The heaviest leaf, denoted l_h, of a rooted phylogenetic tree t is the leaf with the heaviest weight among all leaves of t. If t is in canonical form, then l_h is always the last leaf of t according to its DFT order, that is, l_h is the rightmost leaf of t.

$(k-1)$-prefix tree. Given any k-leaf tree t in canonical form, we define its $(k-1)$-prefix tree to be the $(k-1)$-leaf tree obtained by pruning the rightmost leaf, that is, the heaviest leaf, from t. We use t_{hlp} to represent the $(k-1)$-prefix tree of t.

3.2 Equivalence Class

For two different k-leaf trees t and t' in their canonical forms, respectively, we say they are in the same equivalence class, if their respective $(k-1)$-prefix trees are isomorphic to each other, that is, t and t' share the same $(k-1)$-prefix tree, which is called the core of the equivalence class. (In determining whether a tree is isomorphic to another tree, we take into account not only their topologies but also node labels in them.) The relation "sharing the same $(k-1)$-prefix tree with each other" for a set of k-leaf trees is an equivalence relation, because the relation on these trees is reflective, symmetric, and transitive. The equivalence relation partitions the set of k-leaf trees into disjoint equivalence classes, where each equivalence class is uniquely identified by a core $(k-1)$-prefix tree.

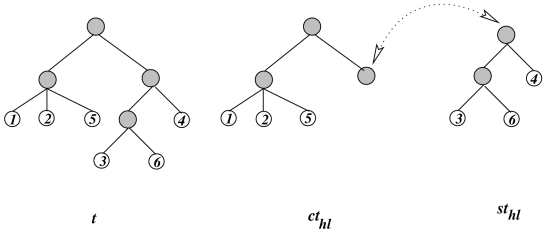

Fig. 5. A tree can be divided into a heaviest subtree and its complementary subtree.

Consider, for example, the trees in Fig. 4. Trees t_1 and t_2 are in an equivalence class, because they share the same $(k-1)$-prefix tree, denoted by $core_1$; t_3 and t_4 are in another equivalence class, since they share the same $(k-1)$-prefix tree, denoted by $core_2$. Note that in tree t_1, after pruning the rightmost leaf labeled 4, the parent, p, of this leaf has a single child labeled 3, violating the property that each internal node must have at least two children. Hence, p is removed, entailing an edge contraction and yield $core_1$. Similarly, in tree t_2, after pruning the rightmost leaf labeled 8, the root has a single child, violating the property that each internal node must have at least two children. Hence, the root is removed too, yielding the subtree in the dashed circle, which is $core_1$.

Heaviest subtree. Given a rooted phylogenetic tree t, the heaviest subtree of t, denoted st_{hl}, is defined as the subtree rooted at the parent of the heaviest leaf of t. The remaining part of the tree t after st_{hl} is taken away is called the *complementary tree* of st_{hl}, denoted by ct_{hl}. For example, in Fig. 5, the leaf labeled 4 is the heaviest leaf, l_h, in t, and st_{hl} is the heaviest subtree of t, whereas ct_{hl} is the complementary tree of the heaviest subtree st_{hl}. The heaviest subtree will be used to describe our candidate generation method, where our main concern is how to join two heaviest subtrees. Notice that when two trees are in the same equivalence class, their differences must be locally restricted to their heaviest subtrees; otherwise, they would not be in the same equivalence class.

3.3 Newick Notation

Phylominer uses Newick notation to represent input trees, intermediate candidate subtrees, and final output trees. Newick notation (http://evolution.genetics.washington.edu/phylip/newicktree.html), widely used in computational phylogenetics [23], represents a tree by a very compact parenthesized string form. The tree (or string) ends with a semicolon. An internal node n is represented by a pair of parentheses, enclosing n's immediate descendants, separated by commas. For example, the Newick formats for trees t_1 and t_2 in Fig. 1 are "(4, (((3, 2), 1), (6, 5)));" and "((4, (3, 2, 1), 6), 5);", respectively. Obviously, the Newick format for a tree is equivalent to, but more succinct than, the in-memory linked-list representation of the tree. It requires only linear time to convert an in-memory linked-list tree into its Newick string, and vice versa. Therefore, most tree manipulating operations used by Phylominer are performed directly on Newick strings to achieve high efficiency.

Procedure: Phylominer(DT, $minsup$)
Input: A set DT of rooted phylogenies on L, and $minsup$.
Output: A set of frequent agreement subtrees in DT.
/* $FAST$ contains frequent agreement subtrees in DT. */
1. $FAST \leftarrow \emptyset$;
2. $F_1 \leftarrow$ {frequent agreement 1-leaf trees};
3. $F_2 \leftarrow$ {frequent agreement 2-leaf trees};
4. $FAST \leftarrow FAST \cup F_1 \cup F_2$;
5. $EC_2 \leftarrow$ {equivalence classes of trees in F_2};
6. $k \leftarrow 2$;
7. **while** $|F_k| \geq k+1$
8. **begin**
9. F_{k+1}, $EC_{k+1} \leftarrow$ Grow_Subtrees(EC_k, k);
10. $FAST \leftarrow FAST \cup F_{k+1}$;
11. $k \leftarrow k+1$;
12. **end**;
13. **return** $FAST$;

Fig. 6. Algorithm for discovering FASTs from a set DT of rooted phylogenetic trees on the leaf label set L.

The Newick string of a tree in canonical form will be called the *canonical Newick string* of the tree. For example, the canonical Newick string of the unordered tree in Fig. 3 is "((1,2,5),((3,6),4));". An operation that would require the in-memory linked-list representation of a tree is the canonical labeling scheme described in Section 3.1. However, such operations are actually never performed, because the canonical form of any candidate subtree is automatically obtained through the joining procedure, as we will explain in Section 3.5. Throughout the paper, Newick notation will be used, whenever appropriate, to illustrate the details of the joining procedure.

3.4 Algorithmic Framework

Phylominer is an Apriori-like data mining method [1], which progressively enumerates all candidate subtrees in a set of rooted phylogenetic trees and checks the occurrence frequency of these candidate subtrees. Its algorithm is summarized in Fig. 6. Initially, Phylominer enumerates $|L|$ 1-leaf trees and all $\frac{|L|*|L-1|}{2}$ 2-leaf trees, which can be obtained by combinatorially assigning two different labels from L to the unlabeled 2-leaf tree skeleton. These 1-leaf and 2-leaf trees must occur in all rooted phylogenetic trees in the input set DT and hence are frequent, that is, their support values are 100 percent and are greater than or equal to $minsup$. The reason is that all trees in the input set DT have exactly the same leaves from L. Consequently, every leaf occurs in every input tree and, hence, every 1-leaf tree has a support value of 100 percent. Now, consider a 2-leaf tree st with two leaves l_1 and l_2 and an input tree t in DT. After pruning all leaves, except l_1 and l_2, from t, the remaining tree t' is isomorphic to st. Thus, st is a subtree of t, that is, st occurs in t. Since all input trees in DT have the same leaves, st is a subtree of all the input trees and hence has a support value of 100 percent too.

Let F_k represent the set of FASTs with k leaves, and let EC_k represent the set of equivalence classes of the subtrees in F_k. During each iteration of the while loop in Phylominer, the algorithm calls the subroutine Grow_Subtrees (line 9 in Fig. 6) to find the set of frequent agreement $(k+1)$-leaf trees, that is, F_{k+1}, from F_k. The subroutine, to be explained in detail in Section 3.5, will also return the set of equivalence classes of the frequent agreement $(k+1)$-leaf trees, that is, EC_{k+1}. Notice that when $|F_k| < k+1$ (line 7 in Fig. 6), we cannot produce any frequent agreement $(k+1)$-leaf tree and hence exit the while loop.

3.5 Candidate Generation

Our candidate generation method adopts a pairwise joining scheme. In order for two frequent agreement k-leaf trees to be eligible for joining, the two trees must be in the same equivalence class and must be on different leaf sets. (Recall that each phylogenetic tree has uniquely labeled leaves. We do not join two frequent agreement k-leaf trees if they are on the same leaf set, as the join cannot produce any $(k+1)$-leaf tree.) Due to the nature of equivalence classes, we adopt a rightmost joining approach to expand pattern trees (reminiscent of the rightmost extension schemes in [3], [34]). Thus, the focus of joining two frequent agreement k-leaf trees would be on how to form a new $(k+1)$-leaf tree by correctly gluing the rightmost leaves of the two k-leaf trees to the isomorphic part of the two trees. The isomorphic part of the two k-leaf trees is the $(k-1)$-prefix tree shared by them. Details and case analyses of joining two frequent agreement k-leaf trees can be found in the Appendix.

The Grow_Subtrees algorithm in Fig. 7 generates all frequent agreement $(k+1)$-leaf trees from frequent agreement k-leaf trees. For each pair of frequent agreement k-leaf trees x, y that are in the same equivalence class and are not on the same leaf set, the subroutine Phylo_Join joins x, y to generate candidate $(k+1)$-leaf trees based on the case analyses presented in the Appendix. For each candidate $(k+1)$-leaf tree c_{k+1} produced by Phylo_Join, we apply the Downward_Closure_Checking procedure [34] to it. This procedure returns a true value if all of the $(k+1)$ k-leaf subtrees of c_{k+1} are frequent, for which case, we invoke Frequency_Count to calculate the support value of c_{k+1}. If the Downward_Closure_Checking procedure returns a false value, c_{k+1} would not be a qualified pattern and is therefore safely discarded.

As an example, consider a candidate 3-leaf tree st with three leaves l_1, l_2, and l_3. It has three 2-leaf subtrees st_1, st_2, and st_3, where st_1 contains leaves l_1, l_2, st_2 contains leaves l_1, l_3, and st_3 contains leaves l_2, l_3. The Downward_Closure_Checking procedure checks the frequency of st_1, st_2, and st_3, respectively. If any one of the 2-leaf subtrees, say, st_3, is infrequent, st would not be frequent. Hence, the Downward_Closure_Checking procedure returns a false value, indicating st can be safely discarded, and we do not need to calculate its support value.

Notice that when a new c_{k+1} is generated, there is no need to check whether or not c_{k+1} is already generated previously, since each particular c_{k+1} can be generated only once based on the equivalence class design. If the support value of the candidate $(k+1)$-leaf tree c_{k+1} is greater than or equal to $minsup$, c_{k+1} is a frequent agreement $(k+1)$-leaf tree and hence added to F_{k+1}. Let c_{hlp} represent the tree c_{k+1} with the heaviest leaf pruned. That is, c_{hlp} is the k-prefix tree of c_{k+1} and will be the core of some equivalence class in EC_{k+1} (cf., Fig. 4). If c_{hlp}'s equivalence class is not already in EC_{k+1}, add

Procedure: Grow_Subtrees(EC_k, k)
Input: The set of equivalence classes of frequent agreement k-leaf trees, EC_k, and the tree size k.
Output: The set of frequent agreement $(k+1)$-leaf trees, F_{k+1}, and EC_{k+1}.
/* C_{k+1} is the set of candidate $(k+1)$-leaf trees generated from trees in EC_k. */
1. $C_{k+1} \leftarrow \emptyset$, $F_{k+1} \leftarrow \emptyset$, $EC_{k+1} \leftarrow \emptyset$;
   /* ec is an equivalence class in EC_k. */
2. **for each** $ec \in EC_k$ **do**
3. **if** $|ec| \geq 2$ **then**
4. **for each** pair of subtrees $x, y \in ec$ that are not on the same leaf set **do**
5. $C_{k+1} \leftarrow$ Phylo_Join(x, y);
6. **for each** $c_{k+1} \in C_{k+1}$ **do**
7. **if** Downward_Closure_Checking(c_{k+1}) = TRUE **then**
8. $freq \leftarrow$ Frequency_Count(c_{k+1});
9. **if** $(freq \geq minsup)$ **then**
10. $F_{k+1} \leftarrow F_{k+1} \cup \{c_{k+1}\}$;
    /* c_{hlp} is the k-prefix tree of c_{k+1}. */
11. **if** c_{hlp}'s equivalence class is not in EC_{k+1} **then**
12. add c_{hlp} as a new equivalence class to EC_{k+1};
13. add c_{k+1} to c_{hlp}'s equivalence class;
14. **return** F_{k+1}, EC_{k+1};

Fig. 7. Algorithm for generating all frequent agreement $(k+1)$-leaf trees from frequent agreement k-leaf trees.

the core c_{hlp} as a new equivalence class to EC_{k+1} (recalling that each equivalence class is uniquely identified by a core k-leaf tree in EC_{k+1}). We also add c_{k+1} to c_{hlp}'s equivalence class. Finally, Grow_Subtrees outputs both F_{k+1} and EC_{k+1}.

3.6 Frequency Counting

Once a candidate $(k+1)$-leaf tree c_{k+1} passes the Downward_Closure_Checking test, Phylominer invokes the Frequency_Count procedure to compute the support value of c_{k+1} by checking its occurrences in the input trees. Given a candidate subtree st on SL and a rooted phylogenetic tree $t \in DT$ on L, $t|_{SL}$ can be obtained by pruning all leaves $l \in L - SL$ from t. The candidate pattern st is a subtree of t if and only if st is isomorphic to $t|_{SL}$. The isomorphism between two trees can be verified by calculating their partition metric value [27]. The partition metric treats each phylogenetic tree as an unrooted tree and analyzes the partitions of taxa resulting from removing one edge at a time from the tree. By removing one edge from a tree, we are able to partition that tree. The metric value between two trees is defined as the number of edges in a tree for which there is no equivalent (in the sense of creating the same partitions) edge in the other tree [27]. Specifically, two trees are isomorphic to each other if and only if the partition metric value of the two trees is 0. The most efficient algorithm for calculating the partition metric of rooted phylogenetic trees has linear time complexity [9], which is the algorithm we adopt for pattern verification in the Frequency_Count procedure.

To further optimize the Phylominer algorithm, a supporting tree ID (STID) list [32], [34] is used to accelerate the process for verifying the presence of a subtree in input trees. Each subtree is associated with an STID list, which is a vector recording a list of identifiers of input trees that support the subtree. Before the frequency of a candidate subtree c_{k+1} is computed, the intersection set J of the STID lists of c_{k+1}'s frequent agreement k-leaf subtrees is computed first. If the cardinality of J is less than $minsup \times |DT|$, c_{k+1} would not be a qualified pattern and is therefore safely discarded. Otherwise, we check the occurrences of c_{k+1} in the trees in J. In fact, there is no need to perform expansion on two frequent agreement k-leaf trees if the cardinality of the interaction set of their STIDs is already less than $minsup \times |DT|$. As an example, consider again the candidate 3-leaf tree st with three leaves l_1, l_2, and l_3 described above. It has three 2-leaf subtrees st_1, st_2, and st_3, where st_1 contains leaves l_1, l_2, st_2 contains leaves l_1, l_3, and st_3 contains leaves l_2, l_3; st_1 and st_2 are frequent, whereas st_3 is infrequent. Suppose the STID list of st_1 is $\{t_1, t_2, t_3, t_4\}$, the STID list of st_2 is $\{t_1, t_2, t_5, t_6\}$, where t_is are identifiers of input trees in DT. That is, st_1 occurs in t_1, t_2, t_3, t_4, and st_2 occurs in t_1, t_2, t_5, and t_6. The intersection set J would be $\{t_1, t_2\}$. The candidate 3-leaf tree st would occur in at most the two trees in J. If the cardinality of J is already less than $minsup \times |DT|$, we do not even need to generate st.

3.7 Correctness and Complexity Analysis

We present in this section a series of lemmas and theorems concerning the proposed Phylominer algorithm.

Lemma 3.2. *Phylominer is correct. That is, any subtree output from Phylominer is a FAST in the given set of rooted phylogenies DT.*

Proof. In order for a candidate subtree st to qualify as a FAST in DT, it has to pass the Frequency_Count test. The Frequency_Count procedure checks if a rooted phylogenetic tree in DT supports the subtree st based on the partition metric, whose correctness is obvious and, hence, the lemma is proved. □

Lemma 3.3. *Phylominer is complete. That is, it does not miss any FAST in the given set of rooted phylogenies DT.*

Proof. We prove this lemma by mathematical induction.

Base step. Clearly, Phylominer finds all frequent agreement 1-leaf and 2-leaf trees, because it generates all such trees using a brute force enumeration method.

Hypothesis step. Assume the lemma holds for FASTs with k leaves, that is, all such trees can be found by Phylominer.

Induction step. We want to show that Phylominer does not miss any FAST with $k+1$ leaves. It suffices to prove that any frequent agreement $(k+1)$-leaf tree can always be generated by two frequent agreement k-leaf trees in some equivalence class.

Suppose a tree cnt of $k+1$ leaves is a FAST in canonical form. (We can canonicalize 2-leaf trees, and any $(k+1)$-leaf tree, $k \geq 2$, generated from the canonical 2-leaf trees is in canonical form.) We will show that cnt cannot be missed by the candidate generation step in Phylominer. Let cnt_{hlp} and cnt_{shlp} be k-leaf trees obtained by pruning the heaviest leaf and the second heaviest leaf from cnt, respectively. Obviously, both cnt_{hlp} and cnt_{shlp} are in their canonical forms according to Properties 1 and 2 in Section 3.1. From the downward closure theory [34], if cnt is frequent, cnt_{hlp} and cnt_{shlp} must be frequent. By the induction hypothesis, Phylominer can find both cnt_{hlp} and cnt_{shlp}. Thus, the two trees are in F_k. Furthermore, these two trees are in the same equivalence class in EC_k. Based on the logic of Phylominer, the two trees cnt_{hlp} and cnt_{shlp} must be joined together by the Phylo_Join procedure. Since this procedure exhaustively considers all possible expansions of trees, cnt must be in the candidate set obtained by joining cnt_{hlp} and cnt_{shlp}. This completes the proof. □

Theorem 1. *Phylominer correctly finds all FASTs in the given set of rooted phylogenies DT.*

Proof. From Lemma 3.2 and Lemma 3.3 and the fact that Phylominer is based on a candidate generation and verification scheme, the theorem is proved. □

Theorem 2. *The time complexity of Phylominer is $O(|F|^2MN)$, where $|F|$ is the cardinality of the FAST set, M is the number of rooted phylogenetic trees in DT, and N is the cardinality of the leaf label set L.*

Proof. Let "a pair of joining" represent the joining two frequent agreement k-leaf trees to obtain candidate $(k+1)$-leaf trees and then calculating the support values of these $(k+1)$-leaf trees. From Lemma 1.1 in the Appendix, we know that it takes $O(k)$ time to join two frequent agreement k-leaf trees to form a $(k+1)$-leaf tree. We can generate at most four candidate $(k+1)$-leaf trees from the two k-leaf trees based on the case analyses in the Appendix. Checking if a candidate $(k+1)$-leaf tree occurs in a phylogenetic tree in DT takes $O(N)$ time. Thus, calculating the support value of the candidate $(k+1)$-leaf tree takes $O(MN)$ time. Therefore, the time involved in a pair of joining is $O(k+MN) \leq O(N+MN) = O(MN)$. There are at most $|F|^2$ valid pairs of joinings and, hence, the total time complexity of Phylominer is $O(|F|^2MN)$. □

Notice that this is a very pessimistic upper bound for two reasons. First, the actual number of rooted phylogenetic trees involved in the verification and frequency counting phase for each candidate subtree is much less than M. With the pattern size growing, the number of rooted phylogenetic trees that need to be checked against each pattern drops quickly. Second, the pairwise joining operation occurs only in the same equivalence class. Consequently, $|F|^2$ is a very loose upper bound for the number of joining operations. Notice also that this is a pseudopolynomial time algorithm, since $|F|$ is not an input parameter but a value derived from the output ($|F|$ is the total number of FASTs discovered from DT). To be more precise, the time complexity of Phylominer is dependent on the number of qualified patterns, which, in the worst case, is exponential with respect to N, the size of the label set L. Therefore, the algorithm requires exponential time in the worst case. In practice, however, the number of qualified patterns is much less, leading to a dramatically low time complexity.

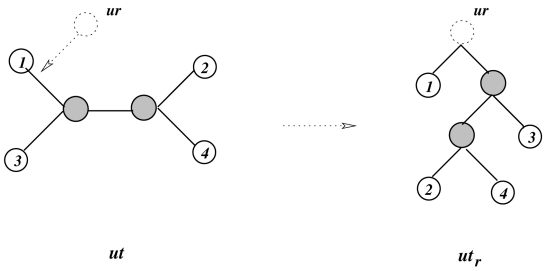

Fig. 8. An unrooted phylogeny and its rooted canonical form.

3.8 Extension to Unrooted Tree Mining

Some phylogeny reconstruction algorithms such as most parsimony and maximum likelihood methods [22], [25] produce unrooted trees, which are also known as undirected acyclic graphs [29]. In this section, we extend our rooted tree mining method to handle unrooted trees. The definitions of FASTs and MASTs for unrooted phylogenetic trees are similar to those for rooted phylogenetic trees given in Section 2. Our unrooted tree mining algorithm, called UPhylominer, works as follows: Given an unrooted tree ut, UPhylominer transforms ut to a rooted tree ut_r by adding a new root to the edge connecting ut's leaf with the smallest label and its neighboring node. (As in rooted phylogenetic trees, the leaves of unrooted phylogenies are labeled with integers that can be sorted.) This procedure of transforming an unrooted phylogenetic tree to a rooted phylogenetic tree is known as *rooting a phylogeny* (see [22] for additional techniques of rooting phylogenetic trees). Fig. 8 shows how an unrooted phylogenetic tree ut, after receiving a new root ur, is transformed to its corresponding rooted phylogeny ut_r in canonical form.

As in Phylominer, UPhylominer adopts a candidate generation and verification scheme for unrooted tree mining. Given two unrooted k-leaf trees, UPhylominer first transforms them to their rooted canonical forms and then uses the Phylo_Join procedure described in Section 3.5 to join the two rooted canonical forms (k-leaf trees) to obtain rooted $(k+1)$-leaf trees. For each rooted $(k+1)$-leaf tree t, UPhylominer transforms it back to an unrooted $(k+1)$-leaf tree as follows: If t's root has two children, the root is removed, and the two children are connected by an edge. If t's root has more than two children, the root becomes an

TABLE 1
Parameters and Default Values Used in the Experiments

Notation	Parameter	Value
$\|DT\|$	The size of the dataset DT	800
$\|L\|$	The cardinality of the leaf label set L	15
LF	The largest fanout of a node	5
SF	The smallest fanout of a node	2
$minsup$	The minimum support for subtrees of interest	30%

internal node, and all the children of the root become the neighbors of the internal node. This procedure is known as *unrooting a phylogeny* [22]. To check whether an unrooted $(k+1)$-leaf tree occurs in a given unrooted phylogenetic tree, UPhylominer calculates the partition metric value between them, as described in Section 3.6.

Recall that in the rooted tree case, the initial set of FASTs consists of all 1-leaf and 2-leaf trees, which are obtained through a brute-force enumeration method. Here, in addition to the enumerated 1-leaf and 2-leaf trees, UPhylominer enumerates all 3-leaf trees by combinatorially assigning three different labels from the leaf label set L to the unlabeled 3-leaf tree skeleton and includes these 3-leaf trees in the initial set. All 3-leaf trees have the same, fixed topology, namely, a star; all of these 3-leaf trees have a support value of 100 percent and must be FASTs in the given set of unrooted phylogenies DT. The reason is that all trees in DT have exactly the same leaves from L. Consider an unrooted 3-leaf tree st with three leaves l_1, l_2, and l_3 and an unrooted input tree t in DT. After pruning all leaves, except l_1, l_2, and l_3 from t, the remaining tree t' is isomorphic to st. Thus, st is a subtree of t, that is, st occurs in t. Since all input trees in DT have the same leaves, st is a subtree of all the input trees and hence has a support value of 100 percent.

With the unrooted 1-leaf, 2-leaf, and 3-leaf trees generated, UPhylominer then produces unrooted k-leaf trees, $k \geq 4$, based on the candidate generation and verification scheme, as described above. As in the rooted tree case, when a new unrooted candidate subtree ut is generated, there is no need to check whether or not ut is already generated previously, since each particular candidate subtree can be generated only once.

Theorem 3. *UPhylominer correctly finds all FASTs in the given set of unrooted phylogenies DT.*

Proof. It is clear that any unrooted subtree discovered by UPhylominer is a FAST in DT. We prove the completeness of UPhylominer by mathematical induction.

Base step. Clearly, UPhylominer finds all FASTs with p leaves, $p \leq 3$, since it uses a brute force method to enumerate all these subtrees.

Hypothesis step. Assume UPhylominer does not miss any FAST with k leaves.

Induction step. We want to show that UPhylominer does not miss any FAST with $k+1$ leaves. Assume, for contradiction, that an unrooted subtree ut with $k+1$ leaves is missed. Consider the rooted canonical form of ut; call it ut_r. Let ut_{hlp} (ut_{shlp}, respectively) be the unrooted k-leaf tree obtained by pruning the heaviest leaf (the second heaviest leaf, respectively) of ut_r from ut. Since ut is a FAST, both ut_{hlp} and ut_{shlp} must be FASTs. By the induction hypothesis, UPhylominer can find both ut_{hlp} and ut_{shlp}. In generating candidate $(k+1)$-leaf subtrees, UPhylominer transforms ut_{hlp} and ut_{shlp} to their rooted canonical forms (k-leaf trees) and uses the Phylo_Join procedure described in Section 3.5 to join these rooted trees. Since the Phylo_Join procedure exhaustively considers all cases of joining rooted k-leaf trees, it can generate the rooted $(k+1)$-leaf tree ut_r. Hence, UPhylominer can find ut, which contradicts the assumption. This completes the proof. □

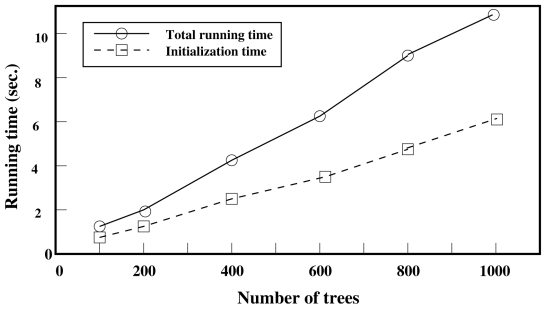

Fig. 9. Effect of the data set size on the runtime of Phylominer.

Theorem 4. *The time complexity of UPhylominer is $O(|F|^2MN)$, where $|F|$ is the cardinality of the FAST set, M is the number of unrooted phylogenetic trees in DT, and N is the size of the label set L.*

Proof. The theorem follows immediately by observing that rooting and unrooting a phylogeny take $O(N)$ time, and UPhylominer uses the same Phylo_Join and Frequency_Count procedures as Phylominer. □

4 Experiments and Results

4.1 Experimental Results on Synthetic Data

We conducted a series of experiments to evaluate the performance of the proposed algorithms. The experiments were performed on a machine with an AMD Athlon(tm) 64 × 2 Dual Core Processor 4200+ (2.20 GHz and 2.00 Gbytes of RAM) with Physical Address Extension running on the Linux operating system (Sabayon Linux distribution). The algorithms were implemented in C++ and compiled by g++ with the -O3 option on the Linux operating system. We also implemented an unordered tree generator in C++, which is similar to, but more powerful than, the one used in Page's COMPONENT tool [22]. COMPONENT generates binary leaf-labeled trees only, whereas our tree generator is able to produce leaf-labeled trees of various degrees by generalizing the algorithm described in [14]. The generated trees are rooted and can be treated as or transformed to unrooted trees through the procedure of unrooting phylogenetic trees described in Section 3.8. Table 1 lists the parameters and their default values used in the experiments, where the $fanout$ of a node is the number of children of that node. We also varied these parameter values in the experiments to evaluate their impact on the proposed algorithms.

Fig. 9 shows how changing the data set size affects the runtime of Phylominer. The 10 data sets generated for this experiment contained different numbers of trees ranging from 100 to 1,000. The other parameters used in the

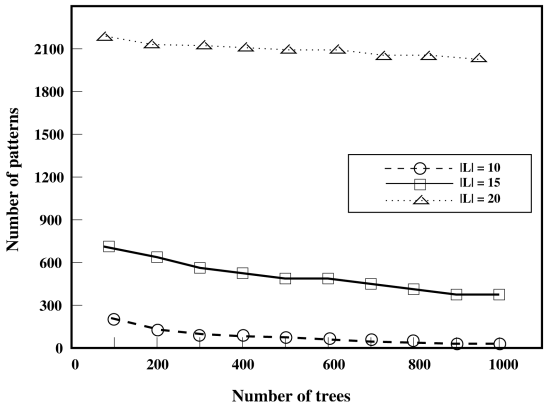

Fig. 10. Effect of the data set size on the number of FASTs discovered by Phylominer for varying $|L|$.

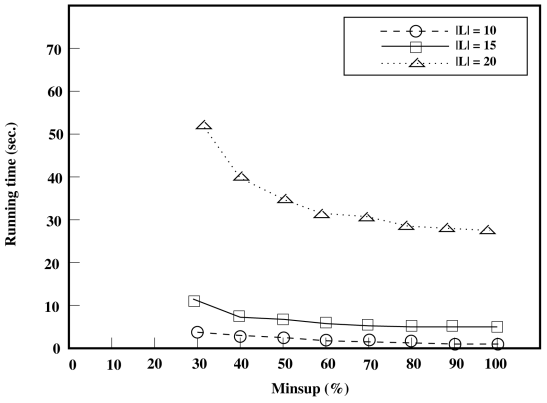

Fig. 12. Effect of $minsup$ on the runtime of Phylominer for varying $|L|$.

experiment had values, as shown in Table 1. It can be seen in Fig. 9 that the runtime of Phylominer scales up linearly with respect to the data set size. This happens because the more trees a data set has, the more time is needed for frequency counting in the data set. The dashed line in Fig. 9 shows that the time spent on the initialization phase of Phylominer also scales up linearly with respect to the data set size. The initialization phase essentially comprises two steps. One is to enumerate all 1-leaf and 2-leaf trees, where the number of these trees is related to the size of the leaf label set only, regardless of how many trees a data set has. The other step is to prepare the STID lists. The more trees a data set has, the more time is needed in preparing these STID lists. This is the reason why the initialization time of Phylominer scales up linearly with respect to the data set size.

Fig. 10 shows the numbers of FASTs obtained from the same experiment for different data set sizes and different cardinalities of the leaf label sets of trees. It can be seen from the figure that with the increasing number of trees in a data set, the number of qualified patterns decreases and eventually reaches a stable value. In general, the more trees a data set has, the less consensus information the data set contains and, hence, the fewer FASTs the data set has. On the other hand, although the number of FASTs with a large number of leaves could drop dramatically to zero due to the increasing number of trees in the data set, the initialization phase guarantees that the output of Phylominer contains at least all 1-leaf and 2-leaf trees, the number of which is a fixed value. This explains why the number of qualified patterns reaches a stable value. Notice that the number of qualified patterns can be exponential with respect to the cardinality of the leaf label set of input trees.

Fig. 11 shows how changing $minsup$ affects the number of FASTs discovered by Phylominer for varying $|L|$ values. The other parameters had the values, as shown in Table 1. It can be seen from the figure that as $minsup$ increases, the number of qualified patterns drops quickly. This happens because when $minsup$ increases, the number of qualified patterns with k-leaves, $k \geq 3$, decreases. Consequently, the number of qualified patterns with $k+1$ leaves decreases. This effect is cascadingly propagated from smaller subtrees to larger ones. Thus, the total number of qualified patterns decreases. It is observed that once the $minsup$ value reaches a certain point, 80 percent in this case, the number of qualified patterns reaches a stable value. This happens because the number of 1-leaf and 2-leaf trees contained in the input trees is always the same, regardless of what the $minsup$ values are, and the support values of these 1-leaf and 2-leaf trees are always 100 percent.

Fig. 12 shows how changing $minsup$ affects the runtime of Phylominer on the same data sets used in this experiment. The figure shows that as $minsup$ increases, the runtime of Phylominer drops quickly. This happens because the number of qualified patterns decreases with the increasing of $minsup$. Consequently, fewer valid pairwise joinings in each equivalence class are performed.

Similar performance results were obtained for UPhylominer and are omitted here. Table 2 compares the mining

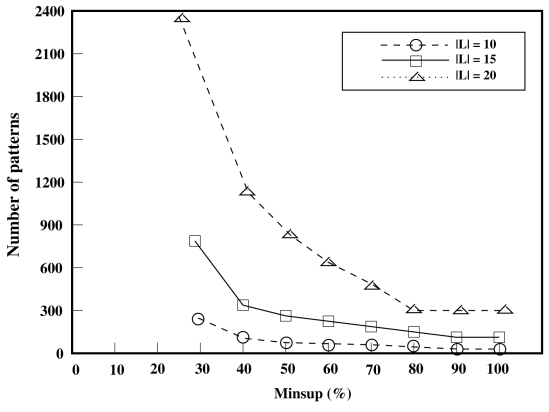

Fig. 11. Effect of $minsup$ on the number of FASTs discovered by Phylominer for varying $|L|$.

TABLE 2
Comparison of Mining Results between Rooted Trees and Unrooted Trees for the Same Set of Phylogenies

$minsup$	30%		50%		70%	
pattern size	r	u	r	u	r	u
1	15	15	15	15	15	15
2	105	105	105	105	105	105
3	284	455	22	455	8	455
4	2	517	0	203	0	48

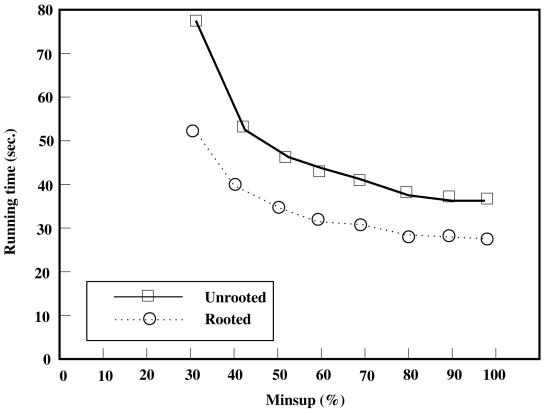

Fig. 13. Comparison of the runtimes of Phylominer and UPhylominer on the same set of phylogenies with respect to different $minsup$ values.

results obtained from the same data set DT with trees in DT being treated as rooted and unrooted, respectively. In the table, "r" represents the rooted tree case, whereas "u" represents the unrooted tree case. The data set DT was generated using the default settings in Table 1; the $minsup$ values were set to 30 percent, 50 percent, and 70 percent, respectively. Table 2 shows that with respect to the same $minsup$ value, fewer qualified patterns are found when the phylogenetic trees are treated as rooted ones. This happens because when the trees are rooted, a candidate k-leaf tree may match fewer phylogenetic trees in the data set due to the fact that more constraints are imposed in matching rooted trees. This situation is best illustrated by the following example in which the pattern size k is 3, and the leaf label set is $L = \{l_1, l_2, l_3\}$. There is only one unrooted 3-leaf tree st on L. The topology of st is a star. On the other hand, there are four rooted 3-leaf trees st_1, st_2, st_3, and st_4 on the same leaf label set L. Suppose two unrooted phylogenetic trees t_1 and t_2 in DT support st. Now, consider two rooted 3-leaf trees, say, st_1 and st_2. The tree t_1, when treated as a rooted tree, supports st_1 but not st_2. The tree t_2, when treated as a rooted tree, supports st_2 but not st_1. Thus, the support of st is larger than the support of either st_1 or st_2. In general, with a large $minsup$ (for example, $minsup \geq 50$ percent), the number of qualified patterns in unrooted trees is greater than that in rooted trees for the same set of phylogenies.

Fig. 13 compares the runtimes of Phylominer and UPhylominer on the data set DT for different $minsup$ values with trees in DT being treated as rooted and unrooted, respectively. It can be seen from the figure that UPhylominer requires more time than Phylominer in pattern discovery as $minsup$ increases. This result is consistent with those in Table 2—when $minsup$ is large, with respect to the same $minsup$ value, UPhylominer finds more qualified patterns and hence has a higher time complexity than Phylominer; cf., Theorems 2 and 4. We have run Phylominer and UPhylominer on different data sets with different trees and $minsup$ values, and the qualitative conclusion is the same.

4.2 Experimental Results on TreeBASE Data

TreeBASE [23] is a relational database of phylogenetic information, storing phylogenetic trees and the data matrices used to generate the trees taken from published research articles. We applied Phylominer to the 12 rooted phylogenetic trees obtained from the study *S*324 [19] stored in TreeBASE. The 12 trees are constructed by biologists based on 21 species, namely, Gonioctena_viminalis, Gonioctena_holdausi, Gonioctena_occidentalis, Gonioctena_rufipes, Gonioctena_linnaeana, Gonioctena_kamikawai, Gonioctena_tredecimmaculata, Gonioctena_rubripennis, Gonioctena_olivacea, Gonioctena_variabilis, Gonioctena_interposita_1, Gonioctena_interposita_2, Gonioctena_pallida_1, Gonioctena_pallida_2, Gonioctena_intermedia, Gonioctena_quinquepunctata, Gonioctena_fornicata_a, Gonioctena_fornicata_b, Gonioctena_nigroplagiata, Oreina_cacaliae, and Chrysomela_tremula. The 12 trees have exactly the same leaves, namely, the 21 species; each of the 12 trees depicts a hypothesis about the evolutionary history of the 21 species. It is worth noting that two of the 12 trees are identical, indicating that two different tree reconstruction methods in fact infer the same phylogeny for the 21 species. To adapt this data to our discovery framework, we used integer numbers, ranging from 1 to 21, to represent the 21 species and assigned a unique identification number to each of the 12 trees. In TreeBASE, this is one of the large families of phylogenetic trees that have the same leaves.

TABLE 3
Data Mining Results on the 12 Rooted Phylogenetic Trees Obtained from the Study *S*324 in TreeBASE

$minsup$	Time (sec.)	Number of FASTs	Number of MFASTs	Size of MFASTs
100%	13	10,459	36	10
80%	79	34,987	12	12
60%	511	114,715	46	13

Table 3 summarizes the experimental result. From the table, it can be seen that with $minsup$ decreasing, the runtime of Phylominer increases, and the total number of qualified patterns increases as well. This result is consistent with the results obtained from the synthetic data described in Section 4.1. The distribution of numbers of qualified patterns follows the combinatorial mathematics calculation of the power set of a set. For example, when $minsup$ is 80 percent, there are a total of 34,987 FASTs; the number of these subtrees, with size being 1 to 12, is 21, 210, 987, 2,821, 5,514, 7,725, 7,871, 5,775, 2,931, 950, 170, and 12, respectively. By comparing Table 3 with Figs. 11 and 12, we see that the number of FASTs in the real phylogenetic trees is much larger than that in the synthetic data. For the real phylogenetic trees in the same family, many of the trees agree with each other and, hence, the size of their MFASTs tends to be large. Consequently, the number of smaller patterns tends to grow exponentially. In contrast, randomly generated trees tend to differ from each other significantly. The size of their MFASTs is smal, and, therefore, few patterns are generated, requiring much less time than the real phylogenetic trees.

Since mining for FASTs is a natural extension of the MAST problem studied in computational phylogenetics, it is interesting to compare the MFASTs found by Phylominer with those calculated by the MAST algorithms [2], [5], [10], [12]. In this data set, Phylominer finds 36 MFAST/MAST

patterns (an MFAST is a MAST when $minsup$ is set to 100 percent), each having 10 leaves. We compared the results with those obtained from the MAST program developed in [5]. To our knowledge, this is the best program for solving the MAST problem. The results from both Phylominer and the MAST program are exactly the same in terms of the 36 MFAST/MAST patterns. Phylominer is practically faster than the MAST program when $minsup$ is set to 100 percent for this data set (13 sec. versus 24 sec.). One reason is that the MAST program is implemented in Perl and deals with string labels directly, whereas Phylominer is implemented in highly efficient C++ and encodes string labels (species names) using integer values.

The space complexity of the MAST program is $O(KN)$, where K is the total number of MAST patterns, and N is the cardinality of the leaf label set L. The space complexity of Phylominer is $O(|F|MN)$, where $|F|$ is the cardinality of the FAST set, and M is the number of input phylogenetic trees. This complexity is obtained based on the observation that a FAST has at most N labels, and its STID list contains at most M trees.

Notice that the 36 MFAST/MAST patterns are only a small portion of the 10,459 FAST patterns found by Phylominer when $minsup$ is set to 100 percent. When $minsup$ is set to 60 percent, Phylominer finds 114,715 FAST patterns, among which there are 46 MFAST patterns with 13 leaves. These 46 MFAST patterns reveal many evolutionary relationships between the 21 species that are suggested in the literature [19] but not revealed by the 36 MAST patterns. These results are consistent with those from the example in Fig. 1. This finding indicates once again that the FAST patterns discovered by our data mining-based approach, Phylominer, are more important than the patterns detected by the nondata mining-based MAST algorithms [2], [5], [10], [12].

The proposed algorithms require that the user input a support value $minsup$. In practice, it is suggested that the user set the support threshold to a reasonably large value (for example, $minsup = 50$ percent), assuming that the data trees under analysis tend to be congruent with each other. Then, use a strategy similar to "divide and conquer" or "binary search" to try different threshold values depending on the number of patterns found in a data set. For example, if there are too many patterns, try a support value of 75 percent; if there are too few patterns, try a support value of 25 percent. The discovered patterns could be used for phylogeny clustering, for example, to construct phylogenetic islands, which is useful in tree surfing [23].

5 Conclusion

We presented Phylominer and UPhylominer for discovering FASTs from rooted and unrooted phylogenies, respectively. To our knowledge, these algorithms are the first data mining techniques for finding interesting and important patterns in multiple phylogenetic trees. The algorithms find many applications in computational phylogenetics. For example, when $minsup$ is set to 100 percent, the MFAST patterns found by our algorithms are exactly MAST patterns. Therefore, with $minsup = 100$ percent, our algorithms can be used to verify other nondata mining-based MAST algorithms. The proposed algorithms can be easily modified to include other user-specified parameters. For example, one can incorporate a parameter $maxsize$ into the algorithms, which specifies the maximum size of patterns of interest. With $maxsize = 4$, UPhylominer is able to find all frequent agreement quartets, which can be used to define the quartet metric between unrooted phylogenetic trees [22], [27]. One can also include another parameter $maxnumber$, which indicates the maximum number of patterns of interest. Our programs will terminate when detecting the number of patterns found so far equals $maxnumber$.

Our algorithms are an upward extension of the MAST algorithms [2], [5], [10], [12], which assume that all phylogenetic trees in a data set are on the same leaf label set. We can relax this assumption by replacing the brute force enumeration method in the initialization phase of Phylominer and UPhylominer with an inverted list technique. For example, when a data set contains unrooted trees of different sizes with different leaf labels, a 3-leaf star tree may no longer be a FAST in the data set. Under this circumstance, we need to modify UPhylominer to obtain frequent agreement 1-leaf trees, 2-leaf trees, and 3-leaf trees through intersecting their inverted lists. These modified tree mining algorithms would be useful in building supertrees of phylogenies [22], [27]. The FAST patterns found by the proposed algorithms could also be used to build phylogenetic islands [18].

The difficulty of the FAST problem, compared with other structured data mining problems such as mining frequent patterns in rooted ordered trees surveyed in Section 1.1, is that the agreement subtrees to be mined for must satisfy properties related to phylogeny. From the tree mining viewpoint, the agreement subtrees in leaf-labeled phylogenetic trees are strictly embedded unordered subtrees, which, to our knowledge, cannot be effectively discovered by the already known algorithms for general trees [7], [34]. One could modify the existing methods by preprocessing and postprocessing pattern trees and apply the modified methods to the FAST problem. However, that would increase inaccuracy and inefficiency in the whole mining process.

The proposed algorithms are based on the Apriori method for frequent pattern mining. It is well known that Apriori is inefficient with large databases because of repeated database scanning. Recent work on FP-trees and FP-growth [13], [26] is intended to avoid redundant database scanning. These methods either avoid candidate generation or generate a limited number of candidate patterns by employing new data structures in the mining process. To apply the FP-growth method to the FAST problem would require new schemes for encoding phylogenetic trees using strings, as well as new data structures such as look-up tables based on hashing functions for guiding the candidate generation process.

In the future, we plan to

1. apply Phylominer and UPhylominer to multiple phylogenies built from different species and study the biological significance of discovered patterns,

2. apply the tree mining methods to tree classification [23], supertree inference [24], and phylogenetic island construction [12],
3. explore alternative strategies (for example, the FP-growth method) for phylogenetic tree mining, and
4. extend the techniques to find frequent substructures in phylogenetic networks [20].

We expect the proposed algorithms to be useful in not only computational phylogenetics but also other domains where data can be modeled as unordered trees.

APPENDIX

CASE ANALYSES OF JOINING FREQUENT AGREEMENT SUBTREES

In this Appendix, we present details concerning how to join FASTs. Joining two frequent agreement k-leaf trees is actually implemented by joining their heaviest subtrees. Although the two k-leaf trees must be in the same equivalence class (cf., line 4 in Fig. 7), their heaviest subtrees may have different sizes and hence may not be in the same equivalence class. (Trees in the same equivalence class must have the same size.) In joining the two k-leaf trees, we need to separate the heaviest subtrees from the two k-leaf trees, respectively. Then, join the two heaviest subtrees to obtain a larger subtree t. Finally, glue t back to the smaller one of the complementary trees of the two heaviest subtrees to obtain a new candidate $(k+1)$-leaf tree.

Depending on what kind of topological relationships the heaviest subtrees of the two k-leaf trees have, there are two cases in which the joining operations are performed differently. In each case, joining two k-leaf trees can produce at most four different candidate $(k+1)$-leaf trees.

- **Case 1.** When the heaviest subtrees of the two k-leaf trees have the same topology, there are two subcases.

  Case 1.1. When both heaviest subtrees of the two k-leaf trees are binary trees, four potential candidates can be generated. In Newick notation, we can use $st_1 = (lt, hl_1)$ and $st_2 = (lt, hl_2)$ to represent the heaviest subtrees of the two k-leaf trees, respectively, where hl_1 and hl_2 are the heaviest leaves of the two k-leaf trees, respectively. lt denotes the left subtree in each of the two heaviest subtrees (the left subtree lt in st_1 must be equivalent to the left subtree lt in st_2). Obviously, in the expanded candidate subtree, hl_1 and hl_2 could be siblings. Two possible candidates having hl_1 and hl_2 as siblings are denoted by $j_{[1]} = (lt, (smaller(hl_1, hl_2), greater(hl_1, hl_2)))$ and $j_{[2]} = (lt, smaller(hl_1, hl_2), greater(hl_1, hl_2))$, respectively. Here, $greater(hl_1, hl_2)$ and $smaller(hl_1, hl_2)$ return whichever is greater and smaller, respectively, between the two integer labels representing the heaviest leaves in the two k-leaf trees. (Each leaf has an integer label and, hence, we can compare two leaves by comparing the corresponding integers.) Notice that we order the return values of the smaller and greater functions to assure that the newly generated candidate subtree is automatically present in its canonical form. Examples of $j_{[1]}$ and $j_{[2]}$ are illustrated by the 4-leaf trees j_{4-1} and j_{4-2}, respectively, in Fig. 14. Notice that in tree $j_{[1]}$ (that is, j_{4-1}), putting hl_1 and hl_2 as siblings introduces a new internal node in $j_{[1]}$.

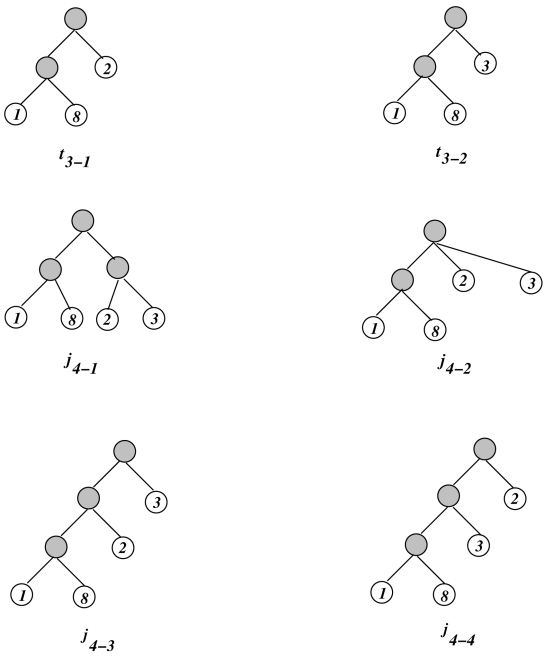

Fig. 14. An example for Case 1.1, which shows that joining t_{3-1} and t_{3-2} can produce at most four candidates j_{4-1}, j_{4-2}, j_{4-3}, and j_{4-4}.

Another way to perform the joining operation on two k-leaf trees is to take one tree as the skeleton, which will then be expanded by adding the heaviest leaf of the other tree to get a $(k+1)$-leaf tree. From Section 2, we know that pruning a leaf from a tree may introduce an edge contraction. It is easy to see that, as the reverse operation of the edge contraction, adding a new leaf to a tree may introduce an additional internal node. Thus, two additional candidates that can be generated are $j_{[3]} = ((lt, hl_1), hl_2)$ and $j_{[4]} = ((lt, hl_2), hl_1)$. Examples of $j_{[3]}$ and $j_{[4]}$ are illustrated by the two 4-leaf trees j_{4-3} and j_{4-4}, respectively, in Fig. 14.

Case 1.2. When both the heaviest subtrees of the two k-leaf trees are multiforked trees, two potential candidates can be generated. Suppose that $(st_1, \ldots, st_m, hl_1)$ and $(st_1, \ldots, st_m, hl_2)$ are the heaviest subtrees of the two k-leaf trees, respectively, where $st_1, \ldots st_m$ are the m sibling subtrees of hl_1 and hl_2, respectively, ($m \geq 2$ since the two heaviest subtrees are multiforked). The expanded candidates can be in either the form of

$$j_{[1]} = (st_1, \ldots, st_m, (smaller(hl_1, hl_2), greater(hl_1, hl_2)))$$

or the form of

$$j_{[2]} = (st_1, \ldots, st_m, smaller(hl_1, hl_2), greater(hl_2, hl_1)).$$

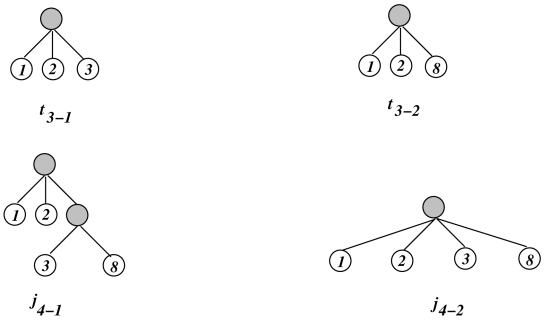

Fig. 15. An example for Case 1.2, which shows that joining t_{3-1} and t_{3-2} can produce at most two candidates j_{4-1} and j_{4-2}.

Examples of $j_{[1]}$ and $j_{[2]}$ are illustrated by the two 4-leaf trees j_{4-1} and j_{4-2}, respectively, in Fig. 15.

It should be pointed out that the two expansions in Case 1.2 are similar to the first two expansions in Case 1.1. However, the latter two expansions in Case 1.1 are no longer applicable in Case 1.2. Assume that the third expansion considered in Case 1.1 would also be applicable in Case 1.2. Thus, for example, referring to Fig. 15, the expanded subtree would be ((1, 2, 3), 8), which would support (1, 2, 3) but not (1, 2, 8). This is because by pruning 3 from the imaginary tree ((1, 2, 3), 8), the resulting subtree would be ((1, 2), 8), not (1, 2, 8). Thus, the third expansion is impossible. A similar argument prohibits the fourth expansion in Case 1.1 from being considered in Case 1.2.

- **Case 2.** When the heaviest subtrees of the two k-leaf trees have different topologies, only one candidate $(k+1)$-leaf tree can be generated. Since the two heaviest subtrees are different from each other, one of them is identified as the larger tree, and the other one as the smaller tree. Formally, let $h(t)$ and $s(t)$ denote the depth and the size of a tree t, respectively. Given two heaviest subtrees t_1 and t_2, t_1 is said to be larger than t_2, if either of the following rules hold.

  Rule 1. $h(t_1) > h(t_2)$. This means the depth of t_1 is greater than that of t_2.

  Rule 2. $s(t_1) > s(t_2)$. This case can happen only when $h(t_1) = h(t_2)$. Note that, in this case, the fanout of the root of t_2 must be 2.

  Let t_1 and t_2 be denoted by (t_{1hlp}, hl_1) and (t_{2hlp}, hl_2), respectively. When t_1 is larger than t_2, hl_1 will be the heaviest leaf in the expanded subtree. There must exist a subtree lst in t_{1hlp} that is isomorphic to t_{2hlp}. We replace lst by t_2. This joining operation can be easily understood if the larger tree t_1 is taken as an umbrella under which a part of t_1 is replaced by the entire smaller tree t_2. Figs. 16 and 17 show examples for Rule 1 and Rule 2, respectively.

It is clear that with the two k-leaf trees being in their canonical forms, the newly generated $(k+1)$-leaf tree must be in canonical form. This automatic canonicalization property is a main factor contributing to the efficiency of the proposed Phylominer algorithm.

Lemma 1.1. *The time complexity of joining two k-leaf trees to form a $(k+1)$-leaf tree is $O(k)$.*

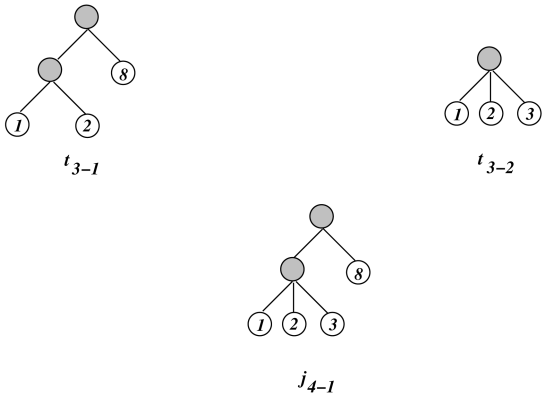

Fig. 16. An example for Rule 1 of Case 2, which shows that joining t_{3-1} and t_{3-2} can produce only one candidate j_{4-1}.

Proof. The joining operation is performed on the Newick strings of the two k-leaf trees with $O(k)$ length. In joining the two k-leaf trees, we need to separate the heaviest subtrees from the two k-leaf trees, respectively. Then, join the two heaviest subtrees to obtain a larger subtree t. Finally, glue t back to the smaller one, ct, of the complementary trees of the two heaviest subtrees to obtain a new candidate $(k+1)$-leaf tree. Separating the heaviest subtree from a k-leaf tree takes $O(k)$ time, since the operation is to extract a substring from a Newick string, which can be done in linear time. Gluing t to ct takes linear time, as it can be accomplished by a substring replacement operation. Thus, the operations used in the joining procedure are string parsing, string extraction, string concatenation, and string replacement, all of which can be done in $O(k)$ time. The lemma is thus proved. □

ACKNOWLEDGMENTS

This work was supported in part by US NSF Grant IIS-9988636. The authors thank the anonymous reviewers for their constructive suggestions, which helped improve the content and presentation of this paper. They also thank Drs. Debashish Bhattacharya, Katherine Herbert, William Piel, Usman Roshan, and David Stockwell for helpful conversations during the preparation of this paper, and Dr. Vincent Berry for his responding to our request for the source code of his MAST program and exchanging interesting ideas on the FAST problem with us.

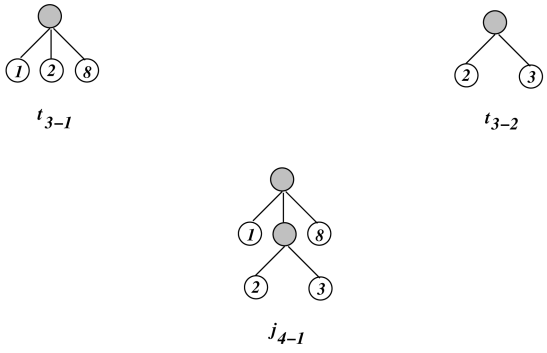

Fig. 17. An example for Rule 2 of Case 2, which shows that joining t_{3-1} and t_{3-2} can produce only one candidate j_{4-1}.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases,* pp. 487-499, 1994.

[2] A. Amir and D. Keselman, "Maximum Agreement Subtree in a Set of Evolutionary Trees," *SIAM J. Computing,* vol. 26, no. 6, pp. 1656-1669, 1997.

[3] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa, "Efficiently Mining Frequent Substructures from Semi-Structured Data," *Proc. Int'l Workshop Information and Electrical Eng.,* pp. 59-64, 2002.

[4] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering Frequent Substructures in Large Unordered Trees," *Proc. Sixth Int'l Conf. Discovery Science,* 2003.

[5] V. Berry and F. Nicolas, "Improved Parameterized Complexity of the Maximum Agreement Subtree and Maximum Compatible Tree Problems," *IEEE/ACM Trans. Computational Biology and Bioinformatics,* vol. 3, no. 3, pp. 289-302, July-Sept. 2006.

[6] Y. Chi, S. Nijssen, R.R. Muntz, and J.N. Kok, "Frequent Subtree Mining—An Overview," *Fundamenta Informaticae,* special issue on graph and tree mining, 2005.

[7] Y. Chi, Y. Xia, Y. Yang, and R.R. Muntz, "Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 2, pp. 190-202, Feb. 2005.

[8] Y. Chi, Y. Yang, and R.R. Muntz, "Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining," *Knowledge and Information Systems,* vol. 8, no. 2, pp. 203-234, 2005.

[9] W.H.E. Day, "Optimal Algorithms for Comparing Trees with Labeled Leaves," *J. Classification,* vol. 1, pp. 7-28, 1985.

[10] M. Farach, T. Przytycka, and M. Thorup, "On the Agreement of Many Trees," *Information Processing Letters,* vol. 55, no. 6, pp. 297-301, 1995.

[11] C.R. Finden and A.D. Gordon, "Obtaining Common Pruned Trees," *J. Classification,* vol. 2, pp. 255-276, 1985.

[12] G. Ganeshkumar and T. Warnow, "Finding a Maximum Compatible Tree for a Bounded Number of Trees with Bounded Degree Is Solvable in Polynomial Time," *Proc. First Int'l Workshop Algorithms in Bioinformatics,* pp. 156-163, 2001.

[13] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery,* vol. 8, no. 1, pp. 53-87, 2004.

[14] S. Holmes and P. Diaconis, "Random Walks on Trees and Matchings," *Electronic J. Probability,* vol. 7, 2002.

[15] J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," *Proc. Third IEEE Int'l Conf. Data Mining,* pp. 549-552, 2003.

[16] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," *Proc. First IEEE Int'l Conf. Data Mining,* pp. 313-320, 2001.

[17] J.T. Li, A.L. Bogle, A.S. Klein, and M.J. Donoghue, "Phylogeny and Biogeography of Hamamelis (Hamamelidaceae)," *Harvard Papers in Botany,* vol. 5, pp. 171-178, 2000.

[18] D.R. Maddison, "The Discovery and Importance of Multiple Islands of Most-Parsimonious Trees," *System Zoology,* vol. 40, pp. 315-328, 1991.

[19] P. Mardulyn, M.C. Milinkovitch, and J.M. Pasteels, "Phylogenetic Analyses of DNA and Allozyme Data Suggest that Gonioctena Leaf Beetles (Coleoptera: Chrysomelidae) Experienced Convergent Evolution in Their History of Host-Plant Family Shifts," *Systematic Biology,* vol. 46, no. 4, pp. 722-747, 1997.

[20] B.M.E. Moret, L. Nakhleh, T. Warnow, C.R. Linder, A. Tholse, A. Padolina, J. Sun, and R. Timme, "Phylogenetic Networks: Modeling, Reconstructibility, and Accuracy," *IEEE/ACM Trans. Computational Biology and Bioinformatics,* vol. 1, no. 1, pp. 13-23, Jan.-Mar. 2004.

[21] S. Nijssen and J.N. Kok, "Efficient Discovery of Frequent Unordered Trees: Proofs," technical report, Leiden Inst. of Advanced Computer Science, Jan. 2003.

[22] R.D.M. Page, "COMPONENT User's Manual (Release 1.5)," Univ. of Auckland, 1989.

[23] W.H. Piel, M.J. Donoghue, and M.J. Sanderson, "TreeBASE: A Database of Phylogenetic Information," *Proc. Second Int'l Workshop of Species,* 2000.

[24] C. Semple and M. Steel, "A Supertree Method for Rooted Trees," *Discrete Applied Math.,* vol. 105, pp. 147-158, 2000.

[25] D. Shasha, J.T.L. Wang, and S. Zhang, "Unordered Tree Mining with Applications to Phylogeny," *Proc. 20th Int'l Conf. Data Eng.,* pp. 708-719, 2004.

[26] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi, "Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining," *Proc. Eighth Pacific-Asia Conf. Knowledge Discovery and Data Mining,* May 2004.

[27] J.T.L. Wang, H. Shan, D. Shasha, and W.H. Piel, "Fast Structural Search in Phylogenetic Databases," *Evolutionary Bioinformatics,* vol. 1, pp. 37-46, 2005.

[28] J.T.L. Wang, B.A. Shapiro, D. Shasha, K. Zhang, and K.M. Currey, "An Algorithm for Finding the Largest Approximately Common Substructures of Two Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 20, no. 8, pp. 889-895, Aug. 1998.

[29] J.T.L. Wang, K. Zhang, G. Chang, and D. Shasha, "Finding Approximate Patterns in Undirected Acyclic Graphs," *Pattern Recognition,* vol. 35, no. 2, pp. 473-483, 2002.

[30] T. Washio and H. Motoda, "State of the Art of Graph-Based Data Mining," *ACM SIGKDD Explorations,* vol. 5, no. 1, July 2003.

[31] Y. Xiao, J. Yao, Z. Li, and M. Dunham, "Efficient Data Mining for Maximal Frequent Subtrees," *Proc. IEEE Int'l Conf. Data Mining,* 2003.

[32] X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* 2003.

[33] L. Yang, M.L. Lee, and W. Hsu, "Efficient Mining of XML Query Patterns for Caching," *Proc. 29th Int'l Conf. Very Large Databases,* 2003.

[34] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications," *IEEE Trans. Knowledge and Data Eng.,* special issue on mining biological data, W. Wang and J. Yang, eds., vol. 17, no. 8, pp. 1021-1035, Aug. 2005.

[35] S. Zhang and J.T.L. Wang, "Mining Frequent Agreement Subtrees in Phylogenetic Databases," *Proc. Sixth SIAM Int'l Conf. Data Mining,* pp. 222-233, 2006.

[36] S. Zhang, K.G. Herbert, J.T.L. Wang, W.H. Piel, and D.R.B. Stockwell, "Phylominer: A Tool for Evolutionary Data Analysis," *Proc. 18th Int'l Conf. Scientific and Statistical Database Management,* pp. 129-132, 2006.

Sen Zhang received the PhD degree in computer science from the New Jersey Institute of Technology, Newark. He is an assistant professor in the Department of Mathematics, Computer Science, and Statistics, the State University of New York College at Oneonta. He is a member of the IEEE.

Jason T.L. Wang received the PhD degree in computer science from the Courant Institute of Mathematical Sciences, New York University. He is a professor of bioinformatics, information technology and computer science, New Jersey Institute of Technology, Newark. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.