Mining Frequent Agreement Subtrees in Phylogenetic Databases

Sen Zhang* Jason T. L. Wang†

Abstract

We present a new data mining problem to discover frequent agreement subtree patterns from a database of rooted phylogenetic trees. This problem is a natural extension of the traditional MAST (maximum agreement subtree) problem. To solve the problem, we first present a novel canonical form for leaf-labeled trees and an efficient tree expansion algorithm for generating candidate subtrees level by level. We then show how to efficiently discover all frequent agreement subtrees from a given set of phylogenetic trees, through an Apriori-like data mining approach. We discuss the correctness and completeness of the proposed method. Experimental results demonstrate that the proposed method can discover interesting patterns from different phylogenetic trees for multiple species. The algorithms were implemented in C++ and integrated into an online toolkit, which is fully operational and accessible on the World Wide Web.

1 Introduction

Different theories concerning the evolutionary history of the same set of species often result in different phylogenetic trees. This leads to a fundamental research problem in phylogenetics: how to determine to what extent two different hypothetical phylogenetic trees regarding the same set of taxa have in common. Traditionally, this problem can be partially answered by computing a maximum agreement subtree (MAST) of the two data trees. An agreement subtree between two trees T_1 and T_2 is a substructure of T_1 and T_2 on which the two trees are the same [2, 9, 10, 11]. A maximum agreement subtree (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 [10]. 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 a tree. Warnow *et al.* [11] later gave an $O(n^2)$ algorithm and Farach *et al.* [9] 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, 9]. For trees with unbounded degrees, this problem is NP-hard [2]. An observation is that a MAST of multiple trees is usually of small size and thus uninformative, especially when a large number of data trees are under consideration [11].

For example, a study $S497$ [20] in TreeBASE [25] shows that biologists built a set of five phylogenetic trees for six Hamamelis-related species. Each of the five trees depicts a hypothesis about the evolutionary history of the six species. The six species are shown in Table 1, and the five phylogenetic trees are shown in the first two rows in Figure 1. The bottom row of Figure 1 shows three subtree patterns: st_1, st_2 and st_3. Here, st_1 and st_2 are MASTs of the five phylogenetic trees, since they are subtrees of all the five phylogenies and no other subtrees occurring in all the five phylogenies have sizes larger than that of st_1 and st_2. The pattern st_3 is a subtree of three data trees only, namely t_1, t_3 and t_5, and therefore not a MAST of the five data trees. Nevertheless, in phylogenetics, st_3 is not necessarily less informative than st_1 or st_2 for two reasons: (i) the number of leaves of st_3 is prominently greater than that of the two MAST patterns st_1 and st_2; and (ii) st_3 occurs in a majority of the data trees. Motivated by this observation, we propose a new tree mining algorithm, called Phylominer, to automatically discover all frequent agreement subtrees from a given set of phylogenies, i.e., our algorithm will find out not only st_1 and st_2, but also st_3, when applied to the above example.

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. Almost in the same period, Zaki independently developed similar techniques capable of finding frequent embedded subtrees in a forest of rooted ordered trees [36]. Yang *et al.* [35] studied the tree mining problem in the context of XML management, by adapting the rightmost expansion scheme to solving a frequent XML query pattern

*Department of Mathematics, Computer Science and Statistics, State University of New York, College at Oneonta, Ravine Parkway, Oneonta, NY 13820, USA (zhangs@oneonta.edu)

†Bioinformatics Center and Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA (wangj@njit.edu)

Table 1: The six species in the study $S497$ stored in TreeBASE.

Species name	Label
Hamamelis_virginiana	1
Hamamelis_vernalis	2
Hamamelis_mexicana	3
Hamamelis_japonica	4
Fothergilla_major	5
Hamamelis_mollis	6

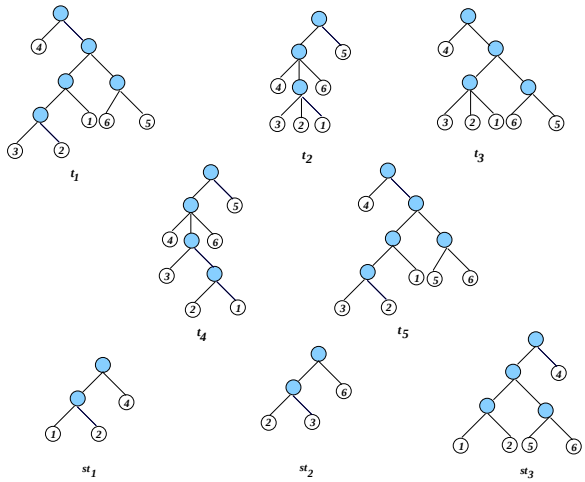

Figure 1: The five data trees obtained from the study $S497$ are displayed in the first two rows. Three subtrees are displayed in the bottom row, where st_1 and st_2 are MASTs, but st_3 is not.

discovery problem. Wang *et al.* [31] presented a dynamic programming algorithm for finding the consensus of two general trees, which was applied to motif finding in RNA secondary structures.

In the area of unordered tree mining, Xiao *et al.* [33] proposed an efficient frequent subtree mining solution through path joining operations. Asai *et al.* [4] and Nijssen *et al.* [23] independently discussed an essentially identical unordered tree enumeration technique for unordered subtree mining. More recently, Chi *et al.* [7] reported a set of more complete algorithms to find frequent induced subtrees in both rooted and unrooted unordered trees. Shasha *et al.* [29] developed methods to find cousin pairs in unordered trees with applications to phylogeny. For a comprehensive survey of tree mining methods and applications, the reader is referred to [5].

In parallel with the tree mining research, graph mining is a closely related field which has also been intensely studied during the past decade. Kuramochi and Karypis extended traditional frequent itemset algorithms [18] to find frequent patterns in graph data. Yan and Han [34] proposed a novel canonical graph form to find closed frequent subgraphs. Huan *et al.* devised a different canonical form to efficiently discover frequent subgraphs in the presence of isomorphism [14]. For the readers who are interested in the state of the art of graph mining, please refer to a survey paper [32] by Washio and Motoda.

Our work differs from the above approaches in two ways. First, in contrast to the general trees [3, 36, 23, 7] studied by previous researchers, we are concerned with leaf-labeled 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 different from the patterns found in all the previous tree mining papers. For example, Zaki's Treeminer is a powerful algorithm to mine embedded subtrees from ordered trees, but his embedded subtree definition is rather tolerant. By contrast, an agreement subtree in the phylogeny context is unordered and demands topological restrictions on valid embeddings. Chi's work is a recent breakthrough in unordered tree mining; however, his algorithms find induced subtrees from unordered trees defined in the generic tree scope, rather than embedded subtrees from leaf-labeled trees used in phylogeny research. Moreover, there are no straightforward ways to adapt previous methods to the problem which the proposed Phylominer is designed for. Our tree mining method thus joins the many others already developed [3, 36, 35, 31, 33, 4, 23, 7, 29]. The difference is that the agreement subtrees to be mined for are unordered and embedded. This makes the problem at hand unique, which is drastically different from previous subtree mining problems. Neither [37] nor [6] can find exactly the frequent agreement subtrees in multiple phylogenies as our algorithms do. Thus we present the first algorithm to tackle this problem in the interdisciplinary field of data mining and computational phylogenetics.

1.2 Novel Contributions of Phylominer The main contributions of this work are highlighted below:

- Proposes and formalizes a unique frequent agreement subtree mining problem for rooted leaf-labeled phylogenetic trees.
- Adopts an effective phylogeny-aware canonical form, which is for the first time to be used in phylogenetic tree mining to facilitate dealing with isomorphism problems.
- Introduces a phylogeny-aware subtree pattern expansion scheme.
- Designs a novel algorithm, Phylominer, which is useful for phylogeny research.

- Analyzes the correctness and complexity of Phylominer.

The rest of the paper is organized as follows. Section 2 presents basic concepts and terminologies. Section 3 describes the Phylominer algorithms. Section 4 shows the correctness and completeness of the algorithms. Section 5 presents experimental results. Section 6 briefly reports the implementation status. Section 7 concludes the paper.

2 Preliminaries

Let L denote a set of labels, with each label representing an evolutionary unit. An evolutionary unit under investigation 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 t on L is a rooted leaf-labeled unordered tree in which (i) each leaf is associated with a unique label drawn from L; (ii) all internal nodes have no labels; and (iii) a special node, denoted $r(t)$, is designated as the root of the tree. Furthermore, the fanout of each internal node is at least two. The *size* of the phylogeny t is the number of its leaves, which equals the cardinality of L. For convenience, a tree t with k leaves is also called a k-leaf tree.

Subtree. A tree st on SL is a *subtree* of t on L, if $SL \subset L$ and st can be obtained by restricting t to the leaf set SL through pruning all leaves $l \in L - SL$. Formally, the above definition can be 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 trees.

Notice that, pruning a leaf may trigger an edge contraction for satisfying the requirement that the fanout of any internal node must be at least 2. Mathematically, let N_t (N_{st}, respectively) represent the set of nodes in t (st, respectively). We say st is a subtree of t, if there exists a mapping 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 $u, v \in N_{st}$:

- $label(f(u)) = label(u)$ (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 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).

Figure 2 shows two different kinds of 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 while st_2 is an embedded subtree due to an edge contraction. Both situations can be handled by our algorithms.

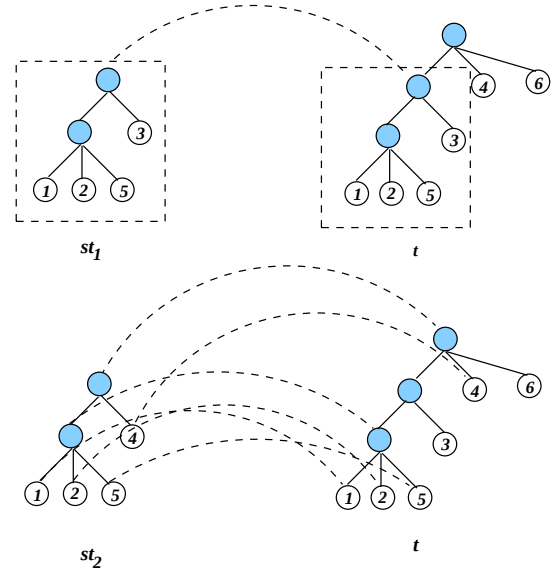

Figure 2: The subtree st_1 is an induced subtree of tree t with no edge contraction and the subtree st_2 is an embedded subtree of t due to edge contractions.

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. Then a leaf-labeled tree st on SL is an *agreement subtree* (or AST) for all $t_i \in DT$, if st is a subtree of every tree in DT, i.e., $t_1|_{SL} \equiv t_2|_{SL} \ldots \equiv t_m|_{SL} \equiv st$.

Maximum agreement subtree. If st has the maximum number of leaves among all agreement subtrees for DT, then st is a *maximum agreement subtree* (MAST) for DT. In Figure 1, both st_1 and st_2 are MASTs of the five data trees in the figure.

Frequent agreement subtree. A subtree or pattern p is said to be supported by a tree 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 $(\sum_{1 \leq i \leq m} supp_{p,i}/|DT|) \times 100\%$. An agreement subtree is *frequent* if its support is greater than or equal to a user-specified minimum support value, *minsup*. Our goal is to find all frequent agreement subtrees (FASTs) from a given set of phylogenies where the size, i.e. the number of the leaves, of the subtrees is greater than or equal to a user-specified parameter value *minsize* and the support of the subtrees is greater than or equal to *minsup*.

Maximum frequent agreement subtree. If st has the maximum number of leaves among all frequent agreement subtrees for DT, then st is a *maximum fre-*

quent agreement subtree (MFAST) for DT. Obviously, a MFAST might have more leaves than a MAST. In Figure 1, st_3 is a MFAST of the five data trees in the figure when the *minsup* is set to 50%.

It should be pointed out that the set of FAST patterns is a super set of the set of MAST patterns. The algorithm for FAST can find frequent agreement subtrees occurring in a small portion of a given database (e.g. with *support* = 10%), whereas the algorithm for MAST always finds the maximum agreement subtrees occurring in all trees in the database.

3 Frequent Agreement Subtree Discovery

To solve the tree isomorphism problem which is generally believed to make pattern mining in unordered trees more sophisticated than that in ordered trees, we propose a new canonical form for leaf-labeled trees. This canonical form will allow us to represent each leaf-labeled unordered tree by using one unique ordered tree. Based on this canonical form, we then introduce the concept of equivalence classes, suggesting an efficient candidate subtree generation strategy.

3.1 Canonical Form The basic assumption of 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, i.e., the ordering of L is $1 < 2 < \ldots < n < n+1 < \ldots$. Based on this leaf label ordering scheme, the canonical form of a leaf-labeled unordered tree t requires an assignment of virtual labels to all originally unlabeled internal nodes. To be more specific, given a phylogenetic tree t, each internal node of t will be assigned to a virtual label, which is the smallest integer label among all the integer labels of its child nodes. Once all internal nodes have obtained virtual labels, we can define a canonical form as follows.

The canonical form of a leaf-labeled unordered tree is a specially designed leaf-labeled ordered tree, in which all sibling nodes (including both leaf nodes and internal nodes) follow a normalized order, such that for every sibling pair (v, u), node v always appears before node u in the depth first traversal (DFT) order if $label(v) < label(u)$.

According to the above definition, it is not difficult to see that any rooted phylogenetic tree with N leaves can be transferred to its canonical form by a straightforward $O(N)$ DFT-like 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 transfer a tree to its canonical form is called a *normalization procedure*.

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

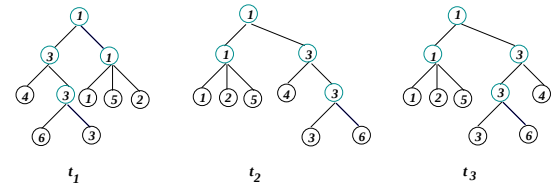

Figure 3: Three ordered representations of the same unordered tree.

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

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

Here, a direct pruning means a pruning of a leaf without further normalizing the resulting tree. In Section 3.4, it will be shown that the direct pruning properties suggest an efficient joining scheme regarding how to arrange the last leaves of two k-leaf subtrees to obtain $(k+1)$-leaf subtrees in their canonical form without going through further normalization.

Once we assign virtual labels to internal nodes and define the canonical form for leaf-labeled trees, the following terms can be clearly defined.

Weight scheme. After all internal nodes are labeled, every leaf $i \in n$ can be associated with a weight, denoted $w(i)$, which is an ordered label list by concatenating the labels of all the nodes along the path from the root to the leaf.

For example, the weights of the leaves of t_3 in Figure 3 are the following: $w(1)$ is "$1, 1, 1$", $w(4)$ is "$1, 3, 4$", $w(5)$ is "$1, 1, 5$", 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 Figure 3 is $w(4) > w(6) > w(3) > w(5) > w(2) > w(1)$.

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

$(k-1)$-prefix tree. Given any k-leaf tree t in its canonical form, we define its $(k-1)$-prefix tree to be the $(k-1)$-leaf subtree obtained by pruning the rightmost leaf (i.e. 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 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, i.e., they share the same $(k-1)$-prefix tree. The relation "sharing the same prefix tree with each other" for a set of subtrees is an equivalence relation, because the relation on these subtrees is reflective, symmetric and transitive. The equivalence relation partitions a set of k-leaf subtrees into disjoint subsets called equivalence classes. Consider the trees in Figure 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 (edge contraction), yielding $core_1$. 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 circle. Note also that in determining whether a subtree is isomorphic to another subtree, we take into account not only their topologies but also node labels in them.

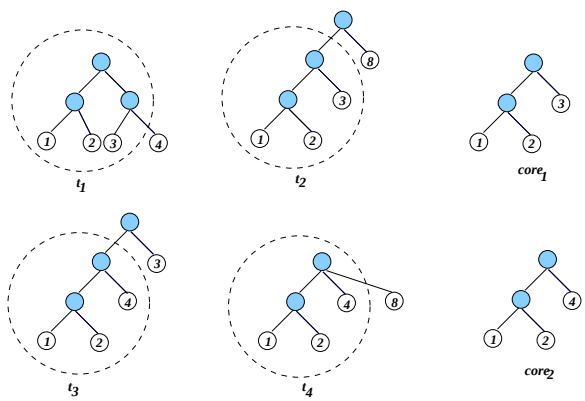

Figure 4: Four trees are grouped into two equivalence classes. Trees t_1 and t_2 are in the same equivalence class, while trees t_3 and t_4 are in another equivalence class.

The heaviest subtree. Given a tree t, the heaviest subtree of t, denoted by st_{hl}, is defined as the subtree rooted at the parent of the heaviest leaf of t. The heaviest subtree will be used to describe our candidate generation algorithm, where our main concern is how to join two heaviest subtrees. This is because when two trees are in the same equivalence class, their differences must have been restricted to the heaviest subtrees. Otherwise, they can not be in the same equivalence class.

Procedure: Phylominer(DT, δ, θ)
Input: DT, a set of phylogenetic trees.
δ, a global variable for *minsup*.
θ, a cutoff value of tree size.
Output: FST, a set of frequent subtrees.
1. $FST \leftarrow \emptyset$;
2. $F_2 \leftarrow$frequent 2-leaf subtrees;
3. $FST \leftarrow FST \cup F_2$;
4. $EClasses_2 \leftarrow$equivalence classes of F_2;
5. k = 2;
6. **while** ($k < \theta$ and $|F_k| \geq k+1$)
7. **begin**
8. F_{k+1} = Grow_Subtrees($EClasses_k$, k);
9. $FST = FST \cup F_{k+1}$;
10. $EClasses_{k+1} \leftarrow$equivalence classes of F_{k+1};
11. $k = k+1$;
12. **end;**
13. **return** FST;

Figure 5: Algorithm for finding frequent subtrees in a database of trees.

3.3 Algorithmic Framework The proposed Phylominer algorithm is an Apriori-like data mining method, which progressively enumerates all candidate subtrees from a set of phylogenetic trees [1, 36]. The algorithm is summarized in Figure 5. Initially, Phylominer enumerates all $\frac{|L|*|L-1|}{2}$ 2-leaf subtrees, which we can obtain by combinatorially assigning 2 different labels from L to a 2-leaf unlabeled tree skeleton. During each of the subsequent iterations, the algorithm calls the subroutine Grow_Subtrees (line 8 in Figure 5) to find frequent subtrees whose sizes are greater than the previous frequent subtrees by one leaf node. Thus, the core of this Apriori-like algorithm is how to design an efficient Grow_Subtrees algorithm to systematically generate candidates, level by level.

3.4 Candidate Generation Our candidate generation method adopts a pairwise joining scheme. In order for two frequent k-leaf trees to be eligible for further joining, the two subtrees must be in the same equivalence class. Since the nature of equivalence classes suggests a pattern expansion scheme through a rightmost joining approach (reminiscent of the rightmost extension schemes in [3, 36]), the focus of joining is thus on how to form a new $(k+1)$-leaf tree by correctly gluing the 2 rightmost leaves of the two k-leaf trees to the isomorphic part of the two k-leaf trees. Note that the isomorphic part of the two k-leaf trees is the $(k-1)$-prefix tree shared by them.

Depending on what kind of topological similarities the two to-be-joined k-leaf trees have, there are two cases in which the joining operations can be performed.

In each case, our analysis shows that joining two k-leaf trees can produce at most 4 different candidate $(k+1)$-leaf trees.

- **Case 1** When the two trees have the same topology, we further consider two sub cases.

  Case 1.1 When both the heaviest subtrees of the two trees are binary trees, four potential candidates can be generated. To facilitate the explanation of our algorithm, we hereafter adopt Newick notation [25] to represent leaf-labeled trees. In Newick notation, we can use (lt, hl_1) and (lt, hl_2) to represent the heaviest subtrees of t_1 and t_2 respectively, where hl_1 and hl_2 are the heaviest leaves in t_1 and t_2 respectively, and lt denotes the left subtree in both heaviest trees since the left subtrees of the two heaviest subtrees must be the same. Obviously, in the expanded candidate tree, 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, the $greater(hl_1, hl_2)$ and $smaller(hl_1, hl_2)$ will return whichever is greater and smaller respectively between the two integers representing the greatest leaves from the two trees. (Each leaf has an integer label and hence we can compare two leaves by comparing their integers.) Examples of $j_{[1]}$ and $j_{[2]}$ are illustrated by the 4-leaf trees j_{4-1} and j_{4-2} respectively in Figure 6. 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 from the other tree to get a $(k+1)$-leaf tree. From Section 2, pruning a leaf from a tree may introduce edge contractions. It is easy to see that, as a reverse operation, to attach a new leaf to a tree may introduce a new internal node as the opposite effect to the edge contraction operation. Therefore, two additional candidates which should also be considered 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 Figure 6. Notice that, although $j_{[2]}$ is generated by putting hl_1 and hl_2 as siblings, $j_{[2]}$ actually introduces a new internal node.

  Case 1.2 When both the heaviest subtrees of the two trees are multi-forked 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 t_1 and t_2 respectively, where $st_1, \ldots st_m$ are the m sibling subtrees (or branches) of hl_1 and hl_2 ($m \geq 2$ since the two trees are multi-forked). The expanded candidates can be in either the form of $j_{[1]} = (st_1, \ldots, st_m, smaller(hl_1, hl_2), greater(hl_2, hl_1))$ or the form of $j_{[2]} = (st_1, \ldots, st_m, (smaller(hl_1, hl_2), greater(hl_1, hl_2)))$. Figure 7 shows examples of $j_{[1]}$ and $j_{[2]}$.

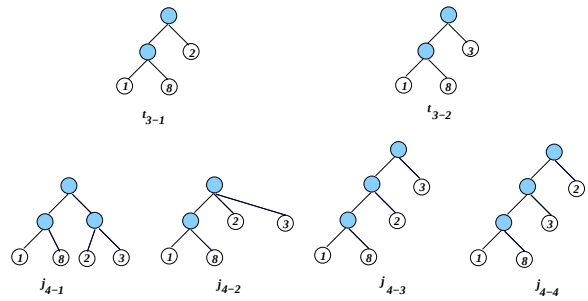

Figure 6: An example for Case 1.1, which shows that joining t_{3-1} and t_{3-2} can produce at most four candidates j_{4-1}, j_{4-2}, j_{4-3} and j_{4-4}.

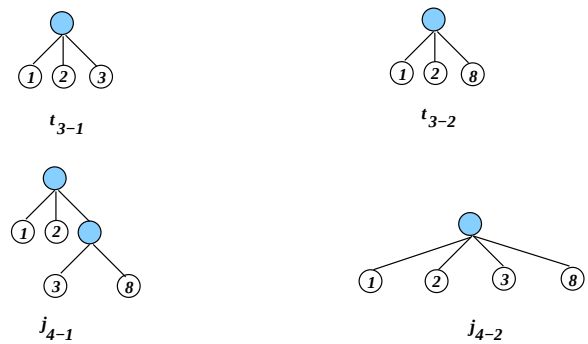

Figure 7: An example for Case 1.2, which shows that joining t_{3-1} and t_{3-2} can produce at most two candidates j_{4-1} and j_{4-2}.

  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. This inapplicability can be seen by contradiction. Let us assume that the third expansion considered in Case 1.1 is also applicable in Case 1.2. Thus, for example, we would expect the merged subtree to be $((1,2,3),8)$, which does support $(1,2,3)$, but not $(1,2,8)$ at the same time. This is because by pruning 3 from the imaginary $((1,2,3),8)$, the resulting subtree should be $((1,2),8)$. Thus the third expansion is impossible. A similar argument prevents the fourth expansion in Case 1.1 from being considered in Case 1.2. In fact, this same argument applies to all other sub cases in which expansion schemes are considered.

- **Case 2** When the two heaviest subtrees 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

can be further identified as the larger tree, and the other one the smaller tree.

Formally, let $h(t)$ and $s(t)$ denote the height and the size of a tree t respectively. Given two trees t_1 and t_2, t_1 is identified to be larger than t_2, if either of the following rules holds.

Rule 1 $h(t_1) > h(t_2)$. It means the nesting level of t_1 is greater than that of t_2.

Rule 2 $s(t_1) > s(t_2)$. This case can happen only when $h(t_1) = h(t_2)$. In this case, the fanout of the root of t_2 must be 2. Otherwise rule 1 will apply.

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 must be the heaviest leaf in the expanded subtree, and there must exist a subtree lst in t_{1hlp} which is isomorphic to t_{2hlp}. Letting lst be replaced by t_2, we can then obtain the $(t_{1hlp} \oplus hl_2, hl_1)$ as the final expanded tree, where $\oplus$ denotes the gluing operation. This joining operation can be easily understood if the larger tree is taken as an umbrella under which a part of it is replaced by the whole smaller tree. Figure 8 and Figure 9 show examples for rule 1 and rule 2 respectively. Please note that in Figure 9, the dotted rectangle delimits the heaviest subtree of tree t_{3_2}.

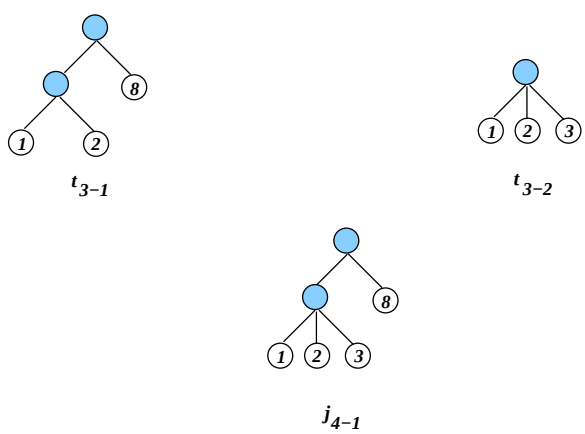

Figure 8: An example for rule 1 of Case 2, which shows that joining t_{3-1} and t_{3-2} can produce only one candidate tree j_{4-1}.

LEMMA 3.1. *The joining operation can be done in $O(k)$ time, where k is the number of leaves of the two data trees.*

The overall algorithm for discovering all frequent $(k+1)$-leaf trees from all k-leaf trees is shown in Figure 10. For each pair of k-leaf frequent subtrees that are in the same equivalence class, the subroutine Phylo_Join is called at line 5 to generate all possible candidate subtrees of

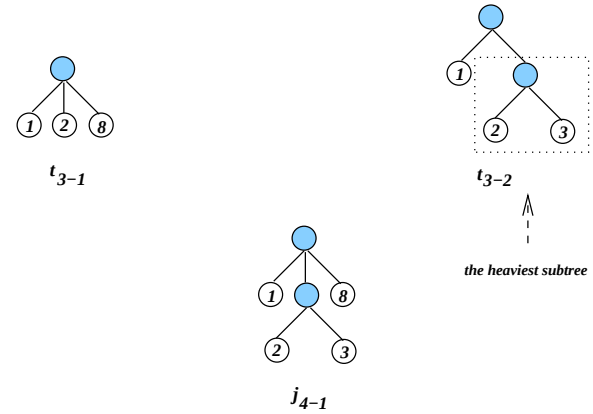

Figure 9: An example for rule 2 of Case 2, which shows that joining t_{3-1} and t_{3-2} can produce only one candidate tree j_{4-1}.

size $k+1$ based on the above case analyses. For each c_{k+1} produced by the Phylo_Join subroutine to become a candidate tree c_{k+1}, it will go through downward closure checking.[1] If the c_{k+1} can pass the downward closure checking, Grow_Subtrees then appends it to C_{k+1} and later it will go through the frequency counting phase; otherwise the c_{k+1} can be safely discarded. There is no need to check whether a c_{k+1} is already in C_{k+1} or not, since each particular c_{k+1} can be generated only once based on the equivalence class design.

3.5 Frequency Counting Once candidate subtrees are generated from each joining iteration, Phylominer computes the support for each candidate by checking the number of its occurrences in the given data trees. Given a candidate tree st on SL and a data tree t on L, $t|_{SL}$ can be obtained by pruning all leaves $l \in L - SL$ from t. Obviously, the candidate pattern st is a subtree of t if and only if $t|_{SL}$ will be isomorphic to st. The isomorphism between two trees can be verified by calculating their partition metric value.[2] To be more specific, 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 to calculate the partition metric has the time complexity of $O(N)$[8], which is the algorithm we adopted for the pattern verification in the frequency counting procedure.

[1] The downward closure checking is performed by hashing on the leaf sets of those k-leaf trees.

[2] 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.

```
Procedure: Grow_Subtrees(EClasses_k, k)
Input: EClasses_k, the equivalence classes for
       frequent k-leaf trees.
       k, tree size.
Output: F_{k+1}, a set of frequent (k+1)-leaf trees.
1.  C_{k+1} ← ∅, EClasses_{k+1} ← ∅;
2.  for each aec ∈ EClasses_k do
3.    if |aec| ≥ 2 then
4.      for each pair of elements x, y ∈ aec
        that are not on the same leaf set do
5.        C_{k+1} ← Phylo_Join(x, y);
6.        for each c_{k+1} ∈ C_{k+1} do
7.          if Downward_Closure_Checking(c_{k+1}) =
            TRUE then
8.            freq ← Frequency_Count(c_{k+1});
9.            if (freq > δ) then
10.             F_{k+1} ← F_{k+1} ∪ c_{k+1};
11.             if c_hlp ∉ EClasses_{k+1} then
12.               EClasses_{k+1} ←
                  EClasses_{k+1} ∪ c_hlp;
13.             register c_{k+1} to c_hlp;
14. F_{k+1} = ∪EClasses_{k+1};
15. return F_{k+1};
```

Figure 10: Algorithm for finding all frequent $(k+1)$-leaf trees based on k-leaf trees.

4 Correctness and Complexity Analysis

We present in this section a series of lemmas and theorems concerning the proposed algorithms, omitting their proofs due to space limitation.

LEMMA 4.1. **(Correctness)** *Any subtree discovered by* Phylominer *is a frequent agreement subtree.*

LEMMA 4.2. **(Completeness)** *It is sufficient to consider only trees in the same equivalence class for joining operations. In other words, the joining operation is complete without missing any frequent subtree.*

LEMMA 4.3. **(Non-Redundancy Candidate Generation)** *Each candidate tree is generated once at most.*

LEMMA 4.4. **(Automatic Canonicalization)** *The candidate trees produced by the* Phylo_Join *procedure are in their canonical form automatically.*

This automatic normalization property is a main factor contributing to the efficiency of the algorithm.

THEOREM 4.1. Phylominer *correctly finds all frequent agreement subtrees.*

THEOREM 4.2. *The time complexity of* Phylominer *is $O(|F|^2MN)$, where $|F|$ is the cardinality of the frequent subtree set, M is the number of data trees, and N is the size of the label set.*

Notice that this is a very pessimistic upper bound for two reasons. First, the actual number of data trees involved in the verification phase for each candidate tree is far less than M. With pattern size growing, the number of data trees that need to be verified 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 pseudo polynomial time algorithm, since $|F|$ is not an input parameter, but a value derived from the output, i.e. the total number of the discovered frequent agreement subtrees. To be more precise, the time complexity of the proposed algorithm is dependent on the number of qualified patterns, which is exponential in terms of the cardinality of the union set of leaf sets of all trees. Therefore, the algorithm has an exponential complexity in the worst case. In a typical case, however, the number of qualified patterns is much less, leading to a dramatically low running time complexity.

5 Experiments and Results

5.1 Synthetic Datasets For correctness verification and performance evaluation purposes, a tree generator is implemented in C++ to generate synthetic datasets subject to user specified parameters. The basic idea behind the data generator is similar to, but more powerful than, the one used in COMPONENT[24]. COMPONENT can generate binary leaf-labeled trees only while the generator developed in this work can generate leaf-labeled trees of various degrees by generalizing the algorithm described in [13]. Table 2 lists the parameters and their default values, where *fanout* of a node is the number of children of that node.

5.2 Performance Analysis The proposed tree mining algorithms were implemented in C++. A series of experiments have been conducted to evaluate the performance of the algorithms on synthetic data, run under the Solaris operating system on a SUN Ultra 60 workstation.

Figure 11 shows how changing the *database size* of synthetic datasets affects the overall running time of the algorithm Phylominer. The eight datasets generated for this experiment contained different numbers of trees ranging from 100 to 800, where each tree had the same

Table 2: Parameters and their default values used in the synthetic dataset generator.

Notation	Parameter	Default setting
$\|DT\|$	The size of the dataset DT	600
$\|S\|$	The cardinality of the leaf label set	10
LF	The largest fanout	5
SF	The smallest fanout	2

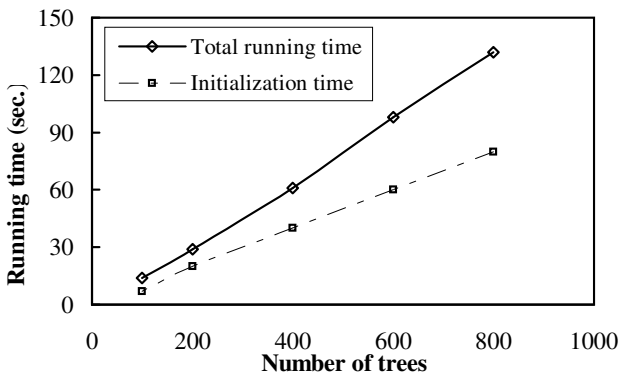

Figure 11: Scalability of Phylominer with respect to the dataset size.

number of leaves (this number was 15). The *minsup* value was set to 30%. Other parameters had default values as shown in Table 2. It can be seen from Figure 11 that the total running time scales up linearly with respect to the size of datasets. This happens because the more trees a dataset contains, the more time is needed for frequency counting in the dataset. Another measurement represented by the dashed line in the figure shows that the time spent on the initialization stage of the algorithm scales up linearly with the size of datasets. This happens because the initialization step essentially comprises two operations. One of them is 2-leaf patterns enumeration, where the number of 2-leaf patterns is related to the size of the label set only regardless of how many trees a dataset contains. On the other hand, the more trees a dataset contains, the more time is needed in preparing the supporting tree ID lists. This is the reason why the initialization time scales up linearly with the size of datasets. In general, when frequent patterns with size larger than 2 is limited, which is usually true, the initialization step dominates the running time.

Figure 12 shows the number of patterns obtained from the same set of experiments. It can be seen from the figure that with the increasing number of trees in different datasets on the same leaf label set, the numbers of patterns decreases to a stable value. The reason

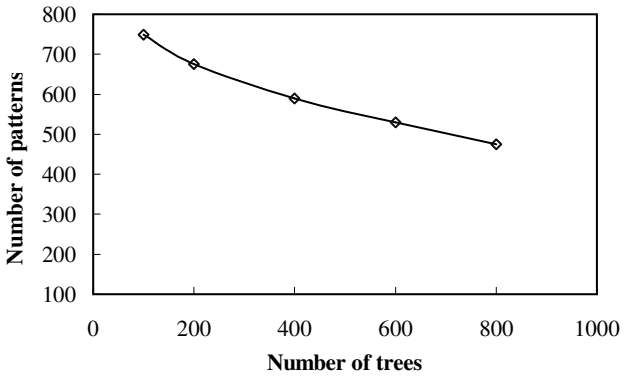

Figure 12: Effect of the database size on the number of frequent patterns discovered.

is that, in general, the more randomly generated trees a dataset has, the less consensus information is embodied in the dataset. This explains why the number of patterns declines with the increase of dataset sizes. On the other hand, although the number of frequent subtrees with more leaves could drop dramatically to zero due to the increasing of *minsup* value, the initialization set will guarantee that the final mining result contains at least all 2-leaf subtrees, the number of which is a fixed value. This explains why the number of qualified patterns reaches a stable value.

Figure 13 shows how changing *minsup* affects the number of patterns discovered by the algorithm. The data used in this experiment contains 200 synthetic trees, with each tree having 15 leaves. The values of the other parameters are as shown in Table 2. It can be seen from the figure that as *minsup* increases, the number of qualified patterns drops quickly. This experimental result can be well justified by the following analysis. When *minsup* goes up, the number of qualified patterns at $k > 3$ level drops. Consequently, the number of patterns with size $k + 1$ will drop in a non-linear way. This effect will be cascadingly propagated from the smaller-sized subtree levels to the larger-sized subtree levels in trees. Finally, the total number of qualified patterns will drop. It can also be observed that once the *minsup* value reaches a certain point, 0.8 in this case, the number of patterns reaches a stable value. This happens because the number of 2-leaf subtrees embedded in these data trees is always the same. As already mentioned, making up the initial set of the mining algorithm, these 2-leaf subtrees will appear in all mining results regardless of what the *minsup* values will be, because their support values are always 100%.

Figure 14 shows how changing *minsup* affects the running time of Phylominer on the same dataset used in the previous experiment. This figure shows that as *minsup* increases, the running time of Phylominer drops quickly. This can be explained by the fact that

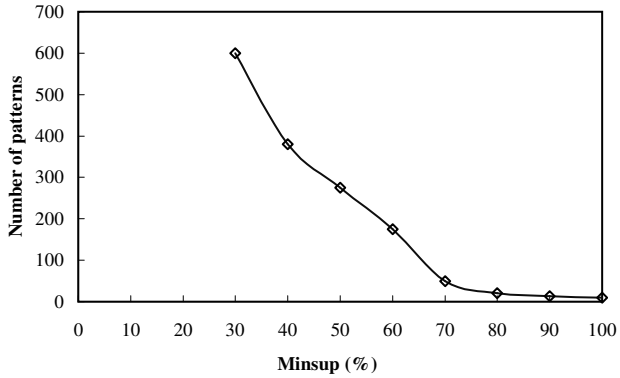

Figure 13: Effect of *minsup* on the number of frequent patterns discovered.

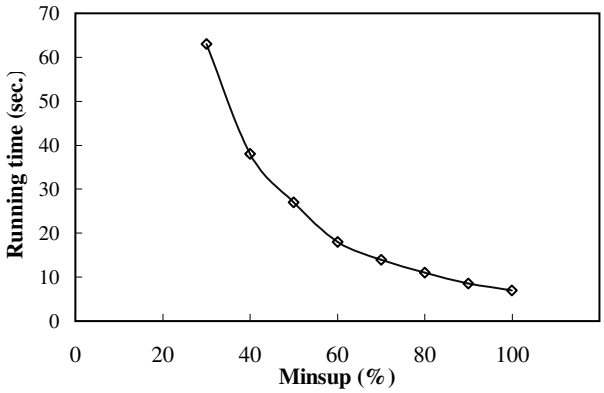

Figure 14: Effect of *minsup* on the running time of Phylominer.

the number of discovered patterns actually decreases with the increasing of *minsup*. Consequently, fewer valid pairwise joinings in each equivalence class will be performed. As a joint result of the above two factors, the overall running time drops quickly.

5.3 Datasets from COMPONENT and TreeBASE We present here the experimental result on the "peg.nex" data obtained from the COMPONENT tool[24]. The file "peg.nex" consists of 9 different trees for 11 species. Table 3 shows the results with different support values. When *minsup* was 33%, Phylominer found a total of 480 frequent subtrees. Among them, the maximum frequent agreement subtrees have 8 leaves (as shown in the "Size of MFAST" column). When *minsup* was 100%, Phylominer found a total of 251 frequent agreement subtrees, among which the maximum frequent agreement subtrees have 7 leaves.

Finally, the algorithm was applied to the dataset in Figure 1. The result is shown in Table 4. From this table, it can be seen that with *minsup* decreasing, the running time goes up, and the total number of interesting patterns goes up as well. The maximum

Table 3: Experimental results on peg.nex.

minsup	Time (ms)	Number of FAST	Number of MFAST	Size of MFAST
100%	34	251	3	7
89%	35	251	3	7
77%	35	251	3	7
66%	63	285	1	8
55%	70	285	1	8
44%	70	285	1	8
33%	150	480	4	8

Table 4: Data mining result on the five trees of the study $S497$ in TreeBASE.

minsup	Time (ms)	Number of FAST	Number of MFAST	Size of MFAST
100%	30	30	9	3
80%	30	30	9	3
60%	40	55	3	5
50%	63	91	9	5

frequent agreement subtrees have three leaves only when *minsup* is set to 100%, while the maximum frequent subtrees have 5 leaves when *minsup* drops to 50%.

Experimental results on these real datasets show that the proposed algorithm can systematically discover all interesting frequent agreement subtree patterns in data trees. These frequent agreement trees are expected to reveal more consensus information which could not be previously discovered by the traditional MAST algorithms. The proposed algorithm requires that the user input a support value. In practice, it is suggested that the user set the support threshold to a reasonably large value (e.g. 50%). Then use a strategy similar to "divide and conquer" or " binary search" to try different threshold values depending on the number of patterns discovered. For example, if there are too many patterns discovered, try a support of 75%; if there are too few patterns found, try a support of 25%, and so on. The patterns could be used for phylogeny clustering, for example, to construct phylogenetic islands, which is useful in tree surfing [25].

6 An Online Tree Mining Engine

An online tree mining toolkit has been developed, which utilizes Perl scripts and HTML pages to wrap up the core mining engine, so that users can easily interact with the mining engine over the Internet. Figure 15 shows the system's interface and the mining result when the

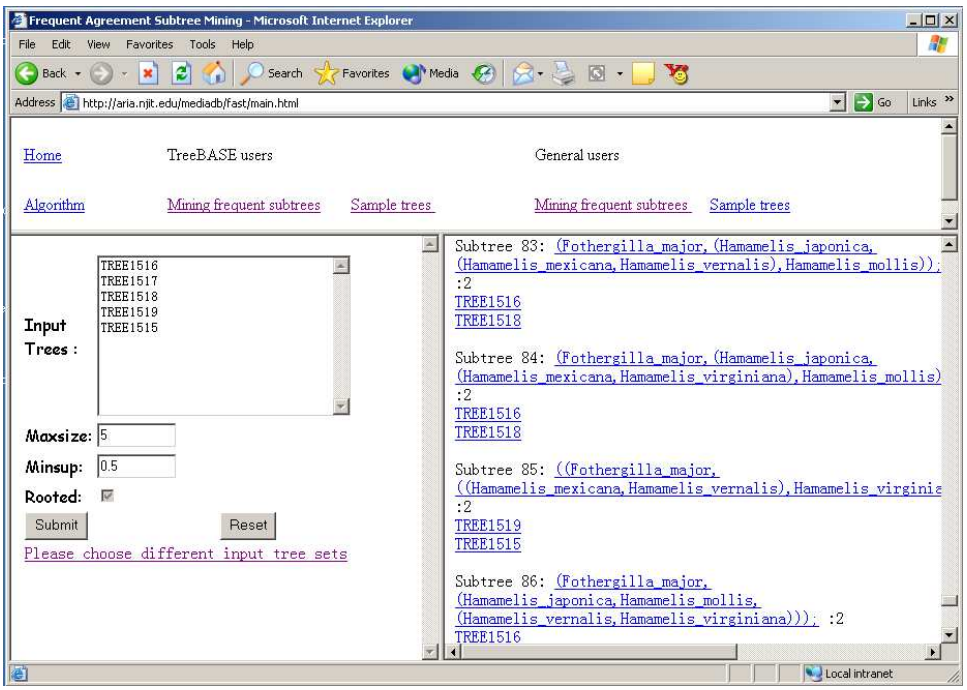

Figure 15: The online tree mining engine interface.

engine works on the dataset shown in Figure 1.

In Figure 15, the main menu of the system is displayed in the top frame; the bottom left frame shows the dataset input with appropriate mining parameter settings; and the right frame shows the mining result. When clicking on any subtree link in the right frame, the user will be redirected to a separate window where the subtree is shown in a Java applet. When the user clicks on the link for a data tree that supports a subtree, the data tree will be displayed in another window, which will highlight the supported subtree by showing its leaves in red color and decorating them with red bullets.

7 Conclusion

We presented in this paper a set of new data mining algorithms, called Phylominer, for automatically discovering frequent agreement subtrees from multiple rooted phylogenetic trees. The correctness and completeness of the algorithms were discussed. The algorithms were implemented in C++ and integrated into our phylogenetic tree mining engine. The experimental results on synthetic leaf-labeled trees and phylogenetic tree data showed the scalability and effectiveness of the proposed algorithms. These algorithms will be useful in not only phylogeny research but also other domains where data can be modeled as leaf-labeled trees. Future work includes (i) applying Phylominer to multiple phylogenies built from different species and studying the biological significance of discovered patterns, (ii) applying the work to tree classification [21] and super-tree construction [26], and (iii) extending the work to find frequent substructures in phylogenetic networks [22]. We also plan to compare alternative pattern generation methods including depth-first, breadth-first and apriori-like techniques and evaluate their relative performance and efficiency.

Acknowledgment

We thank the SDM anonymous reviewers for their constructive suggestions which helped improve the content and presentation of this paper.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th International Conference on Data Engineering*, 1995.

[2] A. Amir. Maximum agreement subtree in a set of evolutionary trees: metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1996.

[3] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa. Efficiently mining frequent substructures from semi-structured data. In *Proc. of International Workshop on Information & Electrical Engineering*, pages 59–64, Suwon, Korea, 2002.

[4] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *Proc. of the 6th International Conference on Discovery Science*, 2003.

[5] 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.

[6] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):190–202, 2005.

[7] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proc. of IEEE International Conference on Data Mining*, 2003.

[8] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 1:7–28, 1985.

[9] M. Farach, T. Przytycka, and M. Thorup. Maximum agreement subtree in a set of evolutionary trees: metrics and efficient algorithms. *Information Processing Letters*, 55:297–301, 1995.

[10] C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985.

[11] G. Ganeshkumar and T. Warnow. Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time. In *Proc. of the 1st International Workshop on Algorithms in Bioinformatics*, pages 156–163, 2001.

[12] M. Garofalakis and A. Kumar. Correlating XML data streams using tree-edit distance embeddings. In *Proc. of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2003.

[13] S. Holmes and P. Diaconis. Random walks on trees and matchings. *Electronic Journal of Probability*, 7, 2002.

[14] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proc. of the 3rd IEEE International Conference on Data Mining*, pages 549–552, 2003.

[15] M. Y. Kao, T. W. Lam, T. M. Przytycka, W. K. Sung, and H. F. Ting. General techniques for comparing unrooted evolutionary trees. In *Proc. of the 29th annual ACM Symposium on Theory of Computing*, pages 54–65, El Paso, Texas, 1997.

[16] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995.

[17] E. Kubicka, G. Kubicka, and F. R. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, 12:91–99, 1995.

[18] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 1st IEEE International Conference on Data Mining*, pages 313–320, 2001.

[19] T. W. Lam, W. K. Sung, and H. F. Ting. Computing the unrooted maximum agreement subtree in sub-quadratic time. *Nordic journal of Computing*, 3(4):295–322, 1996.

[20] J. T. Li, A. L. Bogle, A. S. Klein, and M. J. Donoghue. Phylogeny and biogeography of hamamelis (hamamelidaceae). *Harvard Papers in Botany*, 5:171–178, 2000.

[21] M. Kuramochi, M. Deshpande, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *Proc. of the 3rd IEEE International Conference on Data Mining*, pages 35–42, 2003.

[22] 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 Transactions on Computational Biology and Bioinformatics*, 1(1):13-23, 2004.

[23] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees: Proofs. Technical Report, Leiden Institute of Advanced Computer Science, Netherlands, Jan. 2003.

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

[25] W. H. Piel, M. J. Donoghue, and M. J. Sanderson. TreeBASE: A database of phylogenetic information. In *Proc. of the 2nd International Workshop of Species 2000*, 2000.

[26] C. Semple and M. Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105:147–158, 2000.

[27] R. Shamir and D. Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33(2):267–280, 1999.

[28] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 39–52, 2002.

[29] D. Shasha, J. T. L. Wang, and S. Zhang. Unordered tree mining with applications to phylogeny. In *Proc. of the 20th International Conference on Data Engineering*, pages 708–719, 2004.

[30] M. Steel and D. Penny. Distributions of tree comparison metrics – some new results. *Systematic Biology*, 42(2):126–141, 1993.

[31] 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 Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.

[32] T. Washio and H. Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1), July 2003.

[33] Y. Xiao, J. Yao, Z. Li, and M. Dunham. Efficient data mining for maximal frequent subtrees. In *Proc. of IEEE International Conference on Data Mining*, 2003.

[34] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[35] L. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *Proc. of the 29th International Conference on Very Large Databases*, Berlin, Germany, 2003.

[36] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002.

[37] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transaction on Knowledge and Data Engineering*, Special Issue on Mining Biological Data, W. Wang and J. Yang (eds.), 17(8):1021–1035, 2005.