

EVOMINER: Frequent Subtree Mining in Phylogenetic Databases

Technical Report #11-08, Dept. of Computer Science, Iowa State University

Akshay Deepak * David Fernández-Baca * Srikanta Tirthapura †

Michael J Sanderson ‡ Michelle M McMahon§

Abstract

The problem of mining collections of trees to identify common patterns, called frequent subtrees (FSTs), arises often when trying to make sense of the results of phylogenetic analysis. FST mining generalizes the well-known maximum agreement subtree problem. Here we present EVOMINER, a new algorithm for mining frequent subtrees in collections of phylogenetic trees. EVOMINER is an Apriori-like level-wise method, which uses a novel phylogeny-specific constant-time candidate generation scheme, an efficient fingerprinting-based technique for downward closure operation, and a lowest common ancestor based support counting step that requires neither costly subtree operations nor database traversal. As a result of these techniques, our algorithm achieves speed-ups of up to 100 times or more over Phylominer, another algorithm for mining phylogenetic trees. EVOMINER can also work in vertical mining mode, to use less memory at the expense of speed.

1 Introduction

A *phylogeny* (or phylogenetic tree or evolutionary tree) depicts the evolutionary relationships among a set of species. Phylogenetic research is of immense benefit to the society. Discovery of new forms of life for biotechnology [1], crop improvement [2], snakebite antivenins [3] and tracking spread of epidemic diseases [4] are just a few among many. Phylogenetic analysis typically yields a collection or distribution of trees for the given set of species, rather than a single tree [5]. Making sense of such data is a fundamental problem in phylogenetics. One basic problem is - given a set of phylogenetic trees, what is the common phylogenetic information conveyed

by all the trees? A commonly used approach to mining such common information is to compute a maximum agreement subtree (MAST) of the input trees [6, 7]. For a collection of input trees, MAST is the common embedded subtree on the largest number of leaves. A frequent subtree (FST) on the other hand refers to any embedded subtree common to at least majority of the input trees.

MAST is an important algorithmic problem with a history of more than 35 years [8]. The MAST problem is known to be NP-hard for more than two input trees having unbounded degrees [7]. The requirement that a MAST be supported by all input trees rules out subtrees that might be more informative than the MAST but still frequent (say $> 50\%$) [9] and thus equally important for phylogenetic inference. In this regard, a set of FSTs can reveal undiscovered phylogenetic information in a collection of trees, which would not be possible by considering just a single MAST. For example, consider the sample trees shown in Figure 1a. Figure 1b shows the corresponding MAST for the input trees, while Figure 1c shows the corresponding majority rule tree (MRT)¹ — another commonly used consensus approach [10]. Either of them return a unresolved star-like tree. The resolution of a tree T is defined as

$$(1.1) \quad \frac{|\text{internal edges in } T| \times 100}{|\text{leaves in } T| - 2} \%$$

In essence, resolution captures how informative the tree is. The MRT even though on the complete leaf set does not have any internal edges and hence does not convey any inferable phylogenetic information. Through a star-like structure it simply indicates that species $s_1 - s_6$ descend from a common ancestor - a fact that holds for any set of biological species in this world. Figure 1d shows the two FSTs with support 66% (2 out the 3 input trees support them as subtrees). The FSTs on the other hand have internal edges and thus convey inferable

*Department of Computer Science, Iowa State University, Ames, IA 50011, USA

†Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

‡Department of Ecology and Evolutionary Biology, University of Arizona, Tucson, AZ 85721, USA

§Department of Plant Sciences, University of Arizona, Tucson, AZ 85721, USA

¹MRT is constructed from the bipartitions occurring in majority of the input trees. A bipartition is a split on the leafset caused by removal of an edge.

phylogenetic information. For example the first of the FSTs indicate that w.r.t. the evolutionary history, s_1 and s_2 are closer to each other than each is to any of the species among $s_3 - s_5$. A similar example is given in [9].

Another aspect is that the MAST might only be depicting phylogenetic relationships among a small set of species [11]. Consideration of FSTs on the rest of the species can reveal a more complete phylogenetic picture; in fact, by itself, the MAST can be misleading [5]. For example consider the case shown in Figure 2. There are four MASTs in this case. Neither the MAST nor the MRT give any information among species $s_1 - s_4$. However, the FST shown depicts informative consensus information among the species $s_1 - s_4$.

Further, our current work can also mine subtree patterns on collections of trees having leaf sets that are partially overlapping but not common — a feature not associated with MAST or other consensus algorithms. For example, consider the trees shown in Figure 3. The input trees do not have a common leafset. Thus, known approaches like MAST or majority rule tree cannot be applied here. The common leafset only contains two species s_1 and s_4 . Thus, restricting the input trees to this common leafset and then applying MAST or MRT would not yield any useful information either. However, the FST as shown in Figure 3b contains species $s_1 - s_4$, which is clearly informative. This feature of our current work would be particularly useful in mining phylogenetic databases such as TreeBASE [12] or PhyLoTA [13], which contain trees with different leaf sets. Applicability of EVOMINER in mining such databases is of immense potential.

Phylogenetic analysis typically leads to a collection of trees rather than a single one in at least two important contexts. First, statistical analysis of a single gene using bootstrap or Bayesian inference procedures generates a confidence set or posterior distribution of trees respectively [14]. Second, analyses are increasingly encompassing many genes, in some cases, entire genomes, simultaneously, and biological processes induce weakly to strongly discordant trees for these separate genes [15]. In both contexts the frequency of subtrees in the collection is interesting. In the first it may reveal reliable subtrees overlooked by conventional methods for summarizing the tree distribution. In the second, frequent subtrees could well point to different biological histories for parts of the genome [16]. Finally, we note that FSTs are useful for solving MAST-related problems such as finding a maximum compatible subtree [11] and supertree [17], finding a maximum agreement supertree [18], and computing the kernel of maximum agreement subtrees [5].

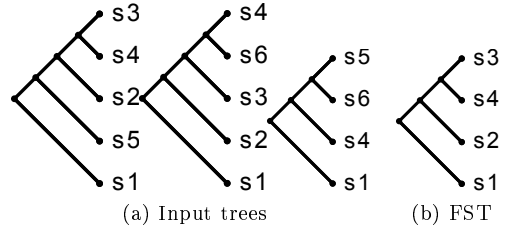


Figure 3: Motivating example 3

1.1 Our Contributions We introduce EVOMINER, an efficient algorithm to mine FSTs in phylogenetic databases. Over a broad range of inputs, EVOMINER is 100 times faster (and often more) than Phylominer [9], another algorithm for the same task. The key features that enable EVOMINER to achieve this speedup are:

1. An efficient phylogeny-specific constant-time candidate generation scheme, which exploits structural properties to produce fewer potential candidates.
2. A novel fingerprinting based scheme for the downward-closure operation, which in linear time checks support for all k of the $(k - 1)$ -leaf subtrees.
3. An efficient lowest common ancestor (LCA) based scheme to count support which neither requires the subtree operation nor a traversal through the database.

EVOMINER works in both a candidate generation based breadth-first enumeration mode (like Apriori [19]) as well as in vertical mining (depth-first enumeration) [20] mode. The first is quicker, while the second uses less memory, enabling it to handle larger trees. Further, EvoMiner can also mine collections of trees having partially overlapping leaf sets. This feature is not available in Phylominer.

1.2 Related Work Tree mining has been an active area of research in the past decade. For a survey of the area, see reference [21]. Based on the type of tree, tree mining tasks can be classified as ordered (TreeMiner [20], FREQT [22]) or unordered (Unot [23], uFreqt [24]), rooted or unrooted (CMTreeMiner [25], DRYADE [26]). Based on the type of subtree, they can be further classified as induced [22], embedded [27], or bottom-up. Based on the relationship among the frequent subtrees, they can be classified as maximal [28] and closed [25].

Loosely speaking, phylogenetic tree mining falls within the category of unordered embedded subtree mining. However, phylogenetic trees possess a very special structure — only leaves are labeled and non-leaf nodes must be of degree two or more (Section 2) —

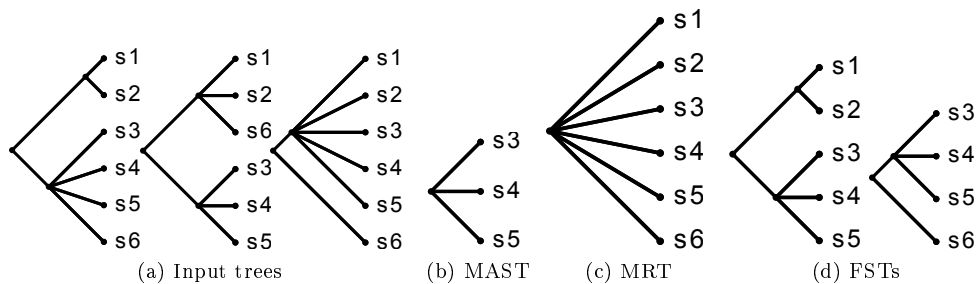


Figure 1: Motivating example 1

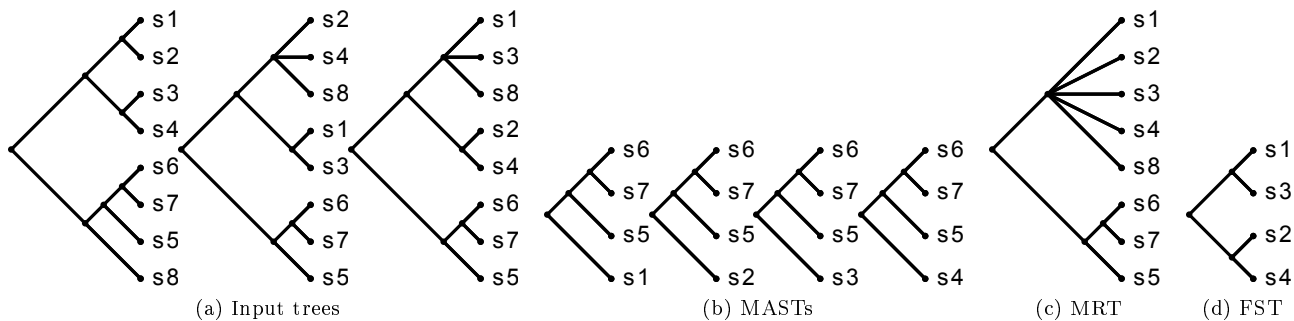


Figure 2: Motivating example 2

which demands a separate approach, and indeed affects the definition of the subtree operation itself (Section 2). Thus, while existing tree mining algorithms such as SLEUTH [27] or CMTreeMiner [25] can be applied for phylogenetic tree mining, the results would be incomplete, and contain redundant and invalid subtrees. To our knowledge Zhang et al.’s Phylominer [9] is the only published algorithm for mining phylogenetic trees; thus, it is the reference point for evaluating our work.

Consensus methods are widely used to summarize the information in a collection of trees. The goal is to find one tree that includes all the species, and captures the common (i.e., consensus) information in the collection of trees. Many different methods to do this have been proposed; for good surveys, see [10, 29]. Though consensus methods can be efficiently computable, the summary tree can significantly lack in resolution [11] i.e. the tree can contain high degree nodes. A MAST on the other hand is typically more resolved but can suffer from lesser number of leaves. FSTs can significantly help in matching the best of both by providing a collection of trees on lesser number of leaves but spanning the entire leafset. Several algorithms have been proposed to find MASTs between pairs of trees [30, 7, 6] and to solve related problems [11, 18, 17, 5]. However, none of these leads to an efficient way of enumerating frequent subtrees in

collections of trees.

2 Preliminaries

A *phylogenetic tree* is an unordered rooted or unrooted leaf-labeled tree. Leaf labels represent the taxonomic units (species) under consideration. A node is *internal* if it is not a leaf node. If a phylogeny T is rooted, we require that each internal node have at least two children; if T is unrooted, every internal node is required to have degree at least three. Note that the definition of a phylogenetic tree is more restrictive than the usual definition of an unordered tree in the data mining literature, since only the leaf nodes are labeled (uniquely) and except for the root node, degree-two internal nodes are not permitted.

Consider a phylogenetic tree T . Let \mathcal{L}_T denote the set of labels of the leaves of T , and Φ_T denote the bijection that maps the leaf nodes to their unique labels. For convenience, we refer to the set of leaf nodes by their labels in \mathcal{L}_T . For brevity, we will often refer to a phylogenetic tree simply as a *tree*.

From this point onwards, unless the context requires making a distinction, we will drop the subscripts in \mathcal{L}_T and Φ_T , and write \mathcal{L} and Φ respectively.

Our current work deals with rooted phylogenetic trees. The usual notions of ancestor/descendant and parent/child for ordinary rooted trees are equally appli-

cable to rooted phylogenetic trees. The single node in a rooted tree with no parent is called the *root* of the tree. The *depth* of a node u , denoted $\text{depth}(u)$, is the number of edges from the root to that node; thus the root node is at depth 0. We denote the lowest common ancestor (LCA) of two nodes u and v by $\text{LCA}(u, v)$. A *k-leaf tree* is a tree with k leaves.

In a graph (not necessarily a phylogenetic tree), suppose u is a degree-two node with neighbors v and w . Then, *suppressing* u means deleting u and replacing edges (v, u) and (u, w) by a single edge (v, w) . Consider a tree T and a subset of its leaves \mathcal{L}' . The *restriction* of T to \mathcal{L}' , denoted by $T|_{\mathcal{L}'}$, is the tree on the leaf set \mathcal{L}' obtained from the minimal subgraph T' of T spanning \mathcal{L}' by suppressing any degree-two node except the root [31]. The root of this restricted tree is the node closest to the root of T . A tree T' is called a *subtree* of another tree T , denoted as $T' \equiv T|_{\mathcal{L}'}$, if $\mathcal{L}_{T'} \subseteq \mathcal{L}_T$ and T' is isomorphic [32] to $T|_{\mathcal{L}'}$. Figure 4 shows examples of subtree operations.

Tree mining is closely related to graph mining [33, 34]. While trees in general are acyclic graphs, phylogenetic trees differ in another notable aspect of edge lengths. Generally phylogenetic trees only represent the branching history of common ancestry, and branch lengths are not considered [35]. Each internal node in a phylogenetic tree represents a hypothetical ancestor or an event (e.g. formation of a new species) in the evolutionary history that separates the ancestor and descendants at that node. For example, for the tree shown in Figure 4b species ‘2’ and ‘4’ are closer to each other in characteristics than each is to species ‘1’. The same information is preserved while considering the subtree without species ‘3’. Thus, considering a subtree preserves this evolutionary information and frequent subtrees are expected to reveal more of such common evolutionary patterns among a set of trees. In some cases branch lengths may represent temporal information of evolution. Such trees are called phylograms. However, the focus of our current work is inference of phylogenetic relationships among the species represented by leaf nodes. For more information on how to read a phylogenetic tree, see reference [35].

2.1 Problem Statement Let $D = \{T_1, T_2 \dots T_n\}$ be a database of n trees on a common label set \mathcal{L} . Let $\text{minSup} \in [1, n]$ be an input parameter. A tree T' with $\mathcal{L}_{T'} \subseteq \mathcal{L}$ is said to be a *frequent subtree* (FST) in D , if there exists $D' \subseteq D$ such that for all $T \in D'$, $T' \equiv T|_{\mathcal{L}'}$ and $|D'| > \text{minSup}$. That is, T' is a subtree to more than $\frac{\text{minSup}}{n}$ fraction of the trees in D . $|D'|$ is called the *support* of T' in D , and is denoted by $\text{sup}(T')$. Our goal is to identify all FSTs in phylogenetic databases.

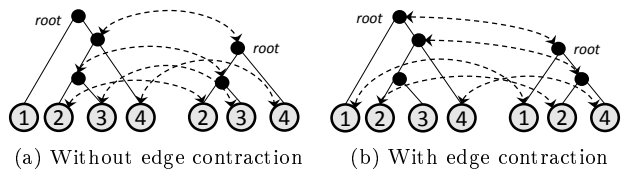


Figure 4: Subtree operation. Arrow mappings indicate the nodes which were retained from the original tree. The topmost node represents the root.

3 Frequent Subtree Mining

To enumerate only one subtree from a collection of isomorphic FSTs, we represent subtrees in a canonical form such that all trees in an isomorphic collection have the same canonical form, and such that trees from different isomorphic collections have different canonical forms [36]. We use the canonical form proposed by [9]. Assume without loss of generality that the leaf label set \mathcal{L} consists of integers in the range $[1, |\mathcal{L}|]$ so that the labels are ordered. The proposed canonical form assigns every internal node a virtual label in \mathcal{L} , which is the minimum among all its leaf descendants. The children of an internal node are ordered from left to right based on the sequence in which they are encountered in an inorder depth first traversal (IDFT), the leftmost child being encountered first. A tree T is in *canonical form* if, for every internal node, its children are ordered from left to right based on their virtual labels. Next, we describe the concept of an equivalence class and of a prefix tree, which are essential for our algorithm.

3.1 Rightmost Leaf, Prefix Tree, and Heaviest Subtree The *rightmost leaf* of tree T is the last leaf encountered in the IDFT of T . A useful property of the mentioned canonical form is that pruning of either the last leaf (deleting the leaf and suppressing the degree two nodes) or the second last leaf encountered in the IDFT, results in a subtree that is also canonical [9]. The resulting subtree after pruning the rightmost leaf is called the *prefix tree*. For a tree T , we refer to its prefix subtree as simply *prefix*. The *heaviest subtree* [9] is the subtree rooted at the parent of the rightmost leaf. Examples of the defined concepts are given in the next section.

3.2 Equivalence Class An *equivalence class* is a set of canonical trees sharing a common prefix tree. Any two trees in an equivalence class differ only w.r.t. their rightmost leaf. We call this common prefix tree the *core tree*. Thus, an equivalence class of k -leaf trees will have a $(k - 1)$ -leaf core tree. Any tree in

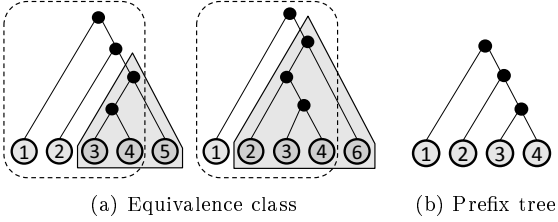


Figure 5: 5a shows two trees belonging to the same equivalence class. The common prefix tree is encircled by a dotted line; the respective rightmost leaves are the ones outside the dotted line. The shaded part represents the respective heaviest subtrees. 5b shows the common prefix tree for the trees in 5a.

the equivalence class can be obtained by attaching a leaf node to the rightmost path of the core tree; i.e., the path from the root node to the rightmost leaf. This leads to an efficient way to store members of an equivalence class: simply store the rightmost leaf and the position at which it is attached in the rightmost path in the core tree. Sharing a common prefix tree indeed defines an equivalence relation among its member trees, since the property is reflexive, symmetric and transitive. This equivalence relation partitions any set of canonical trees into disjoint subsets of equivalence classes, each identified by its unique core tree. This partitioning is the basis for our enumeration approach, which generates larger frequent subtrees by extending the core tree (Section 4.1). Figure 5 shows an example of two trees belonging to the same equivalence class, their rightmost leaves and their heaviest subtrees. Note that any two trees belonging to the same equivalence class differ only w.r.t. their heaviest subtrees.

4 The EVOMINER Algorithm

Figure 6 gives a high-level description of EVOMINER. The algorithm is based on an Apriori-like [19] candidate generation scheme, and uses breadth-first search to enumerate frequent subtrees. EVOMINER begins by computing the *LCA mappings* for every tree in the input database D . That is, for each tree T in D , and every pair $\{u, v\}$ of leaves of T , it computes $LCA(u, v)$ and stores it in a table. After this is done, EVOMINER repeatedly alternates between two steps until all FSTs are enumerated. The first step is **candidate generation**, which provides a set of potential frequent candidate trees. This must be done so that each frequent subtree is enumerated only once. The second step is **frequency counting**, which examines the candidate trees, identifying the frequent subtrees among them. This is a potentially time-consuming operation, since it can in-

```

EVOMINER( $D, \text{minSup}$ )
1: computeLCA_Mappings( $D$ )
2:  $Ft \leftarrow$ 
   enumerateFrequentTriplets( $D, \text{minSup}$ )
3:  $EC_3 \leftarrow$  computeEquivalenceClasses( $Ft$ )
4:  $E_{\text{next}} \leftarrow EC_3, \text{Result} \leftarrow \phi$ 
5: while  $E_{\text{next}} \neq \phi$  do
6:    $E_{\text{next}} \leftarrow$ 
     enumerateNextLevel( $E_{\text{next}}, \text{minSup}$ )
7:    $\text{Result} \leftarrow \text{Result} \cup E_{\text{next}}$ 
8: return  $\text{Result}$ 

enumerateNextLevel( $EC_k, \text{minSup}$ )
1:  $EC_{k+1} \leftarrow \phi$ 
2: for all  $e \in EC_k$  do
3:   for all  $T_x \in e$  do
4:      $e_{\text{next}} \leftarrow \phi$ 
5:     for all  $T_y \in e$  such that  $T_x \neq T_y$  do
6:       candidates  $\leftarrow$  join( $T_x, T_y$ )
7:       for all  $T^{\text{join}} \in$  candidates do
8:         if downwardClosure( $T^{\text{join}}$ ) then
9:           if sup( $T^{\text{join}}$ ) > minSup then
10:             $e_{\text{next}} \leftarrow e_{\text{next}} \cup T^{\text{join}}$ 
11:           $EC_{k+1} \leftarrow EC_{k+1} \cup e_{\text{next}}$ 
12: return  $EC_{k+1}$ 

```

Figure 6: EVOMINER Algorithm

volve frequent traversals through the input database and subsequent subtree operations. We next describe how each of these steps is implemented in EVOMINER.

4.1 Candidate Generation We denote the set of all equivalence classes on frequent k -leaf trees by EC_k . The input for the candidate generation step is an equivalence class in EC_k . The output is a set of potential candidate subtrees on $k + 1$ leaves extending the k -leaf trees in the equivalence class. The candidate generation strategy has two parts. The first is a constant time pairwise joining of frequent subtrees within an equivalence class to produce larger candidate trees (equivalence class based extension [27]). The second is a linear-time pruning of generated candidate trees through a downward closure operation, which tests whether all k -leaf subtrees of a given $(k + 1)$ -leaf tree are frequent [19, 37, 38]. Note that the enumeration of FSTs starts with *triplets* (trees on three leaves) as a triplet is the smallest tree on which evolutionary inference can be meaningful.

4.1.1 Pairwise Extension Pairwise extension involves joining of two frequent k -leaf trees belonging to the same equivalence class to generate larger $(k + 1)$ -

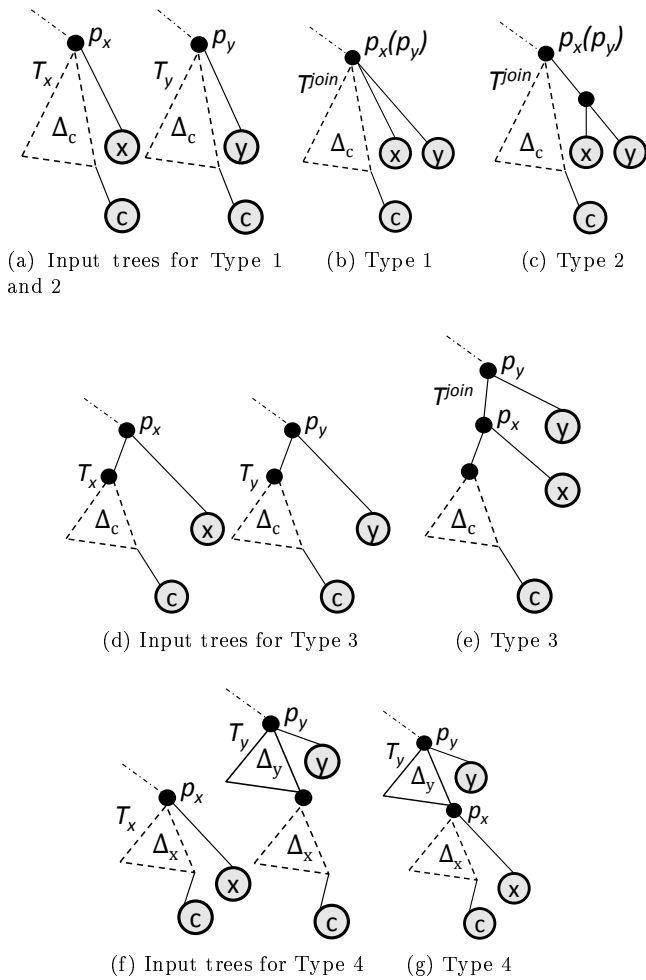


Figure 7: Different types of pairwise join. A dotted triangle represents part of the tree which may be empty while a solid triangle represents a non-empty part of the tree. Δ reflects topologies of the heaviest subtrees. ‘c’ denotes the rightmost leaf of the common core tree.

leaf candidate trees. Consider an ordered pair of trees $\langle T_x, T_y \rangle \in e \in EC_k$ and $T_x \neq T_y$. Let $\text{join}(T_x, T_y)$ denote the set of possible $(k+1)$ -leaf candidate trees by joining the pair. To ensure non-redundant generation of candidates, ordered pairs are considered and the resulting candidates are required to be in their canonical form. For this we require any $T^{\text{join}} \in \text{join}(T_x, T_y)$ to satisfy the following:

$$(4.2) \quad T_x = \text{prefix of } T^{\text{join}} \text{ and } T_y = T^{\text{join}}|_{\mathcal{L}_{T_y}}.$$

That is, every such tree T^{join} is a $(k+1)$ -leaf tree having T_x as its prefix and T_y as its subtree. This also leads to an efficient way to partition the joined trees

as equivalence classes: all such T^{join} s belong to the equivalence class with T_x as its core tree. Since T^{join} contains T_x as its prefix and is required to be canonical, it must be obtained by attaching the rightmost leaf of T_y to the rightmost path of T_x — resulting in a rightmost path extension [23, 24, 27]. T_x and T_y differ only w.r.t. their heaviest subtrees. The topology of the heaviest subtrees can differ in three possible ways leading to four types of joined trees as described next. In the subsequent discussion, let x and y denote the rightmost leaf of T_x and T_y respectively, p_x and p_y denote the parents of x and y respectively, and T^{core} represent the core of the equivalence class e . For an internal node u , let $\text{numChild}(u)$ denote its number of children.

1. $\text{depth}(p_y) = \text{depth}(p_x)$: Figure 7a shows the input trees for this case. Leaves x and y are attached at the same depth on the rightmost path of T^{core} . There are three possible ways in which T_x and T_y can be joined. Figure 7b and 7c show the resulting joins for the first two cases — Type 1 and 2 respectively. In Type 1 and 2, x and y are attached as siblings to the same pendant node in the joined tree. Thus, for the resulting joined tree to be canonical, $\Phi(x) < \Phi(y)$ must hold. The Type 3 join is a special case where p_y becomes the parent of p_x in the resulting tree as shown in Figure 7e. For this to be true $\text{numChild}(p_y) = \text{numChild}(p_x) = 2$ must hold else p_x and p_y cannot exist at the same depth on the rightmost path of T^{core} . Figure 7d shows the input trees for Type 3 join. Clearly, it is a special case of the input trees for Type 1 and 2.
2. $\text{depth}(p_y) < \text{depth}(p_x)$: Figure 7f shows the input trees for this case. On the rightmost path of T^{core} , leaf y is attached at a lesser depth than leaf x . There is only one way in which the join can take place leading to join of Type 4 as shown in Figure 7g. This is because the resulting join must satisfy 4.2. Node p_y becomes an ancestor of p_x in the joined tree.
3. $\text{depth}(p_y) > \text{depth}(p_x)$: A join operation is not possible since T_x cannot be the prefix tree of any T^{join} .

Clearly, we can check for each of the above cases in constant time and the resulting tree in each case is canonical. Hence, a join can be done in constant time for a pair of input trees. This is an important difference w.r.t. Phylominer, where the candidate generation scheme requires comparing the respective topologies of the heaviest subtrees of the input trees, which takes $O(k)$ time. Another difference is that by comparing $\text{depth}(p_y)$ with $\text{depth}(p_x)$, we generate fewer

candidate trees because fewer cases are considered for each possibility. This means that fewer trees go to the frequency counting step, which saves further time.

THEOREM 4.1. *Pairwise extension enumerates all FSTs and each FST is enumerated only once. Moreover the enumerated FSTs are in canonical form.*

Proof. Consider a FST T on k leaves. We shall prove the above by induction on k . If $k = 3$, then all such FSTs are uniquely enumerated during triplet enumeration — starting step of Algorithm 6. Consider $k > 3$. Assume all FSTs on $k - 1$ leaves have been uniquely enumerated. Let T be in its canonical form. Let T_x and T_y respectively be the subtrees obtained by pruning the last leaf and the second last leaf in the IDFT of T . By virtue of the chosen canonical form T_x and T_y are also canonical. Clearly T_x and T_y are FSTs on $k - 1$ leaves and share the same prefix tree. Thus, they must have been uniquely enumerated and must belong to the same equivalence class e (say). Since T satisfies 4.2 w.r.t. T_x and T_y , thus, T must be obtained as a result of joining T_x and T_y , and must be canonical. Thus pairwise extension must enumerate T . Further, if the members of e are considered in an ordered fashion for pairwise extension such that $\langle T_x, T_y \rangle$ is considered only once, then T will also be enumerated only once. Hence proved.

4.1.2 Downward-Closure Operation For a generated k -leaf candidate tree T , we verify if all k of its $(k - 1)$ -leaf subtrees are frequent. For this, all the $(k - 1)$ -leaf frequent subtrees must have been enumerated beforehand. EVOMINER uses an Apriori-like level-wise approach. That is, EC_{k+1} is enumerated only after EC_k has been enumerated. A common approach for the downward-closure operation is to first generate all k of the $(k - 1)$ -leaf subtrees and then check if each subtree is frequent by indexing it into an efficient data structure [19, 9]. This requires at least $O(k^2)$ time as indexing into any data structure will take at least $O(k)$ time and there are $O(k)$ subtrees to check. Things can get more complex if the subtrees themselves need to be checked for isomorphism [9]. We next propose an efficient fingerprinting based scheme that checks in $O(k)$ time whether all k of the $(k - 1)$ -leaf subtrees are frequent.

Fingerprint generation Fingerprinting is a mechanism that maps a large data item to a much shorter bit string. A good fingerprint function has a very small probability of mapping two different data items to the same bit string. This probability is inversely affected by the size of the bit string, which is generally a constant for a given application. Our approach involves generating fingerprints for all frequent subtrees

in EC_{k-1} and storing them in a hash table. Given a k -leaf candidate tree, to check if one of its $(k - 1)$ -leaf subtree is frequent, we check if its fingerprint is present in the hash table. We achieved speed-ups up to 100% when using the fingerprinting based pruning technique as compared to our vertical mining approach.

We employ the fingerprint function used in the Rabin–Karp pattern-matching algorithm [39]. Our approach is based on a string representation of trees based on Euler tours [40, 27]. The Euler tour for a tree is the sequence of nodes encountered in the IDFT of the tree. For example, the string representation of the first tree shown in Figure 5b is ‘D1UDD2UDD3UD4UUU’, where ‘D’/‘U’ respectively represent downward/upward traversal in the tree, and are selected such that $D, U \notin \mathcal{L}$. Clearly the size of the string representation is of the same order as the size of the tree and it uniquely identifies an ordered tree. Since frequent subtrees are enumerated in canonical form, they are always ordered. Thus any frequent subtree is uniquely identified by its string representation. The fingerprint function interprets a b -bit string as a b -bit integer. The fingerprint of a b -bit string $s = (s_1s_2 \dots s_b)$, denoted $F_p(s)$, is computed as:

$$F_p(s) = \left(\sum_{i=1}^b 2^i s_i \right) \bmod p,$$

where p is a random prime. While the cost of computing the fingerprint for a b -bit integer is $\Theta(b)$, a nice property of this fingerprint function is that a fingerprint can be updated in constant time after deletion of a substring [41]. For example given the fingerprint of the string ‘123456’, we can compute the fingerprint of the result of deleting substring ‘34’ by observing that $F_p(\text{‘1256’}) = (F_p(\text{‘123456’}) - F_p(\text{‘1234’}) \times 2^{\text{length}(\text{‘56’})} + F_p(\text{‘12’}) \times 2^{\text{length}(\text{‘56’})}) \bmod p$.

We note that in many applications of fingerprinting, the prime is chosen randomly so as to avoid an attempt by an adversary to select strings s_1 and s_2 such that $s_1 \neq s_2$ but $F_p(s_1) = F_p(s_2)$. In our case, however, it is reasonable to assume that the input distribution is oblivious to p . Hence, we use a fixed prime p . If p is chosen randomly, then the probability that two strings have the same fingerprint is $\frac{1}{p}$ [39, 41]. By selecting a large p , we keep this probability low. The size of the fingerprint is clearly constant for a chosen prime p .

Given a k -leaf tree, calculating the fingerprint of one of its $(k - 1)$ -leaf subtrees involves deleting the corresponding leaf and extracting the fingerprint from the fingerprint of the original tree in constant time. As shown in Figure 8, there are two possible cases for leaf deletion. If there is no edge contraction, the string representation changes from ‘...DD2UD3UD4UU...’ to

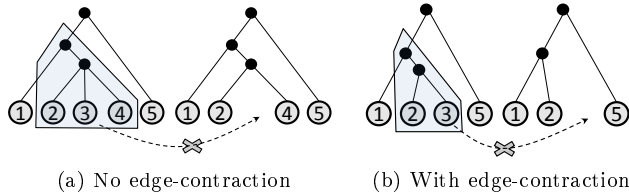


Figure 8: Leaf deletion

‘...DD2UD4UU...’. If there is edge contraction, the string changes from ‘...DD2UD3UU...’ to ‘...D2U...’. In either case, leaf deletion involves deletion of one or two substrings (emphasized in *italics*). For this we first compute the fingerprints of all prefix strings, which takes $O(k)$ time for a k -leaf tree. With this information, computing the fingerprint corresponding to the deletion of a leaf takes constant time. Thus, the fingerprints corresponding to all $(k-1)$ -leaf subtrees can be computed in $O(k)$ time. What remains is a lookup in the hash table to check if the subtree is frequent. If any of the $(k-1)$ -leaf subtrees is not frequent, then the candidate tree is pruned.

When using fingerprints as just described, there is a small probability of a false match; i.e., two different subtrees having the same fingerprint. Since the number of frequent subtree patterns is often large, we want to further strengthen the fingerprinting scheme. To do so, we hash the leaf sets of the subtrees and store the hash codes along with the fingerprints in the hash table. Thus we only compare two subtrees when they share the same leaf set. In our experiments, we never encountered duplicate values in the hash table when using this strengthened scheme. While theoretically there is still a chance of a false match, these errors are eventually detected in the frequency counting step. Further, when amortized, false matches do not affect the overall complexity. This also does not affect the correctness of the algorithm since it never prunes away a frequent subtree.

Note that our fingerprinting scheme does not consider any subtree obtained by deleting a leaf that is the leftmost child of its parent. This is because after deletion of such a leaf, the subtree might need to be canonicalized. Thus, in the worst case, only half of the subtrees are considered. This leads to the possibility of a *false positive* w.r.t. the downward closure operation; i.e., referring a k -leaf candidate to the frequency counting step even though it has an infrequent $(k-1)$ -leaf subtree. However, in our experiments we found a low false positive rate (.01 – 4%) when compared to a complete downward-closure check, which considers all $(k-1)$ -leaf subtrees. The explanation for the low rate

of false positives is that our problem empirically exhibits an *abundance of witnesses* property [42]. A witness here is an infrequent subtree that witnesses against the candidate tree being frequent. Our experiments show that the witness count, even when only considering half of the subtrees, remains abundant. Thus, a complete downward-closure check, while thorough, increases the running time without significantly improving the amount of pruning achieved. Note that this is only a pruning step. A false positive here only means that some candidates that could have been pruned by the exhaustive check were not pruned in this step. This in no way affects the correctness of the algorithm because these false positives will be eventually detected in the next step.

4.2 Frequency Counting For each frequent subtree T , we maintain an occurrence list [20, 22] that contains all the trees in the database that have T as a subtree. While considering a tree $T^{join} \in \text{join}(T_x, T_y)$, we first compute the intersection of the occurrence lists of T_x and T_y . We then iterate through this intersection list to check how many trees contain T^{join} . For this we propose an LCA-based enumeration scheme, which evaluates in constant time whether for a given tree in the intersection list, its subtree on the combined leafset $\mathcal{L}^\oplus = \{\mathcal{L}_{T^{core}} \cup x \cup y\}$ is T^{join} , where T^{core} is the common prefix tree of T_x and T_y . This allows significant speedup w.r.t. Phylominer, where a subtree operation is done w.r.t. every tree in the intersection list and is followed by a check for isomorphism (given in [43]). Both the subtree operation and the isomorphism check take at least linear time, whereas our scheme takes constant time.

We compute the LCA for all pairs of leaves for all the input trees during the initialization phase. For a given database D on the leaf set \mathcal{L} , this one-time task takes $O(|D||\mathcal{L}|^2)$ time. For a given T_x and T_y , and a T_i in the intersection of the occurrence lists of T_x and T_y , the subtree $T_i|_{\mathcal{L}^\oplus}$ must be one of the four types of joins as described in Section 4.1.1. We next give precise conditions based on the LCA values for each of the four cases. The meaning of the symbols c, x, y, p_x and p_y is the same as in Section 4.1.1. Superscripts indicate the reference tree.

LEMMA 4.1. $T_i|_{\mathcal{L}^\oplus}$ is of Type 1 join if and only if

1. $\text{depth}(\text{LCA}^{T_i}(c, x)) = \text{depth}(\text{LCA}^{T_i}(c, y))$
2. $\text{depth}(\text{LCA}^{T_i}(c, x)) = \text{depth}(\text{LCA}^{T_i}(x, y))$
3. $\Phi(x) < \Phi(y)$

Proof. Clearly if $T_i|_{\mathcal{L}^\oplus}$ is of Type 1 join, then it satisfies 1-3. To prove the *only if* part, let 1-3 be satisfied.

Since T_x and T_y are obtained by attaching x and y respectively to the rightmost path of T^{core} , 1 implies that $\text{depth}(p_y) = \text{depth}(p_x)$. Thus, $T_i|_{\mathcal{L}^\oplus}$ is either of Type 1, 2 or 3 join. Type 3 join requires the parent of y to be an ancestor of the parent of x in $T_i|_{\mathcal{L}^\oplus}$ — a case which is ruled out by 1. Further, 2 and 3 imply that the join must be of Type 1. Hence proved.

LEMMA 4.2. $T_i|_{\mathcal{L}^\oplus}$ is of Type 2 join if and only if

1. $\text{depth}(\text{LCA}^{T_i}(c, x)) = \text{depth}(\text{LCA}^{T_i}(c, y))$
2. $\text{depth}(\text{LCA}^{T_i}(c, x)) < \text{depth}(\text{LCA}^{T_i}(x, y))$
3. $\Phi(x) < \Phi(y)$

Proof. The proof proceeds in a similar fashion as that for Lemma 4.4. Again $T_i|_{\mathcal{L}^\oplus}$ can be either of Type 1 or 2 join. Conditions 2 and 3 imply that the join must be of Type 2. Hence proved.

LEMMA 4.3. $T_i|_{\mathcal{L}^\oplus}$ is of Type 3 join if and only if

1. $\text{depth}(\text{LCA}^{T_i}(c, x)) > \text{depth}(\text{LCA}^{T_i}(c, y))$
2. $\text{depth}^{T_x}(p_x) = \text{depth}^{T_y}(p_y)$

Proof. Condition 2 implies that $T_i|_{\mathcal{L}^\oplus}$ must be of Type 1, 2 or 3 join. Condition 1 rules out Type 1 and 2 joins. Thus the join must be of Type 3. Hence proved.

LEMMA 4.4. $T_i|_{\mathcal{L}^\oplus}$ is of Type 4 join if and only if

1. $\text{depth}^{T_x}(p_x) > \text{depth}^{T_y}(p_y)$

Proof. As per 1, $T_i|_{\mathcal{L}^\oplus}$ can only be of Type 4 join. Hence proved.

THEOREM 4.2. *Frequency counting scheme correctly classifies $T_i|_{\mathcal{L}^\oplus}$ as Type 1, 2, 3 or 4 join in constant time.*

The above is a direct consequence of Lemmas 4.1—4.4 as each of the cases can be clearly evaluated in constant time.

Vertical mining The above scheme also leads to a vertical mining algorithm (using depth first search in the enumeration graph [21]), which does not require candidate generation, along the lines of [20]. This is so because the frequency counting step gives sufficient conditions to distinguish among all possible candidates from pairwise-extension. While vertical mining does not benefit from efficient pruning through downward closure (which can be very effective as the number of trees in the database becomes large), it uses less memory than the breadth-first approach, allowing larger trees to be mined. This is because to extend a k -leaf tree, we only

need to store its ancestor equivalence classes, unlike the breadth-first enumeration approach where all the equivalence classes of the previous level must be stored. An experimental evaluation of the trade-offs between vertical and breadth-first mining is given in Section 5.

Extension to Partially Overlapping Leafsets EVOMINER can mine collections of trees having partially overlapping leaf sets. While computing the LCA for all pairs of leaves for all the input trees during the initialization phase, we flag an LCA value if either of the leaf involved in the pair is not present in the input tree. Such flagged LCA values are not considered during frequency counting. This allows to mine partially overlapping leaf sets without compromising the efficiency of common leafsets. Note that such an extension is not possible in Phylominer as the frequency counting step is not based on LCA values.

4.3 Correctness and Complexity Analysis Theorem 4.1 proves that all potential FSTs are enumerated uniquely by the pairwise extension scheme. Theorem 4.2 proves that all true FSTs are correctly identified from the potential candidates by the frequency counting step. Thus, EVOMINER correctly identifies all FSTs in a non-redundant fashion. We next discuss the time complexity of different steps of EvoMiner (EM) and compare it with Phylominer (PM); in the process recapturing the reasons for speedups. Let the input database be D consisting of n trees on a common leafset \mathcal{L} .

Initialization This one time task involves computation of LCA mappings for all possible pairs of leaves for all the input trees and enumeration of all frequent triplets. The former takes $O(n|\mathcal{L}|^2)$ time. Though this step is not there in PM, the time taken is a very small fraction of the total time as the total number of frequent patterns $\mathcal{F} \gg |\mathcal{L}|^2$. The latter takes $O(n|\mathcal{L}|^3)$ time. PM starts with evaluation of all frequent pairs of leaves instead of triplets. However, by virtue of enumerating all FSTs, it does enumerate all frequent triplets. Thus the net complexity for evaluating all frequent triplets is the same in Phylominer.

Candidate Generation Both EM and PM use pairwise extension approach. In EM candidates are generated in constant time. In PM it takes $O(k)$ time for joining two k -leaf candidate trees as it compares the topologies of the heaviest subtrees of the two candidates. Further by exploiting the structural properties of the candidates, EM generates lesser number of potential candidates than PM.

Downward Closure Both EM and PM use downward closure operation for pruning potential infrequent candidates. For a k -leaf candidate tree, PM checks whether all k of its $(k - 1)$ -leaf subtrees are frequent

by referring to hash table which stores the previously enumerated frequent $(k - 1)$ -leaf subtrees. This must take $O(k^2)$ time as there are $O(k)$ number of subtrees and indexing each will take $O(k)$ time. EM does the same operation in $O(k)$ time by deploying an efficient fingerprinting based scheme.

Frequency Counting Both EM and PM use occurrence list based approach. For a potential k -leaf frequent candidate, for every tree T in the intersection of the occurrence lists, PM computes the subtree of T on the same leaf set as that of the candidate and then compares this subtree with the candidate using a linear time isomorphism check. Each of this step takes $O(k)$ time for PM while EM does this check in constant time using LCA based frequency counting approach. Overall for each potential candidate, frequency counting in PM takes $O(nk)$ time while it takes $O(n)$ time in EM.

THEOREM 4.3. *The time complexity of EVOMINER is $O(n\mathcal{F} + |\mathcal{L}|\mathcal{F})$ where \mathcal{F} is the cardinality of the FST set, n is the number of trees in the database and \mathcal{L} is the common leafset.*

Proof. As discussed, the time complexity for initialization step is $O(n|\mathcal{L}|^2) + O(n|\mathcal{L}|^3) = O(n|\mathcal{L}|^3)$. Each k -leaf candidate generation takes $O(k) \leq O(|\mathcal{L}|)$ time. Downward closure operation for each k -leaf candidate takes $O(k) \leq O(|\mathcal{L}|)$ time. The frequency counting step takes $O(n)$ time for each candidate. Thus, the total time complexity to generate all FSTs is $O(n|\mathcal{L}|^3) + O(n\mathcal{F} + |\mathcal{L}|\mathcal{F}) = O(n\mathcal{F} + |\mathcal{L}|\mathcal{F})$ since $\mathcal{F} \gg |\mathcal{L}|^3$.

5 Experiments and Results

We performed a series of experiments to evaluate the performance of EVOMINER using both synthetic and biological datasets. All experiments were performed on an Intel Core2 Duo E8500 @ 3.16 GHz machine running Windows 7 Professional 64 bit edition with 8GB of RAM. Algorithms were implemented in C++ and compiled using Microsoft Visual C++ 2008 (part of Microsoft Visual Studio 2008, Version 9.0.21022.8 RTM). To our knowledge, Phylominer [9] is the only other algorithm for mining frequent phylogenetic trees. We compare the performance of our algorithm with Phylominer using the original C++ implementation of its authors. Our experiments involve four datasets, indexed as $D1$ – $D4$, which are described next.

$D1$ and $D2$ are datasets of synthetic phylogenetic trees. For $D1$, first random binary trees were generated based on the Yule model [44] and then iteratively for 30% of the number of the internal edges, a random internal edge was contracted. This way a random phylogenetic tree was obtained. This closely resembles the dataset used for the performance evaluation of Phy-

lominer. For $D2$, first a random tree was generated (as in $D1$). It was then replicated for the required number of trees. Each replicated tree was then perturbed by randomly contracting 10% of its internal edges and randomly swapping 10% of its leaf labels with another random leaf. This resulted in a set of trees having high commonality, which aligns with one of the utilities of EVOMINER as a consensus tree algorithm. $D3$, and $D4$ are datasets taken from published phylogenetic analyses. $D3$ is from Bayesian analysis of [45], while $D4$ consists of bootstrap trees from [46]. Bayesian analysis and bootstrap trees are typical candidates for consensus tree algorithms [47, 46] and the resulting trees have a very high commonality. To extract datasets of different sizes (in terms of the number of leaves and the number of trees) from $D3$ and $D4$, we randomly selected the required number of trees and restricted them on a random set of leaves of the required size.

Figure 9 compares the performance of EVOMINER with Phylominer on datasets $D1$ – $D4$. For each comparison, three different leafsets of sizes 15, 25 and 35 leaves were considered. For $D1$ and $D2$, the minSup value was 50%. Since, $D3$ and $D4$ have highly similar trees, the minSup value was 99% to single out the highly frequent subtrees among the frequent subtrees. In each of the datasets, EVOMINER performs up to 100 times or more better than Phylominer. For the purpose of comparison, the physical memory was capped at 4GB. This explains the missing entries in the graphs. However, in the vertical mining mode the memory is not a limitation due to the depth first approach for enumeration. If the run time is not a consideration then the user can mine up to 10000 trees on 254 leaves in the current implementation. The reason why EvoMiner is able to handle larger datasets — both in terms of the number of trees and the number of leaves — is because of the use of vertical bitmap representation of database [48]. In the vertical bitmap representation, a bit is reserved for every tree in the database. Generally there is a risk of under utilization of memory if the minimum support value is very less i.e. number of unset bits far exceeds the number of set bits for the occurrence list of a FST. However in our case, the minimum support value is always greater than 50%. Thus, this leads to a very efficient memory utilization while storing occurrence lists as vertical bitmaps.

In addition to accuracy, speed is also very important. As mentioned before, phylogenetic analysis typically yields a collection of trees rather than a single tree. However, a lot of such generated trees are not part of the final output. For example, Markov chain Monte Carlo (MCMC) simulations used for Bayesian phylogenetic inference [49] involves multiple runs until convergence is

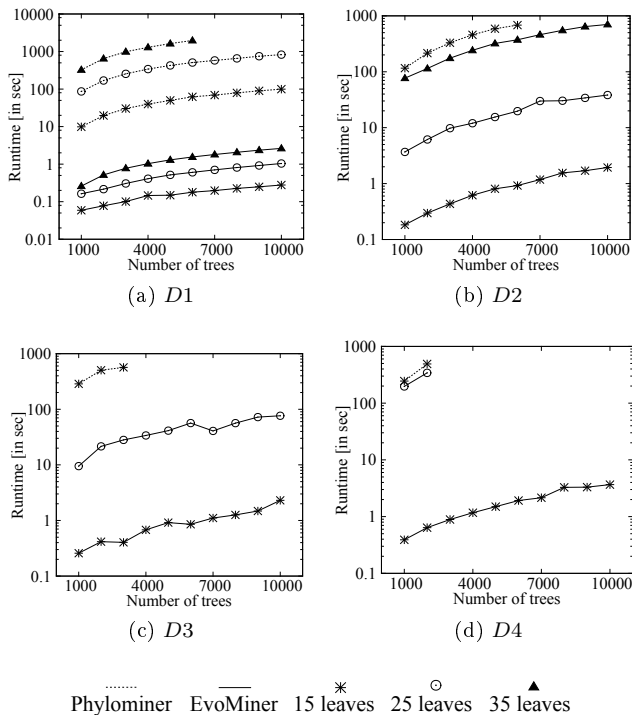


Figure 9: Performance comparison

reached. Each such run can be summarized using a consensus approach such as majority rule tree. In such a case, the consensus approach should be efficient enough to give results in a reasonable amount of time. As shown in Figure 9, the difference in runtimes of EVOMINER and Phylominer frequently reaches 1000 seconds. This further strengthens the need for speed. Another case in point is the use of consensus approach to summarize phylogenetic trees for the purposes of visualization. For example in [50], authors proposed and implemented a linear time algorithm for majority rule tree generation as the previous approaches were not good enough for their interactive visualization application. Use of frequent subtree patterns in the above contexts would certainly demand high efficiency in terms of speed rather than just accuracy. Our current work is an attempt make its use more attractive to phylogeneticists in the above contexts.

Figure 10a confirms the exponential growth of the number of frequent subtree patterns with an increase in the size of the leaf set. Figure 10b shows the corresponding analysis for run time. These experiments were done on Bayesian analysis *D3* with 100 trees and minSup as 99%. The range of the leaf sets and the number of trees reflects the typical sizes on which phylogenetic analyses involving MAST and related problems are done (e.g.

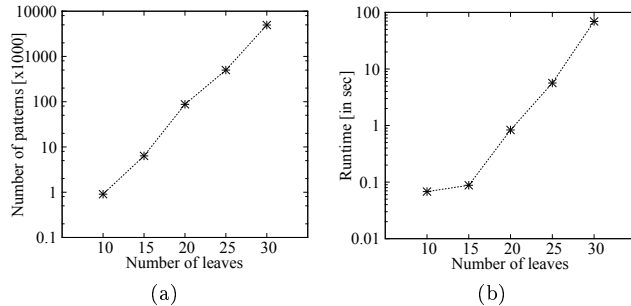


Figure 10: Exponential growth of the number of frequent patterns and its effect on the run time.

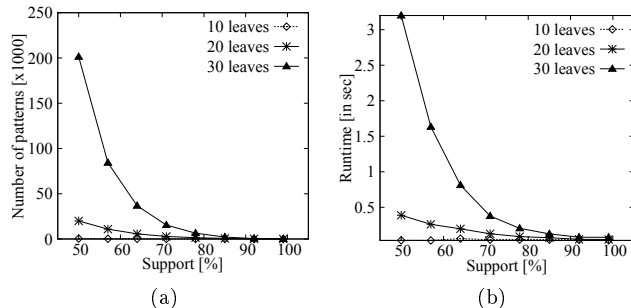


Figure 11: Effect of minSup on the number of frequent subtrees and the run time.

[5, 9]).

Figure 11a shows how the number of frequent subtree patterns varies w.r.t. minSup. The number of frequent subtrees falls exponentially as minSup is increased. This behavior is expected as the pruning of a frequent tree has a cascading effect on all of its frequent supertrees. The run time is also affected in a similar fashion (shown in Figure 11b). Similar behavior of the number of frequent subtrees and the run time w.r.t. minSup indicate how the run time depends directly on the number of frequent subtrees generated. These experiments were done on dataset *D2* with 500 trees.

Figure 12a compares the performance of vertical (depth first) mining with the candidate generation based (breadth first) approach w.r.t. the size of the database. When the number of trees is small, the frequency counting step takes about the same time as the downward-closure operation and hence the latter becomes an overhead. As the number of trees grows, the frequency counting step becomes costlier and it saves time to use downward closure. The cross-over point comes around 200-300 trees for Bayesian analysis dataset *D3*, with minSup set to 99% and leaf set size of 30. Figure 12b illustrates how the effectiveness of downward closure depends on the value of minSup.

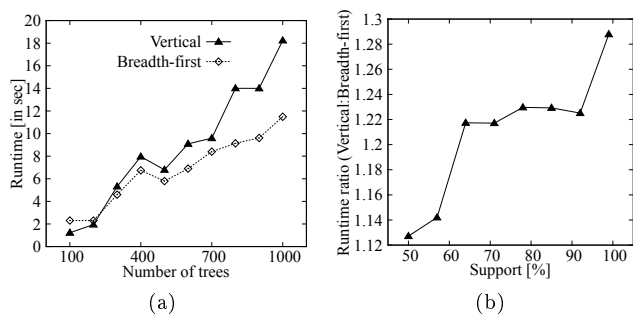


Figure 12: Comparison of Vertical mining with the candidate generation based (breadth first) approach.

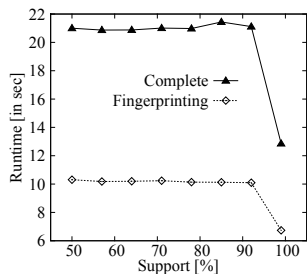


Figure 13: Performance evaluation of the fingerprinting based pruning technique.

When minSup is small, more candidate patterns turn out to be frequent; thus, downward-closure is less effective. As minSup is increased to the maximum value of 100%, more candidate patterns are pruned and the gain in performance due to the downward-closure operation is more pronounced. This experiment was also performed on $D3$, with 1000 trees on 20 leaves.

Figure 13 compares our fingerprinting based pruning technique with a complete downward closure operation. The latter additionally considers any remaining $(k - 1)$ -leaf subtrees for a k -leaf candidate tree, which are not considered by the former. For reasons discussed in Section 4.1.2, the fingerprinting technique is clearly more efficient than the complete operation. This experiment was done on bootstrapped dataset $D4$ with 1000 trees on 20 leaves. The false positive ratio hovered around 3.2% for this experiment.

5.1 Utility on Biological Data To demonstrate the utility of FST mining as an alternate and more informative consensus approach, we apply frequent subtrees, maximum agreement subtree and majority rule tree consensus approaches on biological data and compare the results. The biological dataset consists of bootstrapped trees used in a previous study [46] on majority rule trees. Trees were constructed using 17 DNA align-

ments containing 125 up to 2,554 sequences. It spans a diverse range of sequences including rbcL genes, mammalian sequences, bacterial and archaeal sequences, ITS sequences, fungal sequences, and grasses. We order the alignments based on the number of sequences and refer to the datasets as $A - Q$. For each dataset we randomly selected a set of 15 leaves and a set of 100 trees. We then restricted each of the trees on these 15 leaves to obtain a collection of 100 trees on a common leafset of size 15. We generated 100 such random collections for each of the dataset in $A - Q$ and the experiments results were averaged over these 100 collections.

Frequent Subtree vs. Maximum Agreement Subtree For each collection of 100 trees, we generated all the MAST patterns. We then mined all FSTs in the collection with $\text{minSup} = 50\%$. For each MAST T we selected the FST T' with the maximum number of leaves such that T is a subtree of T' . We then compared the size of T' (i.e. the number of leaves) with T . The first histogram bar in Figure 13 shows this leaf gain, which is defined as

$$\frac{|\text{leaves in } T'| - |\text{leaves in } T| \times 100}{|\text{leaves in } T|} \%$$

Clearly for each of the dataset, FST returns larger subtrees than MAST. In some of the cases the leaf gain is as high as 100%. This clearly demonstrates that FST can return larger subtrees than MAST. Note that the set of all MAST patterns is a subset of FSTs, thus MASTs can never be more informative than FSTs.

Frequent Subtree vs. Majority Rule Tree A majority rule tree (MRT) [10] is a consensus method that necessarily returns a summary tree on the complete leaf set. However, a MRT can suffer from low resolution. We use the tree-resolution measure defined as per 4.2. A similar measure was used by [46] for evaluating their work on improving resolution of MRTs. To compare MRT and FST, we introduce the notion of FST-profile. A FST T , is called *maximal* if there does not exist another FST T' such that T is a subtree of T' . A *FST-profile* is a collection of maximal FSTs such that the combined leafset contains all the species of the corresponding input collection. For example, the FSTs in Figure 1d form a FST-profile for the input collection of trees shown in Figure 1a. For each collection of 100 trees, we computed the MRT using HashCS [47] and generated the corresponding FST-profile from the collection of all FSTs mined using EVOMINER. For each tree T^F in the FST-profile, we restricted the MRT to the same leafset as that of T^F to obtain T^M . We computed the difference in tree-resolution values of T^F and T^M and averaged the difference over all the trees in the FST-profile. We did this for each of the

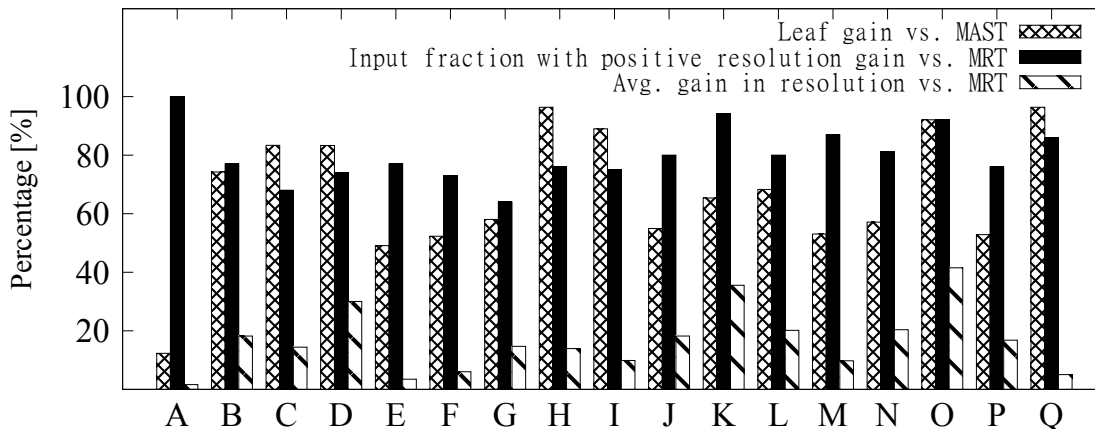


Figure 14: FST vs. MAST and MRT

100 collections. The second histogram bar in Figure 13, denotes the fraction of the input collections that observed a positive resolution gain in the FST-profile over the MRT. Clearly in each of the cases, a high fraction of the 100 collections resulted in better resolved FSTs as compared to the MRT. The third histogram bar shows the average resolution gain. In some of the cases it is more than 30%. These clearly indicate that FSTs can more than often return more resolved or more informative consensus trees than MRT.

6 Conclusion

We introduced a new algorithm for mining frequent subtrees in phylogenetic databases. We compared our work with Phylominer, another algorithm for the same problem, and showed speed-ups up to 100 times and more. Our improvements derive from an efficient phylogeny-specific constant-time candidate generation scheme, a novel fingerprinting based scheme for the downward-closure operation, and an efficient LCA-based scheme to count support. We also demonstrated the utility of FST mining as a more informative consensus approach to MAST and quite often to MRT as well.

We plan to extend EVOMINER to mine closed and maximal subtree patterns [21]. Currently at times, the number of FSTs can get quite large for the user to process them in a reasonable amount of time. Mining of maximal and closed subtrees would be quite useful in this context. Another point would be to consider branch lengths and other nodal attributes specific to certain phylogenetic analyses, in mining frequent subtrees. We also intend to explore frequent phylogenetic subtree mining in other problems related to MAST [11, 18, 17, 5]. It would also be interesting to apply the fingerprinting technique for the downward closure oper-

ation for mining arbitrary trees, not just phylogenies.

Acknowledgments

This work was supported in part by National Science Foundation (grant: XXXX). The authors thank Drs. Sen Zhang and Jason T. L. Wang for sharing the source code of Phylominer and discussions on their work. They also thank Drs. Seung-Jin Sul and Tiffani L. Williams for sharing the Bayesian data sets, and Dr. Nicholas D. Pattengale for sharing the Bootstrapped dataset.

References

- [1] S. Barns, C. Delwiche, J. Palmer, and N. Pace, "Perspectives on archaeal diversity, thermophily and monophyly from environmental rRNA sequences," *Proceedings of the National Academy of Sciences*, vol. 93, no. 17, p. 9188, 1996.
- [2] S. Flint-Garcia, A. Thuillet, J. Yu, G. Pressoir, S. Romero, S. Mitchell, J. Doebley, S. Kresovich, M. Goodman, and E. Buckler, "Maize association population: a high-resolution platform for quantitative trait locus dissection," *The Plant Journal*, vol. 44, no. 6, pp. 1054–1064, 2005.
- [3] J. Slowinski and J. Keogh, "Phylogenetic relationships of elapid snakes based on cytochrome b mtDNA sequences," *Molecular Phylogenetics and Evolution*, vol. 15, no. 1, pp. 157–164, 2000.
- [4] M. Smith and J. Patton, "Phylogenetic relationships and the radiation of sigmodontine rodents in south america: evidence from cytochrome b," *Journal of mammalian evolution*, vol. 6, no. 2, pp. 89–128, 1999.
- [5] K. Swenson, E. Chen, N. Pattengale, and D. Sankoff, "The Kernel of Maximum Agreement Subtrees," in *Proc. International Symposium on Bioinformatics Research and Applications*. Springer, 2011, pp. 123–135.

- [6] V. Berry and F. Nicolas, "Improved parameterized complexity of the maximum agreement subtree and maximum compatible tree problems," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, pp. 289–302, 2006.
- [7] A. Amir and D. Keselman, "Maximum agreement subtree in a set of evolutionary trees," *SIAM Journal on Computing*, vol. 26, pp. 758–769, 1994.
- [8] C. Finden and A. Gordon, "Obtaining common pruned trees," *Journal of Classification*, vol. 2, no. 1, pp. 255–276, 1985.
- [9] S. Zhang and J. Wang, "Discovering frequent agreement subtrees from phylogenetic data," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 1, pp. 68–82, 2008.
- [10] D. Bryant, "A classification of consensus methods for phylogenetics," in *Bioconsensus: DIMACS Working Group Meetings on Bioconsensus*. Amer Mathematical Society, 2003, p. 163.
- [11] G. Ganapathysaravanabavan and T. Warnow, "Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time," in *Proc. International Workshop on Algorithms in Bioinformatics*. Springer, 2001, pp. 156–163.
- [12] W. Piel, M. Donoghue, and M. Sanderson, "Treebase: a database of phylogenetic knowledge," in J. Shimura, K. L. Wilson and D. Gordon, eds. *To the Interoperable "Catalog of Life" with Partners, Species 2000 Asia Oceania*. Research Report from the National Institute for Environmental Studies No. 171, Tsukuba, Japan, 2002, pp. 41–47.
- [13] M. Sanderson, D. Boss, D. Chen, K. Cranston, and A. Wehe, "The PhyLoTA browser: processing GenBank for molecular phylogenetics research," *Systematic Biology*, vol. 57, no. 3, p. 335, 2008.
- [14] J. Felsenstein, *Phylogenetics*. Sunderland, Massachusetts: Sinauer Associates, 2004.
- [15] B. Rannala and Z. Yang, "Phylogenetic inference using whole genomes," *Annu. Rev. Genomics Hum. Genet.*, vol. 9, pp. 217–231, 2008.
- [16] X. Zou, F. Zhang, J. Zhang, L. Zang, L. Tang, J. Wang, T. Sang, and S. Ge, "Analysis of 142 genes resolves the rapid diversification of the rice genus," *Genome Biology*, vol. 9, no. 3, p. R49, 2008.
- [17] V. Hoang and W. Sung, "Improved algorithms for maximum agreement and compatible supertrees," *Algorithmica*, pp. 1–20, 2011.
- [18] S. Guillemot and V. Berry, "Fixed-parameter tractability of the maximum agreement supertree problem," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 7, no. 2, pp. 342–353, 2010.
- [19] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. Verkamo *et al.*, "Fast discovery of association rules," *Advances in knowledge discovery and data mining*, vol. 12, pp. 307–328, 1996.
- [20] M. Zaki, "Efficiently mining frequent trees in a forest: Algorithms and applications," *IEEE Trans. Knowl. Data Eng.*, pp. 1021–1035, 2005.
- [21] Y. Chi, R. Muntz, S. Nijssen, and J. Kok, "Frequent subtree mining-an overview," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2004.
- [22] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," in *Proc. SIAM International Conference on Data Mining*, 2002, pp. 158–174.
- [23] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering frequent substructures in large unordered trees," in *Discovery science*. Springer, 2003, pp. 47–61.
- [24] S. Nijssen and J. Kok, "Efficient discovery of frequent unordered trees," in *International Workshop on Mining Graphs, Trees and Sequences*, 2003.
- [25] Y. Chi, Y. Xia, Y. Yang, and R. Muntz, "Mining closed and maximal frequent subtrees from databases of labeled rooted trees," *IEEE Trans. Knowl. Data Eng.*, pp. 190–202, 2005.
- [26] A. Termier, M. Rousset, and M. Sebag, "Dryade: a new approach for discovering closed frequent trees in heterogeneous tree databases," in *Proc. IEEE International Conference on Data Mining*. IEEE, 2004, pp. 543–546.
- [27] M. Zaki, "Efficiently mining frequent embedded unordered trees," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 33–52, 2004.
- [28] Y. Xiao and J. Yao, "Efficient data mining for maximal frequent subtrees," in *Proc. IEEE International Conference on Data Mining*. IEEE, 2003, pp. 379–386.
- [29] C. Scornavacca, "Supertree methods for phylogenomics," Ph.D. dissertation, Univ. of Montpellier II, Montpellier, France, December 2009.
- [30] M. Steel and T. Warnow, "Kaikoura tree theorems: computing the maximum agreement subtree," *Information Processing Letters*, vol. 48, no. 2, pp. 77–82, 1993.
- [31] M. Steel, "The complexity of reconstructing trees from qualitative characters and subtrees," *Journal of Classification*, vol. 9, no. 1, pp. 91–116, 1992.
- [32] R. Shamir and D. Tsur, "Faster subtree isomorphism," *Journal of Algorithms*, vol. 33, no. 2, pp. 267–280, 1999.
- [33] T. Washio and H. Motoda, "State of the art of graph-based data mining," *Acm Sigkdd Explorations Newsletter*, vol. 5, no. 1, pp. 59–68, 2003.
- [34] D. Cook and L. Holder, *Mining graph data*. Wiley-Blackwell, 2007.
- [35] D. Baum, "Reading a phylogenetic tree: The meaning of monophyletic groups," *Nature Education*, vol. 1, no. 1, 2008.
- [36] Y. Chi, Y. Yang, and R. Muntz, "Canonical forms for labelled trees and their applications in frequent subtree mining," *Knowledge and Information Systems*, vol. 8, no. 2, pp. 203–234, 2005.
- [37] Y. Chi, Y. Yang, and R. R. Muntz, "Indexing and mining free trees," in *Proc. IEEE International Conference on Data Mining*, 2003.
- [38] J. Pei, J. Han, and L. Lakshmanan, "Mining frequent itemsets with convertible constraints," in *Proc. IEEE International Conference on Data Engineering*. IEEE

Computer Society, 2001, pp. 0433–0442.

- [39] R. Karp and M. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
- [40] M. Bender and M. Farach-Colton, “The LCA problem revisited,” *LATIN 2000: Theoretical Informatics*, pp. 88–94, 2000.
- [41] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge: Cambridge Univ, 1995, ch. 7.
- [42] J. Hromkovič, *Design and analysis of randomized algorithms: introduction to design paradigms*. Springer-Verlag New York Inc, 2005, ch. 7.
- [43] J. Wang, H. Shan, D. Shasha, and W. Piel, “Fast structural search in phylogenetic databases,” *Evolutionary Bioinformatics Online*, vol. 1, p. 37, 2005.
- [44] G. Yule, “A mathematical theory of evolution, based on the conclusions of dr. jc willis, frs,” *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, vol. 213, pp. 21–87, 1925.
- [45] L. Lewis and P. Lewis, “Unearthing the molecular phylo-diversity of desert soil green algae (Chlorophyta),” *Systematic Biology*, vol. 54, no. 6, p. 936, 2005.
- [46] N. Pattengale, A. Aberer, K. Swenson, A. Stamatakis, and B. Moret, “Uncovering Hidden Phylogenetic Consensus in Large Datasets,” *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, no. 99, pp. 1–1, 2011.
- [47] S. Sul and T. Williams, “An experimental analysis of consensus tree algorithms for large-scale tree collections,” in *Proc. International Symposium on Bioinformatics Research and Applications*. Springer, 2009, pp. 100–111.
- [48] D. Burdick, M. Calimlim, and J. Gehrke, “Mafia: A maximal frequent itemset algorithm for transactional databases,” in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 443–452.
- [49] B. Mau, M. Newton, and B. Larget, “Bayesian phylogenetic inference via markov chain monte carlo methods,” *Biometrics*, vol. 55, pp. 1–12, 1999.
- [50] N. Amenta, F. Clarke, and K. St. John, “A linear-time majority tree algorithm,” *Algorithms in Bioinformatics*, pp. 216–227, 2003.