Noname manuscript No.
(will be inserted by the editor)

EvoMiner: Frequent Subtree Mining in Phylogenetic Databases

Akshay Deepak · David Fernández-Baca · Srikanta Tirthapura · Michael J. Sanderson · Michelle M. McMahon

the date of receipt and acceptance should be inserted later

Received: Aug 13, 2012
Revised: Apr 01, 2013
Accepted: May 25, 2013

Abstract The problem of mining collections of trees to identify common patterns, called frequent subtrees (FSTs), arises often when trying to interpret 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, and a lowest common ancestor based support counting step that requires neither costly subtree operations nor database traversal. Our algorithm achieves speed-ups of up to 100 times or more over Phylominer, the current state-of-the-art algorithm for mining phylogenetic trees. EvoMiner can also work in depth first enumeration mode, to use less memory at the expense of speed. We demonstrate the utility of FST mining as a way to extract meaningful phylogenetic information from collections of trees when compared to maximum agreement subtrees and majority rule

A. Deepak (✉) · D. Fernández-Baca
Department of Computer Science, Iowa State University, Ames, IA 50011, USA
e-mail:akshayd@iastate.edu

S. Tirthapura
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

M. J. Sanderson
Department of Ecology and Evolutionary Biology, University of Arizona, Tucson, AZ 85721, USA

M. M. McMahon
Department of Plant Sciences, University of Arizona, Tucson, AZ 85721, USA

trees — two commonly used approaches in phylogenetic analysis for extracting consensus information from a collection of trees over a common leaf set.

1 Introduction

A phylogeny (or phylogenetic tree or evolutionary tree) depicts the evolutionary relationships among a set of species [9]. Aside from their intrinsic scientific interest, phylogenies have diverse applications, which include characterizing cryptic biological diversity [8], crop improvement [30], identifying snakebite antivenins [73], tracking the spread of epidemic diseases [74], political science [20], and historical linguistics [34].

Here we consider the problem of identifying common patterns —specifically, frequent subtrees— in collections of phylogenetic trees. A frequent subtree (FST) is a tree t that is "supported" by at least some given fraction of the input trees. That is, t is embedded as a subtree in at least the given fraction of the input trees[1]. The need to mine collections of phylogenies for frequent patterns arises in different contexts. For example, when building evolutionary trees from single-gene data sets, Bayesian inference [43] produces a posterior distribution of trees, while the bootstrap method [27] yields a set of trees, from which confidence intervals are obtained for various groupings of the species. In both cases, frequent subtrees can reveal common patterns overlooked by conventional techniques, such as majority-rule trees or maximum agreement subtrees. Collections of phylogenetic trees also arise in multi-gene (or even whole-genome) studies. Large sets of such trees have been assembled into databases such as TreeBASE [65] and PhyLoTA [68]. In general, these phylogenies overlap only partially in the sets of species they cover. Further, these trees often disagree with respect to the placement of the species they share in common, due to either error or to complex biological processes (e.g., horizontal gene transfer, gene duplication, convergent evolution, and varying evolutionary rates) [67]. Here again, FSTs can be valuable, pointing to different histories for parts of the genome [90].

In this paper, we present EvoMiner, a fast algorithm for enumerating FSTs. Further, we demonstrate experimentally the effectiveness of the FST approach, as compared to other widely-used methods for identifying common phylogenetic information. To put our contributions in context, we first review the existing work.

[1] Strictly speaking, we required that t be *displayed by* at least some given fraction of the input trees. Formal definitions of all the concepts used in this Introduction are given in Sect. 2.

Algorithm	Type of input trees	Type of subtrees
Chopper [79]	Ordered	Embedded
CMTreeMiner [18]	Ordered /Unrooted	Embedded, Maximal/Closed
DRYADE [78]	Unrooted	Maximal/Closed
FreeTreeMiner [15]	Unordered/Unrooted	Induced
FREQT [5]	Ordered	Induced
HybridTreeMiner [17]	Unordered/Unrooted	Induced
PathJoin [83]	Unrooted	Embedded, Maximal/Closed
Phylominer [88]	Unordered /Unrooted	Phylogenetic
POTMiner [46]	Ordered/Unordered/Partially-ordered	Induced/Embedded
SLEUTH [86]	Unordered	Embedded
TreeMiner [87]	Ordered	Embedded
uFreqt [60]	Unordered	Induced
Unot [6]	Unordered	Induced
Xspanner [79]	Ordered	Embedded
EvoMiner	Unordered /Unrooted	Phylogenetic

Table 1 Some frequent subtree-mining approaches

1.1 Related Work

1.1.1 Tree Mining

Tree mining has been an active area of research in the past decade; for a survey, see references [16, 45]. Table 1 summarizes some of the proposed approaches. Based on the type of tree, tree mining tasks can be classified as ordered (TreeMiner [87], FREQT [5], Chopper and XSpanner [79]) or unordered (Unot [6], uFreqt [60], [10]), rooted [10] or unrooted (CMTreeMiner [18], DRYADE [78]). Based on the type of subtree, they can be classified as induced [5], embedded [87, 54], or bottom-up. Based on the relationship among the frequent subtrees, they can be classified as maximal (PathJoin [83], [36]) and closed [18, 28, 81, 59].

A number of approaches have been proposed for mining ordered trees. Asai et al's Freqt method [5] uses rightmost expansion scheme for candidate generation and occurrence lists for frequency counting. Zaki's TreeMiner algorithm [87] introduced a scope list based representation of the occurrence list of a frequent subtree. Coupled with a clever string encoding, this results in efficient candidate generation and frequency counting. The enumeration proceeds in a depth-first manner. Wang et al [79] proposed the Chopper and Xspanner

algorithms to mine subtrees without candidate generation. These scale well for large sets of trees because they use the pattern-growth method [62, 40, 38] for enumeration, which represents the database in a compact form and avoids generating an exponential number of candidates [79]. Yang et al [84] gave algorithms for ordered tree mining in the context of mining frequent XML query patterns. They adopted a rightmost expansion approach for candidate generation.

In the field of unordered tree mining, Xiao et al [83] proposed efficient algorithms for discovering frequent subtrees under the assumption that no two siblings have the same label. Unot, by Asai et al [6], and uFreqt, by Nijssen and Kok [60] — proposed independently — use very similar techniques for unordered frequent subtree mining. Both extend the ordered tree mining approach of [5] by enumerating subtrees in a canonical form. Zaki [86] extended the efficiency of TreeMiner [87] for mining unordered embedded subtrees. Chi et al [15, 17] studied the unordered tree mining problem for rooted and unrooted trees.

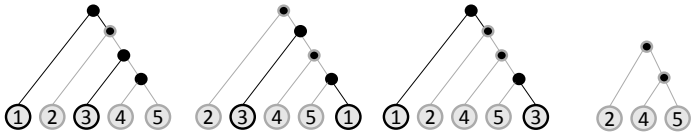
Loosely speaking, phylogenetic tree mining is a kind of unordered embedded subtree mining. However, phylogenetic trees possess a special structure — only leaves are labeled and non-leaf nodes must be of degree two or more (Sect. 2.1) — which affects the definition of the subtree operation itself. This demands a specialized data mining approach. To our knowledge Zhang et al's Phylominer [88] is the only published algorithm for mining phylogenetic trees; thus, it is the reference point for evaluating our work. Phylominer is an Apriori-like approach that uses rightmost path extension for candidate generation and an occurrence list for frequency counting. It introduces a novel phylogeny-specific canonical form for evolutionary trees, which we also use in our approach. The candidate generation strategy highlights structural properties of phylogenetic trees that are useful for efficient candidate generation.

1.1.2 Graph Mining and Itemset Mining

Graph mining [1, 91, 44] is closely related to subtree mining, in fact, tree mining can be viewed as a special case of graph mining. Reference [1] gives a comprehensive survey on the topic. Itemset mining (or mining of association rules) [2, 12, 53] is closely related to both graph mining and tree mining. Graphs and trees can be seen as itemsets with additional constraints resulting from their respective topologies. The classical Apriori [2] algorithm has been the most influential in this field — as duly noted in [82] — and has influenced a lot of subtree mining approaches [16, 45]. It is not surprising that it is also the basis of our work.

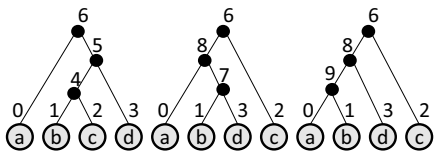
1.1.3 Maximum Agreement Subtrees and Majority Rule Trees

Maximum agreement subtrees (MASTs) [29, 47, 4] and consensus trees [14, 71] are perhaps the approaches most often-used by evolutionary biologists to identify common phylogenetic information in collections of trees. An MAST of a



(a) (b)

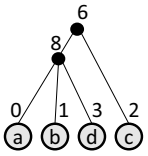
Fig. 1 (a) A collection of input trees. The lightly shaded part indicates the embedding of the common MAST — shown separately in (b).



(a)

Cluster	Frequency
0 - (a)	3
1 - (b)	3
2 - (c)	3
3 - (d)	3
4 - (bc)	1
5 - (bcd)	1
6 - $(abcd)$	3
7 - (bd)	1
8 - (abd)	2
9 - (ab)	1

(b)



(c)

Fig. 2 (a) A collection of input trees. (b) The clusters in the input trees along with their frequencies. Clusters $0-3, 6$ and 8 occur in the majority of the input trees. (c) The MRT.

collection of input trees is a common embedded subtree with the largest number of leaves (see Fig. 1). Consensus methods aim to find one tree that includes all the species, and captures the common information in the collection of trees. Among the most popular such methods is the majority-rule tree (MRT) [55], defined as follows. A *cluster* in a tree is the set of all leaf descendants of some node in the tree. The MRT of a collection of trees is the tree that exhibits all clusters present in the majority of the input trees (see Fig. 2).

The MRT and MAST problems have notably different complexities (see Table 2). The MRT can be constructed in time that is linear in the total size of the input trees [3]. In contrast, while an MAST of two trees can be computed in polynomial time [4, 47], finding an MAST of three or more trees is NP-hard if the trees have unbounded degree [4]. The problem is polynomially-solvable when the maximum node-degree is bounded by a constant [26].

Aside from speed, an important consideration when choosing a method to extract common substructures from collections of trees is the number of leaves in the subtree extracted; the larger the better. Here, MRTs have an advantage over MASTs, since the latter often have fewer leaves. MASTs, on the other hand, are in a sense stronger representatives of the common substructures, because each MAST is embedded as a subtree in each tree in the input collection. This cannot be said about the MRT. In fact, the MRT may not be embedded as a subtree in any of the input trees.

Reference	Consensus Approach	Max. input trees	Max. node-degree	Complexity
Amenta et al [3]	MRT	Any	Any	$O(tn)$
Cole et al [19]	MAST	2	2	$O(n \log n)$
Steel and Warnow [75]	MAST	2	Any	$O(n^{4.5} \log n)$
Bryant [13, pp. 174–182]	MAST	Any	2	$O(tn^3)$
Farach et al [26]	MAST	Any	d	$O(tn^3 + n^d)$
Amir and Keselman [4]	MAST	Any	Any	NP-hard

Table 2 An overview of MAST and MRT approaches. t denotes the number of input trees and n the size of the common leafset.

Another factor in selecting a method is the degree of resolution of the trees it produces. We say a phylogeny is well-resolved if its number of internal edges is high, relative to its number of leaves. More formally, we define the resolution of a tree T as

$$\text{resol}(T) = \frac{|\text{internal edges in } T|}{|\text{leaves in } T| - 2} \times 100\%. \tag{1}$$

(A similar measure was used before by Pattengale et al [61].) The denominator in (1) is simply the maximum number of internal edges a tree can have, and is used to normalize the degree of resolution across trees of different sizes. The more resolved a phylogeny is, the more informative it is considered to be. The least informative tree we can have is a *fan*, i.e., a star-like tree with zero internal edges. A fan states the trivial fact that the species at its leaves descend from a common ancestor. It has been observed that MRTs tend to be poorly resolved [31]; MASTs typically perform better than MRTs in this regard.

FSTs offer several advantages over MASTs and MRTs:

- An FST is usually more resolved than the MRT and always has at least as many leaves as an MAST. This is because an MAST is, by definition, also an FST, because it is supported by every input tree. (Note that the MRT is not guaranteed to be an FST.) Indeed, the requirement that an MAST be supported by all input trees rules out subtrees that might be more informative than any MAST but still quite frequent and thus equally important for phylogenetic inference [88]. An example of this is shown in Fig. 3. Here, the FSTs are better resolved than both the MAST and the MRT, which are fans. For instance, the first FST in Fig. 3 indicates that $s1$ and $s2$ are closer to each other than each is to any of the species among $s3 - s5$.
- FSTs can reveal a more complete phylogenetic picture than an MAST or an MRT. MASTs and MRTs tend to depict phylogenetic relationships among only a limited subset of the species [31]. For example, in Fig. 4 neither the

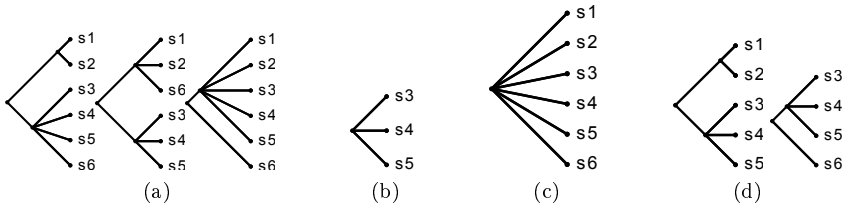


Fig. 3 (a) A collection of input trees, along with (b) their MAST, (c) their MRT, and (d) two FSTs with 67% support (i.e., two out of the three input trees support them).

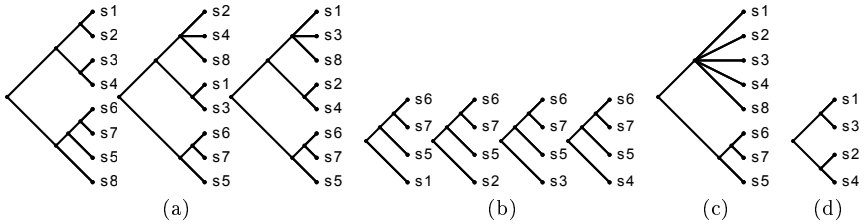


Fig. 4 (a) A collection of input trees, together with (b) their MASTs, (c) their MRT, and (d) an FST with support 67%.

MRT nor any of the MASTs gives information about relationships among species $s1 - s4$, but the FST does. In fact, by itself, an MAST can be misleading [77].

- FSTs are useful for solving MAST-related problems. These include finding a maximum compatible subtree [31], finding a maximum agreement supertree [35], and computing the kernel of maximum agreement subtrees [77].
- FSTs can be used to mine subtree patterns on collections of trees having leaf sets that are partially overlapping but not identical. This ability is particularly useful in mining phylogenetic databases such as TreeBASE [65] and PhyLoTA [68], which contain trees with different leaf sets, and where the number of common leaves is relatively small. Neither MASTs or MRTs are suitable for this purpose. For instance, while we could apply MASTs by simply restricting all the trees to the common leaf set, this would fail to give any results when the common overlap among the leaf sets is low; see Fig. 5. Indeed, just a couple of trees with no or very little overlap can result in such a scenario in a collection with any number of trees.

1.2 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 [88] — the current state-of-the-art algorithm for

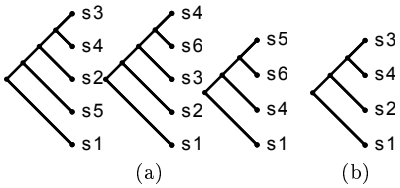


Fig. 5 (a) A collection of input trees on partially overlapping leafsets. Restricting the input trees to the common leafset (i.e. species $s1$ and $s4$) and then applying MAST or MRT will not yield any useful information. However, (b) the FST contains species $s1 - s4$, which is clearly informative.

the same task. The 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 breadth-first candidate enumeration mode (like Apriori [2]) as well as in depth-first enumeration [87] mode. The first is quicker, while the second uses less memory, enabling it to handle larger trees. An implementation of EvoMiner, which also works on collections of trees with partially overlapping leaf sets, is available by request from the authors.

Finally, we demonstrate experimentally that, for a wide collection of biological datasets, FST mining (whether or not it is done with EvoMiner) has significant advantages over the use of MASTs and MRTs. That is, FST mining can produce agreement subtrees that are significantly larger than MASTs and more resolved than the MRT (see Sect. 4.2).

2 Preliminaries

2.1 Phylogenies

As usual, a ***rooted tree*** is a connected acyclic graph, which contains a special node called the ***root***. The ***depth*** of a node u in a rooted tree T, denoted $\text{depth}^T(u)$, 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 in T by $\text{LCA}^T(u, v)$. When the tree T is clear from the context, we drop the superscripts. A ***k-leaf tree*** is a tree with k leaves. An edge is called ***internal*** if neither of its end points is a leaf node. Two nodes are called ***siblings*** if they have a common parent node.

A ***phylogenetic tree*** (or, for brevity, simply a ***tree*** or a ***phylogeny***) is a rooted[2] tree where every internal (i.e., non-leaf) node has at least two children, whose leaves are in a bijection with a set of labels. The labels in a phylogenetic tree represent a set of taxonomic units — species — under consideration, and the branching structure of the tree represents the history of the evolution of the species from a common ancestor [9]. Indeed, each internal node in a phylogenetic tree represents a hypothetical ancestor or an event (e.g., emergence of a new species) in the evolutionary history that separates the ancestor and descendants at that node.

Let $\mathcal{L}_T$ denote the leaf label set of tree T, and ψ_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 $\mathcal{L}_T$. From this point forward, unless the context requires making a distinction, we will drop the subscripts in $\mathcal{L}_T$ and ψ_T, and write $\mathcal{L}$ and ψ respectively. For the rest of the paper, we assume without loss of generality that the leaf label set $\mathcal{L}$ consists of distinct integers in the range $[1, |\mathcal{L}|]$; thus, the labels are ordered.

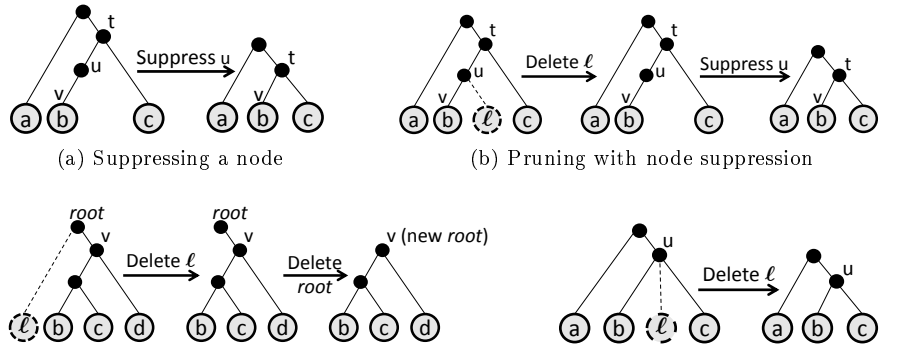
Let u be an internal non-root node in some tree (not necessarily a phylogenetic tree), such that u has only one child v. Then, *suppressing* u means contracting the edge (u, v); i.e., deleting u and adding an edge from the parent of u to v. For example, in Fig. 6a, to suppress u, it is deleted and an edge is added from t to v. Node u is a ***neighbor*** of node v if edge (u, v) exists. To *prune* a leaf ℓ, we first delete it. Let u be ℓ's neighbor. If u is not the root, and the deletion of ℓ makes u a degree-two node, we suppress u (see Fig. 6b). If u is the root and deleting ℓ makes it a degree one node, u is deleted and its neighbor becomes the new root (see Fig. 6c). Otherwise, u remains as it is (see Fig. 6d).

Consider a tree T and a set $\mathcal{L}' \subseteq \mathcal{L}_T$. The *restriction* of T to $\mathcal{L}'$, denoted by $T|_{\mathcal{L}'}$, is the tree obtained by pruning leaves $\mathcal{L}_T - \mathcal{L}'$ from T. We denote the fact that two trees T_1 and T_2 are isomorphic by writing $T_1 \equiv T_2$. Given two phylogenies T and T', we say that T ***displays*** T' —or, equivalently, that T' is ***displayed by*** T— if $\mathcal{L}_{T'} \subseteq \mathcal{L}_T$ and $T' \equiv T|_{\mathcal{L}_{T'}}$ [72]. For example, each of Fig. 7a and Fig. 7b shows two trees such that the tree on the left displays the tree on the right. For phylogenetic trees, the notion of "displayed by" replaces the usual notion of "embedded subtree" that is commonly used in the data mining literature (e.g., as in [87]).

2.2 Frequent Subtree Mining in Phylogenetics

Let $D = \{T_1, T_2 \ldots T_n\}$ be a database of n trees on a common label set $\mathcal{L}$. Let minSup $\in [1, n]$ be an input parameter. A tree T with $\mathcal{L}_T \subseteq \mathcal{L}$ is said to be a (phylogenetic) ***frequent subtree*** (FST) in D, if there exists $D' \subseteq D$ with $|D'| >$ minSup such that for all $T' \in D'$, T' displays T. $|D'|$ is called

[2] Phylogenies can also be unrooted [58], but here we deal exclusively with rooted phylogenies.



(a) Suppressing a node

(b) Pruning with node suppression

(c) Pruning a leaf attached to a degree-two root

(d) Pruning without node suppression

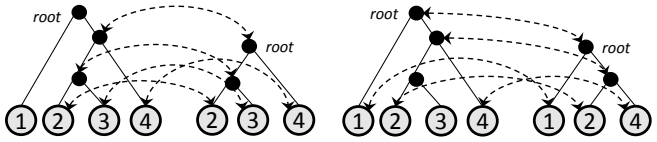
Fig. 6 Pruning in phylogenetic trees

the ***support*** of T in D, and is denoted by $\sup(T)$. The (phylogenetic) ***FST mining problem*** is to identify all the FSTs in D.

There are significant differences between phylogenetic FST mining and unordered tree mining. These dissimilarities stem from the peculiarities of phylogenetic trees — only the leaf nodes are labeled and all internal nodes must have at least two children — and from the fact that we are looking for displayed trees rather than embedded subtrees. When applied to collections of phylogenetic trees, ordinary methods for unordered tree mining can yield trees with redundant internal nodes; i.e., nodes with just one child. Figures 7a and 7b show the same tree with two different displayed trees. Of these two displayed trees, the one in Fig. 7b is not an embedded subtree according to the standard definition, since, to obtain it, we not only need to delete a leaf, but we also have to suppress a node. Mining for displayed trees, rather than embedded subtrees makes sense in phylogenetics, where we are primarily interested in the groupings among species. Thus, for the tree in Fig. 7b species '2' and '4' are evolutionarily closer to each other than either is to species '1'. The same relative evolutionary information is preserved even if we prune species '3'. This characteristic of phylogenies offers some advantages for FST mining, since it enables us to use efficient LCA-based techniques. At same time it limits the application of existing methods in data mining literature for enumerating frequent subtrees.

We should note that the edges of phylogenetic trees sometimes have numerical values associated to them, called branch lengths, representing the time elapsed or other measure of divergence separating ancestor and descendant during evolution. Dealing with branch lengths is beyond the scope of this paper.

Next, we describe several concepts that are essential to EvoMiner, including canonical form, equivalence classes, and prefix trees. Most of these notions are illustrated in Fig. 8.



(a) Without node-suppression (b) With node-suppression

Fig. 7 Phylogenetic subtree. Arrow mappings indicate the nodes that were retained from the original tree. The topmost node represents the root.

2.2.1 Canonical Form

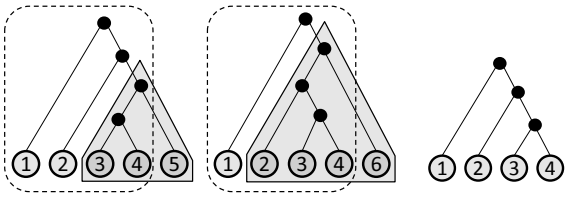
The ***virtual label*** of an internal node v is the minimum label among all leaf descendants of v. The children of an internal node are ordered from left to right based on the sequence in which they are encountered in an inorder depth-first traversal (IDFT), the leftmost child being encountered first. A tree T is in ***canonical form*** [88] if, for every internal node, its children are ordered from left to right by their virtual labels. It can be seen that two trees are isomorphic if and only if they have the same canonical forms. By generating all trees in canonical form, it is straightforward to test if two trees are isomorphic and prevent duplicate enumeration. EvoMiner relies on this property to ensure that each FST is enumerated exactly once.

2.2.2 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 canonical form is that pruning either the last leaf or the second-to-last leaf encountered in the IDFT yields a subtree that is also canonical [88]. The subtree resulting from pruning the rightmost leaf is called the ***prefix tree*** or simply the ***prefix***. The ***heaviest subtree*** [88] is the subtree rooted at the parent of the rightmost leaf.

2.2.3 Equivalence Class

Let R denote the relation: 'sharing a common prefix' between two canonical trees. Note that R is an equivalence relation. Two canonical trees T and T' are said to be equivalent with respect to R (or simply equivalent), denoted $T \sim T'$ if they share a common prefix. An ***equivalence class*** E is a set of canonical trees that are pair-wise equivalent; i.e., all trees in E share a common prefix tree; that is, for every pair $(T, T') \in E, T \sim T'$. Thus, all trees in E share a common prefix tree. The shared $(k-1)$-leaf prefix tree, called the ***core tree***, uniquely identifies the members of an equivalence class. Any two trees in an equivalence class differ with respect to their rightmost leaf and (topologically) with respect to their heaviest subtrees. The partition of the set of frequent k-leaf trees into equivalence classes is the basis for our enumeration approach, which generates larger frequent subtrees by extending the core tree (Sect. 3.1).



(a) Equivalence class (b) Prefix tree

Fig. 8 (a) Two trees belonging to the same equivalence class. The common prefix tree (shown separately in (b)) 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.

3 The EvoMiner Algorithm

Fig. 9 gives a high-level description of EvoMiner. The algorithm uses an Apriori-like [2] 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 $\text{LCA}(u, v)$ and stores it in a three-dimensional array indexed by triplet (T, u, v). In our implementation, these LCA values are computed in quadratic time and space per tree by traversing the tree in a depth-first manner and computing the LCA values of the leaf-descendants at a node. For a given database D on the leaf set $\mathcal{L}$, this one-time task takes $O(|D||\mathcal{L}|^2)$ time. We should point out that it is well-known that one can pre-process a tree in linear time and space to produce a data structure that can answer any LCA query on that tree in constant time [41, 70, 11]. Such algorithms are quite useful when the number of LCA queries is limited and the pre-processing dominates the total time. That is not the case in our application. Indeed, EvoMiner queries all possible LCA values while enumerating all FSTs on three leaves, and then does a constant number of LCA queries for every FST generated thereafter. Although both our three-dimensional array and the specialized LCA data structures [41, 70, 11] offer constant-time access to LCA-values, the former's constant factor is smaller than the latter's, which makes a significant difference in practice.

After the LCA mappings are computed, 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 is 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 involve frequent traversals through the input database and subsequent subtree operations. Next, we describe how each of these steps is implemented in EvoMiner.

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

enumerateNextLevel(EC_k, minSup)
1: $EC_{k+1} \leftarrow \emptyset$
2: **for all** $e \in EC_k$ **do**
3: **for all** $T_x \in e$ **do**
4: $e_{\text{next}} \leftarrow \emptyset$
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^{join}) **then**
9: **if** sup(T^{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}

Fig. 9 The EvoMiner Algorithm

3.1 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 pairwise joining of frequent subtrees within an equivalence class to produce larger candidate trees (sometimes referred to as equivalence class based extension [87]). This is achieved in constant time per pair. The second part 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 [2, 15]. The enumeration of FSTs starts with *triplets* (trees on three leaves), as triplets are the smallest trees that offer meaningful evolutionary information.

3.1.1 Pairwise Extension

Let $\langle T_x, T_y \rangle$ be an ordered pair of distinct frequent k-leaf trees from the same equivalence class e of EC_k. The pairwise extension operation "joins" $\langle T_x, T_y \rangle$ in all possible ways so as to generate a set join(T_x, T_y) of $(k+1)$-leaf candidate trees, called ***joined trees***, such that every $T^{\text{join}} \in$ join(T_x, T_y) satisfies the following:

$$T^{\text{join}} \text{ is in canonical form, } T_x \text{ is the prefix of } T^{\text{join}}, \text{ and } T_y \equiv T^{\text{join}}|_{\mathcal{L}_{T_y}}. \quad (2)$$

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. Thus, T^{join} belongs to the equivalence class with T_x as its core tree. Since T_x and T_y are in the same equivalence class, they differ only with respect to their heaviest subtrees. This fact restricts the possible ways in which they can be joined.

Condition (2) implies that T^{join} is obtained by attaching the rightmost leaf of T_y to the rightmost path of T_x (the path from the root to the rightmost leaf); this is known as a *rightmost path extension* [6]. 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 numChild(u) denote its number of children. To satisfy (2), T^{join} can have one of four possible topologies, which we refer to as type 1–4 joins. These are described next.

Type 1: In this case, as shown in Fig. 10a, the leaves x and y of the participating trees are attached at the same depth on the rightmost path of T^{core}; i.e., depth(p_y) = depth(p_x). In T^{join}, x and y are siblings, and their depths are the same as in T_x and T_y; see Fig. 10b. For T^{join} to be canonical, we must have $\psi(x) < \psi(y)$ (recall that we assume that the labels are distinct numbers). Figures 11a and 11b exemplify type 1 joins.

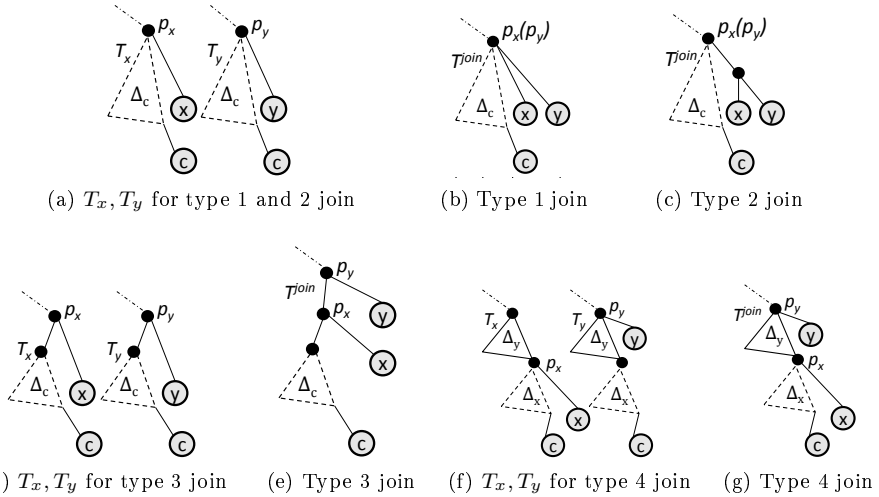
Type 2: As in type 1 joins, depth(p_y) = depth(p_x) and x and y are siblings in T^{join} (Fig. 10a). In T^{join}, however, each of x and y is one level deeper than it was in T_x and T_y; see Figure 10c. Thus, pruning either x or y in T^{join} leaves the parent with only one child, so the parent is suppressed. (This suppression does not occur in Type 1 joins, because the common parent of x and y in T^{join} must have degree greater than two; see Figure 10b.) Figures 11a and 11c exemplify type 2 joins.

Type 3: Fig. 10d shows the participating trees. As in type 1 and 2 joins, depth(p_y) = depth(p_x). In this case, however, p_y is the parent of p_x in T^{join}; see Figure 10e. Since pruning x in T^{join} causes its parent node to be suppressed (see Fig. 10e), the only way we can get this T^{join} is if numChild(p_y) = numChild(p_x) = 2. Figures 11d and 11e exemplify type 3 joins.

Type 4: In this case, as shown in Fig. 10f, depth(p_y) < depth(p_x). Thus, the depth of y on the rightmost path of T^{core} is lower than that of x. As a result there is only one way to join T_x and T_y so as to satisfy condition (2). See Fig. 10g. Note that p_y becomes an ancestor of p_x in T^{join}. Figures 11f and 11g exemplify type 4 joins.

Observe that if depth(p_y) > depth(p_x), we cannot join T_x and T_y while satisfying condition (2), since T_x cannot be the prefix tree in this case. FSTs from such joins are enumerated when considering the ordered pair $\langle T_y, T_x \rangle$.

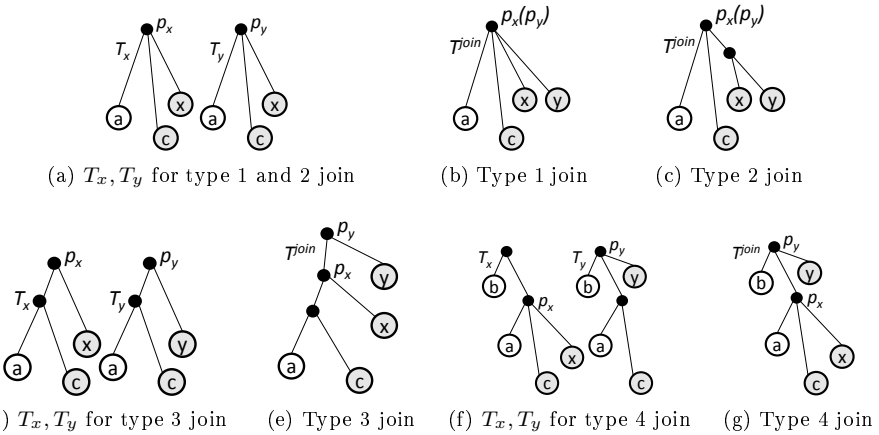
Clearly, we can identify in constant time the type(s) of join resulting from an ordered pair of trees in an equivalence class and the tree resulting from each case is canonical. Hence, a join can be done in constant time for a pair of input trees. This is an important difference with respect to Phylominer, where the candidate generation scheme requires comparing the respective topologies



(a) T_x, T_y for type 1 and 2 join (b) Type 1 join (c) Type 2 join

(d) T_x, T_y for type 3 join (e) Type 3 join (f) T_x, T_y for type 4 join (g) Type 4 join

Fig. 10 Different types of pairwise join. A dotted triangle represents a part of the tree that may be empty, while a solid triangle represents a non-empty part of the tree. Δ reflects the topologies of the heaviest subtrees. 'c' denotes the rightmost leaf of the common core tree.



(a) T_x, T_y for type 1 and 2 join (b) Type 1 join (c) Type 2 join

(d) T_x, T_y for type 3 join (e) Type 3 join (f) T_x, T_y for type 4 join (g) Type 4 join

Fig. 11 An example for each of the join type shown in Fig. 10.

of the heaviest subtrees of the input trees, which takes $O(k)$ time. Another difference is that by comparing depth(p_y) with 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 1 *Pairwise extension enumerates all FSTs and each FST is enumerated only once. Moreover the enumerated FSTs are in canonical form.*

Proof We use induction on k, the number of leaves in an FST. If $k = 3$, then all such FSTs are uniquely generated during triplet enumeration, the starting step of Algorithm 9. Consider $k > 3$. Assume all FSTs on $k-1$ leaves have been uniquely enumerated. Let T be a k-leaf FST in 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. Since T is in canonical form, so are T_x and T_y. 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. Since T satisfies condition (2) with respect to T_x and T_y, we have $T \in \text{join}(T_x, T_y)$. Thus pairwise extension must enumerate T. Further, if the members of e are considered in an ordered fashion for pairwise extension so that $\langle T_x, T_y \rangle$ is considered only once, then T will also be enumerated only once. □

3.1.2 Downward-Closure Operation

The downward closure operation takes a k-leaf candidate tree T (generated by pairwise extension) and checks whether all k of its $(k-1)$-leaf subtrees are frequent. Thus, it requires all $(k-1)$-leaf frequent subtrees to 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 [2, 88]. 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 [88]. We next present 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. In our computational experiments, the fingerprinting based pruning technique enables us to achieve speed-ups up to 100% as compared to an alternative depth-first mining approach (see Sect. 3.5).

We employ the fingerprint function used in the Rabin–Karp pattern-matching algorithm [48]. For this, we represent each tree in the database by the sequence of nodes encountered in the IDFT of the tree (this traversal sequence is called an Euler tour). For example, the string representation of the tree shown in Fig. 8b is 'D1UDD2UDD3UD4UUU', where 'D'/'U' respectively represent

downward/upward traversal in the tree, and are selected such that $\mathrm{D}, \mathrm{U} \notin \mathcal{L}$. 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 $s = (s_1 s_2 \dots s_b)$, denoted $F_p(s)$, is computed as:

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

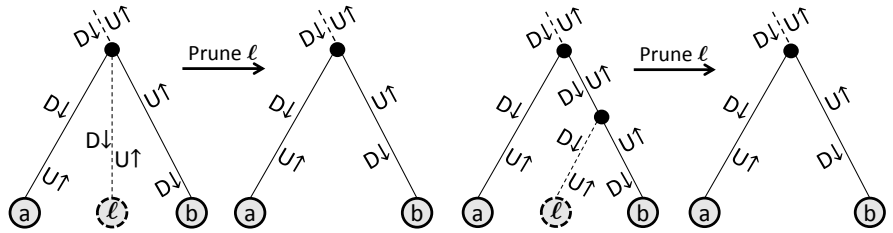
where p is a random prime. While the cost of computing the original fingerprint for a b-bit integer is $\Theta(b)$, a fingerprint can be updated in constant time after deletion of a substring [57]. For example given the fingerprint of the string '123456', we can compute the fingerprint of the result of deleting substring '34' by observing that $F_p(\text{'1256'}) = (F_p(\text{'123456'}) - F_p(\text{'1234'}) \times 2^{\text{length}(\text{'56'})} + F_p(\text{'12'}) \times 2^{\text{length}(\text{'56'})}) \bmod p$, which is easy if the fingerprints for the prefix strings '1234' and '12' have been pre-computed.

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. This allows us to select a large p. This is helpful because for two random input strings and a fixed prime p, the probability that the two strings have the same fingerprint is $\frac{1}{p}$. 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 Fig. 12, there are two possible cases for pruning a leaf ℓ. In the first case (Fig. 12a), no node is suppressed and the string representation changes from '...DDaU$D\ell U$DbUU...' to '...DDaUDbUU...', where a and b are the siblings of ℓ. This involves deletion of substring $D\ell U$ (emphasized in *italics* in the original string). The fingerprint is updated as:

$$\begin{aligned} F_p(\text{'...DDaUDbUU...'}) = (&F_p(\text{'...DDaU}D\ell U\text{DbUU...'}) \\ &- F_p(\text{'...DDaU}D\ell U\text{'}) \times 2^{\text{length}(\text{'DbUU'})} \\ &+ F_p(\text{'...DDaU'}) \times 2^{\text{length}(\text{'DbUU'})}) \bmod p. \end{aligned}$$

In the second case (Fig. 12b), a node is suppressed and the string representation changes from '...DDaUD$D\ell U$DbUUU...' to '...DDaUDbUU...'. This involves deletion of two substrings: '$D\ell U$D' and 'U' (emphasized in *italics* in the original string). Each substring is deleted one at a time. On deleting '$D\ell U$D'



(a) Without node-suppression (b) With node-suppression

Fig. 12 Fingerprint update in pruning a leaf

from '...DDaUD$D\ell UD$bUUU...', the fingerprint is updated as:

$$\begin{aligned} F_p(\text{'...DDaUDbUUU...'}) = (&F_p(\text{'...DDaUD}D\ell UD\text{bUUU...'}) \\ &- F_p(\text{'...DDaUD}D\ell UD\text{'}) \times 2^{\text{length('bUUU')}} \\ &+ F_p(\text{'...DDaUD'}) \times 2^{\text{length('bUUU')}}) \bmod p. \end{aligned}$$

Next, on deleting the second substring 'U' from '...DDaUDbUUU...', the fingerprint is updated as:

$$\begin{aligned} F_p(\text{'...DDaUDbUU...'}) = (&F_p(\text{'...DDaUDb}U\text{UU...'}) \\ &- F_p(\text{'...DDaUDb}U\text{'}) \times 2^{\text{length('UU')}} \\ &+ F_p(\text{'...DDaUDb'}) \times 2^{\text{length('UU')}}) \bmod p. \end{aligned}$$

Each of the above updates can be achieved in constant time if the fingerprints of all the prefixes of the string representation of the original tree is pre-computed. Computing all prefixes 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 for 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 away from enumeration.

When using fingerprints as just described, there is a small probability of a false match[3]; 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.

[3] Recall that this probability is $\frac{1}{p}$ for a chosen prime p (Sect. 3.1.2, page 17).

Note that false matches do not affect the correctness of the algorithm since a false match never prunes away a frequent subtree from enumeration.

Note that any subtree obtained by deleting a leaf that is the leftmost child of its parent may not be canonical. This is because in canonical form the virtual label of a node is the same as its leftmost child. Deleting such a leaf results in a new leftmost child for its parent, and hence, a new virtual label, which is greater than the previous one. This new virtual label of the node may be greater than the virtual label of its next sibling in the IDFT of the tree — requiring repositioning of the node among its siblings to restore the canonical form. This repositioning effect can cascade all the way to the root if each node on the path from the leaf being deleted to the root is the leftmost child of its parent. Thus our fingerprinting scheme does not consider any such subtree. In the worst case, only half of the subtrees are considered. This leads to the possibility of a *false positive* with respect to 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 to 4%) when compared to a complete downward-closure check, which considers all $(k-1)$-leaf subtrees (see Sect. 4.1). The explanation for the low rate of false positives is that our problem empirically exhibits an *abundance of witnesses* property [42, chapter 6]. The goal here is to identify a 'guilty' infrequent candidate k-leaf tree posing as a frequent one. A witness here is an infrequent $(k-1)$-leaf subtree of the candidate tree that witnesses against the latter being frequent. However, every $(k-1)$-leaf subtree is not infrequent, i.e., a witness. Thus, the more $(k-1)$-leaf subtrees are checked for being infrequent, the higher are the chances of coming across a witness. 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 frequency counting step.

3.2 Frequency Counting

For each frequent subtree t, we maintain an occurrence list containing all the trees in the database that have t as a subtree. While considering a tree $T^{\text{join}} \in \text{join}(T_x, T_y)$, we first compute the intersection of the occurrence lists of T_x and T_y. We then count how many trees in the intersection display T^{join}. Let $\mathcal{L}^{\cup} = \{\mathcal{L}_{T^{\text{core}}} \cup x \cup y\}$ denote the leaf set of T^{join}, where T^{core} is the common prefix tree of T_x and T_y. For each tree T in the intersection list, EvoMiner uses an LCA-based scheme to determine in constant time whether T^{join} is displayed by T —i.e., if $T^{\text{join}} \equiv T|_{\mathcal{L}^{\cup}}$— without actually computing $T|_{\mathcal{L}^{\cup}}$. In contrast, Phylominer takes linear time for the corresponding step, because

for every tree T in the intersection list, it explicitly constructs $T|_{\mathcal{L}^\cup}$ and then checks if this tree is isomorphic to T^{join} (using the linear-time algorithm of [80]).

For a given pair $\langle T_x, T_y \rangle$ of trees in an equivalence class, and a tree T in the intersection of the occurrence lists of T_x and T_y, the subtree $T|_{\mathcal{L}^\cup}$ must be the result of one of the four types of joins on $\langle T_x, T_y \rangle$ as described in Sect. 3.1.1. 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 Sect. 3.1.1.

Lemma 1 *$T|_{\mathcal{L}^\cup}$ is the result of a type 1 join if and only if*

1. $\text{depth}(\text{LCA}^T(c, x)) = \text{depth}(\text{LCA}^T(c, y))$,
2. $\text{depth}(\text{LCA}^T(c, x)) = \text{depth}(\text{LCA}^T(x, y))$, *and*
3. $\psi(x) < \psi(y)$.

Proof Clearly if $T|_{\mathcal{L}^\cup}$ is the result of a type 1 join, it satisfies conditions 1–3. To prove the *only if* part, let conditions 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}, condition 1 implies that $\text{depth}(p_y) = \text{depth}(p_x)$. Thus, $T|_{\mathcal{L}^\cup}$ must be a join of type 1, 2 or 3. Now, a type 3 join requires the parent of y to be an ancestor of the parent of x in $T|_{\mathcal{L}^\cup}$ — a case ruled out by condition 1. Further, conditions 2 and 3 imply that the join must be of type 1. □

Lemma 2 *$T|_{\mathcal{L}^\cup}$ is the result of a type 2 join if and only if*

1. $\text{depth}(\text{LCA}^T(c, x)) = \text{depth}(\text{LCA}^T(c, y))$,
2. $\text{depth}(\text{LCA}^T(c, x)) < \text{depth}(\text{LCA}^T(x, y))$, *and*
3. $\psi(x) < \psi(y)$.

Proof The proof is similar to that of Lemma 1. Again $T|_{\mathcal{L}^\cup}$ can be the result of either a type 1 or a type 2 join. Conditions 2 and 3 imply that the join must be of type 2. □

Lemma 3 *$T|_{\mathcal{L}^\cup}$ is the result of a type 3 join if and only if*

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

Proof Condition 2 implies that $T|_{\mathcal{L}^\cup}$ must be the result of a join of type 1, 2 or 3. Condition 1 rules out type 1 and 2 joins. Thus, the join must be of type 3. □

Lemma 4 *$T|_{\mathcal{L}^\cup}$ is the result of a type 4 join if and only if* $\text{depth}^{T_x}(p_x) > \text{depth}^{T_y}(p_y)$.

Proof As per the given condition, $T|_{\mathcal{L}^\cup}$ can only be the result of a type 4 join. □

Theorem 2 *The frequency counting scheme correctly identifies $T|_{\mathcal{L}^\cup}$ as a result of a join of type 1, 2, 3 or 4 in constant time.*

Proof Clearly, Lemmas 1–4 are mutually exclusive and correctly identify $T|_{\mathcal{L}^\cup}$ as a result of a join of type 1, 2, 3 or 4. Further, each condition in the lemmas can be evaluated in constant time. The claim follows.

□

3.3 Correctness and Complexity Analysis

Theorem 1 proves that pairwise extension uniquely enumerates all potential FSTs. Theorem 2 proves that the frequency counting step correctly identifies all true FSTs out of the potential candidates. Thus, EvoMiner identifies all FSTs correctly and non-redundantly. We next discuss the time complexities of the different steps of EvoMiner (EM) and compare the total time with that of Phylominer (PM). In the process, we explain why EM is faster than PM. As before, let the input database be D, consisting of n trees on a common leafset $\mathcal{L}$. Let $\mathcal{F}$ denote the set of all FSTs.

Initialization. This one-time task involves (1) computing LCA mappings for all pairs of leaves for all the input trees and (2) enumerating all frequent triplets. Step 1 takes $O(n|\mathcal{L}|^2)$ time. Though this step is not done by PM, it takes only a small fraction of the total time, as $|\mathcal{L}|^2$ is negligible compared with $|\mathcal{F}|$, the total number of frequent patterns. Step 2 takes $O(n|\mathcal{L}|^3)$ time. PM starts by evaluating all frequent pairs of leaves instead of triplets. However, because it enumerates all FSTs, PM also enumerates all frequent triplets. Thus the net complexity for evaluating all frequent triplets is the same in PM as in EM.

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

Downward Closure. Both EM and PM use downward closure to prune infrequent candidates. For a k-leaf candidate tree, PM checks whether all k of its $(k-1)$-leaf subtrees are frequent by referring to a hash table that stores the previously enumerated frequent $(k-1)$-leaf subtrees. This takes $O(k^2)$ time, as there are $O(k)$ subtrees and indexing each takes $O(k)$ time. EM does the same operation in $O(k)$ time through its fingerprinting scheme.

Frequency Counting. Both EM and PM use occurrence lists. For a potential k-leaf frequent candidate, PM examines every tree T in the intersection of the occurrence lists, computing the restriction of T to the leaf set of the candidate and then comparing this restricted tree to the candidate using a linear-time isomorphism check. PM takes $O(k)$ time for each of these steps. EM avoids computing the restriction, as well as the isomorphism test. Instead, it performs frequency counting in constant time via its LCA-based scheme. Overall, for each potential candidate, frequency counting in PM takes $O(nk)$ time while it takes $O(n)$ time in EM.

Theorem 3 *The time complexity of* EvoMiner *is* $O(n|\mathcal{L}|^3 + n|\mathcal{F}| + |\mathcal{L}||\mathcal{F}|)$ *where* n *is the number of trees in the database,* $\mathcal{F}$ *is the set of FSTs, and* $\mathcal{L}$ *is the common leaf set. When, as typically happens in practice,* $|\mathcal{F}| \gg |\mathcal{L}|^3$*, the time is* $O(n|\mathcal{F}| + |\mathcal{L}||\mathcal{F}|)$.

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 $O(k)$ time, which is $O(|\mathcal{L}|)$. The downward closure operation for each k-leaf candidate takes $O(k)$ time, which is $O(|\mathcal{L}|)$. The frequency counting step takes $O(n)$ time for each candidate. Totaling these estimates, we obtain the claimed bound. □

3.4 Extension to Partially Overlapping Leaf Sets.

So far, we have assumed that all trees in the database have the same leaf set. The sets of trees considered by evolutionary biologists often have only partially overlapping leaf sets. We can easily extend EvoMiner to mine such collections of trees, without any loss in efficiency. We do so as follows. While computing the LCA for all pairs of leaves for all the input trees during the initialization phase, we flag an LCA value if one of the leaves involved in the pair is not present in the input tree. These flagged LCA values are not considered during frequency counting. Note that such an extension is not as direct in Phylominer, as its frequency counting step is not based on LCA values.

3.5 Depth-first Mining

We can use many of the ideas behind EvoMiner in a depth-first mining scheme, along the lines of [87], which uses depth first search in the enumeration graph [16]. This alternative scheme does not require candidate generation (similar to [79, 38, 40]), since the frequency counting step gives sufficient conditions to distinguish among all possible candidates from pairwise-extension. Fig. 13 gives a high-level description of the depth-first enumeration scheme. The existence of a joined-subtree in line 5 can be checked in constant time per tree using the conditions given in Lemmas 1-4. We note that while depth-first 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 very large 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. In the next section, we give a bound on the memory required by the depth-first mining scheme. In Sect. 4, we give the results of an experimental evaluation of the trade-offs between depth-first and breadth-first mining.

3.6 Discussion

Apriori-based methods for mining frequent patterns —such as association rules, frequent closed itemsets, max-patterns, sequential patterns, or constraint-based mining of frequent patterns— can generate an exponential number of candidates for a frequent pattern (see [37, 32]). It turns out, however, that

EvoMinerDF(D, minSup)
1: computeLCA_Mappings(D)
2: $Ft \leftarrow$enumerateFrequentTriplets(D, minSup)
3: $EC_3 \leftarrow$ computeEquivalenceClasses(Ft)
4: **for all** $e \in EC_3$ **do**
5: depthFirstRecursive(e, minSup)

depthFirstRecursive(e, minSup)
1: **for all** $T_x \in e$ **do**
2: **print** T_x
3: $e^{T_x} \leftarrow \emptyset$
4: **for all** $T_y \in e$ such that $T_x \neq T_y$ **do**
5: **if** greater than minSup trees in D exhibit a common subtree T_{xy} over $\mathcal{L}_{T_x} \cup \mathcal{L}_{T_y}$ with T_x as its prefix **then**
6: $e^{T_x} \leftarrow e^{T_x} \cup T_{xy}$
7: **if** $e^{T_x} \neq \emptyset$ **then**
8: depthFirstRecursive(e^{T_x}, minSup)

Fig. 13 EvoMinerDF — depth-first enumeration

EvoMiner does not suffer from a similar combinatorial explosion. In fact, as we show next, the number of candidates generated in the enumeration of an FST is polynomially-bounded.

Theorem 4 *The number of candidates generated in the enumeration of an FST is $O(|\mathcal{L}|^3)$, where $\mathcal{L}$ is the common leaf set of the input trees. The number of candidates generated per FST is $O(|\mathcal{L}|)$.*

Proof An FST can have at most $|\mathcal{L}|$ leaves. Each such leaf is added during a pair-wise join within an equivalence class. Within an equivalence class, the maximum number of pairs that can participate in a join operation is $\binom{|\mathcal{L}|}{2}$, which is less than $|\mathcal{L}|^2$. Each such pair can result in a maximum of three types of joined candidates (Sect. 3.1.1)[4]. Thus, the maximum number of candidates generated in the enumeration of an FST is $3|\mathcal{L}|^2|\mathcal{L}|$, which is $O(|\mathcal{L}|^3)$ as claimed. Note that this is a worst-case estimate, as it includes both frequent and infrequent candidates. If we are to consider the number of candidates generated per FST, we only need to consider the maximum number of pair-wise joins in which an FST can participate within an equivalence class, i.e., $O(|\mathcal{L}|)$, which gives the second part of the result. □

The breadth-first enumeration scheme in EvoMiner requires all the FSTs to be kept in memory. This limits the size of the input trees, because the number of FSTs grow exponentially with the size of the trees (Sect. 4). However, the depth-first enumeration scheme in EvoMinerDFonly stores the FSTs of the ancestor equivalence classes while enumerating a k-leaf FST. The next result shows that the memory required by the depth-first enumeration scheme scales polynomially with the number of input trees and the size of the common leaf set.

[4] This will happen when the pair being considered for join operation has the same topology as the input trees for type 3 join. Such a pair will produce candidates of join types 1, 2, and 3.

Theorem 5 EvoMinerDF *requires* $O(n|\mathcal{L}|^3)$ *space, where* n *is the number of input trees and* $\mathcal{L}$ *is the common leaf set.*

Proof The enumeration tree has depth at most $\mathcal{L}$. Enumerating an FST at this depth will require storing $O(|\mathcal{L}|)$ ancestor equivalences classes, each of which can have at most $|\mathcal{L}|$ FSTs. Thus, the maximum number of FSTs to be stored is $O(|\mathcal{L}|^2)$. Storing each such FST requires $O(|\mathcal{L}|)$ space for the subtree and $O(n)$ space for the occurrence-list. Thus, the maximum space required to store all FSTs is $O(n|\mathcal{L}|^3)$. Adding to this the space required to store LCA mappings, which is $O(n|\mathcal{L}|^3)$, we get the claimed figure. □

To close this section, we note that pattern-growth methods [64, 63, 39] are a potential alternative to EvoMiner's Apriori approach. While the pattern-growth approach has been shown to scale well for large databases [38] and has been used to mine FSTs in collections of ordered trees [79], it is not obvious how to extend the technique to unordered trees. Indeed, although our string encoding of phylogenetic trees is basically an ordered tree representation, our encoding does not satisfy an essential property on which the known pattern-growth methods for unordered tree mining rely. That is, in our approach, the string encoding of a subtree is *not* a prefix of the string encoding of the original tree. In fact, deleting just one leaf from a tree can drastically change its string encoding.

4 Experiments and Results

To evaluate the performance of EvoMiner, as well as to test the effectiveness of the FST approach compared to MASTs and MRTs, we conducted experiments on real and simulated data. 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).

We note that in many of our experiments, we used support values of 99% and 50%. While these values may appear high compared to those commonly used in the data-mining literature (e.g., [79, 18]), they reflect standard practice in phylogenetics, which typically demands a strong consensus among the trees in the input collection. The stronger the consensus, the higher is the confidence that can be placed in the common representative tree. For example, MASTs (Sect. 1.1.3) require 100% support. Strict-consensus trees [14] are another example of 100% support, as they are built from the clusters present in all the input trees. Majority-rule consensus trees (Sect. 1.1.3) are a more relaxed version strict-consensus trees, where one only requires that a cluster be present in a majority of the trees —at least 50% support— for it to be included.

4.1 Performance of EvoMiner

We compare the performance of our algorithm with that of Phylominer [88] using the original C++ implementation of its authors. Our experiments involve four datasets, indexed as $D1-D4$. $D1$ and $D2$ consist of synthetic phylogenetic trees. Data set $D1$ closely resembles the one used to evaluate the performance of Phylominer. Each tree in $D1$ was produced by first generating a random binary tree based on the Yule model [85]; for each such tree, we randomly chose a set of 30% of the internal edges, and contracted all edges in the set. To produce dataset $D2$, we first generated one random tree, which was then replicated to get 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. Datasets $D3$ and $D4$ were taken from published phylogenetic analyses. $D3$ is from the Bayesian analysis of [52], while $D4$ consists of bootstrap trees from [61]. Bayesian analyses and bootstrap trees are typical candidates for consensus tree algorithms [76, 61]; the trees in these datasets have a very high commonality. We extracted datasets of different sizes (in terms of the number of leaves and the number of trees) from $D3$ and $D4$ by randomly selecting the required number of trees and restricting them on a random set of leaves of the required size.

Fig. 14 compares the performance of EvoMiner with Phylominer on datasets $D1-D4$. For each comparison, three different leaf sets of sizes 15, 25 and 35 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. The range of leaf set sizes and the number of trees reflects the typical inputs on which phylogenetic analyses involving MAST and related problems are done; see, e.g., [77, 88]. In each of the datasets, EvoMiner is faster than Phylominer by a factor of up to 100 (sometimes more).

For comparison purposes, the physical memory was capped at 4GB. This explains the missing entries in the graphs. EvoMiner is able to handle larger datasets — both in terms of the number of trees and the number of leaves — because it uses a vertical bitmap representation of the database [7]. In this representation, in the occurrence list of a FST, a bit is reserved for every tree in the database. If the minimum support value is very small, there is a risk of under-utilizing memory, because the number of unset bits would far exceed the number of set bits for the occurrence list of an FST. In our studies, however, the minimum support value is always greater than 50%. Thus, memory utilization was high.

While breadth-first enumeration has the advantage of downward-closure operation and vertical bitmap representation of the database results in a smart utilization of memory, the fact remains that the number of FSTs that can be enumerated is limited by the available memory. However, in depth-first mining mode memory is not a limitation because the enumeration tree is explored in

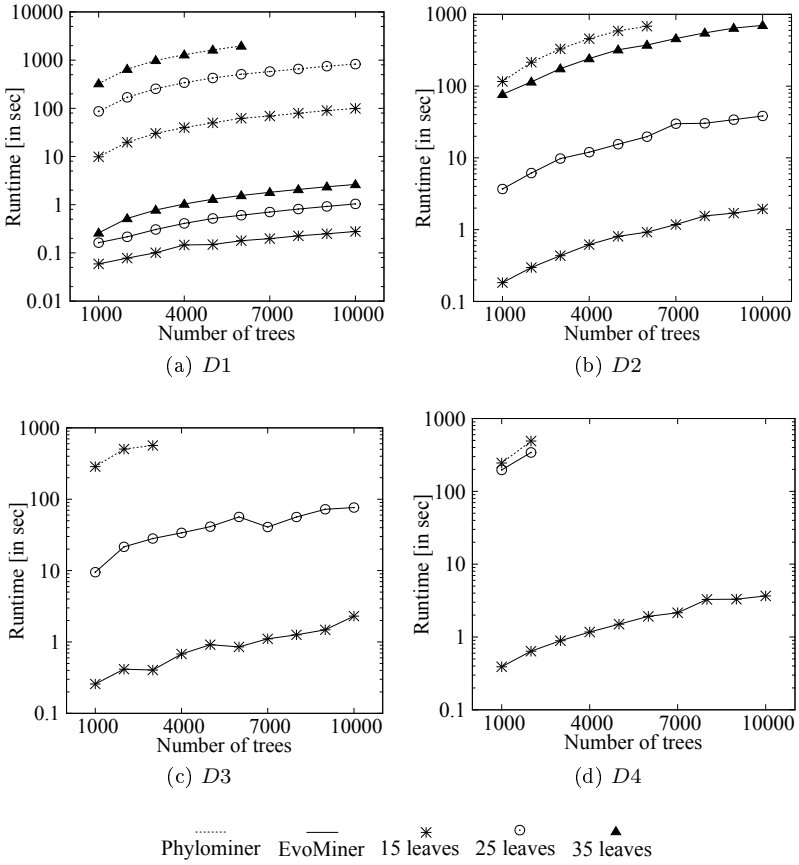


Fig. 14 Performance comparison

a depth-first manner. Thus, if the run time is not a consideration, users can mine up to 10000 trees on 254 leaves with the current implementation.

As shown in Fig. 14, the difference in runtimes of EvoMiner and Phylominer frequently reaches 1000 seconds in our experiments. This improved speed has a practical impact. As mentioned in the Introduction, phylogenetic analysis typically yields a collection of trees rather than a single tree. However, many of the generated trees are not part of the final output. For example, Markov chain Monte Carlo (MCMC) simulations used for Bayesian phylogenetic inference [56] involve multiple runs until convergence is reached. In such cases, it is essential to identify the common information in each run quickly. Speed is also important in summarizing the commonalities among phylogenetic trees during interactive visualizations [3].

Fig. 15a confirms the exponential growth of the number of frequent subtrees with an increase in the size of the leaf set. These experiments were done on the Bayesian analysis dataset $D3$ with 100 trees and minSup as 99%. Fig. 15b

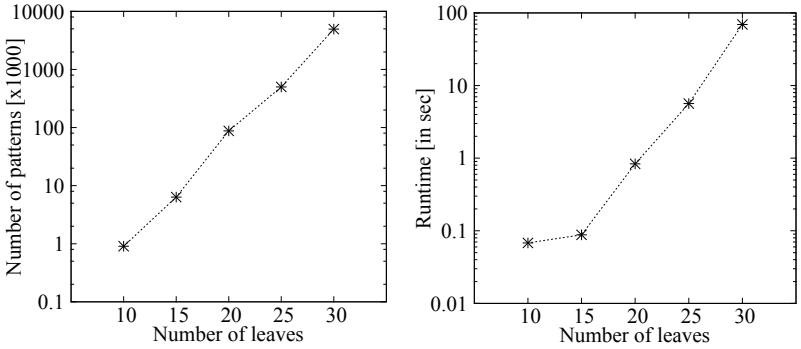
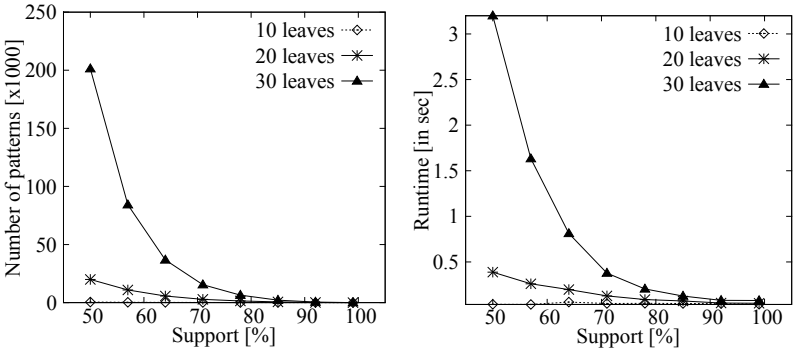


Fig. 15 Exponential growth of the number of frequent patterns and its effect on the run time.

shows the corresponding growth of the run time. The slope of this graph is steeper than that of Fig. 15a because, as indicated by Theorem 3, the run time depends not just on the the size of the leafset, $|\mathcal{L}|$, but also on $|\mathcal{F}|$, the number of frequent subtrees and on n, the number of trees. Theorem 3 also helps to explain the anomaly in the run time graph for $|\mathcal{L}|$ between 10 and 15: When $|\mathcal{F}|$ is small, we cannot assume that $|\mathcal{F}| \gg |\mathcal{L}|^3$, which makes the $n|\mathcal{L}|^3$ term in the running time non-negligible.

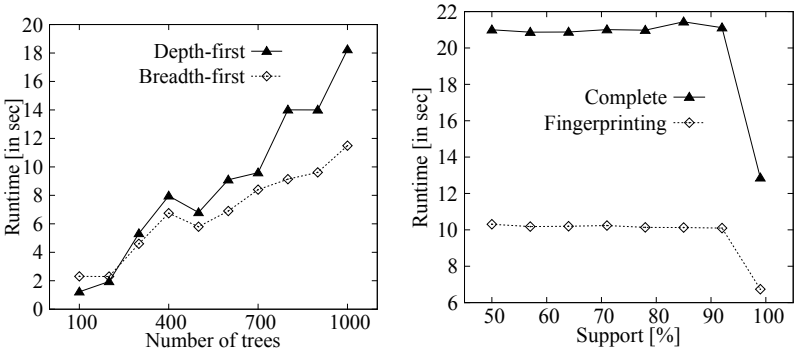
Fig. 16a shows how the number of frequent subtree patterns varies with respect to minSup on dataset $D2$ (500 trees). The exponential decrease in the number of frequent subtrees as minSup increases is to be expected: The pruning of a frequent tree has a cascading effect on all of its frequent supertrees. The run time behaves in a similar fashion (see Fig. 16b). The correlation between the number of frequent subtrees and the run time with respect to minSup confirms that the run time depends directly on the number of frequent subtrees generated.

Fig. 17a compares the performance of depth-first mining with the candidate generation based (breadth first enumeration) approach with respect to 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. Fig. 17b 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 Sect. 3.1.2, the fingerprinting technique is clearly more efficient than the complete operation. This experiment was done on bootstrapped dataset $D4$



(a) (b)

Fig. 16 Effect of minSup on the number of frequent subtrees and the run time.



(a) (b)

Fig. 17 Depth-first mining and fingerprinting based pruning technique.

with 1000 trees on 20 leaves. The false positive ratio hovered around 3.2% for this experiment.

4.2 Experiments on Biological Data

To study the effectiveness of FST mining, we compared the results of applying the FST, MAST, and MRT approaches to biological data. As mentioned in Sect. 1.1.3, both MASTs and MRT are frequently used in phylogenetics. Indeed, a search on "maximum agreement subtree" and "majority-rule tree" on Google Scholar[5] (`http://scholar.google.com/`) returned about 365 and 1140 results respectively. Among other things, MASTs are used as a metric to compare phylogenies [33, 24, 25], to compute the congruence index of

[5] Both terms were searched in double quotes, i.e., all words in the query appear together in all returned documents.

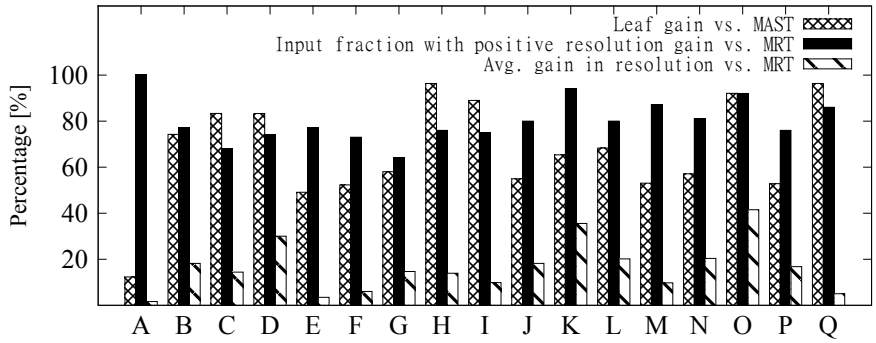


Fig. 18 FST vs. MAST and MRT

phylogenetic trees [22, 51], to identify horizontal gene transfer events [21], to resolve ambiguity in terraces in phylogenetic tree space [69], and as a consensus approach [14]. MRTs are used, among other things, to analyze phylogenetic trees from Bayesian analysis [43] and bootstrapping [27], two widely-used phylogenetic analysis techniques. As we shall see, FST mining can identify representative trees for a collection that are superior to the MAST(s) or the MRT with respect to both size and resolution.

Datasets. Our datasets were derived from bootstrapped trees used in a previous study [61] on majority rule trees. Trees were constructed using 17 DNA alignments containing 125 up to 2,554 sequences. The data spans a diverse range of sequences including rbcL genes, mammalian sequences, bacterial and archaeal sequences, ITS sequences, fungal sequences, and grasses. We ordered the trees 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 leaf set of size 15. We generated 100 such random collections for each of the dataset in *A*–*Q*. The experimental results were averaged over these 100 collections.

Frequent Subtree versus Maximum Agreement Subtree. Note that the set of all MASTs is a subset of the set of FSTs. Thus, for every MAST T there is always an FST that has the same or higher resolution as T. Thus, in comparing MASTs with FSTs. we focus on the gain in the number of leaves.

For each collection of 100 trees, we generated all MASTs. We then mined all FSTs in the collection with 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'. Note that T' is a maximal subtree. We then compared the size of T' (i.e., the number of leaves) with that of T. The first histogram bar in Fig. 18 shows this leaf gain, which is defined as

$$\text{gain}(T, T') = \frac{|\text{leaves in } T'| - |\text{ leaves in } T|}{|\text{leaves in } T|} \times 100\%.$$

In all datasets, we found FSTs larger than any MAST. In some of the cases the leaf gain was as high as 100%.

Frequent Subtree versus Majority Rule Tree. As mentioned in the Introduction, the MRT can be poorly resolved. To explore this issue further, we compared the degrees of resolution of the MRT with that of the FSTs by considering what we call FST profiles. An FST T is *maximal* if there is no other FST T' such that T is a subtree of T'. An *FST-profile* is a collection of maximal FSTs such that the combined leaf set contains all the species present in the input trees. For example, the FSTs in Fig. 3d form an FST-profile for the input collection of trees shown in Fig. 3a. We measure the resolution of a tree T using equation (1) from Sec. 1.1.

For each collection of 100 trees, we computed the MRT using HashCS [76] 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 leaf set as that of T^F to obtain T^M, computed the difference in tree-resolution values of T^F and T^M, and then averaged the difference over all the trees in the FST-profile. The second histogram bar in Fig. 18, denotes the fraction of the input collections that observed a positive resolution gain in the FST-profile over the MRT. In all cases, a high fraction of the 100 collections resulted in FSTs that were better resolved than the MRTs. The third histogram bar shows the average resolution gain, which in some cases exceeds 30%. These measures of gain do not have natural statistical distributions associated with them, but do point the investigator in the direction of the most unexpected solutions, relative to existing methods.

4.3 Discussion

Our experiments indicate that FSTs can be more informative than the classical methods MRT and MAST. It makes further sense to single out those FSTs that accentuate this gain with respect to the classical methods. Here we suggest some tests to only keep such "interesting" FSTs. We discuss this for MRT and MAST separately.

Maximum Agreement Subtree. One of the ways in which we can get better results than MASTs is by looking for FSTs having leafsets not covered by any of the MASTs. We classify such FSTs in two categories. The first category consists of an FST T if it is a proper supertree to an MAST T', i.e., $\mathcal{L}_T \supset \mathcal{L}_{T'}$ and $T' \equiv T|_{\mathcal{L}_{T'}}$. For example, each of the FSTs in Fig. 3 is a supertree to the corresponding MAST. Such FSTs are very interesting because each such FST extends an existing MAST, i.e., it phylogenetically relates the set of leaves $\mathcal{L}_{T'}$ displayed by the MAST to a new set of leaves $\mathcal{L}_T - \mathcal{L}_{T'}$. The second category includes FSTs that display phylogenetic relationships over a set of leaves $\mathcal{L}$ such that there exists no MAST T with $|\mathcal{L}_T \cap \mathcal{L}| \geq 3$. Three is the minimum number of leaves on which a phylogenetic tree can be

meaningful. Thus, any such FST on the leafset $\mathcal{L}$ will give completely new set of phylogenetic relationships, i.e., those not covered by any of the MASTs either in part or total. For example, the FST in Fig. 4 displays relationship among species $s1 - s4$. The corresponding set of MASTs (see Fig. 4) don't include a tree T such that $|\mathcal{L}_T \cap (s1 - s4)| \geq 3$. Thus, none of the MASTs display any of the any of the phylogenetic relationships displayed by the FST.

Clearly, FSTs in both categories can be identified by comparing the set of all MASTs with the set of all FSTs.

Majority Rule Tree. FSTs tend to be more resolved than the MRT. One way to identify such interesting FSTs can be to compare the set of all FSTs with the MRT using (1) and then keep aside the subset of FSTs that have show maximum resolution with respect to the MRT. This can be further refined by analyzing the parts of the MRT that are poorly resolved. For example, in Fig. 4, consider the internal node corresponding to the cluster $\mathcal{C} = \{s1, s2, s3, s4, s8\}$ in the MRT. This internal node exhibits a star-like structure with respect to it's children, i.e., no phylogenetic relationships can be drawn among the leafset $\mathcal{C}$. The remaining part of the MRT is well resolved. Thus, instead of comparing the MRT with all the FSTs, considering only FSTs having leafset subset of $\mathcal{C}$ makes more sense. The FST shown in Fig. 4 is one such FSTs. One can get more precise if the set of triplets in line 2 of the EvoMiner algorithm (see Fig. 9) includes only triplets from the leafset $\mathcal{C}$. This will mine only those FSTs whose leafset is a subset of $\mathcal{C}$.

5 Conclusion

We introduced EvoMiner, 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 of up to 100 times, and sometimes more. We also demonstrated the utility of FST mining as a way to extract meaningful phylogenetic information from collections of trees, making it a valuable alternative to MASTs and MRTs in various settings. We now mention some directions for future work.

The sheer number of FSTs can overwhelm a user. One could instead mine for maximal and closed subtrees [16]. While there has been significant work on this subject [83, 36, 78, 18, 28, 81], for reasons mentioned in Sections 1.1.1 and 2.2, mining maximal phylogenetic subtrees demands a separate approach. We intend to investigate whether we can extend EvoMiner for this purpose. Note that the number of MASTs can grow exponentially with the number of leaves [50]; the number of maximal and closed subtrees will grow at least as fast. Thus, we intend to develop an approach that will mine a 'representative' set of FSTs that will be pair-wise distinct and well sampled from the entire set (along the lines of [89]). Another way to handle large datasets can be to use a parallel implementation [23] or to explore approximate solutions [49].

An important open problem is to derive bounds on the number of FSTs. Such bounds would be useful for EvoMiner and EvoMinerDF, as they would allow us to estimate the progress of those algorithms, giving the end-user the option to mine only some desired fraction of the total. Bounds of this sort have been derived for frequent sequential patterns [66], but it is not obvious to extend these results to frequent subtrees.

In addition to pure topology, phylogenetic trees typically have other attributes, such as support values for nodes and branch lengths. To our knowledge, the problem of mining phylogenies with such attributes has not been studied. Finally, it would be interesting to see if one can apply the fingerprinting technique for the downward closure operation for mining arbitrary trees, not just phylogenies.

Acknowledgments

This work was supported in part by National Science Foundation grant DEB-0829674. 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 datasets from Bayesian analyses, and Dr. Nicholas D. Pattengale for sharing the datasets consisting of bootstrapped trees. A special thanks to the anonymous reviewers at KAIS whose detailed comments helped greatly in improving the paper.

References

1. Aggarwal CC, Wang H (2010) Managing and Mining Graph Data, Advances in Database Systems, vol 40. Springer
2. Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo A (1996) Fast Discovery of Association Rules. Advances in Knowledge Discovery and Data Mining 12:307–328
3. Amenta N, Clarke F, John KS (2003) A linear-time majority tree algorithm. In: Proceedings of the 3rd Workshop on Algorithms in Bioinformatics (WABI'03), pp 216–227
4. Amir A, Keselman D (1994) Maximum agreement subtree in a set of evolutionary trees. SIAM Journal on Computing 26:758–769
5. Asai T, Abe K, Kawasoe S, Arimura H, Sakamoto H, Arikawa S (2002) Efficient substructure discovery from large semi-structured data. In: Proceedings of the SIAM International Conference on Data Mining, pp 158–174
6. Asai T, Arimura H, Uno T, Nakano Si (2003) Discovering frequent substructures in large unordered trees. In: Proceedings of the 6th International Conference on Discovery Science, pp 47–61
7. Ayres J, Flannick J, Gehrke J, Yiu T (2002) Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD

International Conference on Knowledge Discovery and Data Mining, pp 429–435
8. Barns S, Delwiche C, Palmer J, Pace N (1996) Perspectives on archaeal diversity, thermophily and monophyly from environmental rRNA sequences. Proceedings of the National Academy of Sciences 93:9188–9193
9. Baum D (2008) Reading a phylogenetic tree: The meaning of monophyletic groups. Nature Education 1(1)
10. Bei Y, Chen G, Shou L, Li X, Dong J (2009) Bottom-up discovery of frequent rooted unordered subtrees. Information Sciences 179:70–88
11. Bender M, Farach-Colton M (2000) The LCA problem revisited. In: Proceedings of the 4th Latin American Symposium on Theoretical Informatics, pp 88–94
12. Bhaskar R, Laxman S, Smith A, Thakurta A (2010) Discovering frequent patterns in sensitive data. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 503–512
13. Bryant D (1997) Building trees, hunting for trees and comparing trees. PhD thesis, University of Canterbury, New Zealand
14. Bryant D (2003) A classification of consensus methods for phylogenetics. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 61:163–184
15. Chi Y, Yang Y, Muntz RR (2003) Indexing and mining free trees. In: Proceedings of the IEEE International Conference on Data Mining, pp 509–512
16. Chi Y, Muntz R, Nijssen S, Kok J (2004) Frequent Subtree Mining — An Overview. Fundamenta Informaticae 66:161–198
17. Chi Y, Yang Y, Muntz R (2004) Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management, pp 11–20
18. Chi Y, Xia Y, Yang Y, Muntz R (2005) Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Transactions on Knowledge and Data Engineering 17:190–202
19. Cole R, Farach-Colton M, Hariharan R, Przytycka T, Thorup M (2000) An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. SIAM Journal on Computing 30:1385–1404
20. Currie TE, Greenhill SJ, Gray RD, Hasegawa T, Mace R (2010) Rise and fall of political complexity in island South-East Asia and the Pacific. Nature 467:801–804
21. Daubin V, Gouy M, Perrière G (2002) A phylogenomic approach to bacterial phylogeny: evidence of a core of genes sharing a common history. Genome Research 12:1080–1090
22. De Vienne D, Giraud T, Martin O (2007) A congruence index for testing topological similarity between trees. Bioinformatics 23:3119–3124
23. Do T, Laurent A, Termier A (2010) Pglcm: Efficient parallel mining of closed frequent gradual itemsets. In: Proceedings of the 10th IEEE Inter-

national Conference on Data Mining, pp 138–147
24. Dong S, Kraemer E (2004) Calculation, visualization, and manipulation of masts (maximum agreement subtrees). In: Proceedings of the IEEE Computational Systems Bioinformatics Conference CSB, pp 405–414
25. Farach M, Thorup M (1994) Fast comparison of evolutionary trees. In: Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp 481–488
26. Farach M, Przytycka T, Thorup M (1995) On the agreement of many trees. Information Processing Letters 55:297–301
27. Felsenstein J (1985) Confidence limits on phylogenies: An approach using the bootstrap. Evolution 39:783–791
28. Feng B, Xu Y, Zhao N, Xu H (2010) A new method of mining frequent closed trees in data streams. In: Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), pp 2245–2249
29. Finden C, Gordon A (1985) Obtaining common pruned trees. Journal of Classification 2:255–276
30. Flint-Garcia S, Thuillet A, Yu J, Pressoir G, Romero S, Mitchell S, Doebley J, Kresovich S, Goodman M, Buckler E (2005) Maize association population: a high-resolution platform for quantitative trait locus dissection. The Plant Journal 44:1054–1064
31. Ganapathysaravanabavan G, Warnow T (2001) Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time. In: Proceedings of the International Workshop on Algorithms in Bioinformatics, pp 156–163
32. Geerts F, Goethals B, Bussche J (2005) Tight upper bounds on the number of candidate patterns. ACM Transactions on Database Systems (TODS) 30:333–363
33. Goddard W, Kubicka E, Kubicki G, McMorris F (1994) The agreement metric for labeled binary trees. Mathematical Biosciences 123:215–226
34. Gray R, Drummond A, Greenhill S (2009) Language phylogenies reveal expansion pulses and pauses in Pacific settlement. Science 323:479–483
35. Guillemot S, Berry V (2010) Fixed-parameter tractability of the maximum agreement supertree problem. IEEE/ACM Transactions on Computational Biology and Bioinformatics 7:342–353
36. Hadzic F, Tan H, Dillon T, Hadzic F, Tan H, Dillon T (2010) Mining maximal and closed frequent subtrees. In: Mining of Data with Complex Structures, Studies in Computational Intelligence, vol 333, Springer Berlin / Heidelberg, pp 191–199
37. Han J, Pei J (2000) Mining frequent patterns by pattern-growth: methodology and implications. ACM SIGKDD Explorations Newsletter 2:14–20
38. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp 1–12
39. Han J, Pei J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M (2001) Prefixspan: Mining sequential patterns efficiently by prefix-projected pat-

tern growth. In: Proceedings of the 17th International Conference on Data Engineering, pp 215–224
40. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery 8:53–87
41. Harel D, Tarjan R (1984) Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13:338–355
42. Hromkovič J (2005) Abundance of witnesses. In: Design and Analysis of Randomized Algorithms, Texts in Theoretical Computer Science. An EATCS Series, Springer Berlin Heidelberg, pp 183–207
43. Huelsenbeck JP, Ronquist F (2001) MRBAYES: Bayesian inference of phylogenetic trees. Bioinformatics 17:754–755
44. Jia Y, Zhang J, Huan J (2011) An efficient graph-mining method for complicated and noisy data with real-world applications. Knowledge and Information Systems 28:423–447
45. Jimenez A, Berzal F, Cubero J (2010) Frequent tree pattern mining: A survey. Intelligent Data Analysis 14:603–622
46. Jimenez A, Berzal F, Cubero J (2010) Potminer: mining ordered, unordered, and partially-ordered trees. Knowledge and Information Systems 23:199–224
47. Kao M, Lam T, Sung W, Ting H (2001) An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. Journal of Algorithms 40:212–233
48. Karp R, Rabin M (1987) Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31:249–260
49. Ke Y, Cheng J, Yu J (2009) Efficient discovery of frequent correlated subgraph pairs. In: Proceedings of the Ninth IEEE International Conference on Data Mining, pp 239–248
50. Kubicka E, Kubicki G, McMorris F (1992) On agreement subtrees of two binary trees. Congressus Numerantium 88:217–217
51. Lapointe F, Rissler L (2005) Congruence, consensus, and the comparative phylogeography of codistributed species in California. The American Naturalist 166:290–299
52. Lewis L, Lewis P (2005) Unearthing the molecular phylodiversity of desert soil green algae (Chlorophyta). Systematic Biology 54:936–947
53. Liu H, Lin Y, Han J (2011) Methods for mining frequent items in data streams: an overview. Knowledge and Information Systems 26:1–30
54. Liu L, Liu J (2011) Mining frequent embedded subtree from tree-like databases. In: Proceedings of the International Conference on Internet Computing & Information Services (ICICIS), pp 3–7
55. Margush T, McMorris F (1981) Consensus n-trees. Bulletin of Mathematical Biology 43:239–244
56. Mau B, Newton M, Larget B (1999) Bayesian phylogenetic inference via Markov chain Monte Carlo methods. Biometrics 55:1–12
57. Motwani R, Raghavan P (1995) Randomized algorithms, Cambridge: Cambridge Univ, chap 7

58. NCBI (2002) Tree facts: Rooted versus unrooted trees. Online, URL http://www.ncbi.nlm.nih.gov/Class/NAWBIS/Modules/Phylogenetics/phylo9.html
59. Nguyen V, Yamamoto A (2010) Incremental mining of closed frequent subtrees. In: Pfahringer B, Holmes G, Hoffmann A (eds) Discovery Science, Lecture Notes in Computer Science, vol 6332, Springer, Berlin / Heidelberg, pp 356–370
60. Nijssen S, Kok J (2003) Efficient discovery of frequent unordered trees. In: Proceedings of the International Workshop on Mining Graphs, Trees and Sequences, pp 55–64
61. Pattengale N, Aberer A, Swenson K, Stamatakis A, Moret B (2011) Uncovering hidden phylogenetic consensus in large datasets. IEEE/ACM Transactions on Computational Biology and Bioinformatics 8:902–911
62. Pei J, Han J (2002) Constrained frequent pattern mining: A pattern-growth view. ACM SIGKDD Explorations Newsletter 4:31–39
63. Pei J, Han J, Mortazavi-Asl B, Wang J, Pinto H, Chen Q, Dayal U, Hsu M (2004) Mining sequential patterns by pattern-growth: The prefixspan approach. IEEE Transactions on Knowledge and Data Engineering 16:1424–1440
64. Pei J, Han J, Wang W (2007) Constraint-based sequential pattern mining: the pattern-growth methods. Journal of Intelligent Information Systems 28:133–160
65. Piel W, Donoghue M, Sanderson M (2002) 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 , Tsukuba, Japan, 171, pp 41–47
66. Raissi C, Pei J (2011) Towards bounding sequential patterns. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 1379–1387
67. Rannala B, Yang Z (2008) Phylogenetic inference using whole genomes. Annual Review of Genomics and Human Genetics 9:217–231
68. Sanderson M, Boss D, Chen D, Cranston K, Wehe A (2008) The PhyLoTA browser: processing GenBank for molecular phylogenetics research. Systematic Biology 57:335–346
69. Sanderson M, McMahon M, Steel M (2011) Terraces in phylogenetic tree space. Science 333:448–450
70. Schieber B, Vishkin U (1988) On finding lowest common ancestors: simplification and parallelization. SIAM Journal on Computing 17:1253–1262
71. Scornavacca C (2009) Supertree methods for phylogenomics. PhD thesis, University of Montpellier II, Montpellier, France
72. Semple C, Steel M (2003) Phylogenetics. Oxford Lecture Series in Mathematics, Oxford University Press, Oxford
73. Slowinski J, Keogh J (2000) Phylogenetic relationships of elapid snakes based on cytochrome b mtDNA sequences. Molecular Phylogenetics and Evolution 15:157–164

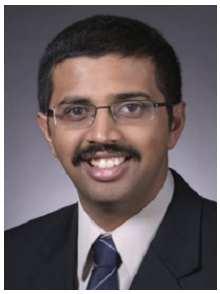
74. Smith M, Patton J (1999) Phylogenetic relationships and the radiation of sigmodontine rodents in South America: Evidence from cytochrome b. Journal of Mammalian Evolution 6:89–128
75. Steel M, Warnow T (1993) Kaikoura tree theorems: computing the maximum agreement subtree. Information Processing Letters 48:77–82
76. Sul S, Williams T (2009) An experimental analysis of consensus tree algorithms for large-scale tree collections. In: Proceedings of the International Symposium on Bioinformatics Research and Applications, pp 100–111
77. Swenson K, Chen E, Pattengale N, Sankoff D (2011) The Kernel of Maximum Agreement Subtrees. In: Proceedings of the International Symposium on Bioinformatics Research and Applications, pp 123–135
78. Termier A, Rousset M, Sebag M (2004) Dryade: a new approach for discovering closed frequent trees in heterogeneous tree databases. In: Proceedings of the IEEE International Conference on Data Mining, pp 543–546
79. Wang C, Hong M, Pei J, Zhou H, Wang W, Shi B (2004) Efficient pattern-growth methods for frequent tree pattern mining. In: Dai H, Srikant R, Zhang C (eds) Advances in Knowledge Discovery and Data Mining, Lecture Notes in Computer Science, vol 3056, Springer, Berlin / Heidelberg, pp 441–451
80. Wang J, Shan H, Shasha D, Piel W (2005) Fast structural search in phylogenetic databases. Evolutionary Bioinformatics Online 1:37–46
81. Wang S, Hong Y, Yang J (2012) XML document classification using closed frequent subtree. In: Bao Z, Gao Y, Gu Y, Guo L, Li Y, Lu J, Ren Z, Wang C, Zhang X (eds) Web-Age Information Management, Lecture Notes in Computer Science, vol 7419, Springer Berlin, Heidelberg, pp 350–359
82. Wu X, Kumar V, Ross Quinlan J, Ghosh J, Yang Q, Motoda H, McLachlan G, Ng A, Liu B, Yu P, et al (2008) Top 10 algorithms in data mining. Knowledge and Information Systems 14:1–37
83. Xiao Y, Yao J (2003) Efficient data mining for maximal frequent subtrees. In: Proceedings of the IEEE International Conference on Data Mining, pp 379–386
84. Yang LH, Lee ML, Hsu W, Acharya S (2003) Mining frequent query patterns from XML queries. In: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications, pp 355–362
85. Yule G (1925) A mathematical theory of evolution, based on the conclusions of Dr. JC Willis, F.R.S. Philosophical Transactions of the Royal Society of London Series B, Containing Papers of a Biological Character 213:21–87
86. Zaki M (2004) Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae 66:33–52
87. Zaki M (2005) Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE Transactions on Knowledge and Data Engineering 17:1021–1035
88. Zhang S, Wang J (2008) Discovering frequent agreement subtrees from phylogenetic data. IEEE Transactions on Knowledge and Data Engineering 20:68–82

89. Zhang S, Yang J, Li S (2009) Ring: An integrated method for frequent representative subgraph mining. In: Proceedings of the Ninth IEEE International Conference on Data Mining, pp 1082–1087
90. Zou X, Zhang F, Zhang J, Zang L, Tang L, Wang J, Sang T, Ge S (2008) Analysis of 142 genes resolves the rapid diversification of the rice genus. Genome Biology 9:R49
91. Zou Z, Gao H, Li J (2010) Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 633–642

Author Biographies

Akshay Deepak is a postdoctoral research associate in the Department of Electrical and Computer Engineering, Iowa State University, USA. His research interests include algorithms in computational biology, frequent subtree mining, software reliability.

David Fernández-Baca is a professor in the Department of Computer Science, Iowa State University, USA. His research interest include combinatorial algorithms, phylogenetic tree construction, sensitivity analysis of optimization problems. Web: http://www.cs.iastate.edu/people/facultyview.jsp?fname=David



Srikanta Tirthapura is an associate professor in the Department of Electrical and Computer Engineering, Iowa State University, USA. His research interests include algorithms and software tools for large scale data analysis, and in particular, real-time analysis of data streams. Web: http://home.eng.iastate.edu/~snt/

Michael J. Sanderson is a professor in the Department of Ecology and Evolutionary Biology, University of Arizona, USA. His research is aimed at developing algorithms and software for assembling data from the large sequences databases for the purpose of building comprehensive phylogenetic trees. Web: http://loco.biosci.arizona.edu/sanderson.html

Michelle McMahon is an associate professor in The School of Plant Sciences, Ecology and Evolutionary Biology, University of Arizona, USA. She researches phylogenetic and phylogenomic methods, systematics of the legume family (Fabaceae), and phylogenetic diversity of regional floras. She also directs the UA Herbarium. Web: http://cals.arizona.edu/spls/node/134