Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications

Mohammed J. Zaki, *Member*, *IEEE*

Abstract—Mining frequent trees is very useful in domains like bioinformatics, Web mining, mining semistructured data, etc. We formulate the problem of mining (embedded) subtrees in a forest of rooted, labeled, and ordered trees. We present TREEMINER, a novel algorithm to discover all frequent subtrees in a forest, using a new data structure called scope-list. We contrast TREEMINER with a pattern matching tree mining algorithm (PATTERNMATCHER), and we also compare it with TREEMINERD, which counts only distinct occurrences of a pattern. We conduct detailed experiments to test the performance and scalability of these methods. We also use tree mining to analyze RNA structure and phylogenetics data sets from bioinformatics domain.

Index Terms—Frequent tree mining, rooted, ordered, labeled trees, subtree enumeration, pattern matching, RNA structure, phylogenetic trees, data mining.

1 Introduction

Frequent Structure Mining (FSM) refers to an important class of exploratory mining tasks, namely, those dealing with extracting patterns in massive databases representing complex interactions between entities. FSM not only encompasses mining techniques like associations [3] and sequences [4], but it also generalizes to more complex patterns like frequent trees and graphs [18], [21]. Such patterns typically arise in applications like bioinformatics, Web mining, mining semistructured documents, etc. As one increases the complexity of the structures to be discovered, one extracts more informative patterns; we are specifically interested in mining tree-like patterns.

As a motivating example for tree mining, consider the problem of mining structural patterns in a data set of Ribonucleic acid (RNA) molecules, which can be represented as trees. To get information about a newly sequenced RNA, researchers may compare it with known RNA structures, looking for common topological patterns, which provide important clues to the function of the RNA [30]. As another example, given several phylogenies (i.e., evolutionary trees) from the Tree of Life [23], indicating evolutionary history of several organisms, one might be interested in discovering if there are common subtree patterns.

In this paper, we introduce TREEMINER, an efficient algorithm for the problem of mining frequent subtrees in a forest (the database). The key contributions of our work are as follows:

1. We introduce the problem of mining *embedded* subtrees in a collection of rooted, ordered, and labeled trees.
2. We use the notion of a *scope* for a node in a tree. We show how any tree can be represented as a list of its node scopes in a novel vertical format called *scope-list*.
3. We develop a framework for nonredundant candidate subtree generation, i.e., we propose a systematic search of the possibly frequent subtrees, such that no pattern is generated more than once.
4. We show how one can efficiently compute the frequency of a candidate tree by joining the scope-lists of its subtrees.
5. Our formulation allows one to discover all subtrees in a forest, as well as all subtrees in a single large tree.

Furthermore, simple modifications also allow us to mine unlabeled subtrees, unordered subtrees, and also frequent subforests (i.e., disconnected subtrees). We also present TREEMINERD, a method that, instead of counting all embeddings, only counts distinct occurrences of a pattern and might be more suitable than TREEMINER for certain data sets. We also contrast TREEMINER with another tree mining algorithm based on pattern matching, PATTERNMATCHER. We present applications of tree mining in bioinformatics, such as mining frequent RNA structures and common phylogenetic tree patterns.

2 Problem Statement

A *rooted, labeled, tree*, $T = (V, E)$ is a directed, acyclic, connected graph with $V = \{0, 1, \cdots, n\}$ as the set of vertices and $E = \{(x, y) | x, y \in V\}$ as the set of edges. One distinguished vertex $r \in V$ is designated the *root*, and for all $x \in V$, there is a *unique* path from r to x. Further, $l : V \rightarrow L$ is a labeling function mapping vertices to a set of *labels* $L = \{\ell_1, \ell_2, \cdots\}$. In an *ordered tree*, the children of each vertex are ordered (i.e., if a vertex has k children, then we can designate them as the first child, second child, and so on, up to the kth child), otherwise, the tree is *unordered*. In this paper, all trees we consider are ordered, labeled, and rooted trees.

• *The author is with the Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180. E-mail: zaki@cs.rpi.edu.*

Manuscript received 23 July 2004; revised 24 Oct. 2004; accepted 24 Feb. 2005; published online 17 June 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-0272-0704.

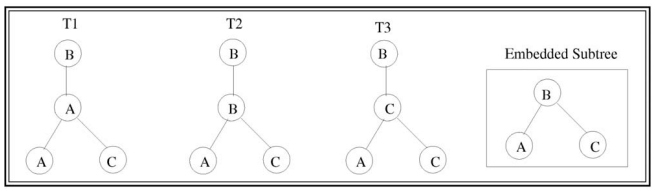


Fig. 1. Embedded subtree.

If $x, y \in V$ and there is a path from x to y, then x is called an *ancestor* of y (and y a *descendant* of x), denoted as $x \leq_p y$, where p is the length of the path from x to y. If $x \leq_1 y$ (i.e., x is an immediate ancestor), then x is called the *parent* of y, and y the *child* of x. If x and y have the same parent, x and y are called *siblings*, and if they have a common ancestor, they are called *cousins*.

We assume that vertex $x \in V$ is synonymous with (or numbered according to) its position n_x in the depth-first (preorder) traversal of the tree T (for example, the root r is vertex n_0). Let $T(x)$ denote the subtree rooted at x, and let y be the rightmost leaf (or highest numbered descendant) under x. Then, the *scope* of x is given as $s(x) = [x, y]$ (or $[n_x, n_y]$). Intuitively, $s(x)$ demarcates the range of vertices under x.

Subtrees. Given a tree $S = (V_s, E_s)$ and tree $T = (V_t, E_t)$, we say that S is an *isomorphic subtree* of T *iff*[1] there exists a one-to-one mapping $\varphi : V_s \rightarrow V_t$, such that $(x, y) \in E_s$ iff $(\varphi(x), \varphi(y)) \in E_t$. If φ is onto, then S and T are called *isomorphic*. S is called an *induced subtree* of $T = (V_t, E_t)$, denoted $S \preceq_i T$, iff S is an isomorphic subtree of T and φ preserves labels, i.e., $l(x) = l(\varphi(x)), \forall x \in V_s$. That is, for induced subtrees, φ preserves the parent-child relationships, as well as vertex labels. The induced subtree obtained by deleting the rightmost leaf in T is called an *immediate prefix* of T. The induced tree obtained from T by a series of rightmost node deletions is called a *prefix* of T.

$S = (V_s, E_s)$ is called an *embedded subtree* of $T = (V_t, E_t)$, denoted as $S \preceq_e T$ iff there exists a 1-to-1 mapping $\varphi : V_s \rightarrow V_t$ that satisfies: 1) $(x, y) \in E_s$ iff $\varphi(x) \leq_p \varphi(y)$ and 2) $l(x) = l(\varphi(x))$. That is, for embedded subtrees, φ preserves ancestor-descendant relationships and labels. Embedded subtrees are thus a generalization of induced subtrees; they allow not only direct parent-child branches, but also ancestor-descendant branches. As such, embedded subtrees are able to extract patterns "hidden" (or embedded) deep within large trees which might be missed by the traditional definition.

As an example, consider Fig. 1, which shows three trees. Let's assume we want to mine subtrees that are common to all three trees (i.e., 100 percent frequency). If we mine induced trees only, then there are no frequent trees of size more than one. On the other hand, if we mine embedded subtrees, then the tree shown in the box is a frequent pattern appearing in all three trees; it is obtained by skipping the "middle" node in each tree. This example shows why embedded trees are of interest. Henceforth, a reference to subtree should be taken to mean an embedded subtree, unless indicated otherwise.

1. If and only if.

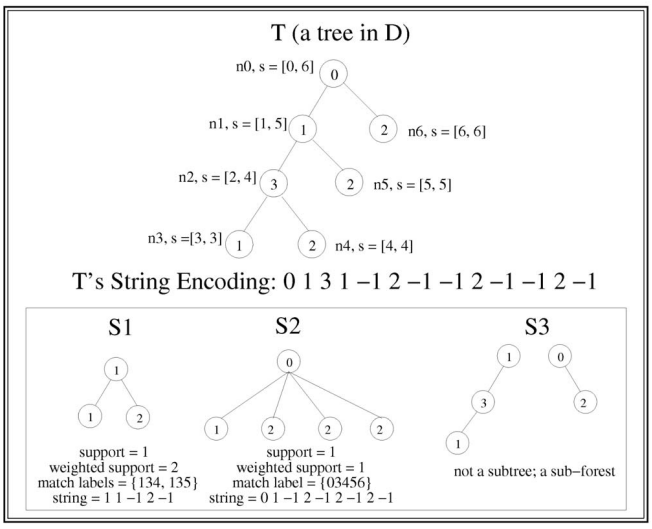


Fig. 2. An example tree with subtrees.

Support. If $S \preceq_{\{i,e\}} T$, we also say that T *contains* S or S *occurs* in T. Note that each occurrence of S in T can be identified by its unique *match label*, given by the sequence $\varphi(x_0)\varphi(x_1)\cdots\varphi(x_{|S|})$, where $x_i \in V_s$. That is, a match label of S is given as the set of matching positions in T. Let $\delta_T(S)$ denote the number of occurrences of the subtree S in a tree T. Let d_T be an indicator variable, with $d_T(S) = 1$ if $\delta_T(S) > 0$ and $d_T(S) = 0$ if $\delta_T(S) = 0$. Let D denote a database (a *forest*) of trees. The *support* of a subtree S in the database is defined as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e., the number of trees in D that contain at least one occurrence of S. The *weighted support* of S is defined as $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., total number of occurrences of S over all trees in D. Typically, support is given as a percentage of the total number of trees in D. A subtree S is *frequent* if its support is more than or equal to a user-specified *minimum support* (*minsup*) value. We denote by F_k the set of all frequent subtrees of size k (also called a k-(sub)tree). In some domains, one might be interested in using weighted support, instead of support. Both of them are allowed in our mining approach, but we focus mainly on support.

Given a database D of trees, our goal is to mine all frequent, labeled, ordered, and embedded subtrees. Consider Fig. 2, which shows an example tree T with node labels drawn from the set $L = \{0, 1, 2, 3\}$. The figure shows, for each node, its label (circled), its depth-first number, and its scope. For example, the root occurs at position $n = 0$, its label $l(n_0) = 0$, and, since the right-most leaf under the root occurs at position 6, the scope of the root is $s = [0, 6]$. Tree $S1$ is a subtree of T; it has a support of 1, but its weighted support is 2, since node n_2 in $S1$ occurs at positions 4 and 5 in T, both of which support $S1$, i.e., there are two match labels for $S1$, namely, 134 and 135 (we omit set notation for convenience). $S2$ is also a valid subtree. $S3$ is not a (sub)tree since it is disconnected; it is a subforest.

3 Generating Candidate Trees

There are two mains steps for enumerating frequent subtrees in D. First, we need a systematic way of generating

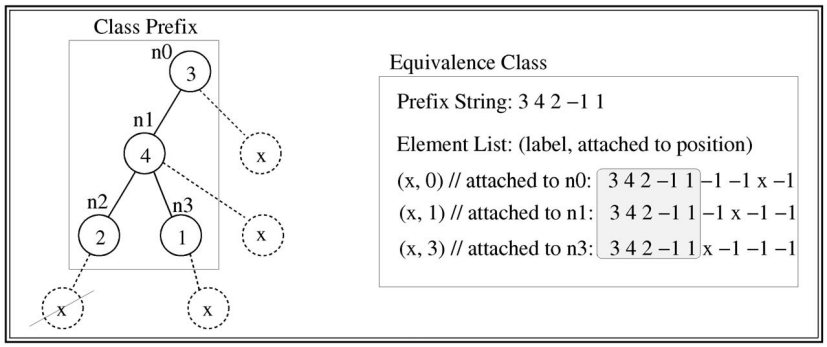


Fig. 3. Prefix equivalence class.

candidate subtrees whose frequency is to be computed. The candidate set should be nonredundant, i.e., each subtree should be generated as most once. Second, we need efficient ways of counting the number of occurrences of each candidate in the database D and to determine which candidates pass the *minsup* threshold. The latter step is data structure dependent and will be treated later. Here, we are concerned with the problem of nonredundant pattern generation. We describe our tree representation and candidate generation procedure below.

Representing Trees as Strings. As described in [40], we represent a tree T by its *string encoding*, denoted $\mathcal{T}$, generated as follows: Add vertex labels to $\mathcal{T}$ in a depth-first preorder traversal of T and add a unique symbol $-1 \notin L$ whenever we backtrack from a child to its parent. This format allows us to conveniently represent trees with an arbitrary number of children for each node. Since each branch must be traversed in both forward and backward direction, the space usage to store a tree as a string is exactly $2m + 1 = 2n - 1$ (where m is the number of edges and n is the number of nodes in T). We use the notation $l(T)$ to refer to the *label sequence* of T, which consists of the node labels of T in depth-first ordering (without backtrack symbol -1), i.e., label sequence ignores tree topology. In Fig. 2, we show the string encodings for the tree T as well as each of its subtrees. For example, subtree $S1$ is encoded by the string $1\ 1\ -1\ 2\ -1$. That is, we start at the root of $S1$ and add 1 to the string. The next node in preorder traversal is labeled 1, which is added to the encoding. We then backtrack to the root (adding -1) and follow down to the next node, adding 2 to the encoding. Finally, we backtrack to the root, adding -1 to the string. Note that the label sequence of $S1$ is given as 112.

3.1 Candidate Subtree Generation

We use the antimonotone property of frequent patterns for efficient candidate generation, namely, that the frequency of a superpattern is less than or equal to the frequency of a subpattern. Thus, we consider only a known frequent pattern for extension. Past experience also suggests that an extension by a single item at a time is likely to be more efficient. Thus, we use information from frequent k-subtrees to generate candidate $(k+1)$-subtrees.

Equivalence Classes. We say that two k-subtrees X, Y are in the same *prefix equivalence class* iff they share a common prefix up to the $(k-1)$th node. Formally, let $\mathcal{X}, \mathcal{Y}$ be the string encodings of two trees and let function $p(\mathcal{X}, i)$ return the prefix up to the ith node. X, Y are in the same class iff $p(\mathcal{X}, k-1) = p(\mathcal{Y}, k-1)$. Thus, any two members of an equivalence class differ only in the position of the last node.

Consider Fig. 3, which shows a class template for subtrees of size 5 with the same prefix subtree P of size 4, with string encoding $\mathcal{P} = 3\ 4\ 2\ -1\ 1$. Here, x denotes an arbitrary label from L. The valid positions where the last node with label x may be attached to the prefix are n_0, n_1, and n_3, since, in each of these cases, the subtree obtained by adding x to P has the same prefix. Note that a node attached to position n_2 cannot be a valid member of class $\mathcal{P}$, since it would yield a different prefix, given as $3\ 4\ 2\ x$. The figure also shows the actual format we use to store an equivalence class; it consists of the class prefix string and a list of elements. Each element is given as an (x, p) pair, where x is the *label* of the last node and p specifies the depth-first position of the node in P to which x is attached. For example, $(x, 1)$ refers to the case where x is attached to node n_1 at position 1. The figure shows the encoding of the subtrees corresponding to each class element. Note how each of them shares the same prefix up to the $(k-1)$th node. These subtrees are shown only for illustration purposes; we only store the element list in a class.

Let P be prefix subtree of size $k-1$; we use the notation $[P]_{k-1}$ to refer to its class (we omit the subscript when there is no ambiguity). If (x, i) is an element of the class, we write it as $(x, i) \in [P]$. Each (x, i) pair corresponds to a subtree of size k, sharing P as the prefix, with the last node labeled x, attached to node n_i in P. We use the notation P_x^i to refer to the new prefix subtree formed by adding (x, i) to P; P_x^i is also called an *extension* of P.

Lemma 1. *Let P be a class prefix subtree with root n_0 and with rightmost leaf n_r. Let $R(P)$ be the rightmost path from the root to n_r, given as $R(P) = \{n_i : n_i \text{ has scope } [i, r]\}$. Then, $(x, i) \in [P]$ iff $n_i \in R(P)$.*

This lemma states that the valid extensions (P_x^i) of P are obtained only by attaching label x to nodes that lie on the path from the root to the rightmost leaf in P. It is easy to see that, if x is attached to any other position, the resulting prefix would be different since x would then be before n_r in depth-first numbering.

Candidate Generation. Given an equivalence class of k-subtrees, how do we obtain candidate $(k+1)$-subtrees? First, we assume (without loss of generality) that the

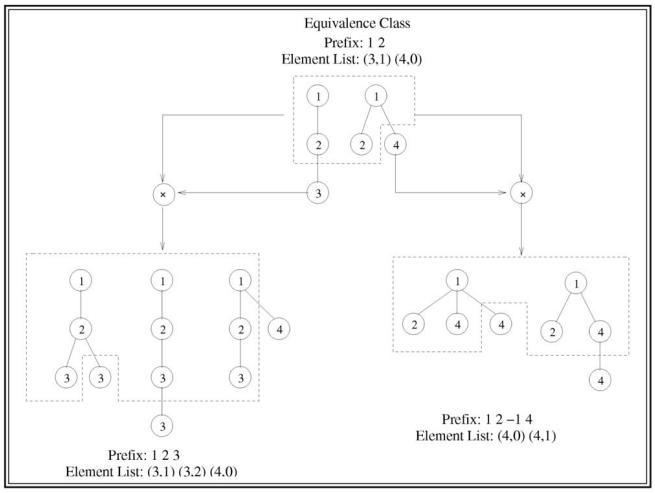


Fig. 4. Candidate generation.

elements, (x, p), in each class are kept sorted by node label as the primary key and position as the secondary key. Given a sorted element list, the candidate generation procedure we describe below outputs a new class list that respects that order, without explicit sorting. The main idea is to consider each ordered pair of elements in the class for extension, including self-extension. There can be up to two candidates from each pair of elements to be joined. The next theorem formalizes this notion.

Theorem 1 (Class Extension). *Let the $k-1$ subtree P be a prefix class with encoding $\mathcal{P}$, and let (x,i) and (y,j) denote any two elements in the class. Let the k-subtree P_x^i be the extension of P with element (x,i). Let $[P_x^i]$ denote the class representing possible extensions of P_x^i. Define a* join operator $\otimes$ *on the two elements, denoted $(x,i) \otimes (y,j)$, as follows:* **Case 1**—*$(i = j)$: a) If $\mathcal{P} \neq \emptyset$, add (y,j) and (y,n_i) to class $[P_x^i]$, where n_i is the depth-first number for node (x,i) in tree P_x^i. b) If $\mathcal{P} \neq \emptyset$, add $(y, j+1)$ to $[P_x^i]$.* **Case 2**—*$(i > j)$: Add (y,j) to class $[P_x^i]$.* **Case 3**— *$(i < j)$: No new candidate is possible in this case. Then, all possible $(k+1)$-subtrees with the common prefix P will be enumerated by applying the join operator to each ordered pair of elements (x,i) and (y,j).*

Proof. Omitted due to lack of space. □

Consider Fig. 4, showing the prefix class $\mathcal{P} = (1\ 2)$, which contains two elements, $(3,1)$ and $(4,0)$. The first step is to perform a self-join $(3,1) \otimes (3,1)$. By Case 1 a), this produces candidate elements $(3,1)$ and $(3,2)$ for the new class $\mathcal{P}_3^1 = (1\ 2\ 3)$. That is, a self-join on $(3,1)$ produces two possible candidate subtrees, one where the last node is a sibling of $(3,1)$ and another where it is a child of $(3,1)$. The leftmost two subtrees in the figure illustrate these cases. When we join $(3,1) \otimes (4,0)$, Case 2 applies, i.e., the second element is joined to some ancestor of the first one, thus, $i > j$. The only possible candidate element is $(4,0)$, since 4 remains attached to node n_0 even after the join (see the third subtree in the left-hand class in Fig. 4).

We thus add $(4,0)$ to class $[P_3^1]$. We now move to the class on the right with prefix $\mathcal{P}_4^0 = (1\ 2\ -1\ 4)$. When we try to join $(4,0) \otimes (3,1)$, Case 3 applies and no new candidate is generated. Actually, if we do merge these two subtrees, we obtain the new subtree $1\ 2\ 3\ -1\ -1\ 4$, which has a different prefix and was already added to the class $[P_3^1]$. Finally, we perform a self-join $(4,0) \otimes (4,0)$ adding elements $(4,0)$ and $(4,2)$ to the class $[P_4^0]$ shown on the right-hand side.

Case 1 b) applies only when we join single items to produce candidate 2-subtrees, i.e., we are given a prefix class $[\emptyset] = \{(x_i, -1), i = 1, \ldots, m\}$, where each x_i is a label and -1 indicates that it is not attached to any node. If we join $(x_i, -1) \otimes (x_j, -1)$, since we want only (connected) 2-subtrees, we insert the element $(x_j, 0)$ to the class of x_i. This corresponds to the case where x_j is a child of x_i. If we want to generate subforests as well, all we have to do is to insert $(x_j, -1)$ in the class of x_i. In this case, x_j would be a sibling of x_i, but since they are not connected, they would be roots of two trees in a subforest. If we allow such class elements, then one can show that the class extension theorem would produce all possible candidate subforests. However, in this paper, we will focus only on subtrees.

Corollary 1 (Automatic Ordering). *Let $[P]_{k-1}$ be a prefix class with elements sorted according to the total ordering $<$ given as follows: $(x,i) < (y,j)$ if and only if $x < y$ or $(x = y$ and $i < j)$. Then, the class extension method generates candidate classes $[P]_k$ with sorted elements.*

Corollary 2 (Correctness). *The class extension method correctly generates all possible candidate subtrees and each candidate is generated at most once.*

4 TREEMINER ALGORITHM

TREEMINER performs depth-first search (DFS) for frequent subtrees, using a novel tree representation called *scope-list* for fast support counting, as discussed below.

Scope-List Representation. Let X be a k-subtree of a tree T. Let x_k refer to the last node of X. We use the notation $\mathcal{L}(X)$ to refer to the *scope-list* of X. Each element of the scope-list is a triple (t, m, s), where t is a tree id (tid) in which X occurs, m is a match label of the $(k-1)$ length prefix of X, and s is the scope of the last item x_k. Recall that the prefix match label gives the positions of nodes in T that match the prefix. Since a given prefix can occur multiple times in a tree, X can be associated with multiple match labels as well as multiple scopes. The initial scope-lists are created for single items (i.e., labels) i that occur in a tree T. Since a single item has an empty prefix, we don't have to store the prefix match label m for single items. We will show later how to compute pattern frequency via joins on scope-lists. Fig. 5 shows a database of three trees, along with the horizontal format for each tree and the vertical scope-lists format for each item. Consider Item 1; since it occurs at node position 0 with scope $[0,3]$ in tree T_0, we add $(0,[0,3])$ to its scope list. Item 1 also occurs in T_1 at position n_1 with scope $[1,3]$, so we add $(1,[1,3])$ to $\mathcal{L}(1)$. Finally, 1 occurs with scope $[0,7]$ and $[4,7]$ in tree T_2, so we add $(2,[0,7])$ and $(2,[4,7])$ to its scope-list. In a similar manner, the scope lists for other items are created.

4.1 Frequent Subtree Enumeration

Fig. 6 shows the high-level structure of TREEMINER. The main steps include the computation of the frequent items

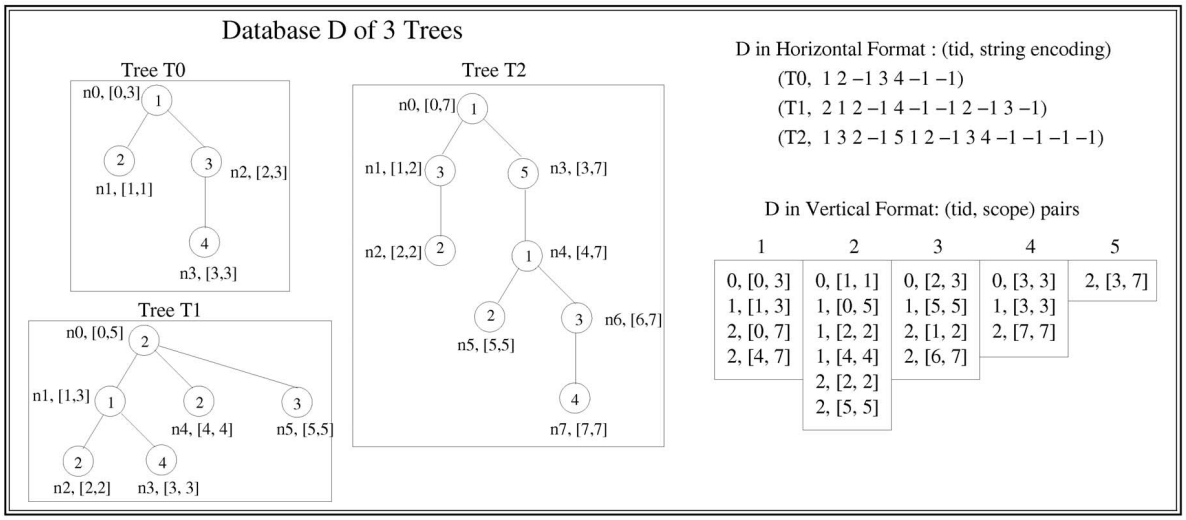


Fig. 5. Scope-lists.

and 2-subtrees and the enumeration of all other frequent subtrees via DFS search within each class $[P]_1 \in F_2$. We will now describe each step in some more detail.

Computing F_1 and F_2. We assume that the input database is in horizontal string encoded format. To compute F_1, for each item $i \in \mathcal{T}$, the string encoding of tree T, we increment i's count in an 1D array. This step also computes other database statistics such as the number of trees, maximum number of labels, and so on. All labels in F_1 belong to the class with empty prefix, given as $[P]_0 = [\emptyset] = \{(i, -1),\ i \in F_1\}$, and the position -1 indicates that i is not attached to any node. Total time for this step is $O(n)$ per tree, where $n = |T|$.

By Theorem 1, each candidate class $[P]_1 = [i]$ (with $i \in F_1$) consists of elements of the form $(j, 0)$, where $j \geq i$. For efficient F_2 counting we compute the supports of each candidate by using a 2D integer array of size $F_1 \times F_1$, where $cnt[i][j]$ gives the count of candidate subtree with encoding $(i\ j\ -1)$. Total time for this step is $O(n^2)$ per tree. While computing F_2 we also create the vertical scope-list representation for each frequent item $i \in F_1$.

Computing $F_k (k \geq 3)$. Fig. 6 shows the pseudocode for the depth-first search for frequent subtrees (ENUMERATE-FREQUENT-SUBTREES). The input to the procedure is a set of elements of a class $[P]$, along with their scope-lists. Frequent subtrees are generated by joining the scope-lists of all pairs of elements (including self-joins). Before joining the scope-lists, a pruning step can be inserted to ensure that subtrees of the resulting tree are frequent. If this is true, then we can go ahead with the scope-list join, otherwise, we can avoid the join. For convenience, we use the set $\mathbf{R}$ to denote the up to two possible candidate subtrees that may result from $(x, i) \otimes (y, j)$, according to the class extension theorem, and we use $\mathcal{L}(\mathbf{R})$ to denote their respective scope-lists. The subtrees found to be frequent at the current level form the elements of classes for the next level. This recursive process is repeated until all frequent subtrees have been enumerated. If $[P]$ has n elements, the total cost is given as $O(ln^2)$, where l is the cost of a scope-list join (given later). In terms of memory management, it is easy to see that we need memory to store classes along a path in DFS search. At the very least, we need to store intermediate scope-lists for two classes, i.e., the current class $[P]$ and a new candidate class $[P_x^i]$.

4.2 Scope-List Joins ($\mathcal{L}(x) \cap_\otimes \mathcal{L}(y)$)

Scope-list join for any two subtrees in a class $[P]$ is based on interval algebra on their scope lists. Let $s_x = [l_x, u_x]$ and $s_y = [l_y, u_y]$ be scopes for nodes x and y. We say that s_x is *strictly less* than s_y, denoted $s_x < s_y$, if and only if $u_x < l_y$, i.e., the interval s_x has no overlap with s_y and it occurs before s_y. We say that s_x *contains* s_y, denoted $s_x \supset s_y$, if and only if $l_x \leq l_y$ and $u_x \geq u_y$, i.e., the s_y is a proper subset of s_x. The use of scopes allows us to compute in constant time whether y is a descendant of x or y is an embedded sibling of x. Recall from the candidate extension theorem (Theorem 1) that when we join elements $(x, i) \otimes (y, j)$, there can be at most two possible outcomes, i.e., we either add $(y, j+1)$ or (y, j) to the class $[P_x^i]$.

In-Scope Test. The first candidate $(y, j+1)$ is added to $[P_x^i]$ only when $i = j$ and, thus, refers to the candidate subtree with y as a child of node x. In other words, $(y, j+1)$ represents the subtree with encoding $(\mathcal{P}_x\ y)$. To check if this subtree occurs in an input tree T with tid t, we search if

```
TREEMINER (D, minsup):
  F_1 = { frequent 1-subtrees };
  F_2 = { classes [P]_1 of frequent 2-subtrees };
  for all [P]_1 ∈ E do Enumerate-Frequent-Subtrees([P]_1);

ENUMERATE-FREQUENT-SUBTREES([P]):
  for each element (x, i) ∈ [P] do
    [P_x^i] = ∅;
    for each element (y, j) ∈ [P] do
      R = {(x, i)⊗(y, j)};
      L(R) = {L(x) ∩_⊗ L(y)};
      if for any R ∈ R, R is frequent then
        [P_x^i] = [P_x^i] ∪ {R};
    Enumerate-Frequent-Subtrees([P_x^i]);
```

Fig. 6. TREEMINER algorithm.

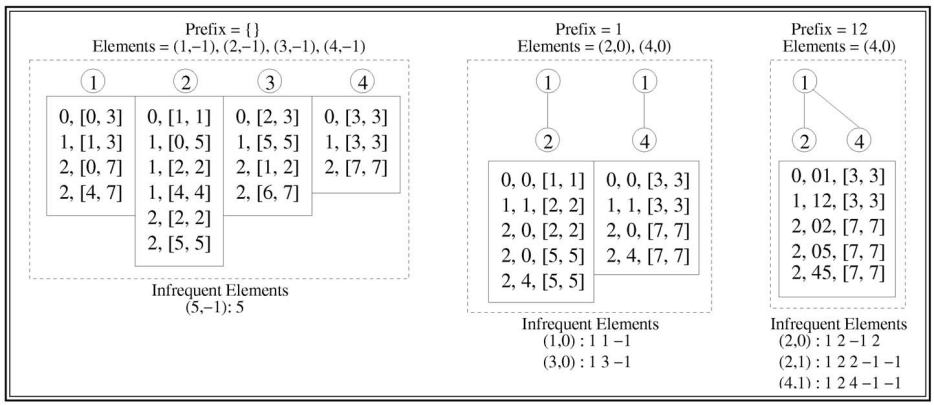


Fig. 7. Scope-list joins: $minsup = 100$ percent.

there exists triples $(t_y, m_y, s_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x) \in \mathcal{L}(x)$, such that: 1) $t_y = t_x = t$, i.e., the triples both occur in the same tree, with tid t. 2) $m_y = m_x = m$, i.e., x and y are both extensions of the same prefix occurrence, with match label m. 3) $s_y \subset s_x$, i.e., y lies within the scope of x.

If the three conditions are satisfied, we have found an instance where y is a descendant of x in some input tree T. We next extend the match label m_y of the old prefix P, to get the match label for the new prefix P_x^i (given as $m_y \cup l_x$) and add the triple $(t_y, \{m_y \cup l_x\}, s_y)$ to the scope-list of $(y, j+1)$ in $[P_x^i]$. We refer to this case as an *in-scope* test.

Out-Scope Test. The second candidate (y, j) represents the case when y is an embedded sibling of x, i.e., both x and y are descendants of some node at position j in the prefix P and the scope of x is strictly less than the scope of y. The element (y, j), when added to $[P_x^i]$, represents the pattern $(\mathcal{P}_x \ -1 \ \ldots \ -1 \ y)$ with the number of -1s depending on path length from j to x. To check if (y, j) occurs in some tree T with tid t, we need to check if there exist triples $(t_y, m_y, s_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x) \in \mathcal{L}(x)$, such that: 1) $t_y = t_x = t$, i.e., the triples both occur in the same tree, with tid t. 2) $m_y = m_x = m$, i.e., x and y are both extensions of the same prefix occurrence, with match label m. 3) $s_x < s_y$, i.e., x comes before y in depth-first ordering, and their scopes do not overlap.

If these conditions are satisfied, we add the triple $(t_y, \{m_y \cup l_x\}, s_y)$ to the scope-list of (y, j) in $[P_x^i]$. We refer to this case as an *out-scope* test. Note that if we just check whether s_x and s_y are disjoint (with identical tids and prefix match labels), i.e., either $s_x < s_y$ or $s_x > s_y$, then the support can be counted for unordered subtrees!

Each application of in-scope or out-scope test takes $O(1)$ time. Let a and b be the number of distinct (t, m) pairs in $\mathcal{L}(x, i)$ and $\mathcal{L}(y, j)$, respectively. Let α denote the average number of scopes with a match label. Then, the time to perform scope-list joins is given as $O(\alpha^2(a+b))$, which reduces to $O(a+b)$ if α is a small constant.

Fig. 7 shows an example of how scope-list joins work, using the database D from Fig. 5, with $minsup = 100$ percent, i.e., we want to mine subtrees that occur in all three trees in D. The initial class with empty prefix consists of four frequent items (1, 2, 3, and 4), with their scope-lists. All pairs of elements are considered for extension, including self-join. Consider the extensions from Item 1, which produces the new class $[1]$ with two frequent subtrees: $(1\ 2\ -1)$ and $(1\ 4\ -1)$. The infrequent subtrees are listed at the bottom of the class. While computing the new scope-list for the subtree $(1\ 2\ -1)$ from $\mathcal{L}(1) \cap_\otimes \mathcal{L}(2)$, we have to perform only in-scope tests, since we want to find those occurrences of 2 that are within some scope of 1 (i.e., under a subtree rooted at 1). Let s_i denote a scope for item i. For tree T_0, we find that $s_2 = [1, 1] \subset s_1 = [0, 3]$. Thus, we add the triple $(0, 0, [1, 1])$ to the new scope list. In like manner, we test the other occurrences of 2 under 1 in trees T_1 and T_2. Note that for T_2, there are three instances of the candidate pattern: $s_2 = [2, 2] \subset s_1 = [0, 7]$, $s_2 = [5, 5] \subset s_1 = [0, 7]$, and $s_2 = [5, 5] \subset s_1 = [4, 7]$. If a new scope-list occurs in at least *minsup* tids, the pattern is considered frequent.

Consider the result of extending class $[1]$. The only frequent pattern is $(1\ 2\ -1\ 4\ -1)$, whose scope-list is obtained from $\mathcal{L}(2, 0) \cap_\otimes \mathcal{L}(4, 0)$, by applications of out-scope test. We need to test for disjoint scopes, with $s_2 < s_4$, which have the same match label. For example, we find that $s_2 = [1, 1]$ and $s_4 = [3, 3]$ satisfy these conditions. Thus, we add the triple $(0, 01, [1, 1])$ to $\mathcal{L}(4, 0)$ in class $[1\ 2]$. Notice that the new prefix match label (01) is obtained by adding to the old prefix match label (0), the position where 2 occurs (1). The final scope list for the new candidate has three distinct tids and is thus frequent. There are no more frequent patterns at $minsup = 100$ percent.

Reducing Space Requirements. Generally speaking, the most important elements of the in-scope and out-scope tests are to make sure that $s_y \subset s_x$ and $s_x < s_y$, respectively. Whenever the test is true, we add $(t, \{m_y \cup l_x\}, s_y)$ to the candidate's scope-list. However, the match labels are only useful for resolving the prefix context when an item occurs more than once in a tree. Using this observation, it is possible to reduce the space requirements for the scope-lists. We add l_x to the match label m_y if and only if x occurs more than once in a subtree with tid t. Thus, if most items occur only once in the same tree, this optimization drastically cuts down the match label size, since the only match labels kept refer to items with more than one occurrence. In the special case that all items in a tree are distinct, the match label is always empty, and each element of a scope-list reduces to a $(tid, scope)$ pair. Consider the scope-list of $(4, 0)$ in class $[12]$ in Fig. 7. Since 4 occurs only once in T_0 and T_1, we can omit the match label from the first

two entries altogether, i.e., the triple $(0, 01, [3,3])$ becomes a pair $(0, [3,3])$, and the triple $(1, 12, [3,3])$ becomes $(1, [3,3])$.

Opportunistic Candidate Pruning. We mentioned above that, before generating a candidate k-subtree, S, we perform a pruning test to check if its $(k-1)$-subtrees are frequent. While this is easily done in a BFS pattern search method like PATTERNMATCHER (see Section 6), in a DFS search, we may not have all the information available for pruning, since some classes at level $(k-1)$ would not have been counted yet. TREEMINER uses an opportunistic pruning scheme whereby it first determines if a $(k-1)$-subtree would already have been counted. If it has been counted but is not found in F_{k-1}, we can safely prune S. How do we know if a subtree was counted? For this, we need to impose an ordering on the candidate generation, so that we can efficiently perform the subtree pruning test. Fortunately, our candidate extension method has the automatic ordering property (see Corollary 1). Thus, we know the exact order in which patterns will be enumerated. To apply a pruning test for a candidate S, we generate each subtree X and test if $X < S$ according to the candidate ordering property. If yes, we can apply the pruning test; if not, we test the next subtree. If S is not pruned, we perform scope-list join to get its exact frequency.

5 TREEMINERD: COUNTING DISTINCT OCCURRENCES

TREEMINER counts all embeddings of a frequent pattern within each database tree using the scope-list joins. The method is thus inherently efficient for counting weighted support, i.e., all possible occurrences over all possible trees in the database. Many applications, however, are only interested in counting the support, that is, instead of finding all embeddings of a subtree in the entire database, we may simply want to know the number of database trees that contain at least one embedding of a subtree. If there are relatively few embeddings per tree, TREEMINER continues to be very effective. On the other hand, if there are many duplicate labels and if the tree is highly branched, the number of embeddings can get very large and, consequently, the scope-lists can become very long, resulting in increased running time. If the application calls for the use of weighted support, the increased cost is acceptable, but if we want only support, it is possible to optimize TREEMINER to count only distinct occurrences of each pattern. We have thus devised a new method called TREEMINERD, which counts only the support rather than all embeddings. The overall algorithmic structure and candidate generation process of TREEMINERD remains the same as in TREEMINER, shown in Fig. 6, but TREEMINERD uses a different scope-list representation and scope-list joins for computing pattern frequency.

The main change in the scope-lists is that TREEMINERD does not maintain the match-labels, which keep track of all embeddings. Instead, TREEMINERD stores the scopes for all nodes on the right-most path within a tree; we call the new scope-lists as *scope-vector-lists* or *SV-lists* for short. Thus, each element of the new list is a pair of the form $(t, \mathbf{s})$, where t is a tree id and $\mathbf{s} = \{s_1, s_2, \cdots, s_m\}$ is the scope-vector of matching node scopes s_i on the rightmost path. Furthermore, $\mathbf{s}$ represents a *minimal* occurrence of the pattern within a database tree, i.e., there does not exist another scope-vector $\mathbf{s}'$ strictly contained in $\mathbf{s}$,[2] such that the pattern also occurs at nodes with scopes given by $\mathbf{s}'$. The in-scope and out-scope tests are then performed on these SV-lists.

5.1 SV-List Joins

Given two trees within the same equivalence class, we perform SV-list joins by looking at an SV-list element $(t_x, \mathbf{s}_x)$ for node x and an element $(t_y, \mathbf{s}_y)$ for node y. Let $\mathbf{s}_x = \{s_x^1, s_x^2, \cdots, s_x^m\}$ and $\mathbf{s}_y = \{s_y^1, s_y^2, \cdots, s_y^n\}$.

In-Scope Test. For the in-scope test, we first make sure that $t_x = t_y$, i.e., both nodes x and y occur in the same database tree. Next, we look at the last node-scope of scope-vectors $\mathbf{s}_x$ and $\mathbf{s}_y$, namely, s_x^m and s_y^n. If $s_y^n \subset s_x^m$ and there does not exist another last node-scope, say s_x^l, in another element of x's SV-list, such that $s_y^n \subset s_x^l \subset s_x^m$ (i.e., this is a minimal occurrence of the pattern), then we add the pair $(t_x, \mathbf{s}' = \{s_x^1, \cdots, s_x^k, \cdots, s_y^n\})$ to the SV-list of the subtree with encoding $(\mathcal{P}_x y)$ (where, $\mathbf{s}'$ represents the scope-vector for only those nodes on the rightmost path of the pattern).

Out-Scope Test. Given a pair of SV-list elements, one from x and one from y, namely, $(t_x, \mathbf{s}_x)$ and $(t_y, \mathbf{s}_y)$, for the out-scope test, we begin by ensuring that $t_x = t_y$. Second, we make sure that $s_x^m < s_y^n$, i.e., the last nodes of each element are disjoint and y node happens after x. Note that the equivalence class member (y, j) denotes that fact that the new candidate has prefix $\mathcal{P}_x$ and has rightmost node y attached to node j in the prefix. The final step in the out-scope test is to compare the scopes at position j in both x and y, i.e., s_x^j and s_y^j, and s_y^n. There are two cases when the out-scope test is satisfied: 1) if $s_y^n \subset s_x^j$ and $s_y^n > s_x^{j+1}$, i.e., the last node of y is contained in the jth node of x (with label z), whereas it is not contained in, but rather after, the $(j+1)$th node of x, or 2) if $s_y^n > s_x^j$ and $s_x^j \subset s_y^j$, i.e., the last node of y is after the jth node of x (which has label z) and the jth node of x is contained in the jth node of y (which also has label z). If 1) is true, then we add the pair $(t_x, \{s_x^1, \cdots, s_x^j, s_y^n\})$ to the SV-list of the new candidate, or if 2) is true, we add $(t_x, \{s_y^1, \cdots, s_y^j, s_y^n\})$. To maintain minimality, we store the pair only for the nearest jth node to y in a database tree.

Fig. 8 shows an example of how SV-list joins work, using the database D from Fig. 5, with $minsup = 100$ percent. The initial SV-lists are the same as the item scope-lists in Fig. 7.

While computing the new SV-lists for the subtrees $(1\ 2\ -1)$ and $(1\ 4\ -1)$, we have to perform only in-scope tests, since we want to find those occurrences of 2 (or 4) that are within some scope of 1 (i.e., under a subtree rooted at 1). The key is to keep only minimal occurrences. For example, in tree id 2, the node scopes $[0,7]$ and $[4,7]$ for label 1 both contain the scope $[5,5]$ for label 2. In this case, the SV-list for $(1\ 2\ -1)$ contains only the pair $(2, [4,7], [5,5])$. The SV-lists for both patterns are shown in the figure. Using these two, to compute the frequency of pattern $(1\ 2\ -1\ 4\ -1)$, we need to perform out-scope test. In our example, all tree ids

2. We say that a scope-vector $\mathbf{s}' = \{s'_1, s'_2, \cdots, s'_n\}$ is contained within another scope-vector $\mathbf{s} = \{s_1, s_2, \cdots, s_m\}$ if $(s_1 < s'_1 \wedge s'_n \leq s_m)$ or $(s_1 \leq s'_1 \wedge s'_n < s_m)$.

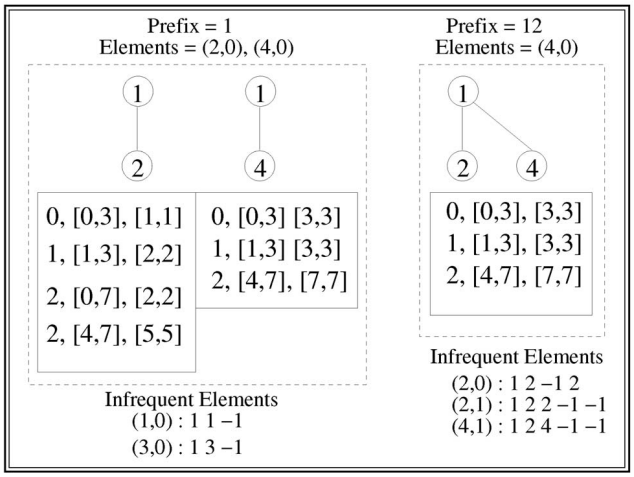


Fig. 8. SV-list joins: $minsup = 100$ percent.

belong to case 1) of the out-scope test. For example, for tree id 2, node label 4 has scope $[7, 7]$, whereas node label 2 has occurrences at scopes $[2, 2]$ and $[5, 5]$. Here, $j = 0$ and, thus, 4's scope $[7, 7]$ is contained in 2's jth node's scope $[0, 7]$ and, also, it is after 2's $(j + 1)$th node's scope $[2, 2]$. The out-scope test is true, but it is not minimal, since the test is also satisfied for 2's scope $[4, 7]$ and, thus, we add $(2, [4, 7], [7, 7])$ to the new candidate's SV-list.

6 PATTERNMATCHER ALGORITHM

PATTERNMATCHER serves as a base pattern matching algorithm to compare TREEMINER against. PATTERNMATCHER employs a breadth-first iterative search for frequent subtrees. Its high-level structure, as shown in Fig. 9, is similar to Apriori [3]. However, there are significant differences in how we count the number of subtree matches against an input tree T. For instance, we make use of equivalence classes throughout, and we use a prefix-tree data structure to index them, as opposed to hash-trees. The details of pattern matching are also completely different.

Due to lack of space, we describe only the main features of PATTERNMATCHER; see [39] for details. PATTERNMATCHER assumes that each tree T in D is stored in its string encoding (horizontal) format (see Fig. 5). The initial steps of the algorithm involve finding F_1 and F_2, which are computed as in TREEMINER.

Pattern Pruning. Before adding each candidate k-subtree to a class in C_k, we make sure that all its $(k-1)$-subtrees are also frequent. To efficiently perform this step, during creation of F_{k-1} (line 8), we add each individual frequent subtree into a hash table. Thus, it takes $O(1)$ time to check each subtree of a candidate and, since there can be k subtrees of length $k-1$, it takes $O(k)$ time to perform the pruning check for each candidate.

Prefix Tree Data Structure. Once a new candidate set has been generated, for each tree in D, we need to efficiently find matching candidates. We use a prefix tree data structure to index the candidates (C_k) to facilitate fast support counting. Furthermore, instead of adding individual subtrees to the prefix tree, we index an entire class using the class prefix.

PATTERNMATCHER (D, $minsup$):
1. $F_1 = \{$ frequent 1-subtrees $\}$;
2. $F_2 = \{$ classes of frequent 2-subtrees $\}$;
3. **for** $(k = 3; F_{k-1} \neq \emptyset; k = k + 1)$ **do**
4. $\quad C_k = \{$ classes $[P]_{k-1}$ of candidate k-subtrees $\}$;
5. $\quad$ **for** all trees T in D **do**
6. $\quad\quad$ Increment count of all $S \preceq T$, $S \in [P]_{k-1}$
7. $\quad C_k = \{$ classes of frequent k-subtrees $\}$;
8. $\quad F_k = \{$ hash table of frequent subtrees in $C_k\}$;
9. Set of all frequent subtrees $= \bigcup_k F_k$;

Fig. 9. PATTERNMATCHER algorithm.

Thus, if the prefix does not match the input tree T, then none of the class elements would match either. This allows us to rapidly focus on the candidates that are likely to be contained in T. Let $[P]$ be a class in C_k. An internal node of the prefix tree at depth d refers to the dth node in P's label sequence. An internal node at depth d points to a leaf node or an internal node at depth $d + 1$. A leaf node of the prefix tree consists of a list of classes with the same label sequence, thus, a leaf can contain multiple classes. For example, classes with prefix encodings $(1\ 2\ -1\ 4\ 3)$, $(1\ 2\ 4\ 3)$, $(1\ 2\ 4\ -1\ -1\ 3)$, etc., all have the same label sequence 1243 and, thus, belong to the same leaf.

Storing equivalence classes in the prefix tree as opposed to individual patterns results in considerable efficiency improvements while pattern matching. For a tree T, we can ignore all classes $[P]_{k-1}$ where $P \not\preceq T$. Only when the prefix has a match in T do we look at individual elements. Support counting consists of three main steps: 1) to find a leaf containing classes that may potentially match T, 2) to check if a given class prefix P exactly matches T, and 3) to check which elements of $[P]$ are contained in T.

Finding Potential Matching Leafs. Let $l(T)$ be the label sequence for a tree T in the database. To locate matching leafs, we traverse the prefix tree from the root, following child pointers based on the different items in $l(T)$, until we reach a leaf. This identifies classes whose prefixes have the same label sequence as a subsequence of $l(T)$. This process focuses the search to some leafs of C_k, but the subtree topology for the leaf classes may be completely different. We now have to perform an exact prefix match. In the worst case, there may be $\binom{n}{k} \approx n^k$ subsequences of $l(T)$ that lead to different leafs. However, in practice, it is much smaller, since only a small fraction of the leafs match the label sequences, especially as pattern length increases. The time to traverse from the root to a leaf is $O(k \log m)$, where m is the average number of distinct labels at an internal node. The total cost of this step is thus $O(kn^k \log m)$.

Prefix Matching. Matching the prefix P of a class in a leaf against the tree T is the main step in support counting. Let $X[i]$ denote the ith node of subtree X and let $X[i, \ldots, j]$ denote the nodes from positions i to j, with $j \geq i$. We use a recursive routine to test prefix matching. At the rth recursive call, we maintain the invariant that all nodes in $P[0, 1, \ldots, r]$ have been matched by nodes in $T[i_0, i_1, \ldots, i_r]$, i.e., prefix node $P[0]$ matches $T[i_0]$, $P[1]$ matches $T[i_1]$, etc., and, finally, $P[r]$ matches $T[i_r]$. Note that while nodes in P are traversed consecutively, the matching nodes in T can be far apart. We

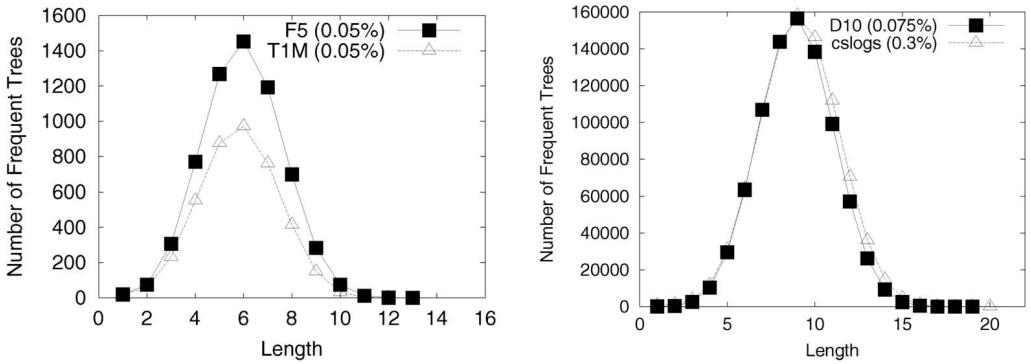


Fig. 10. Distribution of frequent trees by length.

thus have to maintain a stack of node scopes, consisting of the scope of all nodes from the root i_0 to the current rightmost leaf i_r in T. If i_r occurs at depth d, then the scope stack has size $d+1$.

Assume that we have matched all nodes up to the rth node in P. If the next node $P[r+1]$ to be matched is the child of $P[r]$, we likewise search for $P[r+1]$ under the subtree rooted at $T[i_r]$. If a match is found at position i_{r+1} in T, we push i_{r+1} onto the scope stack. On the other hand, if the next node $P[r+1]$ is outside the scope of $P[r]$ and is instead attached to position l (where $0 \leq l < r$), then we pop from the scope stack all nodes i_k, where $l < k \leq r$, and search for $P[r+1]$ under the subtree rooted at $T[i_l]$. This process is repeated until all nodes in P have been matched. This step takes $O(kn)$ time in the worst case. If each item occurs once, it takes $O(k+n)$ time.

Element Matching. If $P \preceq T$, we search for a match in T for each element $(x,k) \in [P]$ by searching for x starting at the subtree $T[i_{k-1}]$. (x,k) is either a descendant or embedded sibling of $P[k-1]$. Either check takes $O(1)$ time. If a match is found, the support of the element (x,k) is incremented by one. For counting support (at least one occurrence in T), the count is incremented only once per tree, or else, for weighted support (all occurrences in T), we continue the recursive process until all matches have been found.

7 Performance Results

Before looking at some application of tree mining in bioinformatics, we first compare TREEMINER versus PATTERNMATCHER and study their properties. All these experiments were performed on a 3.2GHz Pentium PC with 1GB memory running RedHat Linux 6.0. Timings are based on total wall-clock time and include preprocessing costs (such as creating scope-lists for TREEMINER).

Synthetic Data Sets. We wrote a synthetic data generation program to create a forest of trees. The program first constructs a master tree, $\mathcal{W}$, based on parameters supplied by the user. These parameters include the maximum fanout F of a node, the maximum depth D of the tree, the total number of nodes M in the tree, and the number of node labels N. We allow multiple nodes in the master tree to have the same label. The master tree is generated using the following recursive process. At a given node in the tree $\mathcal{W}$, we decide how many children to generate. The number of children is sampled uniformly at random from the range 0 to F. Before processing children nodes, we assign random probabilities to each branch, including an option of backtracking to the node's parent. The sum of all the probabilities for a given node is 1. The probability associated with a branch $b=(x,y)$ indicates the likelihood of going to child y from node x. As long as tree depth is less than or equal to maximum depth D, this process continues recursively. Once the master tree has been created we create as many subtrees of $\mathcal{W}$ as specified by the parameter T. To generate a subtree, we repeat the following recursive process starting at the root: Generate a random number between 0 and 1 to decide which child to follow or to backtrack. If a branch has already been visited, we select one of the other unvisited branches or backtrack. We used the following default values for the parameters: the number of labels $N=100$, the number of nodes in the master tree $M=10,000$, the maximum depth $D=10$, the maximum fanout $F=10$, and total number of subtrees $T=100,000$. We use three synthetic data sets: The $D10$ data set had all default values, $F5$ had all values set to default except for fanout $F=5$, and, for $T1M$, we set $T=1,000,000$, with remaining default values.

CSLOGS Data Set. This data set consists of Web logs files collected over one month at the CS department. The logs touched 13,361 unique Web pages within our department's Web site. After processing the raw logs, we obtained 59,691 user browsing subtrees of the CS department website. The average string encoding length for a user subtree was 23.3.

Fig. 10 shows the distribution of the frequent subtrees by length for the different data sets used in our experiments; all of them exhibit a symmetric distribution. For the lowest minimum support used, the longest frequent subtree in $F5$ and $T1M$ had 12 and 11 nodes, respectively. For cslogs and $D10$ data sets, the longest subtree had 18 and 19 nodes.

Performance Comparison. Fig. 11 shows the performance of PATTERNMATCHER versus TREEMINER and TREEMINERD. On the real cslogs data set, we find that TREEMINER and TREEMINERD are about two times faster than PATTERNMATCHER until support 2 percent. After that, the number of embeddings becomes very large and TREEMINERD outperforms TREEMINER. For $T1M$, TREEMINER is faster than PATTERNMATCHER by a factor of 4 and better than TREEMINERD. For $D10$, TREEMINER is 10 times faster than PATTERNMATCHER and about 1.5 times better than TREEMINERD. A similar trend is observed for

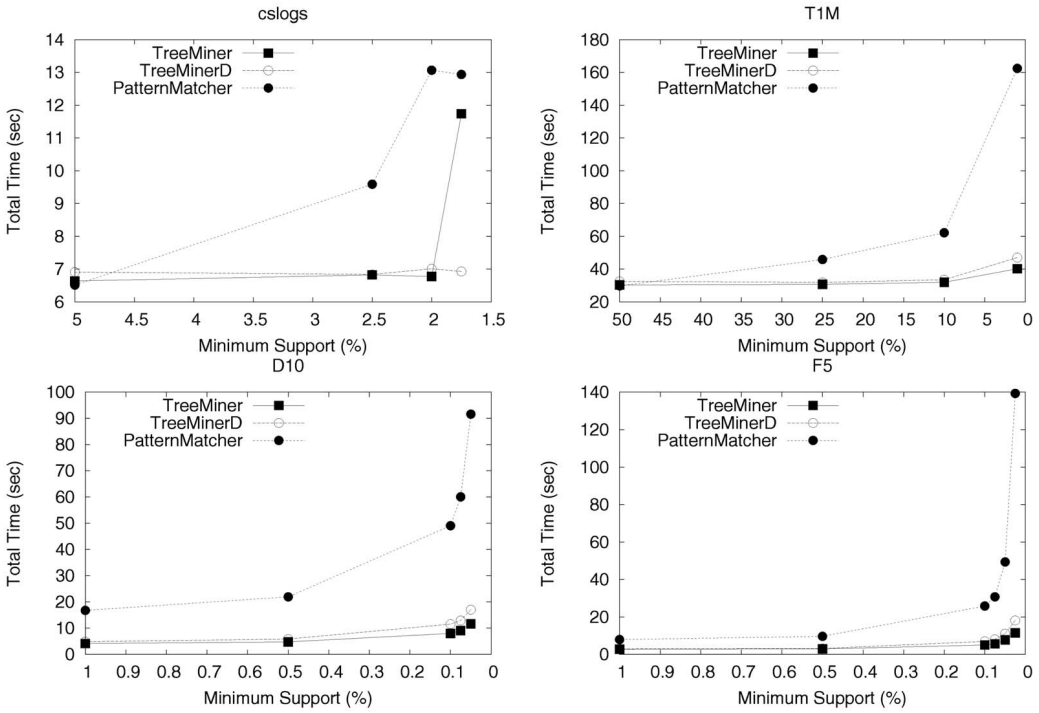


Fig. 11. Performance comparison.

$F5$. These experiments clearly indicate the superiority of scope-list based-method over the pattern matching method. Furthermore, TREEMINERD has clearly superior performance if there are many tree embeddings.

Scaleup Comparison. Fig. 12 shows how the algorithms scale with increasing number of trees in the database D, from 10,000 to one million trees. At a given level of support, we find a linear increase in the running time with increasing number of transactions for all algorithms; TREEMINER is 6 times faster than PATTERNMATCHER and 3 times faster than TREEMINERD.

Effect of Pruning. In Fig. 13, we evaluated the effect of candidate pruning on the performance of PATTERNMATCHER and TREEMINER. We find that PATTERNMATCHER (denoted PM in the graph) always benefits from pruning, since the fewer the number of candidates, the lesser the cost of support counting via pattern matching. On the other hand, TREEMINER (labeled TM in the graph) does not always benefit from its opportunistic pruning scheme. While pruning tends to benefit it at higher supports, for lower supports, its performance actually degrades by using candidate pruning. TREEMINER with pruning at 0.1 percent support on $D10$ is 2 times slower than TREEMINER with no pruning. There are two main reasons for this: First, to perform pruning, we need to store F_k in a hash table, and we need to pay the cost of generating the $(k-1)$ subtrees of each new k-pattern. This adds significant overhead, especially for lower supports when there are many frequent patterns. Second, the vertical representation is extremely efficient; it is actually faster to perform scope-list joins than to perform pruning test.

Fig. 14 shows the number of candidates generated on the $D10$ data set with no pruning, full pruning (in

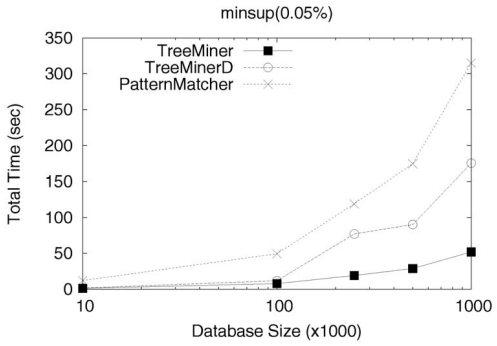


Fig. 12. Scaleup.

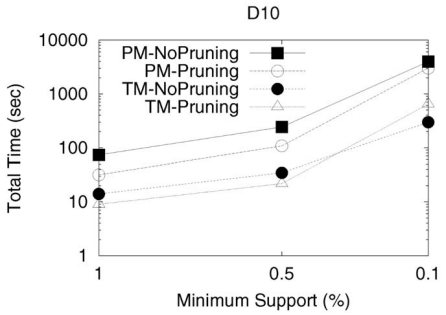


Fig. 13. Pruning.

minsup	No Pruning	Full Pruning	Opportunistic
1%	14595	2775	3505
0.5%	70250	10673	13736
0.1%	3555612	481234	536496

Fig. 14. Full versus opportunistic pruning.

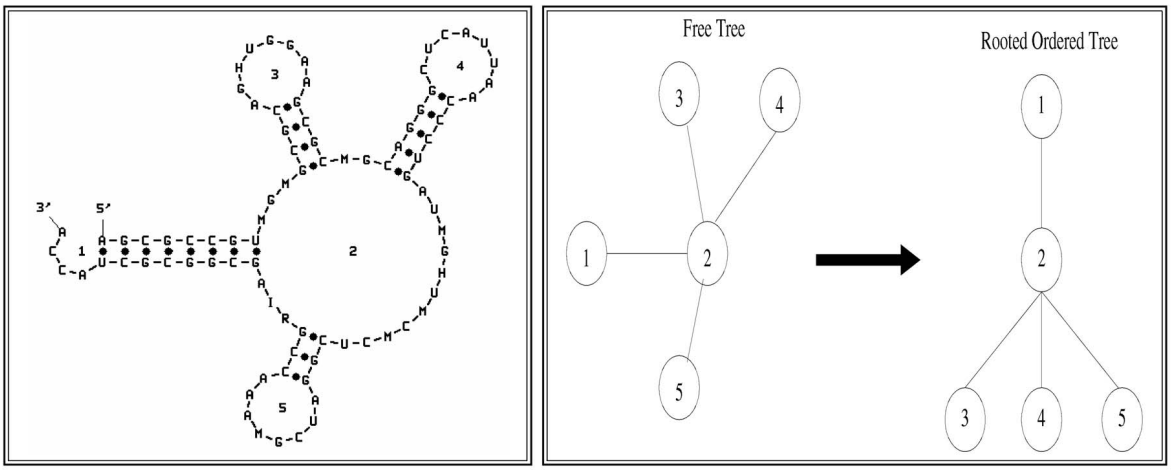


Fig. 15. An example RNA structure and its tree representation.

PATTERNMATCHER), and with opportunistic pruning (in TREEMINER). Both full pruning and opportunistic pruning are extremely effective in reducing the number of candidate patterns, and opportunistic pruning is almost as good as full pruning (within a factor of 1.3). Full pruning cuts down the number of candidates by a factor of 5 to 7! Pruning is thus essential for pattern matching methods and may benefit scope-list method in some cases (for high support).

8 Tree Mining Applications in Bioinformatics

In this section, we look at two applications of tree mining within bioinformatics domain: RNA structure and phylogenetic tree analysis.

8.1 RNA Structure

RNA molecules perform a variety of important biochemical functions, including translation, RNA splicing and editing, and cellular localization. Predicting RNA structure is thus an important problem; if a significant match to an RNA molecule of known structure and function is found, then a query molecule may have similar role. Here, we are interested in finding common motifs in a database of RNA structures [16].

Whereas RNA has a three-dimensional (3D) shape, it can be viewed in terms of its secondary structure, which is composed mainly of double-stranded regions formed by folding the single-stranded RNA molecule back on itself. To produce these double-stranded regions, a subsequence of bases (made up of four letters: A, C, G, U) must be complementary to another subsequence so that base-pairing can occur (G-C and A-U). It is these pairings that contribute to the energetic stability of the RNA molecule. Moreover, bulges may also form, for example, when the middle portion of a complementary subsequence doesn't participate in the base-pairing. Thus, there are different RNA secondary structures that are possible, such as: single-stranded RNA, double-stranded RNA helix, stem and loop or hairpin loop, bulge loop, interior loop, junction and multi-loops, etc. [24]. In addition, there may be tertiary interactions between RNA secondary structures, e.g., pseudo-knots, kissing hair-pins, hairpin-bulge contacts, etc. Fig. 15 shows a two-dimensional (2D) representation of a (transfer) RNA secondary structure. There are five loops (as numbered in the center); loop 1 is a bulge loop, 3, 4, and 5 are hairpin loops, and 2 is a multijunction loop.

To mine common RNA motifs or patterns, we use a tree representation of RNA secondary structure obtained from the RNA Matrix method used in the RAG (RNA-as-graph) database [15]. In the RNA tree, a nucleotide bulge, hairpin loop, or internal loop is considered a vertex if there is more than one unmatched nucleotide or noncomplementary base pair. The 3′ and 5′ ends of a helical stem are considered vertices, and so is a junction. An RNA stem with complementary base pairs (more than one) is considered an edge. The resulting free tree captures the topological aspects of RNA structure. To turn the free tree into a rooted labeled tree, we label each vertex from 1 to n, numbered sequentially from the 5′ to the 3′ end of the RNA strand. We choose the root to be vertex 1 and children of a node are ordered by their label number. For example, Fig. 15 shows the RNA free tree representing the RNA secondary structure and its rooted version.

We took 34 Eukarya RNA structures from the Ribonuclease P (Rnase P) database [7]. Rnase P is the ribonucleoprotein endonuclease that cleaves transfer (and other) RNA precursors. Rnase P is generally made up of two subunits, an RNA and a protein, and it is the RNA subunit that acts as the catalytic unit of the enzyme. The RNase P database is a compilation of currently available RNase P sequences and structures. For a given RNase P RNA subunit, we obtained a free tree using the RNA Matrix program[3] and then converted it into a rooted ordered tree. The resulting RNA tree data set has 34 trees, with the smallest having two vertices and the largest having 12 vertices. We then ran TREEMINER on this RNA tree data set. Fig. 16 shows the total time taken to mine the data set and the number of patterns found at different values of minimum support. We observe that mining at minimum support of one occurrence took less than 0.1 seconds and found 5,593 total patterns. An example of a common topological RNA pattern is also shown (rightmost figure); this pattern appears in at least 10

3. http://monod.biomath.nyu.edu/rna/analysis/rna_matrix.php.

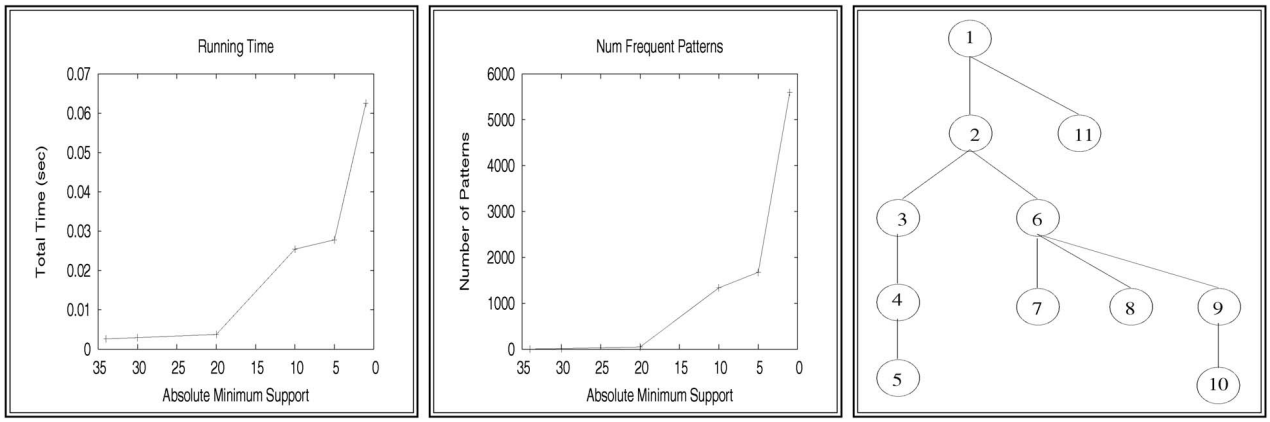


Fig. 16. RNase P database: (a) time, (b) num. patterns, and (c) example pattern.

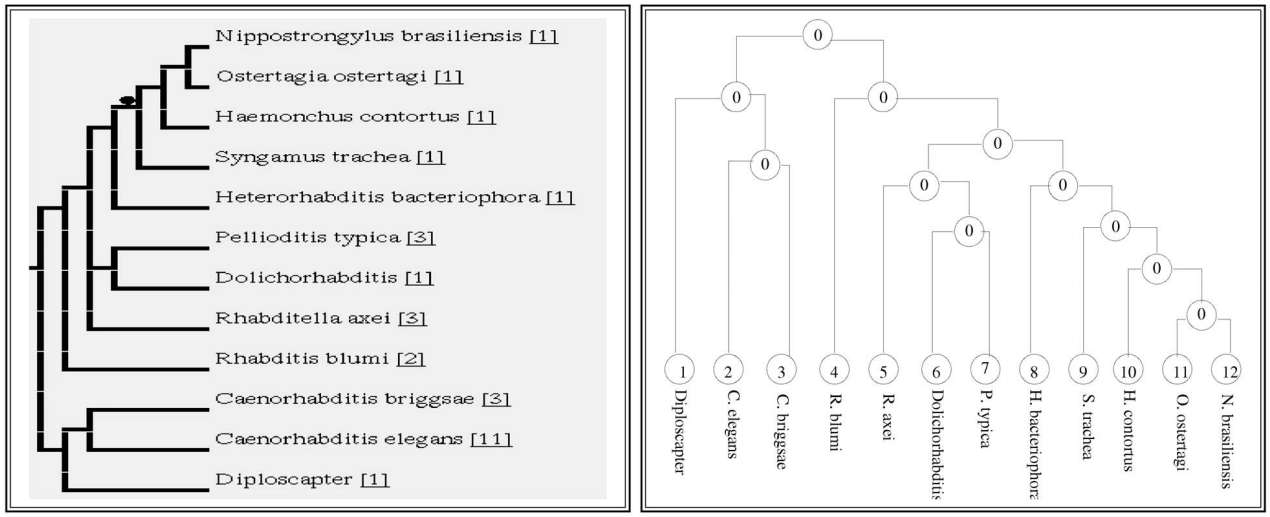


Fig. 17. (a) Part of the phylogenetic tree of phylum Nematoda. (b) Tree for mining.

of the 34 Eukarya RNA. By applying tree mining, it is thus possible to analyze RNA structures to enumerate all the frequent topologies. Such information can be a useful step in characterizing existing RNA structures and may help in structure prediction tasks [16]. Enumerating frequent RNA trees also helps in cataloging the kinds of RNA structures seen in nature [15].

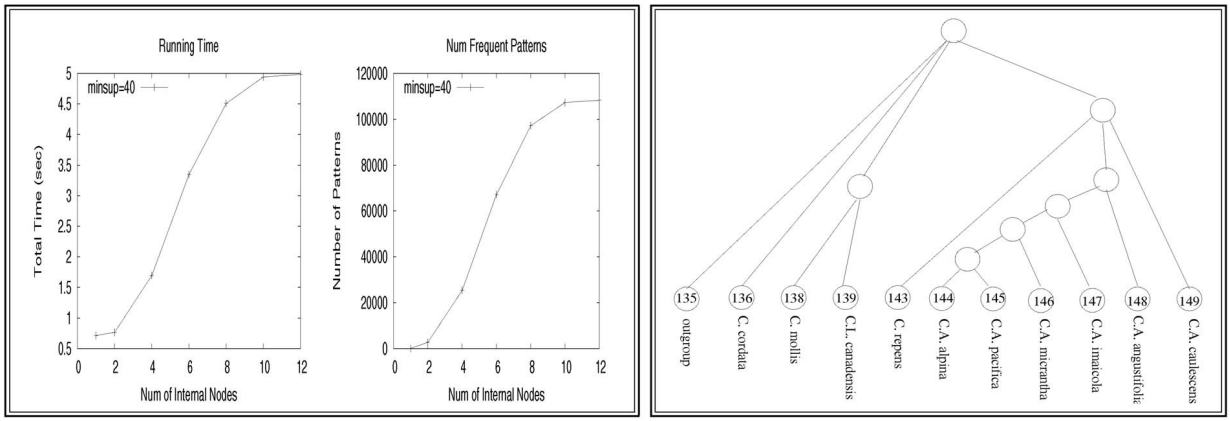
8.2 Phylogenetic Trees

Given several phylogenies (i.e., evolutionary trees) from the Tree of Life [23], indicating evolutionary history of several organisms, one might be interested in discovering if there are common subtree patterns. This is an important task, since there are many algorithms for inferring phylogenies, and biologists are often interested in finding consensus subtrees (those shared by many trees) [27]. Tree mining can also be used to mine cousin pairs in phylogenetic trees [31]. A cousin pair is essentially a pair of siblings, and mining pairs that share common ancestors gives important clues about the evolutionary divergence between two organisms or species.

TreeBASE is a relational database designed to manage and explore information on phylogenetic relationships.[4] It stores phylogenetic trees and data matrices used to generate them from published research papers. It includes bibliographic information on phylogenetic studies, as well as details on taxa, methods, and analyses performed; it contains all types of phylogenetic data (e.g., trees of species, trees of populations, trees of genes) representing all biotic taxa. The database is ideally suited to allow retrieval and recombination of trees and data from different studies; it thus provides a means of assessing and synthesizing phylogenetic knowledge.

Fig. 17 shows part of the evolutionary relationship between organisms of the plylum Nematoda taken from the TreeBase site. This tree was produced using a parsimony-based phylogenetic tree construction method [24]; using different algorithms may produce several variants of the evolutionary relationships. Tree mining can help infer the parts of the phylogeny that are common among many alternate evolutionary trees.

4. http://www.treebase.org/.



(a) (b)

Fig. 18. Phylogenetic data: (a) runtime and num. patterns and (b) example pattern.

We took 1,974 trees from the TreeBase data set and converted them into a format suitable for mining. We give each organism a unique label (e.g., C. elegans has label 2), and we give each internal node the same label (e.g., 0). Given the resulting database of 1,974 trees, we mine for frequent patterns. To prevent combinatorial effects, we also impose a constrain on the number of internal nodes allowed in the mined patterns; this constraint is incorporated during mining for efficiency reasons (as opposed to postprocessing). Fig. 18 shows the running time and number of patterns found for an absolute support value of 40, as the number of internal nodes increase from 1 to 12. As we allow more internal nodes, more patterns are found. Since there are many internal nodes with label 0, we used TREEMINERD for fast mining. An example of a mined frequent pattern (with frequency 42) is also shown; this pattern shows the evolutionary relationship between members of the Circaea plant family. Notice how the most closely related organisms, e.g., Circaea Alpina (C.A.), group together (right branch under the root).

9 Related Work

Tree mining, being an instance of frequent structure mining, has obvious relation to association [3] and sequence [4] mining. Frequent tree mining is also related to tree isomorphism [29] and tree pattern matching [11]. Given a pattern tree P and a target tree T, with $|P| \leq |T|$, the subtree isomorphism problem is to decide whether P is isomorphic to any subtree of T, i.e., there is a one-to-one mapping from P to a subtree of T, that preserves the node adjacency relations. In tree pattern matching, the pattern and target trees are labeled and ordered. We say that P matches T at node v if there exists a one-to-one mapping from nodes of P to nodes of T such that: 1) the root of P maps to v, 2) if x maps to y, then x and y have the same labels, and 3) if x maps to y and x is not a leaf, then the ith child of x maps to the ith child of y. Both subtree isomorphism and pattern matching deal with induced subtrees, while we mine embedded subtrees. Further, we are interested in enumerating all common subtrees in a collection of trees. The tree inclusion problem was studied in [19], i.e., given labeled trees P and T, can P be obtained from T by deleting nodes? This problem is equivalent to checking if P is embedded in T. The paper presents a dynamic programming algorithm for solving ordered tree inclusion, which could potentially be substituted for the pattern matching step in PATTERNMATCHER. However, PATTERNMATCHER utilizes prefix information for fast subtree checking, and its three step pattern matching is very efficient over a sequence of such operations.

Recently, tree mining has attracted a lot of attention. We developed TREEMINER [39], [40] to mine labeled, embedded, and ordered subtrees. The notions of scope-lists and rightmost extension were introduced in that work. TREEMINER was also used in building a structural classifier for XML data [41]. This current paper focuses on bioinformatics applications and presents a new algorithm called TREEMINERD to mine distinct occurrences of trees. Asai et al. [5] presented FreqT, an apriori-like algorithm for mining labeled ordered trees; they independently proposed the rightmost candidate generation scheme. Wang and Liu [34] developed an algorithm to mine frequently occurring subtrees in XML documents. Their algorithm is also reminiscent of the level-wise Apriori [3] approach, and they mine induced subtrees only. There are several other recent algorithms that mine different types of tree patterns, which include FreeTreeMiner [9] which mines induced, unordered, free trees (i.e., there is no distinct root); FreeTreeMiner for graphs [28] for extracting free trees in a graph database; and PathJoin [35], uFreqt [26], uNot [6], and HybridTreeMiner [10], which mine induced, unordered trees. TreeFinder [32] uses an Inductive Logic Programming approach to mine unordered, embedded subtrees, but it is not a complete method, i.e, it can miss many frequent subtrees, especially as support is lowered or when the different trees in the database have common node labels. SingleTreeMining [31] is another algorithm for mining rooted, unordered trees, with application to phylogenetic tree pattern mining. Recently, XSpanner [33], a pattern-growth-based method, has been proposed for mining embedded ordered subtrees. They report that XSpanner outperforms TREEMINER, however, note that TREEMINER mines all embeddings, whereas XSpanner counts only the

distinct trees. Therefore, we plan to compare TREEMINERD with XSpanner in the future.

There has been active work in indexing and querying XML documents [2], [14], [22], [42], which are mainly tree (or graph) structured. To efficiently answer ancestor-descendant queries various node numbering schemes similar to ours have been proposed [1], [22], [42]. Other work has looked at path query evaluation that uses local knowledge within data graph based on path constraints [2] or graph schemas [14]. The major difference between these works and ours is that instead of answering user-specified queries based on regular path expressions, we are interested in finding all frequent tree patterns among the documents. A related problem of accurately estimating the number of matches of a small node-labeled tree in a large labeled tree was presented in [8]. They compute a summary data structure and then give frequency estimates based on this summary, rather than using the database for exact answers. In contrast, we are interested in the exact frequency of subtrees. Furthermore, their work deals with traditional (induced) subtrees, while we mine embedded subtrees.

There has also been recent work in mining frequent graph patterns. The AGM algorithm [18] discovers induced (possibly disconnected) subgraphs. The FSG algorithm [21] improves upon AGM and mines only the connected subgraphs. Both methods follow an Apriori-style level-wise approach. Recent methods to mine graphs using a depth-first tree-based extension have been proposed in [36], [37]. Another method uses a candidate generation approach based on Canonical Adjacency Matrices [17]. The GASTON method [25] adopts an interesting step-wise approach using a combination of path, free tree, and, finally, graph mining to discover all frequent subgraphs. There are important differences in graph mining and tree mining. Our trees are rooted and, thus, have a unique ordering of the nodes based on depth-first traversal. In contrast, graphs do not have a root and allow cycles. For mining graphs, the methods above first apply an expensive *canonization* step to transform graphs into a uniform representation. This step is unnecessary for tree mining. Graph mining algorithms are likely to be overly general (thus, not efficient) for tree mining. Our approach utilizes the tree structure for efficient enumeration.

The work by Dehaspe et al. [13] describes a level-wise Inductive Logic Programming-based technique to mine frequent substructures (subgraphs) describing the carcinogenesis of chemical compounds. Work on molecular feature mining has appeared in [20]. The SUBDUE system [12] also discovers graph patterns using the Minimum Description Length principle. An approach termed Graph-Based Induction (GBI) was proposed in [38], which uses beam search for mining subgraphs. However, both SUBDUE and GBI may miss some significant patterns, since they perform a heuristic search. In contrast to these approaches, we are interested in developing efficient algorithms for tree patterns.

10 Conclusions

In this paper, we introduced the notion of mining embedded subtrees in a (forest) database of trees. Among our novel contributions is the procedure for systematic candidate subtree generation, i.e., no subtree is generated more than once. We utilized a string encoding of the tree that is space-efficient to store the horizontal data set and we use the notion of a node's scope to develop a novel vertical representation of a tree called scope-lists. Our formalization of the problem is flexible enough to handle several variations. For instance, if we assume the label on each node to be the same, our approach mines all unlabeled trees. A simple change in the candidate tree extension procedure allows us to discover subforests (disconnected patterns). Our formulation can find frequent trees in a forest of many trees or all the frequent subtrees in a single large tree. Finally, it is relatively easy to extend our techniques to find unordered trees (by modifying the out-scope test) or to use the traditional definition of a subtree. To summarize, this paper proposes a framework for tree mining which can easily encompass most variants of the problem that may arise in different domains.

We introduced a novel algorithm, TREEMINER, for tree mining. TREEMINER uses depth-first search; it also uses the novel scope-list vertical representation of trees to quickly compute the candidate tree frequencies via scope-list joins based on interval algebra. We extended the approach to TREEMINERD, an algorithm that counts only distinct occurrences using the notion of scope-vector lists. We compared their performance against a base algorithm, PATTERNMATCHER. Experiments on real and synthetic data confirmed that TREEMINER outperforms PATTERNMATCHER from a factor of 2 to 10 and scales linearly in the number of trees in the forest. We studied two applications of tree mining: finding common RNA structures and mining common phylogenetic subtrees.

For future work, we plan to extend our tree mining framework to incorporate user-specified constraints. Given that tree mining, though able to extract informative patterns, is an expensive task, performing general unconstrained mining can be too expensive and is also likely to produce many patterns that may not be relevant to a given user. Incorporating constraints is one way to focus the search and to allow interactivity. We also plan to develop efficient algorithms to mine maximal frequent subtrees from dense data sets which may have very large subtrees. Finally, we plan to apply our tree mining techniques to other compelling applications, such as the extraction of structure from XML documents and their use in classification, clustering, etc.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) CAREER Award IIS-0092978, the US Department of Energy (DOE) Career Award DE-FG02-02ER25538, and NSF grants EIA-0103708 and EMT-0432098.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo, "Compact Labeling Schemes for Ancestor Queries," *Proc. ACM Symp. Discrete Algorithms,* Jan. 2001.

[2] S. Abiteboul and V. Vianu, "Regular Path Expressions with Constraints," *Proc. ACM Int'l Conf. Principles of Database Systems,* May 1997.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo, "Fast Discovery of Association Rules," U. Fayyad et al., eds., *Advances in Knowledge Discovery and Data Mining,* pp. 307-328, Menlo Park, Calif.: AAAI Press, 1996.

[4] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. 11th Int'l Conf. Data Eng.,* 1995.

[5] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa, "Efficient Substructure Discovery from Large Semi-Structured Data," *Proc. Second SIAM Int'l Conf. Data Mining,* Apr. 2002.

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

[7] J.W. Brown, "The Ribonuclease P Database," *Nucleic Acids Research,* vol. 27, no. 1, pp. 314-315, 1999.

[8] Z. Chen, H.V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava, "Counting Twig Matches in a Tree," *Proc. 17th Int'l Conf. Data Eng.,* 2001.

[9] Y. Chi, Y. Yang, and R.R. Muntz, "Indexing and Mining Free Trees," *Proc. Third IEEE Int'l Conf. Data Mining,* 2003.

[10] Y. Chi, Y. Yang, and R.R. Muntz, "HybridTreeMiner: An Efficient Algorihtm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms," *Proc. 16th Int'l Conf. Scientific and Statistical Database Management,* 2004.

[11] R. Cole, R. Hariharan, and P. Indyk, "Tree Pattern Matching and Subset Matching in Deterministic $o(n \log^3 n)$-Time," *Proc. 10th Symp. Discrete Algorithms,* 1999.

[12] D. Cook and L. Holder, "Substructure Discovery Using Minimal Description Length and Background Knowledge," *J. Artificial Intelligence Research,* vol. 1, pp. 231-255, 1994.

[13] L. Dehaspe, H. Toivonen, and R. King, "Finding Frequent Substructures in Chemical Compounds," *Proc. Fourth Int'l Conf. Knowledge Discovery and Data Mining,* Aug. 1998.

[14] M. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas," *Proc. IEEE Int'l Conf. Data Eng.,* Feb. 1998.

[15] H.H. Gan, D. Fera, J. Zorn, N. Shiffeldrim, M. Tang, U. Laserson, N. Kim, and T. Schlick, "RAG: RNA-As-Graphs Database—Concepts, Analysis, and Features," *Bioinformatics,* vol. 20, no. 8, pp. 1285-1291, 2004.

[16] H.H. Gan, S. Pasquali, and T. Schlick, "Exploring the Repertoire of RNA Secondary Motifs Using Graph Theory with Implications for RNA Design," *Nucleic Acids Research,* vol. 31, pp. 2926-2943, 2003.

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

[18] A. Inokuchi, T. Washio, and H. Motoda, "Complete Mining of Frequent Patterns from Graphs: Mining Graph Data," *Machine Learning,* vol. 50, no. 3, pp. 321-354, 2003.

[19] P. Kilpelainen and H. Mannila, "Ordered and Unordered Tree Inclusion," *SIAM J. Computing,* vol. 24, no. 2, pp. 340-356, 1995.

[20] S. Kramer, L. De Raedt, and C. Helma, "Molecular Feature Mining in HIV Data," *Proc. Int'l Conf. Knowledge Discovery and Data Mining,* 2001.

[21] M. Kuramochi and G. Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 9, pp. 1038-1051, Sept. 2004.

[22] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proc. 27th Int'l Conf. Very Large Data Bases,* 2001.

[23] V. Morell, "Web-Crawling up the Tree of Life," *Science,* vol. 273, no. 5275, pp. 568-570, Aug. 1996.

[24] D.W. Mount, *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Press, 2001.

[25] S. Nijssen and J.N. Kok, "A Quickstart in Frequent Structure Mining Can Make a Difference," *Proc. ACM SIGKDD Int'l Conf. KDD,* 2004.

[26] S. Nijssen and J.N. Kok, "Efficient Discovery of Frequent Unordered Trees," *Proc. First Int'l Workshop Mining Graphs, Trees, and Sequences,* 2003.

[27] R.D. Page and E.C. Holmes, *Molecular Evolution: A Phylogenetic Approach.* Blackwell Science, 1998.

[28] U. Ruckert and S. Kramer, "Frequent Free Tree Discovery in Graph Data," *Special Track on Data Mining, Proc. ACM Symp. Applied Computing,* 2004.

[29] R. Shamir and D. Tsur, "Faster Subtree Isomorphism," *J. Algorithms,* vol. 33, pp. 267-280, 1999.

[30] B. Shapiro and K. Zhang, "Comparing Multiple RNA Secondary Strutures Using Tree Comparisons," *Computer Applications in Biosciences,* vol. 6, no. 4, pp. 309-318, 1990.

[31] D. Shasha, J. Wang, and S. Zhang, "Unordered Tree Mining with Applications to Phylogeny," *Proc. Int'l Conf. Data Eng.,* 2004.

[32] A. Termier, M-C. Rousset, and M. Sebag, "Treefinder: A First Step Towards XML Data Mining," *Proc. IEEE Int'l Conf. Data Mining,* 2002.

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

[34] K. Wang and H. Liu, "Discovering Typical Structures of Documents: A Road Map Approach," *Proc. ACM SIGIR Conf. Information Retrieval,* 1998.

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

[36] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," *Proc. IEEE Int'l Conf. Data Mining,* 2002.

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

[38] K. Yoshida and H. Motoda, "CLIP: Concept Learning from Inference Patterns," *Artificial Intelligence,* vol. 75, no. 1, pp. 63-92, 1995.

[39] M.J. Zaki, "Efficiently Mining Trees in a Forest," Technical Report 01-7, Computer Science Dept., Rensselaer Polytechnic Inst., July 2001.

[40] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest," *Proc. Eighth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* July 2002.

[41] M.J. Zaki and C.C. Aggarwal, "XRules: An Effective Structural Classifier for XML Data," *Proc. Ninth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* Aug. 2003.

[42] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Managment Systems," *Proc. ACM Int'l Conf. Management of Data,* May 2001.

Mohammed J. Zaki received the PhD degree in computer science from the University of Rochester in 1998. Presently, he is an associate professor of computer science at RPI. His research interests focus on developing novel data mining techniques for bioinformatics, performance mining, Web mining, etc. He has published more than 100 papers on data mining, coedited 11 books (including *Data Mining in Bioinformatics*, Springer-London, 2005), served as guest editor for several journals, served on the program committees of major international conferences, and cochaired many workshops (BIOKDD, HPDM, DMKD, etc.) in data mining. He is currently an associate editor for *IEEE Transactions on Knowledge and Data Engineering*, action editor for *Data Mining and Knowledge Discovery: An Int'l Journal*, and editor for *Int'l Journal of Data Warehousing and Mining*, and *ACM SIGMOD Digital Symposium Collection*. He received the US National Science Foundation CAREER Award in 2001 and the Department of Energy Early Career Principal Investigator Award in 2002. He also received the ACM Recognition of Service Award in 2003. He is a member of the IEEE.