New Techniques for Mining Frequent Patterns in Unordered Trees

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

***Abstract*—We consider a new tree mining problem that aims to discover restrictedly embedded subtree patterns from a set of rooted labeled unordered trees. We study the properties of a canonical form of unordered trees, and develop new Apriori-based techniques to generate all candidate subtrees level by level through two efficient rightmost expansion operations: 1) pairwise joining and 2) leg attachment. Next, we show that restrictedly embedded subtree detection can be achieved by calculating the restricted edit distance between a candidate subtree and a data tree. These techniques are then integrated into an efficient algorithm, named frequent restrictedly embedded subtree miner (FRESTM), to solve the tree mining problem at hand. The correctness of the FRESTM algorithm is proved and the time and space complexities of the algorithm are discussed. Experimental results on synthetic and real-world data demonstrate the effectiveness of the proposed approach.**

***Index Terms*—Apriori algorithm, pattern matching, pattern mining and recognition, rooted labeled unordered trees.**

I. Introduction

ROOTED labeled unordered trees are trees in which there is a root, each node has a label, and the left-to-right order among siblings is unimportant. Such trees find many applications where they are used to represent industrial parts [16], web connectivity [18], semi-structured XML data [11], RNA [7], phylogeny [17], [21], [25], prerequisite trees, and chemistry compound data [19]. Tree matching and pattern matching in general are very useful operations in these applications [9], [12]. The reader is referred to [16], [23], and [25] for the various algorithms designed for tree matching. In this paper, we extend tree matching to tree mining, aiming to find novel subtree patterns frequently occurring in multiple rooted labeled unordered trees.

During the past decade, considerable research has been conducted in the tree mining field. According to the type of trees, tree mining problems can be broadly classified into ordered tree mining (e.g., TreeMiner [21], FREQT [1], OInduced [3]), unordered tree mining (e.g., Unot [2], uFreqt [15]), rooted or unrooted tree mining (e.g., CMTreeMiner [5], HybridTreeMiner [6]). According to the type of subtree patterns, tree mining problems can be further classified into induced subtree discovery [1], [3], [6], [26], or embedded subtree discovery [10], [13], [21]. Here, we are interested in mining a new type of patterns, namely restrictedly embedded subtrees, in rooted labeled unordered trees.

In contrast to embedded subtrees, restrictedly embedded subtrees provide a middle ground which can properly capture most hidden relationships in the unordered trees while at the same time excluding some over-relaxed patterns. Specifically, induced subtrees preserve parental relationships where any parent-child relationship of a subtree pattern must be strictly present in a data tree. Therefore induced subtrees are rigid patterns that cannot reveal hidden relationships such as multiple generation inheritance in a data tree. By contrast, embedded subtrees generalize induced subtrees in the sense that the parent-child relationship is relaxed to an ancestor-descendant relationship, i.e., a parent of a node in a subtree pattern may be an ancestor of that node in a data tree. The embedded subtrees are expected to be able to exhibit patterns "hidden" (or embedded) deeply within large data trees which might be missed by the induced subtrees [21]. However, some hidden patterns might be too relaxed to have meaningful branching structures in some applications [7], [18]. This motivates the need of restrictedly embedded subtrees in data trees. Conceptually, a restrictedly embedded subtree s is an embedded subtree that satisfies the least common ancestor constraint, i.e., if two nodes are in s, then their least common ancestor in a data tree must also be in s.

Consider, for example, the three data trees t_1, t_2, t_3 and three subtree patterns s_1, s_2, and s_3 in Fig. 1. Here, s_1 is clearly an induced subtree of t_1, an embedded subtree of t_2 due to relaxing the parent-child relationship to the ancestor-descendant relationship [10], [21], and a restrictedly embedded subtree of t_3 due to that s_1 satisfies the least common ancestor constraint imposed by t_3. For the same reasons, s_2 is an embedded subtree of t_2 and a restrictedly embedded subtree of t_3. Please note that an induced subtree is always a restrictedly embedded subtree, which in turn is always an embedded subtree, but not vice versa. As for s_3, it is an induced subtree of t_1 but not an induced or embedded subtree of t_2 and t_3. Mining restrictedly embedded subtrees is useful in those applications where the least common ancestor relationship of nodes must

Manuscript received September 10, 2013; revised May 17, 2014 and July 27, 2014; accepted July 29, 2014. Date of publication August 14, 2014; date of current version May 13, 2015. This work was supported in part by the Tsinghua Global Scholars Fellowship Program under Grant 201108220001, in part by the National Nature Science Foundation of China under Grant 6136309, Grant 61272087, Grant 61073008, Grant 60773148, and Grant 60503039, and in part by the Beijing Natural Science under Grant 4122039 and Grant 4082016. This paper was recommended by Associate Editor H. Wang.

S. Zhang is with the Department of Mathematics, Computer Science and Statistics, State University of New York, Oneonta, NY 13820 USA (e-mail: zhangs@oneonta.edu).

Z. Du is with the Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: duzh@tsinghua.edu.cn).

J. T. L. Wang is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102 USA (e-mail: wangj@njit.edu).

Digital Object Identifier 10.1109/TCYB.2014.2345579

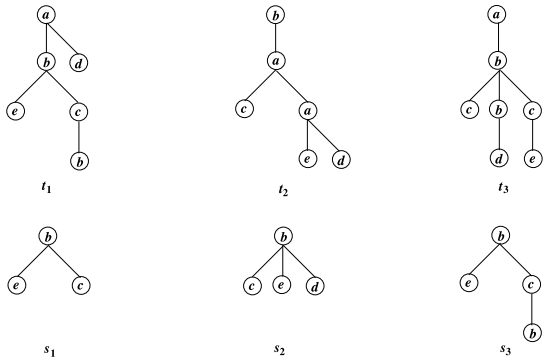

Fig. 1. Set of three rooted labeled unordered trees $TS = \{t_1, t_2, t_3\}$ and three subtree patterns s_1, s_2, and s_3.

be preserved [14], [18]. For example, a student may want to find out frequent restrictedly embedded subtrees from multiple prerequisite trees for a course as we will show later.

Our work makes several contributions to the tree mining field. They include the following.

1) Formulation of a new frequent restrictedly embedded subtree mining problem.
2) Design of a candidate generation algorithm using both rightmost leg attachment and rightmost joining methods.
3) Design of an edit distance based subtree detection algorithm.
4) Integral design of a set of techniques based on the Apriori principle.
5) Prototype implementation of the proposed algorithms.
6) Experimental comparison of our method with a closely related tree mining algorithm.

The rest of this paper is organized as follows. Section II presents basic concepts, definitions and terms. Section III presents our approach to solving the tree mining problem at hand. Section IV reports experimental results. Section V concludes the paper.

II. Preliminaries

To facilitate the analysis of the problem at hand and the presentation of our algorithms, we adopt the following formal definitions and notation that have been introduced in the tree mining literature [2], [15]. Let $\lambda = \{l_1, l_2, \ldots\}$ be a finite set of labels built upon an alphabet $\sum$ whose elements follow an alphabetical order, so a total order $\leq_{\sum}$ can be extended to $\leq_{\lambda}$. A rooted labeled ordered tree t is a directed acyclic graph described by a 5-tuple, $t = (V, r, E, S, L)$, where V, r, E, S, and L denote respectively the set of nodes, a distinguished root node, the set of parent-child edges, the sibling relation, and the labeling function of V. More specifically, $V = \{v_1, \ldots, v_k\}$ is the set of nodes; k denotes the size of the tree which is simply the size of the node set, i.e., $k = |t| = |V|$. Without loss of generality, the elements in V are uniquely and consecutively indexed according to the preorder depth-first traversal of t. Therefore, V can also be represented by their indexes $\{1, \ldots, k\}$. Obviously, the index of the root of t is 1 and the index of the rightmost leaf, i.e., the last node, is k. $E \subset V \times V$ is the set of parent-child edges. The root r has no parent and a leaf node has no children. If $(u, v) \in E$ then we say that u is the parent of v, and v is a child of u, denoted by $u = parent(v)$ and $v \in children(u)$. For every node $v \in V$, there is a unique path from the root r to v. If a node u is on the path from the root r to a node v, the node u is an ancestor of the node v, denoted by $u \in ancestors(v)$, and v is a descendant of u, denoted by $v \in descendants(u)$. S represents the left-to-right sibling ordering. If $(v_1, v_2) \in S$, then we say that v_1 is the immediate left sibling of v_2. L is a mapping function assigning labels to nodes: $V \rightarrow \lambda$. The label of a node with index i is denoted by $L(i)$, or $label(i)$. Consider, for example, s_1 in Fig. 1. The indexes for the root, the left child and the right child are 1, 2 and 3, respectively, and $label(1) = b$, $label(2) = e$ and $label(3) = c$. The depth of v is the number of nodes from r to v. Let u, v and w be three nodes in the tree t. If u is the least common ancestor of v and w, the relationship among them is denoted by $u = lca(v, w)$. A k-subtree (or k-pattern) is a subtree having exactly k nodes.

Our work considers unordered trees where the ordering among sibling nodes is insignificant or even unavailable [15], [20]. We denote by $\Phi(t) = (V, r, E, L)$ the unordered tree obtained from the above ordered tree t by ignoring the order S over the siblings in t. Two ordered trees t_1 and t_2 are equivalent to each other when treated as unordered trees, denoted by $t_1 \equiv t_2$, if $\Phi(t_1) = \Phi(t_2)$ holds. The patterns we intend to discovery from data trees can be defined as follows.

Definition 1 (Restrictedly embedded subtrees): Let s and t be the pattern tree and the data tree respectively, and let N_s and N_t be the sets of nodes of s and t respectively. An injective mapping function $f : N_s \rightarrow N_t$ is a restrictedly embedding of s in t, if for all nodes $u, v \in N_s$ the following conditions hold.

1) f preserves labels, i.e., $label(f(u)) = label(u)$.
2) f preserves ancestors, i.e., $f(u) \in ancestors(f(v))$ if and only if $u \in ancestors(v)$.
3) f preserves least common ancestors, i.e., $label(lca(f(u), f(v))) = label(lca(u, v))$.

If there is a restrictedly embedding of s in t, we say that t restrictedly embeds s or t supports s; and s is restrictedly embedded in t, or simply s is a restrictedly embedded subtree of t. Please note that if f preserves labels and ancestors only, the above would be the definition of embedded subtrees. If f preserves labels and parents only, we would have the definition of induced subtrees. Refer to Fig. 1 again, where s_1 isn't qualified to be an induced subtree, but is a restrictedly embedded subtree, of t_3. On the other hand, s_2 isn't qualified to be a restrictedly embedded subtree, but is an embedded subtree, of t_2, since $a = lca(d, e)$ in t_2 is not equal to $b = lca(d, e)$ in s_2. One can also verify that s_2 is a restrictedly embedded subtree of t_3.

Definition 2 (Frequent subtrees): Let TS denote a set of data trees $TS = \{t_1, t_2, \ldots, t_m\}$ and s denote a subtree pattern. We say $Supp(s, t) = 1$ if s is restrictedly embedded in t. The support of s in the set TS is defined as $Supp(s, TS) = \sum_{t \in TS} Supp(s, t)/|TS|$. A subtree is frequent if its support is greater than or equal to a user-specified minimum support threshold, denoted by *minsup*. A set of all frequent subtrees of size k is denoted by FST_k.

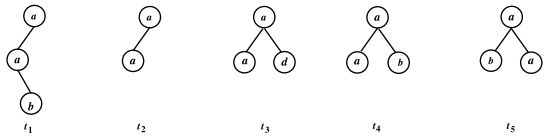

Fig. 2. Five unordered trees where t_4 and t_5 are two different ordered trees but represent the same unordered tree.

Definition 3 (FRESTM): Given a tree set *TS* and a user specified *minsup* value, the frequent restrictedly embedded subtree mining (FRESTM) problem aims to find the largest subtree set *S* such that any element in *S* must be a restrictedly embedded subtree of at least *minsup* $\times$ $|TS|$ trees in *TS*.

III. FRESTM Algorithm

A. Canonical Form of Tree Representation

As discussed before, an unordered tree may appear as different ordered trees, which are considered to be equivalent to each other. For example, in Fig. 2, t_4 and t_5 are two different ordered trees that actually represent the same unordered tree. Therefore, it is crucial for us to adopt a canonical form to uniquely code each unordered subtree and systematically differentiate all unordered subtrees, as well as to identify all equivalent mappings from different data trees to the same subtree. In this regard, we adopt the canonical form used by Nijssen and Kok [15].

Given a node v_i, its depth-label pair is denoted as $dl(v_i) = (depth(v_i), label(v_i))$, or simply $dl(i) = (d_i, l_i)$ when the context is clear. For two tree nodes i and j, we say $dl(i) < dl(j)$ if either (i) $d_i > d_j$ or (ii) $l_i < l_j$ if $d_i = d_j$. When $l_i = l_j$ and $d_i = d_j$, we say $dl(i) = dl(j)$. The first condition may sound anti-intuitive, but it actually reflects that the closer to the root a node is, the higher rank the node has and the heavier the node is. Given an ordered k-tree t, its depth-label sequence is denoted as $dls(t)$, which is a concatenation of the depth label pairs of all the nodes of the tree visited by the preorder left-to-right depth-first tree traversal: $dls(t) = dl(1), dl(2) \ldots, dl(k)$. Let t_1 and t_2 be two labeled ordered trees; t_1 is called a prefix subtree of t_2 if $dls(t_1)$ is a prefix of $dls(t_2)$. We say $dls(t_1) > dls(t_2)$, i.e., t_1 is heavier than t_2, if either t_1 is a prefix of t_2 or the two trees differ at the leftmost position i by having $(d_i, l_i)^{t_1} > (d_i, l_i)^{t_2}$, where the superscripts refer to the two trees respectively.

An unordered tree t is in its canonical form if no equivalent ordered tree t' exists with $dls(t') < dls(t)$, i.e., the canonical form of an unordered tree should result in the lightest *dls* among all of its equivalent ordered trees. Considering Fig. 2, the depth-label sequences of the five trees are $dls(t_1) = (0, a), (1, a), (2, b)$, $dls(t_2) = (0, a), (1, a)$, $dls(t_3) = (0, a), (1, a), (1, d)$, $dls(t_4) = (0, a), (1, a), (1, b)$, and $dls(t_5) = (0, a), (1, b), (1, a)$, respectively. We have the following observations: 1) since t_2 is a prefix tree of t_1, t_2 is heavier than t_1, i.e., $dls(t_2) > dls(t_1)$ and 2) t_4 and t_5 are all possible ordered trees of the same unordered tree and t_4 is lighter than t_5. Therefore, t_4 is in the canonical form of the unordered tree.

Since a tree can be recursively defined on its downward subtrees, when a tree t is in its canonical form, each of its downward subtrees must also be in its canonical form. Here, the downward subtree rooted at a node v, denoted by $st(v)$, is the induced subtree consisting of all the descendant nodes of the node v. As an example, s_3 is a downward subtree of t_1 in Fig. 1. Nijssen and Kok [15] have obtained the following lemma and observation regarding the canonical form.

Lemma 1 (Rightmost heaviest): A labeled ordered tree t is the canonical representation of an unordered tree iff t is rightheaviest, that is, for any node $v_1, v_2 \in V$, $(v_1, v_2) \in S$ implies $dls(st(v_1)) \leq dls(st(v_2))$.

Observation 1: Directly removing the last node of a canonicalized tree t will result in a residue tree still in its canonical form. Here "directly removing" means removing a node without further canonicalizing the resulting tree.

Therefore, if t is an unordered tree in its canonical form, then every downward subtree and every prefix of t is also automatically in its canonical form.

For the purpose of developing right-most joining based algorithms, we further investigate the canonical status of the residue tree after the second to the last node is removed. Here the second to the last node is counted according to the *dls* concatenation order, i.e., the preorder of the depth-first traversal. It turns out that depending on how the second to the last node is topologically related to the last node, directly removing the second to the last node from a canonicalized tree may or may not result in a residue tree still in its canonical form.

In the first situation, if the second to the last node is also a leaf node, then the resulting tree is automatically in its canonical form. This is described in the following lemma.

Lemma 2: Letting t be a tree in its canonical form with its second to the last node also being a leaf, the residue tree after the second to the last node is removed must still be in its canonical form.

Proof: First, consider the case where the second to the last node is the immediate left sibling of the last node. In this case it is trivial to see that the residue tree must be in its canonical form. Therefore, we focus on the nontrivial case where the second to the last node is not the immediate left sibling of the last node. We will prove that after the second to the last node is removed, all the influenced parts remain canonically stable. That is, the influenced parts are still in their canonical form. Let the last node, the second to the last node and the immediate left sibling node of the last node, respectively, be denoted by l, sl and ill. Further, let *lcm* denote the least common ancestor of l and sl, which is also the parent of l and ill. The removal of the second to the last node is an operation whose influence can be restricted to the local scopes that are marked as C and D, respectively in Fig. 3. A and B are also marked to show the adjacent structures of D and C respectively. Let C denote the downward subtree rooted at ill and C' denote the subtree after sl has been removed. Since sl is the last node of C, after it is removed, C' remains to be canonical according to the prefix rule. Notice that the presence of B is optional. When B does appear, we also need to show B and C' are canonically stable. Since $dls(C) \geq dls(B)$ and the prefix is heavier than the original tree, we know $dls(C') > dls(C) \geq dls(B)$, and hence C' and B are canonically stable (based on Lemma 1).

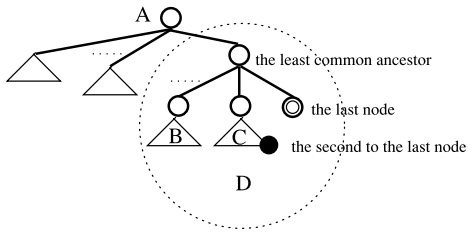

Fig. 3. C and D are the structures of a tree affected by removing the second to the last node of the tree. B is the immediate left sibling of C, and A is the complementary part of D.

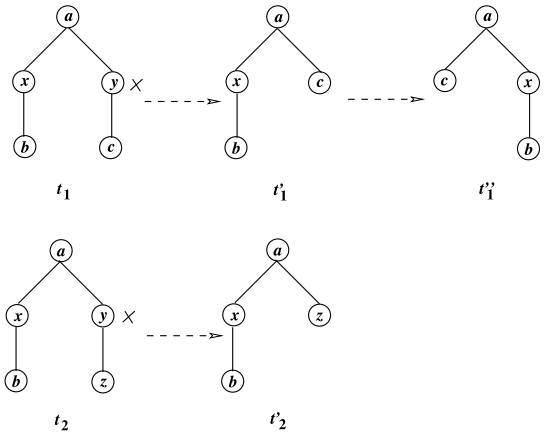

Fig. 4. t_1' is not in its canonical form, while t_2' is in its canonical form.

Now consider the subtree rooted at *lcm*. We know $dls(st(l)) \geq dls(C)$ and $depth(l) = depth(ill)$; therefore, $label(l) \geq label(ill)$. After *sl* is removed, the ordering between $dls(st(l))$ and $dls(st(ill))$ is still decided by the labels of the two nodes, i.e., l and ill; therefore $dls(st(l)) \geq dls(C')$. Let D' denote the subtree after the removal of *sl*. D' remains to be canonical. Furthermore, since $dl(l) > dl(sl)$, we know that after *sl* is removed, $dls(D')$ is even heavier than $dls(D)$. Therefore, after *sl* is removed, the whole tree remains to be rightmost heaviest (again based on Lemma 1) considering the ordering between D' and its immediate left sibling within the complementary A part, and is thus still canonical. Hence the proof is completed. ■

In the second situation, if the second to the last node is not a leaf, then the node must be the parent of the last node and have the last node as the sole child. For this case, we have the following observation.

Observation 2: If the second to the last node is the parent having the last node as the sole child, removal of the second to the last node may or may not result in a tree still in its canonical form.

Fig. 4 shows two possibilities.

1) After the second to the last node is removed from t_1, the residue tree $t_1^{'}$ is not in its canonical form. Instead, $t_1^{''}$ is in the canonical form.
2) After the second to the last node is removed from t_2, the residue tree $t_2^{'}$ is still in its canonical form.

As for whether or not directly appending a new node to a canonicalized tree will automatically lead to a canonicalized tree, we have the following observation on the canonical status after directly appending a new node to a canonicalized tree.

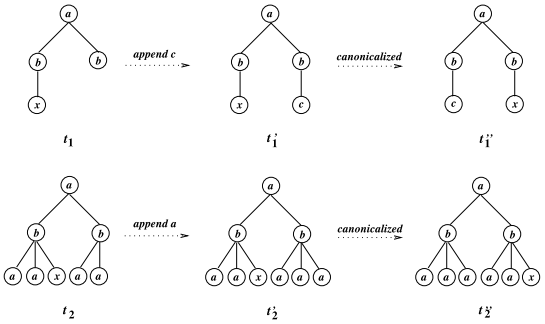

Fig. 5. Appending a new node as a child or as a sibling to the last node of a canonicalized subtree may result in a tree not in its canonical form.

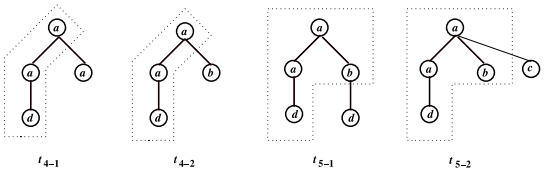

Fig. 6. t_{4-1} and t_{4-2} are in the same equivalence class, and t_{5-1} and t_{5-2} are in the same equivalence class, with the cores surrounded by dotted lines, respectively.

Observation 3: Appending a node to a canonicalized k-tree either as a child of the last node or as a sibling of the last node may or may not produce a canonical $(k+1)$-tree.

Fig. 5 shows the two situations where appending new nodes to originally canonicalized trees may lead to trees that are not in the canonical form. Please note that t_2' in Fig. 5 is not automatically canonicalized.

B. Equivalence Class

Two unordered k-trees are said to be in the same equivalence class, if their canonical forms share exactly the same $k-1$ *dls* prefix. Thus, any two members of an equivalence class differ only at the rightmost position. For example, Fig. 6 shows two equivalence classes, each containing two members. Specifically, t_{4-1} and t_{4-2} are in the same equivalence class, represented by $core_1$: $(0,a)(1,a)(2,d)$, and t_{5-1} and t_{5-2} are in another equivalence class, represented by $core_2$: $(0,a)(1,a)(2,d)(1,b)$. As we will see in the next subsection, an efficient joining expansion technique will be used to grow new patterns from trees in the same equivalence class.

C. Candidate Generation Via Rightmost Growth

Our candidate generation scheme consists of two types of rightmost growth operations (reminiscent of the rightmost growth schemes in [1], [21], and [25]): pairwise joining for horizontal growth and leg attachment for vertical growth.

1) Pairwise Joining: In order for two different k-subtrees t_1 and t_2 to be eligible for further joining, the two subtrees must be in the same equivalence class. Assume the *dls* of t_1 and t_2 are $(d_1^{t_1}, l_1^{t_1})(d_2^{t_1}, l_2^{t_1}) \ldots (d_{k-1}^{t_1}, l_{k-1}^{t_1})(d_k^{t_1}, l_k^{t_1})$ and $(d_1^{t_2}, l_1^{t_2})(d_2^{t_2}, l_2^{t_2}) \ldots (d_{k-1}^{t_2}, l_{k-1}^{t_2})(d_k^{t_2}, l_k^{t_2})$, respectively, where a superscript refers to a tree and a subscript refers to a node in

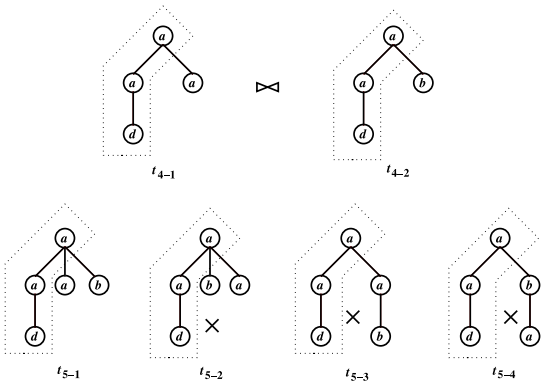

Fig. 7. Two different k-subtrees in the same equivalence class with the same topology can be joined to produce one $(k+1)$-subtree. Here, two 4-subtrees t_{4-1} and t_{4-2} in the same equivalence class are joined to obtain one candidate 5-subtree t_{5-1}.

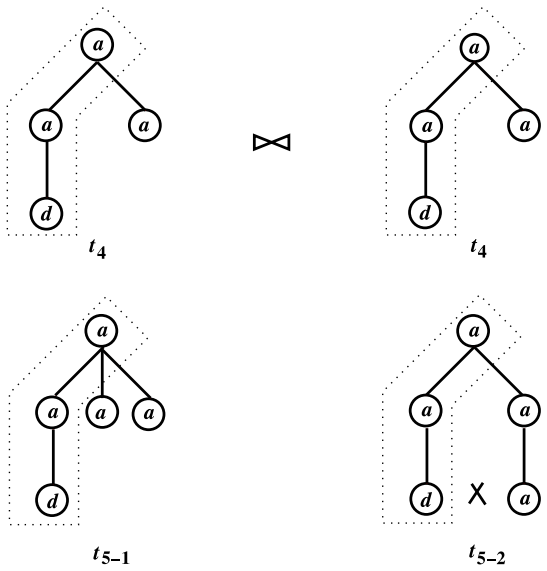

Fig. 8. k-subtree can be joined with itself to produce one $(k+1)$-subtree. Here, a 4-subtree t_4 is joined with itself to obtain one candidate 5-subtree t_{5-1}.

a tree. When two k-subtrees are in the same equivalence class, we know they must share the same $k-1$ prefix tree. Therefore we have $d_i^{t_1} = d_i^{t_2}$ and $l_i^{t_1} = l_i^{t_2}$ where $1 \le i \le k-1$. However, the topologies of the two k-subtrees themselves can still differ from each other. Depending on whether the topologies of the two k-subtrees are the same or not, the joining operations can be formally classified into two cases which are elaborated below.

1) *Case 1: Same Topology:* Since the topologies of the two joined trees are the same, the two rightmost leaves must be at the same depth, i.e., $d_k^{t_1} = d_k^{t_2}$. Without lose of generality, we assume $dls(t_1) < dls(t_2)$, which means $l_k^{t_1} < l_k^{t_2}$. This decides the order in which the two rightmost leaves are glued back to the same core shared by the two trees for generating the $(k+1)$-subtree. The resulting tree is $(d_1, l_1)(d_2, l_2) \ldots (d_{k-1}, l_{k-1})(d_k, l_k^{t_1})(d_k, l_k^{t_2})$. Please note that the $k-1$ prefix is exactly the same; therefore, in the first $k-1$ depth-label pairs, no superscripts are attached to differentiate between the two trees. Also, the depths of the last nodes of the two trees are the same due to the same topology. As a result, we only need to use the ids of the two k-subtrees as the superscripts to identify the last two labels of the resulting tree.

   Fig. 7 shows that one candidate tree, i.e., the first one t_{5-1}, can be generated for this case. The second one is ignored because it will not be in the canonical form. The third and the fourth ones cannot be obtained through joining but instead are created due to the leg attachment rule which will be introduced later.

   Since the same label is allowed to appear multiple times in a tree, and every tree always has the same topology as itself, self-joining should also be considered. In fact, self-joining is perfectly supported here, as illustrated in Fig. 8. When $t_1 = t_2$, the result simply is: $(d_1, l_1)(d_2, l_2) \ldots (d_{k-1}, l_{k-1})(d_k, l_k)(d_k, l_k)$.

2) *Case 2: Different Topologies:* In this case, the topologies of the two joined trees are different. Since their $k-1$ prefix subtrees are the same, we must have $d_k^{t_1} \neq d_k^{t_2}$ in order for the two trees to present different topologies. In the newly generated candidate tree, the last leaves from the two joined trees will not have any parent-child or sibling relationship; instead, the last leaf (also the last node from one joined tree) must be a xth cousin once removed of the second to the last leaf (the last node from the other joined tree), where $x \ge 1$. We compare the depths of the two last leaves of the two joined trees. If $d_k^{t_1} < d_k^{t_2}$, then the candidate tree will be $(d_1, l_1)(d_2, l_2) \ldots (d_{k-1}, l_{k-1})(d_k^{t_2}, l_k^{t_2})(d_k^{t_1}, l_k^{t_1})$, where t_1 plays the umbrella role; otherwise, the candidate tree is $(d_1, l_1)(d_2, l_2) \ldots (d_{k-1}, l_{k-1})(d_k^{t_1}, l_k^{t_1})(d_k^{t_2}, l_k^{t_2})$, where t_2 plays the umbrella role. Fig. 9 illustrates how to join two trees in the same equivalence class that have different topologies.

2) Leg Attachment: Notice that the above two joining cases only expand trees horizontally. In order to grow a tree vertically, for each k-subtree, we attach to its rightmost leaf any frequent single label. Consider Fig. 7 again; t_{5-3} and t_{5-4} are crossed out because they cannot be obtained through joining. However, they can be created through the leg attachment rule. In fact, all frequent labels besides "a" and "b" can be attached to the last node of t_{4-1} or t_{4-2} to grow more trees vertically. As another example, in Fig. 8, t_{5-2} can be generated through the leg attachment rule too.

D. Pattern Discarding

According to Observation 3 given in Section III-A, we know a subtree expanded using either the pairwise joining algorithm or the leg attachment rule may or may not conform to the canonical form definition, i.e., the generated trees may not be automatically canonicalized (see Fig. 5 for two examples). Therefore, for a newly generated $(k+1)$-subtree, we need to test whether or not it is in its canonical form. If the tree is already in its canonical form, it can be kept as a candidate for the subsequent embedding detection; otherwise, the subtree can be safely discarded.

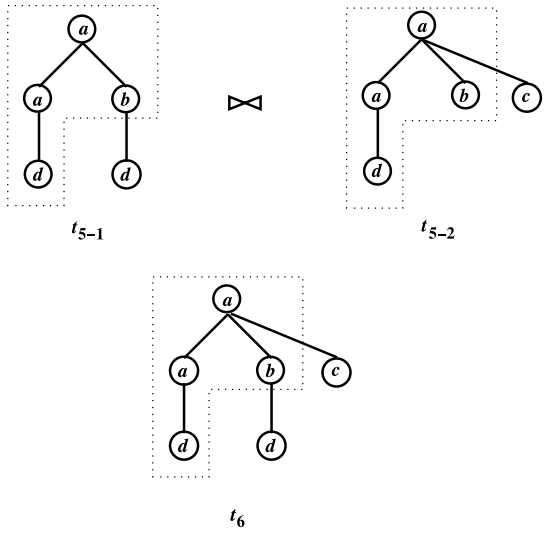

Fig. 9. Two k-subtrees with different topologies may produce one possible $(k+1)$-subtree, which may or may not be in its canonical form. Here, two 5-subtrees t_{5-1} and t_{5-2} with different topologies are joined to obtain one 6-subtree t_6.

The canonical test can be conducted efficiently because the candidate $(k+1)$-subtree t' is expanded from an already canonicalized k-subtree t. Since t is already canonicalized, the majority of nodes of t' must have been in order. After a new node is appended at the end of the tree, we only need to trace through the rightmost path starting from the last node all the way up to the root to compare the *dls* between the affected subtrees and their immediate left sibling subtrees. For each node on the rightmost path, if the *dls* of the rightmost subtree is heavier than the *dls* of its immediate left sibling subtree, then the new tree is automatically canonicalized. Otherwise, the tree is not canonicalized. The idea here is similar to the canonical test procedure described in [22], which adopts a different canonical form. It is worth mentioning that instead of canonicalizing them, these noncanonical trees can be just safely discarded. This is because every discarded pattern will still be generated in its canonical form from other equivalence classes. Consider Fig. 5 again as an example; $t_1^{'}$ and $t_2^{'}$ will be discarded. However, they will still be generated in their canonical forms $t_1^{''}$ and $t_2^{''}$ from other equivalence classes. For example, $t_2^{''}$ will be generated in its canonical form shown as t_9 in Fig. 10.

E. Tree Embedding Detection

After an expanded subtree has passed the canonical test discussed above, it should be compared against every data tree to see whether or not it is restrictedly embedded in each data tree. Our detection algorithm is developed from the restricted edit distance work done by Shasha *et al*. [16], Zhang [23], and Zhang *et al*. [24].

The edit distance between two trees t_1 and t_2 is defined as the minimum cost edit operations sequence S that transforms t_1 to t_2 [23]. The edit operations (substitution, deletion, and insertion as defined in [23]) give rise to a bijective mapping specifying how the edit operations apply to each node in the two unordered trees. Furthermore, the edit distance with unit

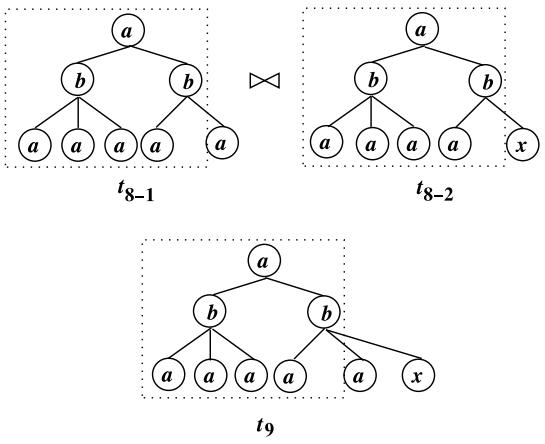

Fig. 10. Example showing a joining case that leads to a canonical form.

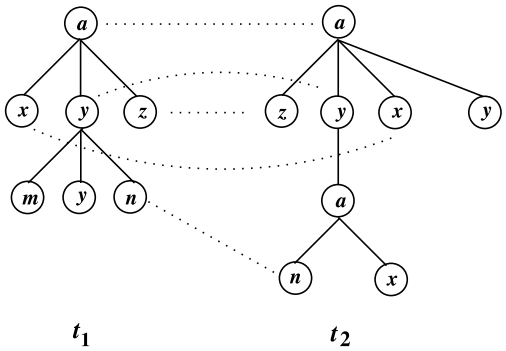

Fig. 11. Pictorial description of the bijective mapping between t_1 and t_2 where t_1 is transformed to t_2. A dotted line between a node u of t_1 and a node v of t_2 indicates that u should be changed to v if $u \neq v$, or that u remains unchanged if $u = v$. The nodes of t_1 not touched by dotted lines are to be deleted and the nodes of t_2 not touched by dotted lines are to be inserted.

cost (i.e., the cost of each edit operation is 1) between trees t_1 and t_2 can be calculated by taking the summation of the number of insertion operations, the number of deletion operations, and the number of substitution operations in the sequence S that transforms t_1 to t_2.

For example, Fig. 11 shows that t_1 can be transformed to t_2 through 0 substitution (of those nodes touched by the dotted lines with different labels), 2 deletions (of those nodes in t_1 that are not touched by the dotted lines), and 3 insertions (of those nodes in t_2 that are not touched by the dotted lines). Under the assumption that all edit operations have unit cost, it is easy to see that the edit distance between t_1 and t_2 is $0+2+3=5$.

Now let us look at how restricted edit distance can help on restrictedly embedding detection. We use the following lemma to determine whether or not a pattern tree s is restrictedly embedded in a data tree t.

Lemma 3 (Restrictedly embedding detection): The restricted edit distance between s and t is equal to the size difference between s and t if and only if s is restrictedly embedded in t.

F. Support Counting

In order to count the support, i.e., calculate the occurrence number, of a candidate k-subtree pattern in the whole data set, intuitively, we should run the restrictedly embedding detection

```
FRESTM(TS, minsup)
    ▷ TS is the set of trees.
    ▷ minsup is the minimum support threshold.
    ▷ The output is a set of frequent restrictedly embedded subtrees discovered from TS.
 1  inverted_index ← nodes appearing in all data trees and their occurrence numbers;
 2  FST_1 ← frequent labels from inverted_index;
 3  FST_2 ← frequent 2-subtrees generated from frequent labels in FST_1;
 4  FST ← FST_1 ∪ FST_2;
 5  k ← 2;
 6  while (|FST_k| ≥ 1)
 7      do
 8          FST_{k+1} ← GENERATE_SUBTREES(k, FST_k, FST_1, minsup * |TS|);
 9          FST ← FST ∪ FST_{k+1};
10          k ← k + 1;
11  return FST
```

Fig. 12. Algorithm for discovering all frequent subtrees.

subroutine on the candidate pattern tree against all data trees one by one. If the occurrence number falls below *minocc*, which is defined as *minsup* $*$ |*TS*| where *TS* is the set of data trees, the candidate pattern tree can be discarded; otherwise, the candidate must be frequent. All the frequent *k*-subtrees will then be used to generate larger candidates for the subsequent pattern growth process.

The above process can be further optimized by taking advantage of two Apriori-based properties. The first property says that a $(k+1)$-tree cannot be frequent if any of its *k*-subtrees already isn't frequent, which is a partial reason why the use of the equivalence class is efficient in our joining method. The second property says, from the supporting data tree viewpoint, the supporting trees of the $(k+1)$-subtree must be in the intersection set of the supporting trees of all its *k*-subtrees. This property actually suggests an occurrence list based pruning technique where the occurrence list of a subtree *s* is the list of data trees that support *s* (i.e., the list of supporting trees of *s*). Specifically, given two *k*-subtrees that are in the same equivalence class, we first find the intersection list of the occurrence lists of the two trees and then compare the cardinality of the intersection list with *minocc*. If the cardinality is already less than *minocc*, we do not even need to join the two subtrees. Otherwise, we join them to obtain a $(k+1)$-subtree, which then needs to pass the canonical test in order to be treated as a candidate tree to go through the support counting phase.

As for the leg attachment rule, a similar pruning technique applies. When we try to attach a frequent single label (as a node) to a *k*-subtree, we first find the intersection list of the occurrence lists of both the *k*-subtree and the frequent single label (node). If the cardinality of the intersection list is already smaller than *minocc*, we do not even need to perform the attachment. Otherwise, we expand the tree by using the leg attachment rule.

G. Algorithmic Framework of FRESTM

The proposed algorithm is an Apriori-based data mining method, which progressively enumerates all candidate subtrees from a given set of unordered trees, level by level, using the rightmost expansion methods. Fig. 12 summarizes the algorithm.

In the initialization phase, frequent 1-subtrees and 2-subtrees are discovered first. To enumerate all frequent 1-subtrees, i.e., frequent single labels, we traverse every node of every tree to create an inverted index structure for each unique label appearing in the trees. Specifically, for each unique label, we maintain a list of IDs of supporting trees, denoted by *STL*, in which the label appears. By comparing its |*STL*| with the given *minsup*, we can decide whether the label is frequent or not. After all frequent 1-subtrees have been discovered, we can use the leg attachment rule to attach one node to another to form a 2-subtree having a parent-child pair. For each of the 2-subtrees, we perform the embedding detection test and support counting to find out whether the pattern is frequent or not. All 2-subtrees are automatically canonicalized, because there is only one topology for any 2-subtree, where one node is the root and the other node is the child of the root. Notice that different nodes may have the same label; thus, all-to-all attachments are used here to avoid missing any candidate 2-subtrees.

Starting from frequent 2-subtrees, during each of the subsequent iterations, the algorithm calls the subroutine GENERATE_SUBTREES to grow frequent subtrees level by level through pairwise joining and leg attachment methods. Notice that when $|FST_k|$ reaches zero, no more frequent $(k+1)$-subtrees can be generated and hence the discovering process terminates. Please notice that $|FST_k|$ can be as small as one to allow self-joining and leg attachment operations.

The GENERATE_SUBTREES module in Fig. 13 is the essential part of the algorithm. This module is comprised of the following functions: 1) equivalence class preparation; 2) candidate generation (expansion); and 3) candidate embedding detection, all of which have been discussed in the previous subsections. The first function divides the frequent *k*-subtrees into different equivalence classes. The second and third functions are executed sequentially on each newly generated $(k+1)$-subtree; the candidate generated from the expansion function will be passed to the candidate embedding detection function. The algorithm in Fig. 13 also shows how the intersection list of two occurrence lists (i.e., supporting

```
GENERATE_SUBTREES(k, FST_k, FST_1, minocc)
    ▷ k is the size of frequent subtrees based on which (k + 1)-subtrees are generated.
    ▷ FST_k is the set of frequent subtrees of size k.
    ▷ FST_1 is the set of frequent labels.
    ▷ minocc is the minimum occurrence number.
    ▷ EC_k is a local variable representing the set of equivalence classes of frequent k-subtrees.
    ▷ The output is a set of frequent restrictedly embedded (k + 1)-subtrees.
 1  FST_{k+1} ← ∅, EC_k ← ∅;
 2  for each t ∈ FST_k
 3       do ▷ prepare equivalence classes for all k-subtrees
 4          core ← Extract_k-1_Prefix(t);
 5          if core is not in EC_k
 6            then
 7                 add core as a new equivalence class to EC_k;
 8          register t to the equivalence class of core;
 9  for each ec ∈ EC_k
10       do
11          for each pair of k-subtrees x, y ∈ ec ▷ self-join as a special case when x = y
12               do
13                  ISTL ← Intersection(STL(x), STL(y)); ▷ STL refers to the supporting trees list
14                  if |ISTL| ≥ minocc ▷ pruned otherwise
15                    then ▷ pairwise joining
16                         c_{k+1} ← PairwiseJoin(x, y);
17                         occurrence ← 0;
18                         for each tree t ∈ ISTL
19                              do ▷ embedding detection
20                                 if (Restrictedly_Embedding_Detection(c_{k+1}, t))
21                                   then
22                                        occurrence ← occurrence + 1;
23                         if (occurrence ≥ minocc) ▷ support check
24                           then
25                                FST_{k+1} ← FST_{k+1} ∪ {c_{k+1}};
26          for each k-subtree x ∈ ec and for each frequent label y ∈ FST_1
27               do
28                  ISTL ← Intersection(STL(x), STL(y));
29                  if |ISTL| ≥ minocc ▷ pruned otherwise
30                    then ▷ leg attachment
31                         c_{k+1} ← LegAttach(x, y);
32                         occurrence ← 0;
33                         for each tree t ∈ ISTL
34                              do ▷ embedding detection
35                                 if (Restrictedly_Embedding_Detection(c_{k+1}, t))
36                                   then
37                                        occurrence ← occurrence + 1;
38                         if (occurrence ≥ minocc) ▷ support check
39                           then
40                                FST_{k+1} ← FST_{k+1} ∪ {c_{k+1}};
41  return FST_{k+1}
```

Fig. 13. Algorithm for generating all frequent $(k+1)$-subtrees from frequent k-subtrees.

trees lists) should be used to skip unnecessary expansions and how support counting is done for a candidate pattern.

H. Correctness

The correctness of embedding detection relies on the restricted edit distance algorithm, which has been well studied in [16], [18], and [24]. Therefore we will mainly consider the correctness of candidate generation, i.e., the algorithm can systematically discover all candidate subtrees without missing any, through the pairwise joining and the leg attachment operations.

Theorem 1: The FRESTM algorithm discovers all frequent restrictedly embedded subtrees.

Proof: We prove the theorem by induction.

Base Case 1: All 1-subtrees can be discovered correctly without missing any (see Section III-G).

Base Case 2: All 2-subtrees can be generated correctly (see Section III-G).

Now, assume $n = k$ holds (induction hypothesis), i.e., all frequent k-subtrees can be generated correctly. We want to show $n = k + 1$ holds, i.e., all frequent $(k + 1)$-subtrees can be generated from all frequent k-subtrees correctly.

Consider a frequent $(k + 1)$-subtree, t_{k+1}, in its canonical form (a tree can always be canonicalized) in which the last node and the second to the last node are denoted as l and sl respectively. While it is trivial to see that l must be the last leaf on the rightmost path, sl may be found in several different locations with respect to l. Depending on the topological relationship between sl and l, there are three possible types (Fig. 14).

1) *Type 1:* The last two nodes, l and sl, have a parent-child relationship. Let us remove the last node; we get a k-subtree, t_k, and the parent of the removed last node must be the last, also the rightmost, node of t_k. Based on Lemma 1, we know t_k must be in its canonical form. Now we know t_k must be frequent based on the

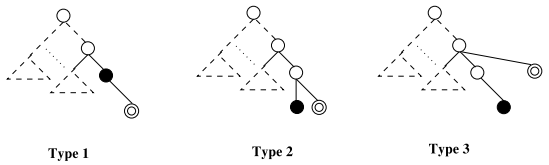

Fig. 14. Three types of relationships between the last two nodes of a tree where the last node of the tree is double circled and the second to the last node is filled with solid black.

Apriori principle. Furthermore based on the induction hypothesis, we know all frequent k-subtrees must have been discovered. We know for each subtree, the leg attachment rule always applies: attach every possible frequent label to the rightmost leaf of the tree. Therefore, tree t_{k+1}, which is generated from t_k, cannot be missed.

2) *Type 2:* l and sl are siblings. In this case, sl must be the immediate left sibling of l. Let us remove the last node and the second to the last node respectively to get t'_k and t''_k. According to Observation 1 and Lemma 2, we know t'_k and t''_k must be in their canonical forms respectively. Based on the induction hypothesis, we know all frequent k-subtrees must have been discovered. Therefore t'_k and t''_k must have been discovered and in the same equivalence class. According to our pairwise joining algorithm Case 1 (see Section III-C1), we know t'_k and t''_k will be joined to generate t_{k+1}.
3) *Type 3:* l and sl are neither siblings nor having a parent-child relationship. In this case, it is easy to see that the parent of l must be a nonparent ancestor of sl. We also know a tree can always be canonicalized. The basic idea is the same as Type 2, except that the pairwise joining algorithm Case 2 applies here (see Section III-C1).

Thus, we conclude that any frequent $(k+1)$-subtree, t_{k+1}, must be generated from frequent k-subtrees. Hence the proof for the theorem is completed. ■

I. Time Complexity

From [23], the calculation of the edit distance between a k-subtree t_k and a data tree t takes $|t_k| * |t| * max\{deg(t_k), deg(t)\} * log_2(max\{deg(t_k), deg(t)\})$ time, where $|t_k|$ is k, and $deg(t_k)$ and $deg(t)$ denote the maximum degrees of trees t_k and t, respectively. This is also the time complexity for detecting restrictedly embedding of t_k in t. To calculate the intersection list of supporting trees lists (i.e., occurrence lists) takes $O(|TS|)$ time in the worst case, where TS denotes the set of data trees. Generating each candidate tree involves the pairwise joining operation or the leg attachment operation, which in the worst case takes $O(|t|)$ time. After a k-subtree, t_k, is generated, it needs to pass the canonical form test, which takes $O(|k|^2)$ time. As a result, to discover a frequent k-subtree takes $O(k^2 + k + |TS| + |TS| * k * |t| * max\{deg(t_k), deg(t)\} * log_2(max\{deg(t_k), deg(t)\}))$ time. To get all the frequent subtrees for one level, it takes, in the worst case, $O(|FST_k| * |FST_1|(k^2 + k + |TS| + |TS| * k * |t| * max\{deg(t_k), deg(t)\} * log_2(max\{deg(t_k), deg(t)\})))$ time. To get all the frequent subtrees for all levels, it takes, in the worst case, $O(\sum_{k=1}^{k=M} |FST_k| * |FST_1|(k^2 + k + |TS| + |TS| * k * |t| * max\{deg(t_k), deg(t)\} * log_2(max\{deg(t_k), deg(t)\})))$ time, where M is the size of the largest frequent subtree. Please note that this gives a very pessimistic upper bound time complexity analysis where the worst cases are considered for all steps. The algorithm, however, should run much faster on most real-world data sets due to Apriori properties.

J. Space Complexity

The FRESTM method integrates multiple algorithms of tree pattern growth, canonical tree test, tree distance based pattern verification, tree distance calculation, etc. However, at any single moment, each of these operations involves only a couple of trees, for which memory can be dynamically allocated and released at the run time. For example, when two trees are joined to produce a new candidate tree, only three trees coexist in the memory. When the edit distance between a pattern tree and a data tree is calculated, only two trees are involved. In our implementation, to trade space for speed, we maintain an occurrence list for each frequent subtree. Therefore, the space complexity is $O(|FST|(|t| + |TS|))$, where $|FST|$ is the number of frequent subtrees, $|t|$ is the average size of trees, and $|TS|$ is the total number of data trees. Note that this is a pseudo polynomial space complexity, because the number of frequent subtrees is data dependent and the support for each frequent subtree varies [25].

IV. Experiments and Results

A. Performance Evaluation on Synthetic Data

The FRESTM method is implemented in C++. We have conducted a series of experiments to evaluate the performance of FRESTM, and to compare it with a closely related method using both synthetic and real-world data. All the experiments were carried out on a MacBook with a 2.4 GHz Intel Core 2 Duo processor and 4 GB 1067 MHz DDR3 RAM, running OSX 10.8.3 (12D78) system.

The synthetic datasets were created by using the master tree guided generator given in [21]. This tree generator has five parameters: 1) the number of node labels (N); 2) the size of the master tree (M); 3) the maximum fan-out of the master tree (F); 4) the maximum depth of the master tree (D); and (5) the total number of trees in the dataset (T).

In the first experiment, we created a dataset, called D10, by using the following default values for the parameters of the tree generator: N = 100, M = 10 000, D = 10, F = 10, and T = 10 000. Then from D10 we obtained four increasingly larger datasets with 2000, 4000, 6000, and 8000 trees, respectively. These four datasets, together with D10, were collectively referred to as D10s. We ran FRESTM on D10s ten times, with the *minsup* value fixed at 5%, and took the average. Fig. 15 shows the running times of FRESTM on the datasets. It can be seen from the figure that the time needed by FRESTM scales up linearly with respect to the dataset size, which is consistent with the time complexity analysis in Section III-I.

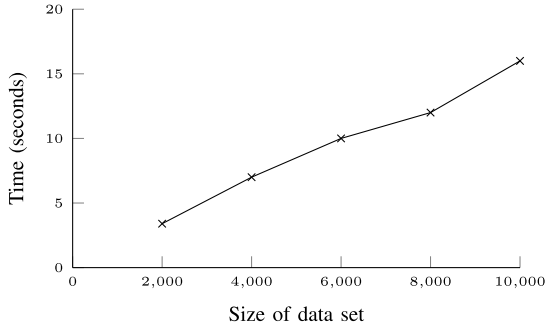

Fig. 15. Effect of dataset size on the running time of FRESTM.

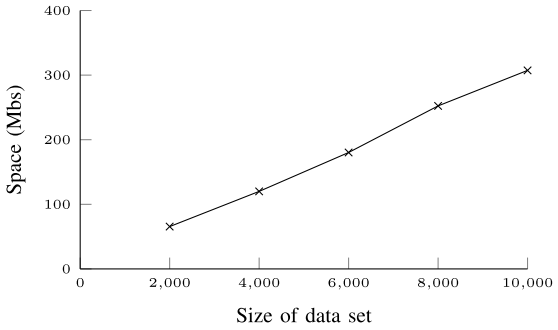

Fig. 16. Effect of dataset size on the space used by FRESTM.

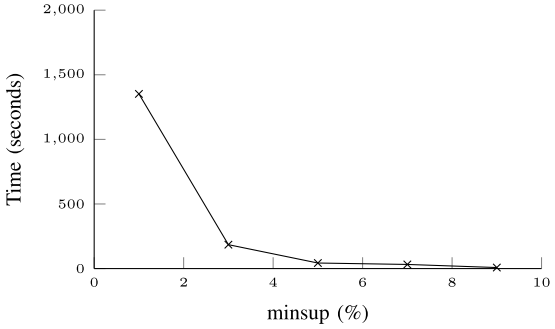

Fig. 17. Effect of *minsup* on the running time of FRESTM.

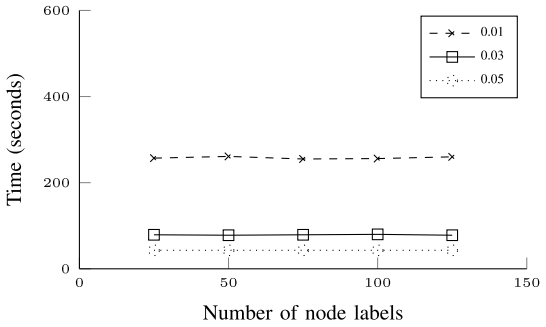

Fig. 18. Effect of the number of node labels on the running time of FRESTM.

This happens because the more trees a dataset has, the more time is needed for calculating occurrence numbers of candidate subtrees in the dataset. In this experiment, the numbers of patterns discovered from the five datasets were relatively stable, fluctuating between 43 and 47. These numbers are consistent with the characteristics of the synthetic trees, which are generated from the same master tree to yield the same amount and the same types of patterns.

Fig. 16 shows the space needed when running FRESTM on D10s, where the space refers to the heap peak memory (measured in Mb) used by the program. It can be seen from the figure that the memory used increases proportionally to the dataset size, which is consistent with the space complexity analysis in Section III-J. Here, the average size of trees and the number of frequent patterns remained constant for the five datasets generated from the same master tree.

To study how the *minsup* value affects the performance of FRESTM, we ran the algorithm on D10 ten times by gradually increasing the *minsup* value, and took the average. Fig. 17 summarizes the results. With a small *minsup* value, many long patterns with different labels were discovered by our algorithm. As a consequence, much time was spent in finding these long patterns. On the other hand, with a large *minsup* value, the running time of our algorithm decreased as few patterns qualified to be solutions.

Fig. 18 shows the impact of changing the number of node labels, N, on the running time of FRESTM. The other parameter values were fixed as follows: T = 5000, F = 10, D = 10, and M = 10 000. It can be seen from Fig. 18 that as N increases from 25 to 125, the running time of FRESTM does not change much. This happens because when N becomes large, the number of patterns increases but the support of each pattern decreases. With a fixed *minsup* value (e.g., *minsup* $= 0.01$), the number of frequent patterns is almost the same for the different N values, and as a consequence the running time remains constant. This result is consistent with the time complexity analysis in Section III-I where the time of FRESTM is shown to be irrelevant to the number of node labels.

In the next experiment, we investigated how changing the size of data trees affects the performance of our algorithm. The master tree guided generator given in [21] is not applicable here, because the parameter M controls the size of the master tree, rather than the size of data trees. Thus, we implemented another tree generator capable of producing trees with different sizes. This program generates trees in a bottom up manner by first creating leaf nodes, then randomly selecting a random number of nodes to merge into siblings sharing the same parent. This process continues until a single root node is obtained. Table I shows how changing the size of data trees affects the running time of FRESTM, where the minimum degree of a tree, the maximum degree of a tree, the number of node labels and the dataset size were fixed at 1, 6, 10, and 1000, respectively. As the tree size becomes larger, more patterns are found, and consequently more time is spent in the mining process. In general, with a fixed number of node labels, the more nodes a tree has, the higher chance each label has in appearing in the tree, and hence more frequently these labels appear in the whole dataset. This means more single node trees appear frequently in the dataset, and consequently, more frequent k-subtrees, $k \geq 2$, exist. Thus, more time is spent on discovering the frequent subtrees.

TABLE I
EFFECT OF TREE SIZE ON THE RUNNING TIME OF FRESTM WITH $minsup = 0.1$

Average tree size	Maximum tree size	Maximum pattern size	Number of patterns	Time (sec.)
6.652	19	2	12	0.477
10.245	24	2	110	3.105
13.751	29	2	110	12.801
17.385	36	3	273	26.156

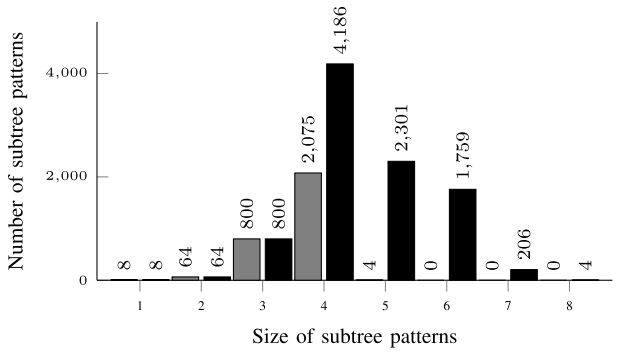

Fig. 19. Distribution of frequent patterns of different sizes with *minsup* = 1%. FRESTM and Sleuth are represented by gray and black bars, respectively.

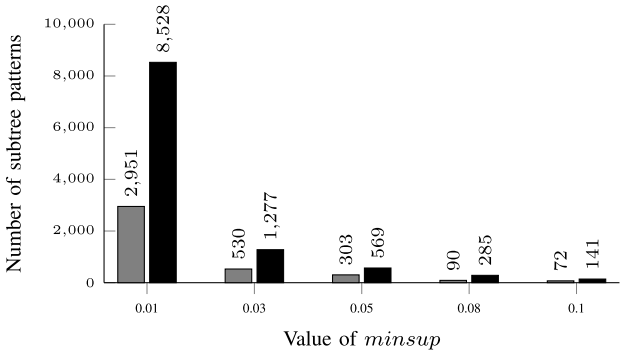

Fig. 20. Effect of *minsup* on the number of frequent patterns. FRESTM and Sleuth are represented by gray and black bars, respectively.

TABLE II
RESULTS OF RUNNING FRESTM ON THE DTDS DATASET WITH DIFFERENT *minsup* VALUES

$minsup$	Maximum pattern size	Number of patterns	Time (sec.)
0.10	2	11	0.023
0.09	2	17	0.056
0.08	3	21	0.083
0.07	10	427	11.754
0.06	13	2072	93.08

TABLE III
RESULTS OF RUNNING FRESTM ON THE CSLOGS DATASET WITH DIFFERENT *minsup* VALUES

$minsup$	Maximum pattern size	Number of patterns	Time (sec.)
0.10	1	2	0.292
0.05	1	6	0.458
0.04	1	12	0.725
0.03	3	33	2.689
0.02	5	93	9.962
0.01	6	293	25.49

B. Comparison with Sleuth

As surveyed in Section I, many tree mining algorithms have been developed. In contrast to the existing algorithms, the proposed method, FRESTM, aims to detect a novel type of patterns, namely restrictedly embedded subtrees, in unordered trees. The work most closely related to ours is the Sleuth algorithm [21], which is capable of detecting embedded subtrees in unordered trees. Thus, our work imposes a constraint on the patterns found by Sleuth, requiring the least common ancestor relationship to be preserved in the patterns. The experimental results, obtained from both the master tree guided generator [21] and our own tree generator, confirmed that the patterns found by Sleuth form a superset of the patterns found by FRESTM.

For example, Fig. 19 shows the numbers of frequent patterns detected by Sleuth and FRESTM, respectively, with respect to different sizes of patterns in the trees produced by our tree generator. Fig. 20 shows the numbers of frequent patterns detected by the two algorithms for varying *minsup* values. It can be seen from the figures that Sleuth detects much more patterns than FRESTM, making it more difficult to perform further (manual) checking of the relevance and meaning of the patterns. Fig. 19 indicates that the number of patterns increases as the size of patterns changes from 1 to 3, peaks when the size is 4, and then decreases. This is consistent with the distribution of frequent subtree patterns, where most patterns have the median size [21], [25]. Most time is spent in mining these patterns of the median size.

C. Experiment Results on Real-World Data

In this group of experiments, we ran our algorithm on three real-world datasets: a set of 210 XML DTDs [11], a set of 10 000 CSLOGs [21], and a set of prerequisite trees. Table II summarizes the results of running FRESTM on the XML DTDs set with different *minsup* values, showing the size of the largest frequent pattern, the number of frequent patterns, and the execution time. When *minsup* was 0.08, the largest pattern discovered by our algorithm was AUTHOR(ADDRESS, NAME). When *minsup* was reduced to 0.07, one of the largest patterns was PATIENT[ADDRESS, GENDER, ID, NAME, PHONE, SERVICES (DATE, PRODUCT, SERVICE)]. When *minsup* was further reduced to 0.06, one of the largest patterns was PATIENTADDRESS, GENDER, ID, NAME, PHONE, SERVICES [DATE, PRODUCT, SERVICE (NAME, PRICE, TIME)].

Table III summarizes the results of running FRESTM on the CSLOGs dataset [21]. This dataset contains the web logs of 59 691 user browsing access patterns on 13 361 different web pages, collected over a month by the Computer Science Department at Rensselaer Polytechnic Institute. On average, a tree in the *CSLOGs* dataset has 12.94 nodes, and the largest tree has 428 nodes. When *minsup* was set to 0.01, the largest pattern discovered by our algorithm was 1155(2603(4768(5996), 4769, 4770)) appearing in 116 trees. When *minsup* was 0.02, the largest pattern found by the

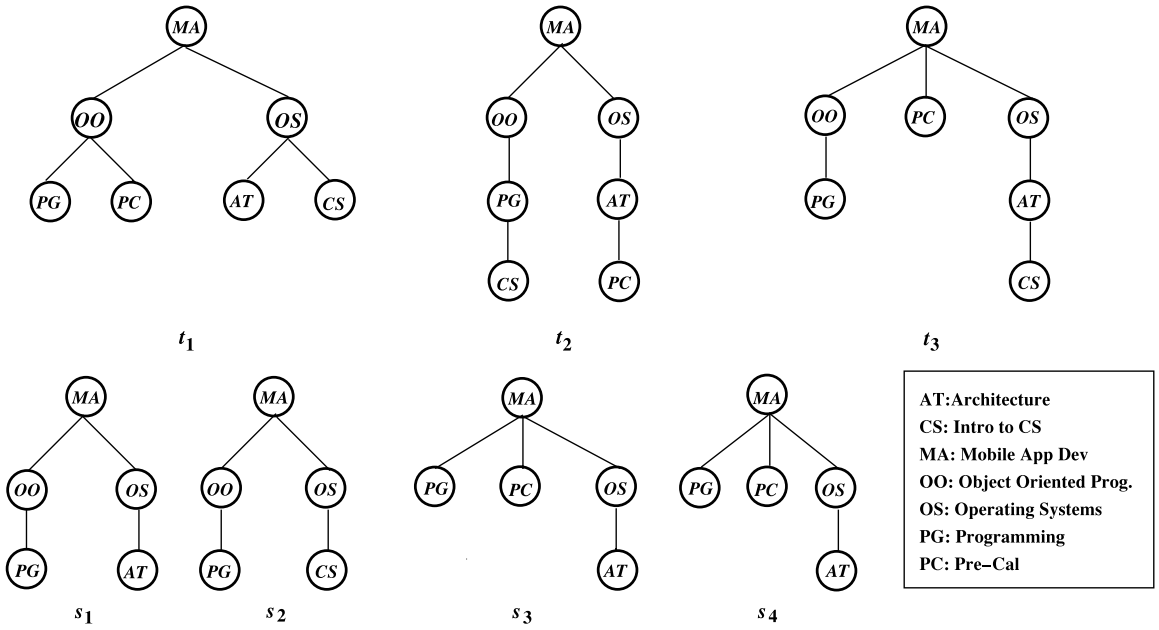

Fig. 21. Three prerequisite trees concerning the Mobile App Development course and the subtrees of size five mined from the prerequisite trees.

algorithm was 2603(4768(5996), 4769, 4770) appearing in 202 trees.

In the last experiment, we collected three prerequisite trees used in curriculum analysis. The nodes in the trees range from 13 to 15, where each node represents a different course. The trees describe the direct and indirect prerequisites of the computer science capstone course offered by three different computer science programs. With *minsup* fixed at 0.7 in the hope to identify consensus patterns from at least two prerequisite trees, Sleuth detected 423 frequent embedded subtrees, which is a rather challenging number for further perusal. On the other hand, FRESTM detected 76 frequent restrictedly embedded subtrees, which are much more manageable for further analysis.

To illustrate how the two algorithms differ, consider the three partial prerequisite trees related to the mobile app development (MA) course shown in the top row of Fig. 21. FRESTM detected 26 patterns and Sleuth detected 32 patterns in the three partial trees in addition to the seven single node patterns and nine 2-subtree patterns found by both algorithms. The bottom row of Fig. 21 lists the subtrees of size 5 found by Sleuth. Among the subtrees, s_1 and s_2 were also detected by FRESTM, whereas s_3 and s_4 were only reported by Sleuth. We can see from Fig. 21 that s_3 and s_4 are embedded subtrees but are misleading in this application. In s_3 and s_4, the fact that a student needs to take the OO course after taking the PG and PC courses is missing. On the other hand, with the constraint that the least common ancestor must be preserved in restrictedly embedded subtrees, FRESTM is able to filter out the misleading patterns. As in the results for synthetic data, the patterns found by Sleuth formed a superset of the patterns found by FRESTM on all the three real-world datasets. The large number of patterns detected by Sleuth, some of which may be misleading as illustrated in the above example, can pose a challenge for domain experts to make further investigation. By contrast, FRESTM is able to discover fewer and yet more meaningful patterns.

V. Conclusion

In this paper, we formalize a restrictedly embedded subtree mining problem, which has potential applications in many domains where data can be naturally represented as rooted labeled unordered trees. We study the properties of the canonical form of unordered trees and propose novel rightmost tree expansion techniques that can systematically, correctly, and efficiently generate all candidate subtrees. Then, we introduce a restricted edit distance based technique to detect the restrictedly embedding relationship between a pattern tree and a data tree. Finally, we design an Apriori based algorithm, FRESTM, to solve the tree mining problem at hand. To the best of our knowledge, this is the first algorithm for finding restrictedly embedded subtree patterns in multiple unordered trees. Experimental results based on synthetic and real-world data demonstrate the good performance of the proposed algorithm.

Our method differs from previous tree mining algorithms [1], [6], [15], [17], [21], [22], [25] in several ways. Our leg attachment method is inspired by [6] but differs from it by considering expansion at the rightmost leaf only, while the work in [6] considers leg attachment at every leaf. Our pairwise joining approach is efficient, because our pairwise joining operation generates one candidate tree only. This is an improvement over the equivalence class-based extension initially proposed in [22], which generates up to 5 candidates by including incomplete leg attachment expansions. Our work also differs from phylogenetic tree mining algorithms [17], [22], [25] where trees are unordered but only

leaf nodes have labels. It also differs from OInduced [3], which discovers induced subtrees from ordered trees. The techniques proposed here can be further extended for designing new tree mining algorithms. For example, the efficient rightmost expansion techniques may be extended to mine induced subtrees. In terms of using the edit distance for pattern verification purposes, our method may be applicable to other types of subtrees by adopting different tree distance definitions.

In future work we plan to: 1) investigate other types of constraints imposed on subtree patterns and design efficient algorithms for finding those patterns; 2) explore the inherent parallelism of the FRESTM method; 3) collaborate with domain experts to apply FRESTM to problems arising in different domains; and 4) extend the FRESTM method to unrooted trees and graphs.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and the handling editor for their useful comments and constructive suggestions that have helped to improve the presentation and content of this paper. They would also like to thank Prof. M. J. Zaki at Rensselaer Polytechnic Institute, Prof. L. M. Li at the National University of Singapore for providing us the data and programs used in this paper, and Dr. Y. Chi at Yahoo! Labs for helpful conversations.

REFERENCES

[1] T. Asai *et al.*, "Efficiently mining frequent substructures from semi-structured data," in *Proc. Int. Workshop Inf. Elect. Eng.*, Suwon, Korea, May 2002, pp. 59–64.

[2] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering frequent substructures in large unordered trees," in *Proc. 6th Int. Conf. Discov. Sci.*, Sapporo, Japan, Oct. 2003.

[3] M. H. Chehreghani, C. Lucas, and M. Rahgozar, "OInduced: An efficient algorithm for mining induced patterns from rooted ordered trees," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 41, no. 5, pp. 1013–1025, Sep. 2011.

[4] Y. Chi, S. Nijssen, R. R. Muntz, and J. N. Kok, "Frequent subtree mining—An overview," *Fundam. Inf.*, vol. 66, nos. 1–2, pp. 161–198, 2005.

[5] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz, "Mining closed and maximal frequent subtrees from databases of labeled rooted trees," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 2, pp. 190–202, Feb. 2005.

[6] Y. Chi, Y. Yang, and R. R. Muntz, "HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms," in *Proc. 16th Int. Conf. Sci. Statist. Datab. Manage.*, Santorini Island, Greece, Jun. 2004.

[7] L. Hua *et al.*, "A method for discovering common patterns from two RNA secondary structures and its application to structural repeat detection," *J. Bioinform. Comput. Biol.*, vol. 10, no. 4, 2012, Art. ID 1250001.

[8] A. Jiménez, F. Berzal, and J. Cubero, "POTMiner: Mining ordered, unordered, and partially-ordered trees," *Knowl. Inf. Syst.*, vol. 23, no. 2, pp. 199–224, May 2010.

[9] K. G. Khoo and P. N. Suganthan, "Structural pattern recognition using genetic algorithms with specialized operators," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 33, no. 1, pp. 156–165, Feb. 2003.

[10] P. Kilpelainen and H. Mannila, "Ordered and unordered tree inclusion," *SIAM J. Comput.*, vol. 24, no. 2, pp. 340–356, 1995.

[11] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang, "XClust: Clustering XML schemas for effective integration," in *Proc. 11th ACM Int. Conf. Inf. Knowl. Manage.*, McLean, VI, USA, Nov. 2002.

[12] C. H. Leung and C. Y. Suen, "Matching of complex patterns by energy minimization," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 28, no. 5, pp. 712–720, Oct. 1998.

[13] L. Liu and J. Liu, "Mining frequent embedded subtree from tree-like databases," in *Proc. Int. Conf. Internet Comput. Inf. Serv.*, Hong Kong, Sep. 2011.

[14] B. Moayer and K. S. Fu, "A tree system approach for fingerprint pattern recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 3, pp. 376–387, May 1986.

[15] S. Nijssen and J. N. Kok, "Efficient discovery of frequent unordered trees," in *Proc. 1st Int. Workshop Mining Graphs, Trees, Sequences (MGTS)*, 2003.

[16] D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih, "Exact and approximate algorithms for unordered tree matching," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, no. 4, pp. 668–678, Apr. 1994.

[17] D. Shasha, J. T. L. Wang, and S. Zhang, "Unordered tree mining with applications to phylogeny," in *Proc. IEEE Int. Conf. Data Eng.*, Boston, MA, USA, 2004, pp. 708–719.

[18] J. T. L. Wang, K. Zhang, G. Chang, and D. Shasha, "Finding approximate patterns in undirected acyclic graphs," *Pattern Recognit.*, vol. 35, no. 2, pp. 473–483, 2002.

[19] J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A system for approximate tree matching," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 4, pp. 559–571, Aug. 1994.

[20] K. Wang and H. Liu, "Discovering typical structures of documents: A road map approach," in *Proc. 21st Annu. Int. ACM SIGIR Conf. Res. Dev. Inf. Retriev.*, 1998, pp. 146–154.

[21] M. J. Zaki, "Efficiently mining frequent trees in a forest: Algorithms and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 8, pp. 1021–1035, Aug. 2005.

[22] M. J. Zaki, "Efficiently mining frequent embedded unordered trees," *Fundam. Inf.*, vol. 65, nos. 1–2, pp. 33–52, Mar./Apr. 2005.

[23] K. Zhang, "A new editing based distance between unordered labeled trees," in *Proc. 4th Annu. Symp. Combin. Pattern Match.*, Padova, Italy, Jun. 1993.

[24] K. Zhang, J. T. L. Wang, and D. Shasha, "On the editing distance between undirected acyclic graphs," *Int. J. Found. Comput. Sci.*, vol. 7, no. 1, pp. 43–58, 1996.

[25] S. Zhang and J. T. L. Wang, "Discovering frequent agreement subtrees from phylogenetic data," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 1, pp. 68–82, Jan. 2008.

[26] L. Zou, Y. Lu, H. Zhang, R. Hu, and C. Zhou, "Mining frequent induced subtree patterns with subtree-constraint," in *Proc. 6th IEEE Int. Conf. Data Mining (ICDM) Workshop*, Hong Kong, Dec. 2006.

Sen Zhang (M'07) received the Ph.D. degree in computer science from the New Jersey Institute of Technology, Newark, NJ, USA.

He is an Associate Professor of Computer Science with the State University of New York, Oneonta, NY, USA.

Zhihui Du (M'99) received the Ph.D. degree in computer science from Peking University, Beijing, China.

He is an Associate Professor of Computer Science with Tsinghua University, Beijing.

Jason T. L. Wang (M'87) received the Ph.D. degree in computer science from the Courant Institute of Mathematical Sciences, New York University, New York, NY, USA.

He is a Professor of Bioinformatics and Computer Science with the New Jersey Institute of Technology, Newark, NJ, USA.