OInduced: An Efficient Algorithm for Mining Induced Patterns from Rooted Ordered Trees

Mostafa Haghir Chehreghani, Morteza Haghir Chehreghani, Caro Lucas, and Masoud Rahgozar

Abstract—Frequent tree patterns have many practical applications in different domains such as XML mining, web usage analysis, etc. In this paper, we present OInduced, a novel and efficient algorithm for finding frequent ordered induced tree patterns. OInduced uses a breadth-first candidate generation method and improves it by means of an indexing scheme. We also introduce frequency counting using tree encoding. For this purpose, we present two novel tree encodings, m-coding and cm-coding, and show how they can restrict nodes of input trees and compute frequencies of generated candidates. We perform extensive experiments on both real and synthetic datasets to show efficiency and scalability of OInduced.

Index Terms—Rooted ordered labeled tree, frequent tree pattern, induced subtree, breadth first candidate generation, frequency counting, tree encoding.

I. INTRODUCTION

MINING frequent tree patterns is very useful in domains such as user web log analysis, XML document mining, web mining, bioinformatics and network routing. For example, in [35], tree patterns are used as a powerful tool to distinguish users according to their behavior on the web. In this work, first, log data are converted into rooted ordered trees and a set of frequent patterns is extracted from them. Then, based on these patterns, a structural classifier is built to classify different users. Structural classifiers show higher performance compared to traditional classifiers which treat each tree as a bag of words [35].

In this paper, we focus on the problem of extracting induced patterns from a database of rooted ordered trees. Several algorithms have been proposed to find induced patterns from a collection of rooted ordered tree. The well-known algorithm in this context is FREQT [2]. FREQT uses an occurrence-list based approach for frequency counting. For each subtree, all the nodes in the database are stored in a list in which the rightmost node of the subtree can be mapped. The size of the occurrence list kept for each frequent pattern can be large ($O(|V|)$, where $|V|$ is the number of nodes of the database). This makes the algorithm inefficient, especially for dense datasets in which the correlation among trees is very high.

Recently, iMB3Miner [22] tries to restrict invalid candidates using a tree model guided approach. For frequency counting, iMB3Miner uses the information gathered for guided candidate generation. However, the amount of this information is high. Each occurrence of a candidate $C$ is encoded as an occurrence coordinator whose size is $|C|$.

In this paper, we develop more efficient data structures for storing the information used in frequency counting. To do so, we initiate frequency counting based on tree encoding. The key contributions of our work are as follows:

1) We develop a new equivalence class extension to extend each candidate by only frequent trees. We use breadth first search and take advantage of an indexing scheme to perform the class extension, effectively.

2) We present two new tree encodings and accordingly, develop a novel and efficient approach for frequency counting. We show that successful occurrences of a candidate must satisfy a number of conditions and the presented tree encodings can check the conditions, efficiently. The size of each occurrence in the proposed method is $O(1)$.

3) We introduce a new and efficient algorithm, called OInduced, for the problem of finding all the frequent induced ordered tree patterns from a single tree or from a forest of trees. We compare OInduced with most efficient previous works, and by performing extensive experiments, we show that OInduced provides significant improvements for both real data and synthetic data.

The rest of this paper is organized as follows. In section 2, some preliminaries and definitions related to tree mining and tree patterns are given. In section 3, we have a brief overview on the related works. Section 4 describes our proposed candidate generation method. In section 5 we present two new tree encodings as well as the method used for frequency counting. We experimentally evaluate the effectiveness of OInduced in section 6. Finally, the paper is concluded in section 7.

II. PRELIMINARIES AND PROBLEM STATEMENT

To explain the problem of mining frequent tree patterns in a collection of trees we provide the following definitions:

a) Rooted labeled tree: A rooted labeled tree $T = (V, E, L)$ is a connected directed acyclic graph (DAG) with $V$ as the set of nodes and $E = \{(x, y) | x, y \in V\}$ as the set of edges. $L : V \rightarrow N$ is a labeling function that assigns an integer to each node of the tree. A distinguished node $r$ is
considered as the root, and for any other node \( x \), there is a unique path from \( r \) to \( x \). A rooted labeled ordered tree has a left-to-right ordering among each set of siblings.

b) Zaki’s string representation: Zaki’s string representation \( S \) for a tree \( T \) is defined as follow: labels of the nodes are added to \( S \) in the preorder traversal of \( T \), and when a backtracking from a child to its direct parent occurs, a unique symbol (e.g. -1) is added to \( S \) [32]. For convenience, through the paper, we present each tree by its string representation. For example, tree \( T \) of Figure 1 is presented as "1 2 -1 3 -1 3 3".

c) Induced subtree: For a rooted labeled tree \( T = (V, E, L) \), a rooted labeled tree \( T' = (V', E', L') \) is an induced subtree of \( T \) (or \( T' \) is isomorphic to a subtree of \( T \)), if and only if: (1) \( V' \subseteq V \), (2) \( E' \subseteq E \), (3) \( L' \subseteq L \) and the labeling of \( V' \) in \( T \) is preserved in \( T' \) and (4) if defined for rooted ordered trees, the left-to-right ordering among the siblings in \( T \) is preserved among the corresponding nodes in \( T' \).

If a \( k \)-candidate (a candidate tree with \( k \) nodes) \( C_k \) is an induced subtree of an input tree \( T \), an occurrence \( O_k \) of \( C_k \) in \( T \) is the subtree of \( T \) which is isomorphic to \( C_k \). Two distinct occurrences can share some nodes in common, but they cannot consist of entirely the same nodes. For example, in Figure 1, \( T \) is an input tree, \( C \) is a candidate, and \( O \) and \( O' \) are two occurrences of \( C \) in \( T \). \( O \) and \( O' \) share two nodes in common: the nodes with labels 1 and 2.

d) Embedded subtree: For a rooted labeled tree \( T = (V, E, L) \), a rooted labeled tree \( T' = (V', E', L') \) is an embedded subtree of \( T \) if and only if: (1) \( V' \subseteq V \), (2) \( v_1 \) is the parent of \( v_2 \) in \( T' \) if \( v_1 \) is an ancestor of \( v_2 \) in \( T \), (3) \( L' \subseteq L \) and the labeling of \( V' \) in \( T \) is preserved in \( T' \) and (4) if defined for rooted ordered trees, the left-to-right ordering among the siblings in \( T \) is preserved among the corresponding nodes in \( T' \).

e) Per-tree support (per-tree frequency, per-transaction frequency) and occurrence-match support (occurrence-match frequency): Given a database \( D \) consisting of rooted ordered labeled trees and a subtree \( S \) the per-tree support (or per-tree frequency) of \( S \) is the number of trees in \( D \) for which \( S \) is an induced subtree. The occurrence-match support (or occurrence-match frequency) of \( S \) is defined as the number of occurrences of \( S \) in \( D \). Per-tree support can be expressed more formally as follows:

\[
\text{support}_T(S, D) = \sum_{T \in D} \text{IsInd}(S, T)
\]

where \( \text{IsInd}(S, T) \) is 1 if \( S \) is the induced subtree of \( T \) and 0 otherwise. Occurrence-match support can be represented as follows:

\[
\text{support}_O(S, D) = \sum_{T \in D} \text{NumInd}(S, T)
\]

where \( \text{NumInd}(S, T) \) is the number of occurrences of \( S \) in \( T \).

f) Frequent tree: Tree \( C \) is frequent if its per-tree support (occurrence-match support) is more than or equal to a user-specified per-tree (occurrence-match) \( \text{minsup} \) value. The problem of mining frequent tree patterns from a database of tree-structured data is concerned with finding all frequent trees. The desired type of patterns in the mining process can differ based on the type of the application. In this paper, our concern is mining frequent induced patterns from rooted ordered labeled trees. Both of per-tree frequency and occurrence-match frequency are allowed in this work. There is no overall agreement on the definition of support for different applications. It seems that occurrence-match frequency is more applicable for structured data [22]. For simplicity, through the paper, we use the term frequency (support) to refer to occurrence-match frequency (occurrence-match support); unless we explicitly say that frequency (support) refers to per-tree frequency (per-tree support).

III. RELATED WORKS

Recently, many algorithms have been proposed in the literature for finding frequent tree patterns from a collection of trees. Wang et al. [26] motivate the schema discovery in the general setting. They also investigate discovering typical structures from web documents and propose algorithms for discovering similar structures and structural association rules among a collection of tree-structured data [27] and [28].

Feng et al. [9] introduce an XML-enabled association rule template which is flexible to represent both simple and complex rules. They continue the work by presenting template models to help users to specify the interesting XML associations to be mined and propose techniques for template-guided mining of association rules [8].

Zaki introduces TreeMiner [32] to mine embedded ordered frequent tree patterns. For frequency counting, he uses a new data structure called \( \text{scope-list} \) and defines join operations for vertical frequency counting. TreeMiner stores each occurrence in \( O(k) \) space, where \( k \) is the size of the tree. He also introduces the rightmost path extension to generate non-redundant candidates. Later, he proposes \( \text{SLEUTH} \) for mining embedded unordered tree patterns [33]. Asai et al. [2] independently propose the rightmost candidate generation. They developed \( \text{FREQT} \) for mining frequent induced ordered tree patterns. In FREQT, for each occurrence, a list stores all nodes in the database for which the rightmost node of the occurrence can be mapped.

Independently, Asai et al. and Nijssen et al. extend FREQT to discover induced unordered tree patterns and present \( \text{Unot} \) [3] and \( \text{uFreqt} \) [17] algorithms. For frequency counting, Unot uses an occurrence list based approach in which each occurrence is stored in \( O(k) \) space, where \( k \) is the size of the tree. uFreqt uses a different occurrence list based approach.
for frequency counting that its size is bounded by the product of the size of the database and the size of the pattern.

HybridTreeMiner [6] discovers induced unordered tree patterns and uses a breadth-first candidate generation method. However, occurrence lists in HybridTreeMiner must record occurrences of a candidate in all possible orders. PathJoin [29] assumes that labels for the children of each node are unique and finds induced unordered maximal patterns. The number of maximal patterns is much less than the number of all the frequent tree patterns.

Chi et al. [5] propose FreeTreeMiner for mining induced unordered free trees. To compute the frequency of a candidate $C$, FreeTreeMiner uses a tree isomorphism algorithm based on bipartite graph matching. Its time complexity is $O(|T| \times |C| \times \sqrt{|C|})$, where $|T|$ and $|C|$ are the sizes of $T$ and $C$, respectively.

TreeFinder [24] uses an Inductive Logic Programming approach to mine unordered, embedded subtrees, but it is not a complete method and may loose many frequent trees. SingleTreeMining [20] is an algorithm proposed for mining rooted unordered trees with application to phylogenetic. Chi et al. propose CMTreeMiner [7] for mining both closed and maximal frequent trees. This algorithm traverses an enumeration tree that systematically enumerates all subtrees, and uses an enumeration DAG to prune the branches of the enumeration tree that do not correspond to closed or maximal frequent subtrees.

Xiao et al. [30] propose TreeGrow for mining unordered maximal embedded tree patterns. However, TreeGrow assumes that the labels for the children of each node are unique. Their candidate generation method is localized so as to avoid unnecessary computational overhead.

The methods of [15], [16] and [21] discover frequent tree patterns in web documents by using tag tree patterns as hypotheses. A tag tree pattern is an edge labeled tree which has structured variables and a variable can match to an arbitrary subtree.

XSPanner [25] is a pattern growth-based method and can mine embedded ordered trees. The pseudo-projection step in XSPanner is expensive that reduces its performance. Tatikonda et al. [23] propose a generic approach for mining tree patterns. They develop TRIPS and TIDES algorithms using two sequential encodings of trees to systematically generate and evaluate the candidate patterns. However, TRIPS and TIDES can only work with per-tree support. Tan et al. [22] present a unique embedding list representation of the tree structure, which enables efficient implementation of their Tree Model Guided (TMG) candidate generation.

To find frequent unordered tree patterns, most of the proposed algorithms use a canonical form and extend only candidates that are in the canonical form. A canonical form is a unique way to represent a labeled tree. Luccio et al. [13], [14] define sorted pre-order string method. This method for a rooted unordered tree is defined as the lexicographically smallest one among those pre-order strings of the ordered trees that can be obtained from the unordered tree. They show that for a rooted unordered tree, its canonical representation based on the pre-order traversal can be obtained in linear time, using the tree isomorphism algorithm of Aho [1]. Later, Asai et al. [3], Nijssen et al. [17], and Chi et al. [5] independently define similar canonical representations.

Efficient algorithms for mining frequent graph patterns which are the general form of frequent tree patterns can be found in [10], [12] and [31]. In [10] a graph transaction is represented by an adjacency matrix and frequent patterns appearing in the matrices are mined using the basket analysis algorithms. Kuramochi et al. [12] propose FSG to find all connected subgraphs that appear frequently in a large graph database. FSG incorporates some optimizations for candidate generation and counting to scale to large graph databases. Yan et al. [31] present CloseGraph for mining closed graph patterns and develop pruning techniques based on early termination.

The tree matching problem, i.e. finding occurrences of a pattern tree in a target tree is studied in [11], and several dynamic programming methods are presented. Shasha et al. [18] survey the algorithms proposed for processing queries on trees and describe algorithms for search in graphs. In [19] the authors present an algorithm to the nearest neighbor search problem for unordered labeled trees. Their algorithm is based on storing the paths of the trees in a suffix array and then counting the number of mismatching paths between a query tree and a data tree.

In general, finding frequent patterns includes two main steps: candidate generation and frequency counting. The well-known method for candidate generation in trees is the rightmost path extension method, and equivalence class extension has widely been used in embedded pattern mining algorithms to improve rightmost path extension. Initial frequency counting methods, in fact, are tree matching algorithms which compute frequencies of patterns, independently. Later, vertical frequency counting methods are introduced that are highly data structure dependent. They usually define join operations on the used data structure and compute frequencies of larger candidates by joining occurrences of smaller ones.

IV. CANDIDATE GENERATION

Our candidate generation method, which is in fact an extension of the well-known rightmost path extension method, generates candidates in a breadth-first way. The rightmost path extension is shown to be complete and non-redundant for generating embedded and induced candidates [2], [32] and [34]. In this method a node is added anywhere in the rightmost path of a $k$-candidate $C$ and generates a $k+1$-candidate $C'$. In its simple form, it extends each candidate by connecting all frequent nodes to all nodes of the rightmost path.

Algorithms such as [32] try to improve candidate generation using equivalence class extension. The main observation behind equivalence class extension is that only known frequent elements are used to extend a candidate [32]. An equivalence class is defined as follows: two trees $C$ and $C'$ are in the same equivalence class if they differ only in the rightmost node. Equivalence class extension has been vastly used to improve embedded candidate generation. In the following, a new equivalence class based extension method is presented for induced candidate generation. Our method extends a candidate...
In equivalence class extension, two \( k \)-candidates \( C \) and \( C' \) join together and the rightmost node of \( C' \), (the second tree) is added to a position in the rightmost path of \( C \) (the first tree). In the extended \( k+1 \)-candidate, the parent of the rightmost node of \( C' \) is either the rightmost node of \( C \) or another node in the rightmost path of \( C \). The first case, denoted by \textit{rn\_extension}, generates deeper candidates and in the second case, denoted by \textit{rp\_extension}, the number of children of the rightmost path increases (wider candidates are generated).

### A. rp\_extension

**Definition 1:** Position of \( x \in V(T) \), denoted by \( \text{post}(x) \), is defined as its depth in \( T \). The position can uniquely distinguish a node in a path. Let \( X \) be the node in the position \( p-1 \) of the rightmost path of \( T \). When we say node \( N \) is added to the position \( p \) of the rightmost path of \( T \), we mean that \( N \) becomes the rightmost child of \( X \).

For \textit{rp\_extension}, our method acts as [34] proposed for embedded candidate generation: for every two candidates \( C \) and \( C' \) belonging to a same equivalence class, the rightmost node of \( C' \) is added to the position \( \text{pos}_{C'}(\text{rightmost node of } C') \) of the rightmost path of \( C \). If \( \text{pos}_{C'}(\text{rightmost node of } C') \) refers to the rightmost node of \( C \), the extension is invalid. So, we will have the following restriction for the \textit{rp\_extension}: \( \text{pos}_{C'}(\text{rightmost node of } C') \geq \text{pos}_{C'}(\text{rightmost node of } C') \).

Figure 2 shows an example of \textit{rp\_extension} in which \( T_0 \) and \( T_1 \) are two input trees, \( \text{minsup} \) is equal to 2, and level 3 contains all the frequent candidates with 3 nodes. Since only frequent candidates are used for future extension, non-frequent candidates are deleted after applying a direct frequency counting operation. For example consider the tree "1 2 3" belonging to equivalence class 0. Since the position of the rightmost node of "1 2 3", (i.e. 2) is greater than the position of the rightmost node of "1 2 -1 3" (i.e. 1), "1 2 3" can join with "1 2 -1 3". The resultant candidate, "1 2 3 -1 -1 3", is generated by adding the rightmost node of "1 2 -1 3", (i.e. "3") to the position 1 of "1 2 3", "1 2 3" can also join with itself and generate candidate "1 2 3 -1 3".

Extension of each candidate generates a new equivalence class. Figure 2 contains all 4-candidates (frequent and non-frequent) generated through \textit{rp\_extension}.

### B. rn\_extension

**Definition 2:** Index of an equivalence class, denoted by \( E \), is defined as the tree consisting of the first \( k-1 \) nodes which are shared among all members of the class.

**Definition 3:** First \( k-1 \) subtree of tree \( T \), denoted by \( \text{first}_{k-1}(T) \), is the subtree generated by removing the rightmost node of \( T \).

**Definition 4:** If tree \( T \) has more than one leaf, its second rightmost leaf, denoted by \( srl \), is defined as the leaf which has the greatest preorder number among all the leaves except the rightmost node.

**Definition 5:** The last \( k-1 \) subtree of tree \( T \), denoted by \( \text{last}_{k-1}(T) \), is the subtree generated by removing either: 1) the root of \( T \) (if \( T \) has only one leaf), or 2) the \( srl \) of \( T \) (if \( T \) has more than one leaf).

For example, in Figure 2, \( \text{first}_{k-1} \) of "1 2 3 -1 -1 3" is "1 2 3", its \( srl \) is the node "3" in position 2 and its \( \text{last}_{k-1} \) is "1 2 -1 3". The \( \text{last}_{k-1} \) of "1 2 3" is "2 3", since it has only one leaf.

Theorem 6 helps us to find the equivalence class that \textit{rn\_extension} extends a candidate.

**Theorem 6:** \( k \)-candidate \( C_k \) can be \textit{rn\_extended} if there exists a \( k \)-candidate \( C_k' \) such that \( \text{last}_{k-1}(C_k) \) and \( \text{first}_{k-1}(C_k') \) are identical.

**Proof:** Consider candidate \( C_{k+1} \) generated by adding a child \( N \) to the rightmost node of \( C_k \). If another node \( M \), \( M \neq N \), is deleted from \( C_{k+1} \), candidate \( C_k' \) is generated. Then, \( C_{k+1} \) can be generated by joining \( C_k \) and \( C_k' \).

- \( M \) cannot be an intermediate node (intermediate node is neither root nor leaf); because in this situation, removing \( M \) converts a parent-child relation into an ancestor-descendant relation and for induced patterns these relations are not equivalent.
- \( M \) can be the root of \( C_{k+1} \). If the root of \( C_{k+1} \) has only one child, no problem arises. However, if the root of \( C_{k+1} \) has more than one child, removing the root generates a forest in which the size of each tree is smaller than \( k \), instead of generating a single \( k \)-candidate.
- \( M \) can be an arbitrary leaf node, e.g. the \( srl \). If \( C_{k+1} \) has more than one leaf, no problem arises. However, if \( C_{k+1} \) has only one leaf, nodes \( N \) and \( M \) will be equivalent and therefore, in this state \( M \) can not be removed.

If \( C_{k+1} \) has one leaf, its root will have only one child. So, in this case the root can be deleted. Now we can claim that...
C. Finding the equivalence class that \textit{rn\_extends} a tree

An important issue is finding the equivalence class \( E' \) that \textit{rn\_extends} \( C_k \). An inefficient solution is to compare \( last_{k-1}(C_k) \) with all class indices, until the satisfying one is found. The class indices of a specific level and as well as the trees of a single equivalence class can be generated in an ordered way. This can improve the search process. However, still there exists a problem: although members of an equivalence class are ordered and they share \( k \)-1 prefix, their \( last_{k-1} \) are not ordered. The reason is that for each tree the node which is deleted and generates \( last_{k-1} \) can be either the root or the \( srl \).

Here, we propose a simple and efficient indexing scheme to find the equivalence class \( \text{rn\_extends} \) a tree. Lemma 9 and Theorem 10 provide the rationale behind the indexing scheme.

\textbf{Lemma 9:} Assume that tree \( C_{k-1} \) is \textit{rp\_extended} by tree \( C'_{k-1} \) and generates tree \( C_k \). Then, \( C'_{k-1} \) will be \( last_{k-1}(C_k) \).

\textbf{Proof:} Since the rightmost node of \( C'_{k-1} \) is added to a non-leaf node of \( C_{k-1} \) and generates a new leaf, \( C_k \) has more than one leaf. On the other hand, when the rightmost leaf of \( C_{k-1} \) is added to \( C_k \), the rightmost leaf of \( C_{k-1} \) will be the second rightmost leaf of the resultant tree \( C_k \). Since \( C_{k-1} \) and \( C'_{k-1} \) belong to the same equivalence class, they share the first \( k \)-1 nodes. So removing the node corresponding to the rightmost node of \( C_{k-1} \) from \( C_k \) (which is the \( srl \) of \( C_k \)), will generate \( C'_{k-1} \). This means that \( C'_{k-1} \) is \( last_{k-1}(C_k) \).

\textbf{Theorem 10:} Suppose that tree \( C_{k-1} \) is \textit{rn\_extended} (via either \textit{rn\_extension} or \textit{rn\_extends}) by tree \( C'_{k-1} \) and generates tree \( C_k \). \( C_k \) can be \textit{rn\_extended} by the class whose index is \( C'_{k-1} \).

\textbf{Proof:} 1) First, assume that \( C_{k-1} \) is \textit{rp\_extended} by \( C'_{k-1} \). According to Lemma 9, \( C'_{k-1} \) becomes \( last_{k-1}(C_k) \). Therefore, \( C'_{k-1} \) will be the index of the class which \textit{rn\_extends} \( C_k \) and generates candidates with \( k \)-1 nodes.

2) Then, assume that \( C_{k-1} \) is \textit{rn\_extended} by \( C'_{k-1} \). There are two possible situations:
Extend
1: Require: candidate \( C_k \);
2: Ensure: all \((k+1)\)-extensions of \( C_k \);
3: Output \( \Rightarrow \emptyset \);
4: for all candidates \( C'_k \) in the equivalence class of \( C_k \) do
5: if \( \text{pos}_{C_k}(\text{rightmost node of } C_k) \geq \text{pos}_{C'_k}(\text{rightmost node of } C'_k) \) then
6: Generate candidate \( C_{k+1} \) by adding the rightmost node of \( C'_k \) to \( \text{pos}_{C'_k}(\text{rightmost node of } C'_k) \) of \( C_k \);
7: Output \( \Rightarrow \) Output \( \cup C_{k+1} \);
8: end if
9: end for
10: Find the equivalence class \( E' \) that its index satisfies the condition of Theorem 6.
11: for all candidates \( C'_k \in E' \) do
12: Generate candidate \( C_{k+1} \) by adding the rightmost node of \( C'_k \) to \( C_k \) as the child of the rightmost node of \( C_k \);
13: Output \( \Rightarrow \) Output \( \cup C_{k+1} \);
14: end for
15: return Output;

Fig. 4: High level pseudo code of the candidate generation method.

a) \( C_k \) might have more than one leaf. As a result, \( C_{k-1} \) will have more than one leaf, and \( C_{k-1} \) and \( C_k \) will have the same slr. On the other hand, \( \text{last}_{k-2}(C_{k-1}) \) and \( \text{first}_{k-2}(C'_{k-1}) \) are identical and since \( V(C_k) \setminus V(C_{k-1}) \) is the rightmost node of \( C'_{k-1} \), \( \text{last}_{k-1}(C_k) \) will be generated by adding the rightmost node of \( C'_{k-1} \) to \( \text{last}_{k-2}(C_{k-1}) \), and this tree is \( C'_{k-1} \).

b) \( C_k \) might have one leaf. Then \( C_{k-1} \) will have one leaf and the roots of \( C_{k-1} \) and \( C_k \) will be the same. On the other hand, \( \text{last}_{k-2}(C_{k-1}) \) and \( \text{first}_{k-2}(C'_{k-1}) \) are identical and since \( V(C_k) \setminus V(C_{k-1}) \) is the rightmost node of \( C'_{k-1} \), \( \text{last}_{k-1}(C_k) \) will be generated by adding the rightmost node of \( C'_{k-1} \) to \( \text{last}_{k-2}(C_{k-1}) \), and this tree is \( C'_{k-1} \).

For example, in Figure 3, "1 2" is \( \text{rn}_{\text{extended}} \) by "2 3" and generates "1 2 3". The class \( \text{rn}_{\text{extending}} "1 2 3" \) is the class whose index is "2 3", "2 3" is \( \text{rp}_{\text{extended}} \) by "2 3" and generates "2 3 -1 3". The class \( \text{rn}_{\text{extending}} "2 3 -1 3" \) is the class whose index is "2 3".

To find the class which \( \text{rn}_{\text{extends}} \) a candidate \( C_k \), two new integers are assigned to \( C_k \): \( \text{Id1} \) and \( \text{Id2} \). \( \text{Id1} \) determines \( C_k \) which tree of level \( k \), and \( \text{Id2} \) determines \( \text{last}_{k-1}(C_k) \) which class of level \( k-1 \). \( \text{last}_{k-1}(C_k) \) is the index of the class which \( \text{rn}_{\text{extends}} \) \( C_k \). Assume that \( C_{k+1} \) is a new tree generated by joining \( C_k \) (as the first subtree) with \( C'_{k} \) (as the second subtree). The \( \text{Id2} \) of \( C_{k+1} \) is set to the \( \text{Id1} \) of \( C'_{k} \). Theorem 10 provides the rationale behind this assignment. The \( \text{Id1} \) of \( C_{k+1} \) can be easily determined by means of a counter that increases by one for each generated tree at level \( k+1 \).

To correctly refer to the equivalence class \( \text{rn}_{\text{extending}} \) \( C_k \), we need to generate all the classes at level \( k-1 \), even those having no member. If so, \( \text{Id2} \) will refer directly to the equivalence class \( \text{rn}_{\text{extending}} \) the tree. In general, the number of classes at level \( k \) must be equal to the number of frequent trees at level \( k-1 \).

Fig. 5 shows how the indexing scheme can be applied to our running example. At level 3 all the classes are generated, even those without any member. "1 2" is \( \text{rn}_{\text{extended}} \) by "2 3" and generates "1 2 3". "2 3" is the third tree of level 2 (so its \( \text{Id1} \) would be 2), therefore "1 2 3" will be \( \text{rn}_{\text{extended}} \) by the third class of level 3. "2 3" is \( \text{rp}_{\text{extended}} \) by "2 3".
Encoding
1: Require: an input tree $T$.
2: Ensure: m-coding and cm-coding of nodes of $T$.
3: $mid ← 0$.
4: m-coding(root($T$)) ← 0.
5: for all nodes $x$ in preorder traversal of $T$ do
6: for all children $r$ of $x$ in right-to-left order do
7: $mid ← mid + 1$.
8: m-coding($r$) ← mid.
9: end for
10: cm-coding($x$) ← mid.
11: end for
12: return m-coding and cm-coding.

Fig. 6: High level pseudo code of m-coding and cm-coding

and generates ”2 3 -1 3”. ”2 3” is the third tree at level 2, therefore, ”2 3 -1 3” will be rm_extended by the third class at level 3.

V. FREQUENCY COUNTING
In this section, we develop a new method for frequency counting which is based on tree encodings. We first introduce two new tree encodings, and then explain how these encodings among with an already proposed encoding can be used to compute frequencies of candidates.

A. M-coding
In this encoding, an auxiliary integer, called $mid$, is used which is initiated by 0. M-coding of the root is set to 0. The tree is traversed in preorder and when a node $x$ is met: the children of $x$ are scanned from right to left and for each child $r$: $mid$ is increased by one and the m-coding of $r$ is set to the new value of $mid$. Since the nodes of the tree are traversed in preorder, when determining the m-coding of the children of a node, its m-coding has already been determined.

B. Cm-coding
Cm-coding of node $x$ in input tree $T$ is m-coding of its leftmost child, i.e. the greatest m-coding among its children. When a node is met in preorder traversal of the tree, m-coding of its children are assigned, therefore cm-coding of each node can be determined in $O(1)$ time complexity. Figure 6 presents the high level pseudo code of determining m-coding and the cm-coding. By one scan of $T$, m-coding and cm-coding of all nodes of $T$ are determined.

As an example of the tree encodings, consider Figure 7 which presents the p-coding, m-coding and cm-coding of the input trees of our running example. P-coding refers to the preorder number of a node in an input tree. While p-coding is a depth-first traversal, m-coding and cm-coding are combined depth-first/breadth-first traversals.

C. Frequency counting
As mentioned above, tree $C_k$ can be extended in two different ways: rp_extension and rm_extension. Each extension requires its particular method for frequency counting. In the rest of this section, we use the following assumptions and notations. We assume that occurrence $O_k$ of $k$-candidate $C_k$, occurrence $O_N$ of node $N$ and occurrence $O_{k+1}$ of $k+1$-candidate $C_{k+1}$ occur in the input tree $T$. RN refers to the rightmost node of $O_k$ and RP refers to the rightmost path of $C_k$ excluding its rightmost node, i.e. $V(RP) \cup V(RN)$ forms the nodes of the rightmost path of $O_k$. $O_RN$ refers to the rightmost node of $O_k$ and $O_RP$ refers to the rightmost path of $O_k$ excluding its rightmost node, i.e. in $O_k$, $O_RN$ and $O_RP$ are the occurrences of $RN$ and $RP$, respectively. We use the notation $part_T(x)$ to refer to the parent of node $x$ in tree $T$.

1) Frequency counting for rp_extension candidates: Suppose that $C_{k+1}$ is generated by adding node $N$ to $C_k$ via rp_extension. We want to know if adding $O_N$ to the rightmost node of $O_k$ generates occurrence $O_{k+1}$. The input tree $T$ can be divided into the partitions depicted in Figure 8. $B1$ is the path between the root of $T$ and the root of $O_k$. $RC$ includes the right children of the nodes of $B1$ and the right children of the nodes of $O_RP$. Let $a$ be a node on $B1$ and assume that its child $b$ belongs to $B1$, too. Right children of $a$ are the children whose preorder numbers are greater than the preorder number of $b$. Now, let $a$ be a node in $O_RP$ and assume that its child $b$ belongs to $O_RP$, too. Right children of $a$ are the children of $a$ whose preorder numbers are greater than the preorder number of $b$. $B2$ is the path between $O_RN$ and $z$, where $z$ is the last node met before $O_RP$ in the preorder traversal of $T$.

To generate an occurrence $O_{k+1}$ of $C_{k+1}$, $O_N$ must belong to the dotted region. For this purpose, $O_N$ must satisfy Properties 11, 12 and 13.

Property 11: \( p\text{-coding}(O_N) > p\text{-coding}(O_{RN}) \).

Proof: Assume that $O_N$ is added to node $x$ in $O_k$, $x$ is an ancestor of $O_{RN}$, and $O_N$ is a right child of $x$, therefore, the preorder number of $O_N$ is greater than the preorder number of $O_{RN}$.

Property 12: \( m\text{-coding}(O_N) < m\text{-coding}(O_{RN}) \).

Proof: There exist two possible situations: 1) $O_N$ is not
Fig. 8: Partitioning an input tree $T$.

(a) Hachured parts are eliminated by applying Property 11.

(b) Hachured parts are eliminated by applying Property 12.

Fig. 9: The restrictions of properties 1 and 2 on the input tree $T$.

added to the parent of $O_{RN}$: since the $p$-coding of the parent of $O_N$ is smaller than the $p$-coding of the parent of $O_{RN}$, therefore, the $m$-coding of $O_N$ will be smaller than the $m$-coding of $O_{RN}$. 2) $O_N$ is added to the parent of $O_{RN}$: since $O_N$ is the right sibling of $O_{RN}$, the $m$-coding of $O_N$ will be smaller than the $m$-coding of $O_{RN}$.

$O_N$ can be anywhere in $T$. It can be seen easily that if Property 11 is applied to $O_N$, it can not be selected from the hachured parts of Figure 9a. Property 12 limits $O_N$ to the non-hachured parts of Figure 9b. Intersection of non-hachured parts of Figures 9a and 9b is the $RC$ area, i.e. applying Properties 11 and 12 to $O_N$ restricts it to the $RC$ area. It is necessary to apply another restriction on $O_N$ to limit it to the dotted region.

**Property 13:** $post(T) - post(O_{RN}) = posc_k(N) - posc_k(RN)$

**Proof:** The length of the path between every pair of nodes in $O_k$ is equal to the length of the path between the corresponding nodes in $T$. Since $O_k$ is an occurrence of $C_k$ in $T$, the length of the path between every pair of nodes in $C_k$ is equal to the length of the path between the corresponding nodes in $T$. Therefore:

$$post(N) - posf(par_T(O_{RN})) = posc_k(N) - posc_k(par_{C_k}(RN))$$

Furthermore:

$$post(par_T(O_{RN})) = post(O_{RN}) - 1$$

$$post(par_T(RN)) = post(RN) - 1$$

Therefore:

$$post(T) - post(O_{RN}) = posc_k(N) - posc_k(RN)$$

If $O_N$ satisfies Properties 11, 12 and 13, it can generate an occurrence of $C_{k+1}$ by appending to $O_k$.

As an example, consider tree “2 3 3 -1 -1 3” of Figure 5 generated through $rn$ extension. Encodings of the rightmost node of “2 3 3” in $T_0$ are: $p$-coding=4, $m$-coding=6, $cm$-coding=6 and position of the rightmost node of “2 3 3” in $T_0$ is 3. Encodings of the rightmost node of “2 3 3” in $T_1$ are: $p$-coding=4, $m$-coding=5, $cm$-coding=5 and position of the rightmost node of “2 3 3” in $T_1$ is 3. Figure 10 shows different occurrences of “3”. Only one occurrence satisfies all the conditions mentioned in Properties 11-13. Therefore, “2 3 3 -1 -1 3” will have one occurrence in the input trees of our running example.

2) **Frequency counting for $rn$ extended candidates:** Assume that $C_{k+1}$ is generated by adding node $N$ to $C_k$ through $rn$ extension. We want to see if adding $O_N$ to the rightmost node of $O_k$ generates the occurrence $O_{k+1}$. Tree $T$ can be divided into the partitions depicted in Figure 11. This partitioning is slightly different from the partitioning of Figure 8, especially $RC$ contains the right children of the nodes of $B1$ and the right children of the nodes of $O_{RP}$ and all children of $O_{RN}$. $B1$, $B2$ and $z$ are defined similar to Figure 8.

$O_N$ can be anywhere in $T$. In order to generate an occurrence $O_{k+1}$ of $C_{k+1}$, it must belong to the dotted region of
Figure 11. For this purpose, \( O_N \) must satisfy Properties 14 and 15.

Property 14: \( \text{cm-coding}(O_N) \leq \text{m-coding}(O_{RN}) \).

Proof: Directly from the definition of cm-coding.

Property 15: \( \text{m-coding}(O_N) > \text{m-coding}(O_{RN}) \).

Proof: When \( O_N \) is a child of \( O_{RN} \), the parent of \( O_N \) is met after the parent of \( O_{RN} \) in the preorder traversal, therefore, \( \text{m-coding} \) of \( O_N \) will be greater than \( \text{m-coding} \) of \( O_{RN} \).

It can be seen easily that if Property 14 is applied to \( O_N \), it can not be selected from the hachured parts of Figure 12a. Property 15 limits \( O_N \) to non-hachured parts of Figure 12b. Intersection of non-hachured parts of Figures 12a and 12b is the dotted region. This means that \( O_N \) can generate an occurrence of \( C_{k+1} \) by appending to \( O_k \) iff it satisfies Properties 14 and 15.

Figure 13 shows how Properties 14 and 15 can be used to determine frequencies of \( \text{m-extended} \) candidates. Consider tree "1 2 3 3" which is generated via \( \text{m-extension} \) of "1 2 3". "1 2 3" has 4 occurrences in the input trees, 2 occurrences in \( T_0 \) and 2 occurrences in \( T_1 \). For each occurrence of "1 2 3" in \( T_i \) \((i \in \{0, 1\})\) all occurrences of "3" occurring in \( T_i \) are tested to determine which one satisfies Properties 14 and 15. Figure 13 presents these 4 different cases. For each case, the occurrences of "3" with strong lines satisfy the conditions. As depicted in the figure, two occurrences of "3" satisfy the conditions, therefore, "1 2 3 3" would have 2 occurrences in the input trees.

Exertion of the conditions presented in Properties 11-15 requires storing \( p\text{-coding}, m\text{-coding}, \text{cm-coding} \) and \( \text{position} \) of the rightmost node of each occurrence. After extending an occurrence \( O_k \) by \( O_N \), \( O_N \) will be the rightmost node of the resultant occurrence \( O_{k+1} \), therefore the encodings and the position of \( O_N \) will be assigned to \( O_{k+1} \). Our algorithm for frequency counting works very efficient: it can compute frequency of a candidate by storing only 4 integers per each occurrence.

The \text{OlInduced} \ algorithm takes as input an integer value \( \text{minsup} \) defined by the user and a forest of rooted ordered labeled trees in Zaki’s string representation format. The \( \text{minsup} \) value can be selected to be either per-tree or occurrence-match. \text{OlInduced} \ performs a breadth-first search in the state space of candidates and determines frequency of each candidates according to the before mentioned encodings. Figure 17 shows the high level pseudo code of \text{OlInduced}.

VI. EXPERIMENTAL RESULTS

We perform extensive experiments to evaluate the efficiency of the proposed algorithm using data from real applications as well as synthetic datasets. We do our experiments on a 1.8GHz Intel Pentium IV PC with 2GB main memory, running UNIX operating system. All the algorithms are implemented in C++ using standard template libraries. For our comparison, we select \text{IMB3Miner} [22] and \text{FREQT} [2] which are the well-known algorithms developed to find induced patterns from rooted ordered trees. \text{OlInduced}, \text{FREQT}, and \text{IMB3Miner} can work with both per-tree frequency and occurrence-match frequency. Here, due to lack of space, we only report results on occurrence-match frequency. Similar results can be obtained for per-tree frequency.

The widely used real dataset is CSLOGS [34]. This dataset contains the web access trees of the CS department of the
In [35], log file of each week is separated into a different dataset and three different datasets are generated: CSLOG1 for the first week, CSLOG2 for the second week and CSLOG3 for the third week. Furthermore, they generated a new dataset called CSLOG12 by combining CSLOG1 and CSLOG2. CSLOG1 contains 8,074 trees, CSLOG2 contains 7,404 trees, CSLOG3 contains 7,628 trees, and CSLOG12 contains 13,934 trees. Here, we use these datasets to evaluate our proposed algorithm. Figure 15 compares OInduced against iMB3Miner and FREQT over CSLOG1, CSLOG2, CSLOG3, and CSLOG12, respectively. Over all the datasets, OInduced significantly outperforms iMB3Miner and FREQT, especially for the lower values of minsup. For example, on CSLOG1 and at minsup = 10, OInduced works more than 18 times faster than FREQT and more than 20 times faster than iMB3Miner.

The second real dataset used in this paper is the Multicast dataset which consists of MBONE multicast data measured during the NASA shuttle launch between the 14th and 21st of February, 1999 [4]. It has 333 distinct vertices where each vertex takes the IP address as its label. The Multicast dataset was sampled from this NASA dataset with 10 minutes sampling interval and has 1,000 transactions. In this dataset, there exist strong correlations among transactions and very large frequent patterns occur even at a high minsup. Figure 16 compares performance of the algorithms over the Multicast dataset. On this dataset, OInduced outperforms the other algorithms, especially; it significantly outperforms the iMB3Miner algorithm. For example, at minsup = 750, OInduced works more than 5 times faster than FREQT and more than 20 times faster than iMB3Miner.

We also evaluate the efficiency of OInduced using synthetic

---

**OInduced**

1. **Require:** a database $D$ consisting of rooted ordered labeled trees, a user defined minsup (either per-tree or occurrence-match).
2. **Ensure:** All frequent induced tree patterns.
3. **Output** $\rightarrow \emptyset$.
4. $F1\_SET \leftarrow \text{the set of all frequent nodes and their encodings.}$
5. $F2\_SET \leftarrow \emptyset$.
6. **while** $F1\_SET \neq \emptyset$ **do**
7. **for all** $P_k \in F1\_SET$ **do**
8. $Ext \leftarrow \text{Extend}(P_k)$.
9. **for all** $P_{k+1} \in Ext$ **do**
10. **if** $\text{support}(P_{k+1}) \geq \text{minsup}$ **then**
11. $F2\_SET \leftarrow F2\_SET \cup P_{k+1}$.
12. **end if**
13. **end if**
14. **end for**
15. $Output \leftarrow Output \cup F1\_SET$.
16. $F1\_SET \leftarrow F2\_SET$.
17. $F2\_SET \leftarrow \emptyset$.
18. **end while**
19. **return** $Output$.  

---

Fig. 14: High level pseudo code of OInduced.
datasets which are generated by the method described in [34].

The synthetic data generation program mimics the web site browsing behavior of the user. First a master web site browsing tree is built and then the subtrees of the master tree are generated. The synthetic tree generation program is adjusted by 5 parameters: 1) the number of labels ($N$), 2) the number of nodes in the master tree ($M$), 3) the maximum fan-out of a node in the master tree ($F$), 4) the maximum depth of the master tree ($D$), and 5) the total number of trees in the dataset ($T$).

The first synthetic dataset is $D_{10}$ and uses the following default values for the parameters: $N = 100$, $M = 10,000$, $D = 10$, $F = 10$, $T = 100,000$. Figure 17a compares the running time of the algorithms on $D_{10}$. As depicted in the figure, $OInduced$ always outperforms $iMB3Miner$ and $FREQT$.

We generate $F_{3}$ as a narrow dataset and set all values to the default expect for $F = 3$. As depicted in Figure 17b, over this dataset $OInduced$ works faster than $iMB3Miner$ and $FREQT$, and $FREQT$ outperforms $iMB3Miner$. For example at $\text{minsup} = 100$, $OInduced$ outperforms $FREQT$ by a factor of 4 and outperforms $iMB3Miner$ by a factor of 8.

In $N_{1M}$, $N$ is set to 1,000,000, so the average frequency of distinct labels becomes very low (i.e. $M ÷ N = 10,000 ÷ 1,000,000 = 0.01$). Figure 17c presents the efficiency of $OInduced$ against $iMB3Miner$ and $FREQT$ over $N_{1M}$. Similar to the previous comparisons, $OInduced$ outperforms the other algorithms.

To study how the algorithms behave on very large datasets, we compare them on $T_{1M}$. For $T_{1M}$, the parameters are set as follows: $N = 100$, $M = 10,000$, $D = 10$, $F = 10$, $T = 1,000,000$. Figure 17d compares $OInduced$ against $iMB3Miner$ and $FREQT$ over $T_{1M}$. As depicted in the figure, $OInduced$ always outperforms $iMB3Miner$ and $FREQT$.

Finally, to show how the algorithms scale, we generate three datasets with different sizes (different values for $T$), while the other parameters are set to the default values. At a fixed $\text{minsup}$ (i.e. 2), as depicted in Figure 18, we can see a linear increase in both running time and the number of patterns with increasing the number of trees for $OInduced$, $iMB3Miner$ and $FREQT$. $OInduced$ is more efficient than $iMB3Miner$ and
FREQT. Both of horizontal and vertical axes in Figure 18 are depicted in logarithmic scale.

VII. CONCLUSION

In this paper, we introduced OInduced to discover all frequent induced patterns from a collection of rooted, ordered and labeled trees. OInduced uses breadth-first search to generate candidates and takes advantage of equivalence classes to extend each candidate by only known frequent candidates. Then, an indexing scheme is used to improve the breadth-first equivalence class extension. We also presented two new tree encodings, m-coding and cm-coding, which are based on combined depth-first/breadth-first traversals of input trees. OInduced benefits from these encodings to restrict the nodes of input trees and quickly compute frequencies of candidates. We compared OInduced with the well-known algorithms, MB3Miner and FREQT. Experiments on both real and synthetic data show that OInduced significantly reduces the running time and scales linearly with respect to the size of input trees.

ACKNOWLEDGMENT

We are grateful to the reviewers of the paper for their thoughtful efforts and useful comments. We would thank to Professor Mohammed Javeed Zaki for providing the CSLOGS dataset, the datasets used by Xrule, and the TreeGenerator program. We are also thankful to Dr Henry Tan and Dr Fedja Hadzic for providing us the iMB3Miner source code. Finally we are thankful to Taku Kudo for making his FREQT implementation available online.

Mostafa Haghir Chehreghani was partially supported by the ERC Starting Grant 240186 ‘MiGraNT’.

REFERENCES

[31] X. Yan and J. Han, CloseGraph: Mining Closed Frequent Graph Patterns, ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, August 2003.
Fig. 18: Scale up comparison. Minimum-support is equal to 2. Both of horizontal and vertical axes are in logarithmic scale.

Mostafa Haghir Chehreghani received his BSc in Computer Engineering from Iran University of Science and Technology in 2004, and his MSc from University of Tehran in 2007. From January 2010, he joined as a PhD student to the department of Computer Science, Katholieke Universiteit Leuven. His research interests include data mining, database systems, and mining and learning with graphs and networks.

Morteza Haghir Chehreghani received his BSc in Computer Engineering from Amirkabir University of Technology in 2005. He finished his MSc at Sharif University of Technology in Dec. 2008. Now, he is doing his PhD on stability analysis of combinatorial structures. His research interests include areas in machine learning, statistical physics, and data mining.

Caro Lucas received the M.S. degree from the University of Tehran, Iran, in 1973 and the Ph.D. degree from the University of California, Berkeley, in 1976. He is a Professor in the Department of Electrical and Computer Engineering (ECE), University of Tehran, Iran, as well as a Researcher at the School of Intelligent System (SIS), Institute for Studies in Theoretical Physics and Mathematics (IPM), Tehran, Iran. He served as the Director of SIS (1993-1997), Chairman of the ECE Department at the University of Tehran (1986-1988), Managing Editor of the Memories of the Engineering Faculty, University of Tehran (1979-1991), Reviewer of Mathematical Reviews (since 1987), Associate Editor of journal of Intelligent and Fuzzy System (1995-1999), and Chairman of the IEEE, Iran Section (1990-1992). He was also a Visiting Associate Professor at the University of Toronto (Summer, 1989-1990), University of California, Berkeley (1988-1989), an Assistant Professor at Garyounis University (1984-1985), University of California, Los Angeles (1975-1976), a Senior Researcher at the International Center for Theoretical Physics and the International Center for Genetic Engineering and Biotechnology, both in Trieste Italy, the Institute of Applied Mathematics, Chinese Academy of Sciences, Harbin Institute of Electrical Technology, a Research Associate at Manufacturing Research Corporation of Ontario, and a Research Assistant at the Electronic Research Laboratory, University of California, Berkeley. He holds a patent on speaker independent Farsi isolated word neurorecognizer. His research interests include biological computing, computational intelligence, uncertain systems, intelligent control, neural network, multiagent systems, data mining, business intelligence, financial modeling, image processing, and knowledge management. Dr. Lucas has served as Chairman of several international conferences. He was the founder of the SIS, and has assisted in founding several new research organizations and engineering disciplines in Iran. He is the recipient of several research grants at the University of Tehran and SIS.

Masoud Rahgozar is an Independent Consultant and an Assistant Professor of Computer Science in the Faculty of Engineering, University of Tehran, Iran. Previously, he worked for French software house companies as R&D manager, Senior Consultant, etc., for about 20 years. His interests include modernization of legacy applications and designing CASE tools for object-oriented programming and database normalization. He has received his MSc and PhD on database systems from Paris-6 University in France.