Efficiently Mining Unordered Trees

Mostafa Haghir Chehreghani
Department of Computer Science
Katholieke Universiteit Leuven, Belgium
Mostafa.HaghirChehreghani@cs.kuleuven.be

Abstract—Frequent tree patterns have many applications in different domains such as XML document mining, user web log analysis, network routing and bioinformatics. In this paper, we first introduce three new tree encodings and accordingly present an efficient algorithm for finding frequent patterns from rooted unordered trees with the assumption that children of every node in database trees are identically labeled. Then, we generalize the method and propose the UITree algorithm to find frequent patterns from rooted unordered trees without any restriction. Compared to other algorithms in the literature, UITree manages occurrences of a candidate tree in database trees more efficiently. Our extensive experiments on both real and synthetic datasets show that UITree significantly outperforms the most efficient existing works on mining unordered trees.

Keywords—Frequent tree patterns, rooted unordered trees, tree encoding, candidate generation, frequency counting.

I. INTRODUCTION

The problem of finding frequent patterns from a database of graphs has several important applications in different areas like bioinformatics, user web log analysis, web mining and network routing. It is also a fundamental problem in many other data mining tasks such as association rule mining, classification and clustering. Although in recent years finding *condensed representations* of frequent patterns (for example *closed* patterns) has found more interest, developing efficient algorithms for finding frequent patterns is still important, because the efficiency of the algorithms of finding condensed representations depends on the efficiency of the frequent pattern mining algorithms.

Trees are an important class of graphs having many applications such as XML documents, World Wide Web and computer vision. In this paper, we focus on rooted unordered trees.

Several algorithms are in the literature for finding frequent patterns from rooted unordered trees. Asai et al. and Nijssen et al. proposed *Unot* [2] and *uFreqt* [14], respectively. For frequency counting, Unot uses an occurrence list based approach in which each occurrence is stored in $O(L)$ space (L is the size of the candidate 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 candidate. The more recent well known algorithm for mining induced patterns is HybridTreeMiner [6] which uses the breadth-first canonical form. However,

for frequency counting, HybridTreeMiner deals with the occurrence-list of every candidate. PathJoin [25] assumes that children of every node are labeled identically and finds maximal patterns. More recently, UNI3 [8] was proposed for unordered candidate generation, however, it counts ordered frequency of every candidate.

In [4], the authors introduced two tree encodings for rooted ordered trees and accordingly, developed an efficient algorithm for finding frequent patterns from rooted ordered trees. However, the method of [4] can not easily be extended to mine unordered trees. In this paper, we follow the idea of storing a few encodings, instead of dealing with the occurrences of a candidate tree in a database tree, and perform the frequency counting step by means of efficient comparisons over these encodings. Our key contributions are as follows:

- We introduce three novel tree encodings, and accordingly, develop a new frequency counting algorithm for mining unordered trees with unique labels on children of every node.
- Then, we generalize the method and propose the UITree algorithm to find frequent patterns from rooted unordered trees without any restriction.
- We perform extensive experiments on both synthetic and real datasets and show that UITree significantly outperforms the well-known algorithms in the literature.

The rest of this paper is organised as follows. In Section II, preliminaries and definitions related to the problem of finding frequent tree patterns are given. Section III describes the proposed algorithm including the candidate generation method, the new tree encodings, and the frequency counting method. We experimentally evaluate the efficiency of *UITree* and compare it against the existing efficient methods in Section IV. In Section V, we have a brief overview on the related works. Finally, in Section VI, the paper is concluded and some possible extensions of the current work are presented.

II. PRELIMINARIES AND PROBLEM STATEMENT

A rooted labeled unordered tree $T = (V, E, L)$ is a connected DAG with V as the set of nodes, $E = \{(u, v) | u, v \in V\}$ as the set of edges, and L as the set of labels. A distinguished node r is considered as the root, and for any other node v , there is a unique path from r to v . Further,

there is a function $\lambda : V \rightarrow L$ which maps nodes to labels. A rooted labeled ordered tree has a left-to-right ordering among the siblings.

The *degree* of a node v in a tree T , denoted by $\deg(T, v)$, is defined as the number of edges incident to v . The *depth* of a node v in a tree T , denoted by $\text{dep}(T, v)$, is the length of the path from the root to v . We say the node v is in the *position* n of a path p , if $\text{dep}(T, v)$ is equal to n .

For a labeled tree T , Zaki's string representation, denoted by $S(T)$ is defined as follows: the labels of the nodes of T are added to $S(T)$ in the preorder traversal of T , and when a backtracking from a child to its parent is occur a unique symbol (e.g. -1) is added to $S(T)$ [28].

For a rooted labeled unordered tree $T = (V, E, L)$, a rooted labeled unordered tree $T' = (V', E', L')$ is an induced subtree of T , also called T' is *isomorphic* to a subtree of T , if and only if: (1) $V' \subseteq V$, (2) $E' \subseteq E$, and (3) $L' \subseteq L$ and the labeling of V' in T is preserved in T' . An *occurrence* O' of T' in T is the set of nodes and edges of T to which T' is mapped. We note T' might have zero, one, or several occurrences in T . T' is a subtree isomorphic of T if and only if T' has at least one occurrence in T . Two different occurrences of T' in T might share some nodes (and edges) in common, but they can not be the same, entirely.

Given a database $\mathcal{D}$ consisting of rooted labeled unordered trees and a subtree S , the *per-tree support* (or *per-tree frequency*) of S is defined as the number of trees in $\mathcal{D}$ 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 $\mathcal{D}$. More formally:

$$\text{per-tree support}(S) = \sum_{T \in \mathcal{D}} \text{IsInduced}(S, T)$$

and

$$\text{occurrence-match support}(S) = \sum_{T \in \mathcal{D}} \text{OccNum}(S, T)$$

where $\text{IsInduced}(S, T)$ is 1 if S is the induced subtree of the database tree T otherwise 0, and $\text{OccNum}(S, T)$ returns the number of occurrences of S in T . A tree is frequent if its *per-tree support* (*occurrence-match support*) is greater than or equal to a user-specified *per-tree* (*occurrence-match*) *min_sup* value.

The desired class of patterns can differ based on the kind of the application. In this paper, our concern is frequent induced unordered tree patterns. Both of per-tree frequency and occurrence-match frequency are allowed in this work, but we mainly concentrate on occurrence-match frequency. Through this paper, for simplicity we use the term of frequency (support) instead of occurrence-match frequency (occurrence-match support); unless we explicitly say that frequency (support) refers to per-tree frequency (per-tree support).

III. MINING UNORDERED TREES

In this section, first, we present the candidate generation method. Then, we introduce three new tree encodings: *s-coding*, *is-coding* and *cs-coding*, which are assigned to nodes of database trees. Next, we present a frequency counting method for unordered trees having unique labels on children of every node. Next, we generalize the frequency counting method for all rooted unordered trees. Finally, we present the UITree algorithm.

A. Candidate generation

For candidate generation, we use a variation of the well-known *rightmost path extension* method. Let C be a tree having k nodes. We call it a k -tree. Using *rightmost path extension*, C can be extended to generate new trees in two different ways. In the first way, called *rn_extension*, a new node N is added to the rightmost node of C . In the second way, called *rp_extension*, a new node N is added to a node in the rightmost path of C , except its rightmost node [4]. In both cases, N will be the rightmost node of the generated $k + 1$ -tree. For ordered trees, this method of candidate generation is complete as well as non-redundant, however, for unordered trees, it might generate redundant trees. Duplicate candidates can be detected and ignored by means of a *canonical form*.

For *rp_extension*, we use the technique proposed in [29] to make the candidate generation method more efficient.

Definition 1: Two k -trees C_1 and C_2 are in the same *equivalence class*, denoted by $C_1 \simeq C_2$, if they share the first $k - 1$ nodes in common.

Definition 2: Assume that C_1 and C_2 are two k -trees and $C_1 \simeq C_2$. Let v_2 be the rightmost node of C_2 and v_1 be the node in the position $\text{dep}(C_2, v) - 1$ of the rightmost path of C_1 . The *join* of C_1 and C_2 , denoted by $C_1 \odot C_2$ is a $k + 1$ -tree C' generated by adding v_2 to C_1 as the rightmost child of v_1 .

Clearly, the join operation is defined if there is a node in the position $\text{dep}(C_2, v) - 1$ of the rightmost path of C_1 , and this means that the following condition should be held: $\text{dep}(C_1, n_1) \leq \text{dep}(C_2, n_2)$, where n_1 and n_2 are the rightmost nodes of C_1 and C_2 , respectively.

We note that the join operation defined in Definition 2 is not commutative, i.e. $C_1 \odot C_2$ and $C_2 \odot C_1$ might be different trees, or only one of them might be defined.

In summary, our method for extending a candidate C is as the following:

- *rp_extension* of C : for all trees C' such that $C \simeq C'$ (including C itself), $C \odot C'$ is generated as a new candidate.
- *rn_extension* of C : every frequent label is added to the rightmost node of C as a child and generates a larger candidate.

In the rest of this paper, the *rightmost path* and the *rightmost node* of an *unordered candidate tree* refer respectively

to the last path and the last node appended to the candidate in its extension chain.

B. Three new encodings

Let T be a database tree and $\prec$ be a left-to-right ordering among children of nodes of T . The *ordered form* of T with respect to $\prec$, denoted by $T_{\prec}$, refers to the **ordered** tree resulted by applying $\prec$ to the nodes of T .

s-coding

Definition 3: Let T be a database tree, $\prec$ be a left-to-right ordering among children of nodes of T , and sid be an auxiliary integer initiated by 0. For every node $v \in V(T)$, its *s-coding* with respect to $\prec$, denoted by $SC_{\prec}(T, v)$ is defined as the following:

- 1) if v is the root of T , $SC_{\prec}(T, v)$ is defined equal to 0.
- 2) $T_{\prec}$ is traversed in preorder and when a node v is met, if $deg(T_{\prec}, v) > 1$, sid is increased by 1, and for any child u of v , $SC_{\prec}(T, u)$ is defined equal to the new value of sid .

All children of a node have the same *s-coding*. Since the nodes of the tree are traversed in preorder, when assigning *s-coding* to the children of a node, its *s-coding* has already been determined. Figure 1 shows an example of *s-coding*. In this figure, T_0 and T_1 are *ordered forms* of two database trees, and the trees below them show the *s-codings* of their nodes.

Figure 1: An example of s-coding.

is-coding

First, we introduce the *r-preorder traversal*.

Definition 4: *R-preorder* traversal of a rooted ordered tree is a depth-first traversal which is defined recursively as follows:

- 1) visit the root first; and then
- 2) do an r-preorder traversal each of the subtrees of the root one-by-one from right to left.

In r-preorder traversal, the children are met from right to left which is the key difference compared to the preorder traversal.

Definition 5: Let T be a database tree, $\prec$ be a left-to-right ordering among children of nodes of T , and sid be an auxiliary integer initiated by 0. For every node $v \in V(T)$, its *is-coding* with respect to $\prec$, denoted by $IS_{\prec}(T, v)$ is defined as follows:

- 1) if v is the root of T , $IS_{\prec}(T, v)$ is defined equal to 0.
- 2) $T_{\prec}$ is traversed in r-preorder and when a node v is met, if $deg(T, v) > 1$, sid is increased by 1, and for any child u of v , $IS_{\prec}(T, u)$ is defined equal to the new value of sid .

s-coding and *is-coding* only differ in the way of traversing ordered forms of database trees. Figure 2 shows the is-encoding of the ordered forms of the database trees presented in Figure 1.

Figure 2: An example of is-coding.

cs-coding

Definition 6: Let $\prec$ be a left-to-right ordering among children of nodes of T . The *cs-coding* of a node v in a database tree T with respect to $\prec$, denoted by $CS_{\prec}(T, v)$, is defined as $SC_{\prec}$ of its children.

When a node is met in preorder traversal of the tree, the *s-codings* of its children are assigned, therefore *cs-coding* of all nodes (with respect to an ordering $\prec$) can be determined in $O(|V(T)|)$. Figure 3 shows the cs-encoding of the ordered forms of the database trees presented in Figure 1.

Figure 3: An example of cs-coding.

C. The frequency counting method

Each candidate C_{k+1} can be generated by either `rp_extension` or `rn_extension`. In this section, we present new frequency counting methods for `rp_extended` candidates

and $rn_extended$ candidates, based on the tree encodings presented in Section III-B.

The following notions and notations are used in this section: we assume that C_k and C'_k are two trees belonging to the same equivalence class, and $C_k \odot C'_k$ is possible and it generates the tree C_{k+1} . C_{k+1} is generated by adding the node N to C_k . We also assume that the occurrence O_k of C_k , the occurrence O_N of N , and the occurrence O_{k+1} of C_{k+1} all occur in the database tree T . CN refers to the rightmost node of C_k and CP refers to the rightmost path of C_k excluding its rightmost node, i.e. $CP \cup CN$ forms the rightmost path of C_k . O_{CN} is the node in O_k corresponding to the node CN of C_k , and O_{CP} is the path in O_k corresponding to CP . We note that since C_k is an unordered occurrence of C_k , it is possible that O_{CN} and O_{CP} are not the rightmost node and the rightmost path of O_k , respectively.

Frequency counting for $rp_extended$ candidates

C_{k+1} is generated by adding node N to C_k via $rp_extension$ and we want to see whether or not adding O_N to a node in O_{CP} generates an occurrence O_{k+1} of C_{k+1} . T can be divided into the partitions depicted in Figure 4. $B1$ is the path between the root of T and the root of O_k . C contains the children of the nodes of $B1$ and the children of the nodes of O_{CP} . $B2$ is the path between O_{CN} and z , where z is the last node met before O_{CN} in preorder traversal of T . To generate an occurrence O_{k+1} of C_{k+1} , O_N must belong to the dotted region depicted in Figure 4. For this purpose, O_N must satisfy Properties 1, 2 and 3.

Figure 4: A partitioning of a database tree.

Property 1: $SC_{\prec}(O_N) \leq SC_{\prec}(O_{CN})$.

Property 2: $IS_{\prec}(O_N) \leq IS_{\prec}(O_{CN})$.

where $\prec$ is any arbitrary ordering on the nodes of the database tree.

O_N can be anywhere in T . It can be shown that if Property 1 is applied on O_N , it can not be selected from the grey parts of Figure 5(a). We also can show that Property 2 expels O_N from the grey parts of Figure 5(b). The intersection of non-grey parts of Figures 5(a) and 5(b) is the region C , i.e. applying Properties 1 and 2 to O_N , restricts it to C . Property 3 restricts O_N to the dotted region, which is the desired region.

Property 3: $dep(T, O_N) - dep(T, O_{CN}) = dep(C_k, N) - dep(C_k, CN)$.

If the nodes in a database tree are allowed to have duplicate labels, but the labels for the children of every node are assumed to be unique, Properties 1, 2 and 3 are sufficient to count frequencies of $rp_extended$ candidates. Assume that C_{k+1} is generated by adding node N to C_k via $rp_extension$. To see whether adding O_N to the rightmost node of O_{C_k} generates O_{k+1} , it is sufficient to check if:

- 1) O_N and O_k occur in the same database tree,
- 2) $SC_{\prec}(O_N) \leq SC_{\prec}(O_{CN})$,
- 3) $IS_{\prec}(O_N) \leq IS_{\prec}(O_{CN})$, and
- 4) $dep(T, O_N) - dep(T, O_{CN}) = dep(C_k, N) - dep(C_k, CN)$.

($\prec$ is any arbitrary ordering on the nodes of database trees.)

If these conditions are satisfied, O_{k+1} is generated by attaching O_N to O_k . s -coding, is -coding and cs -coding of O_N are stored for O_{k+1} to be used in future extensions of O_{k+1} .

As an example, consider tree "b b -1 c" which is generated by $b \ b \odot \ b \ c$. As depicted in Figure 6, "b b" has one occurrence in T_0 and two occurrences in T_1 . Node "c" occurs for 5 times in T_0 and for 4 times in T_1 . Figure 6 shows how the occurrences of "b b -1 c" can be determined by means of occurrences of "b b" and "c". Left side trees highlight the occurrences of "b b" and right side trees show different occurrences of "c". Texts on the side of the hachured occurrences of "c" present the unsatisfied properties. Other occurrences of "c", depicted by double circles, are occurrences that satisfy all the conditions; therefore they can join with an occurrence of "b b" and generate an occurrence of "b b -1 c".

Frequency counting for $rn_extended$ candidates

The following theorem can help us to count frequencies of $rn_extended$ candidates either when database trees have unique labels on the children of every node or when they are general unordered trees.

Theorem 1: Assume that C_k is generated by adding node N to C_k via $rn_extension$. To see whether adding O_N to the rightmost node of O_k generates O_{k+1} , it is sufficient to check if:

- 1) O_N and O_k occur in the same database tree, and

Figure 5: Restrictions applied by Properties 1, 2 and 3 on the nodes of the database tree T .

Figure 6: An example of frequency counting for $rp_extended$ candidates.

$$2) \mathcal{CS}_{\prec}(O_{CN}) = \mathcal{SC}_{\prec}(O_N).$$

where $\prec$ is any arbitrary ordering among the nodes of the database tree.

If the conditions of Theorem 1 are satisfied, O_{k+1} is generated and the s -coding, is -coding and cs -coding of O_N are stored for O_{k+1} to be used in future extensions of O_{k+1} .

Figure 7 shows an example of frequency counting for $rn_extended$ candidates. $Rn_extension$ of "a b" by means of "c" generates the tree "a b c". As depicted in Figure 7, "a b" has one occurrence in T_0 and one occurrence in T_1 . For each occurrence, only one occurrence of "c" satisfies both conditions. Therefore, "a b c" have two occurrences in the database trees.

D. Generalizing the Frequency Counting Method

In this section, we assume that the database trees are general trees and their nodes can have children with the same label. For frequency counting of $rn_extended$ candidates, this generalization does not make any problem; however, for $rp_extended$ candidates we must revise the frequency counting algorithm.

Figure 7: An example of frequency counting for $rn_extended$ candidates.

For generalized $rp_extended$ frequency counting, it is possible that O_N belongs to the dotted region of Figure 4 and it has already been appended to O_k . In this case, the above mentioned frequency counting method is not enough restricting for selecting correct occurrences of O_N .

Definition 7: Tree candidate C' is *self-extended* if it is generated by $rp_extension$ of a candidate C'' with itself. In the consequence of this extension, which is called *self-extension*, the rightmost node and the second rightmost node¹ of the resultant candidate find the same label. This same label is called *self-extension label*.

Theorem 2: For self-extended candidates, to guarantee that O_N has not already been appended to the dotted region of C_k , more than the 4 conditions already mentioned, the following condition should also be held:

$$\mathcal{P}(T_{\prec}, O_{CN}) < \mathcal{P}(T_{\prec}, O_N)$$

where for a node $v \in V(T_{\prec})$, $\mathcal{P}(T_{\prec}, v)$ is the preorder number of v in $T_{\prec}$.

In this case, in order to correctly count frequencies of candidates extended from C_k , both the encodings of O_N and the encodings of O_{CN} are stored for C_k , if they have not already been stored.

¹The second rightmost node is the rightmost node of the tree generated from removing the rightmost node.

As an example, consider Figure 8 in which the database tree is "c b d -1 a -1 b -1 a -1 b -1" and we want to compute the frequency of "c b d -1 b -1 b -1". "c b d -1 b -1 b -1" is a self-extended candidate which is generated by appending node "b" to "c b d -1 b -1". "c b d -1 b -1" has two occurrences in the database tree. For the first occurrence, the rightmost occurrence of "b" satisfies the condition of Theorem 2 as well as all of the 4 conditions already mentioned for *rp_extended* candidates. For the second occurrence, none of the occurrences of "b" satisfy the condition of Theorem 2. Therefore, "c b d -1 b -1" has one occurrence in the database tree.

Figure 8: An example of generalized frequency counting for self-extended candidates.

Theorem 2 solves the problem for self-extended candidates, however, the problem still remains for other *rp_extended* candidates.

Assume that candidate $C_{k+1} = C_k \odot C'_k$ is generated via *rp_extended*. In every occurrence O'_k of C'_k , we refer to the image of N by $O_N : O'_k$. Without going into details and proofs (due to lack of space), we present Theorem 3 for frequency counting of *rp_extended* candidates which are not self-extended.

Theorem 3: Frequency of a *rp_extended* candidate $C_{k+1} = C_k \odot C'_k$ (C_{k+1} is not self-extended) in a database tree T can be correctly computed as the following:

for every two occurrences O_k of C_k and O'_k of C'_k in T , if:

- 1) $O_N : O'_k$ is not the rightmost node of any already generated occurrence of C_{k+1} ,²
- 2) $\mathcal{CS}_{\prec}(O_N : O'_k) \neq \mathcal{SC}_{\prec}(O_{CN})$,
- 3) $\mathcal{SC}_{\prec}(O_N : O'_k) \leq \mathcal{SC}_{\prec}(O_{CN})$,

²Since the occurrences of a candidate tree can be generated in an *ordered* manner, this condition can be tested very efficiently by only 1 comparison. Due to lack of space we omit details.

$$4) \mathcal{IS}_{\prec}(O_N : O'_k) \leq \mathcal{IS}_{\prec}(O_{CN}), \text{ and}$$

$$5) \text{dep}(T, O_N : O'_k) - \text{dep}(T, O_{CN}) = \text{dep}(C_k, N) - \text{dep}(C_k, CN).$$

an occurrence O_{k+1} of C_{k+1} is generated ($\prec$ is an arbitrary ordering on the nodes of database trees).

For example, in Figure 9 "c b d -1 -1 b -1 b -1" and "c b d -1 -1 b -1 a -1" join together and generate "c b d -1 -1 b -1 b -1 a -1". "c b d -1 -1 b -1 a -1" has 4 occurrences in the database tree "c b d -1 -1 a -1 b -1 a -1 b -1". In the first and third occurrences, occurrence "a" satisfies all the above mentioned conditions. In the second occurrence, occurrence "a" is the same as in the first occurrence, and in the fourth occurrence, occurrence "a" is the same as in the third occurrence. Therefore, occurrence "a" in the second and fourth occurrences can not participate in the join operation.

Figure 9: An example of frequency counting for non self_extended candidates in which nodes might have children with the same labels.

E. The UITree Algorithm

Figure 10 shows the high level pseudo code of the UITree algorithm. First, the set of frequent labels (frequent patterns of size 1) are generated. Then, larger patterns are generated via recursive calls of the REFINEMENT method.

In the REFINEMENT method, a pattern is extended if it is in the *canonical form*. Here, we use the canonical form presented in [29]. If a pattern C is in the *canonical form*, it is *rn_extended* using frequent labels and frequency of every resultant candidate tree is computed according to Theorem 1; and C is *rp_extended* using patterns which are equivalent to it and frequency of every resultant candidate tree is computed according to Theorems 2 and 3.

To guarantee the completeness of the candidate generation step, C is *rp_extended* by every pattern C' which is in the equivalence class of C (either C' is in the *canonical form* or is not). In this way of candidate generation, only

known frequent patterns are used to $\text{rp_extend } C$ (C itself is frequent). In this method, candidates which are frequent but are not in the *canonical form* are used for rp_extension of other candidates.

Another possible way for candidate generation is generating only candidates which are in the *canonical form* and rp_extending them by all frequent labels. In this way, it is possible that a candidate which is already known to be infrequent (since it has an infrequent subtree), is generated.

As discussed in [29], there is a trade-off between the number of redundant candidates (which are not in the *canonical form*) generated in the first method, and the number of already known infrequent candidates generated in the second method. In [29] these two different methods were experimentally evaluated, and it was found that the first method of candidate generation is more efficient than the second method. We note that at the end, both methods generate the same set of frequent patterns.

UITREE (a database $\mathcal{D}$ consisting of rooted unordered trees, an integer min_sup)

```

1:            $\triangleright \mathcal{F}$  refers to the set of frequent patterns
2:  $\mathcal{C} \leftarrow$  the set of frequent labels
3: for all  $C \in \mathcal{C}$  do
4:   REFINEMENT( $C$ )
5: end for
```

REFINEMENT (a tree C)

```

1: if  $C$  is in the canonical form then
2:    $\mathcal{F} \leftarrow \mathcal{F} \cup C$ 
3:                                      $\triangleright \text{rn\_extension:}$ 
4:   for all frequent label  $N$  do
5:     generate  $C_{k+1}$  by adding  $N$  to the rightmost
       node of  $C$ 
6:     compute  $\text{support}(C_{k+1})$  using Theorem 1
7:     if  $C_{k+1}$  is frequent then
8:       REFINEMENT( $C_{k+1}$ )
9:     end if
10:  end for
11:                                      $\triangleright \text{rp\_extension:}$ 
12:  for all  $C'$  such that  $C \simeq C'$  do
13:     $C_{k+1} \leftarrow C \odot C'$ 
14:    compute  $\text{support}(C_{k+1})$  using Theorems 2 and
       3
15:    if  $C_{k+1}$  is frequent then
16:      REFINEMENT( $C_{k+1}$ )
17:    end if
18:  end for
19: end if
```

Figure 10: High level pseudo code of UITree.

IV. 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 2.93GHz Intel Pentium IV PC with a 8GB main memory, running UNIX operating system. All the algorithms are implemented in C++ using standard template libraries.

The well-known algorithm in the literature for mining unordered trees is *HybridTreeMiner* [6]. [7] and [3] have been reported to be more efficient, but they find frequent *closed* and *maximal* patterns. Here, we select *HybridTreeMiner* for our comparisons.

The CSLOGS dataset [30] is a real dataset which contains the web access trees of the CS department of the Rensselaer Polytechnic Institute during one month. This dataset has 59,691 transactions, 716,263 nodes and 13,209 unique vertex labels. Each distinct label corresponds to the URLs of a web page.

In [31], the log file of every week is separated into a different dataset and 3 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. We use these real datasets to evaluate the efficiency of the proposed algorithm.

Figure 11 compares UITree against HybridTreeMiner over CSLOG1. As depicted in the figure, UITree significantly outperforms HybridTreeMiner, for example in $\text{min_sup} = 13$, UITree outperforms HybridTreeMiner by a factor of 300 (the left side diagram). Two algorithms use almost equal amounts of memory (the right side diagram).

Figures 12, 13 and 14 compare UITree against HybridTreeMiner over CSLOG2, CSLOG12 and CSLOG3, respectively. Over CSLOG2, in $\text{min_sup} = 10$, HybridTreeMiner can not generate the output after several hours. In $\text{min_sup} = 12$, UITree works more than 250 times faster than HybridTreeMiner. Over CSLOG12, for the values of min_sup smaller than 21, HybridTreeMiner can not generate the output after a reasonable time (several hours). In $\text{min_sup} = 21$, UITree works more than 200 times faster than HybridTreeMiner. Over CSLOG3, for the values of min_sup smaller than 15, HybridTreeMiner can not generate the output after a reasonable time. In $\text{min_sup} = 15$, UITree works more than 150 times faster than HybridTreeMiner.

We also evaluate the efficiency of the proposed algorithm using synthetic datasets which are generated by the method described in [30]. 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

Figure 11: Comparison over CSLOG1.

Figure 14: Comparison over CSLOG3.

Figure 12: Comparison over CSLOG2.

Figure 15: Comparison over D10.

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 D10 and uses the following default values for the parameters: $N = 100$, $M = 10,000$, $D = 10$, $F = 10$, $T = 100,000$. Figure 15 compares two algorithms on D10. As depicted in the figure, UITree always outperforms HybridTreeMiner.

F3 uses the default value expect for $F = 3$. As depicted in Figure 16, over this dataset, UITree works faster than HybridTreeMiner. M100K uses the default values expect for $M = 100,000$. Figure 17 reports the efficiency of UITree compared to HybridTreeMiner over this dataset.

In N10K, N is set to 10,000. In Figure 18, we compare two algorithms over this dataset, and as depicted in the diagram, UITree significantly outperforms HybridTreeMiner. For example, in $min_sup = 4$, UITree works more than 35

times faster than HybridTreeMiner.

V. RELATED WORKS

Recently, many algorithms have been proposed to find frequent patterns from a collection of database trees. Wang et al. proposed algorithms for discovering similar structure and structural association rules among a collection of tree structured data [22], [23] and [24].

Zaki introduced TreeMiner [28] to mine embedded ordered frequent tree patterns. For frequency counting, he used an efficient data structure, called scope-list, and proposed rightmost path extension to generate non-redundant candidates. Later, by proposing the SLEUTH algorithm, he extended the work to find embedded patterns from rooted unordered [29]. Asai et al. [1] independently proposed the rightmost candidate generation and developed FREQT for mining frequent induced ordered tree patterns. Chi et al. [5] proposed FreeTreeMiner for mining induced unordered free

Figure 13: Comparison over CSLOG12.

Figure 16: Comparison over F3.

Figure 17: Comparison over M100K.

Figure 18: Comparison over N10K.

trees. Other algorithms for mining induced unordered tree patterns are PathJoin [25], uFreqt [14], uNot [2] and HybridTreeMiner [6]. TreeFinder [20] uses an Inductive Logic Programming approach to mine unordered embedded trees but it is not a complete method and can miss some frequent subtrees. SingleTreeMining [15] is an algorithm for mining rooted unordered trees with application to phylogenetic tree pattern mining.

Chi et al. proposed CMTreMiner [7] for mining both closed and maximal frequent subtrees in a database of rooted unordered trees. This algorithm traverse an enumeration tree that systematically enumerates all subtrees. Xiao et al. [26] proposed TreeGrow for mining unordered maximal embedded tree patterns. However, TreeGrow assumes that the labels for the children of every node are unique. The candidate generation method is localized so as to avoid unnecessary computational overhead. [13] and [16] developed other algorithms for finding maximal embedded tree patterns.

XSpanner [21] is a pattern growth-based method and can mine embedded ordered subtrees. Tatikonda et al. [18] proposed a generic approach that can be used to mine embedded or induced subtrees that can be labeled, unlabeled, ordered, unordered, or edge-labeled. They developed TRIPS and TIDES algorithms using two sequential encodings of trees to systematically generate and evaluate the candidate patterns. Tan et al. [17] presented a unique embedding list representation of the tree structure, which enables efficient implementation of their Tree Model Guided (TMG) candidate generation. TMG can enumerate all the valid candidates that conform to the structural aspects of the input data.

UNI3 [8] generates unordered candidate while it performs ordered frequency counting. In [19], the authors developed efficient pattern mining algorithms specialized for multi-core hardwares.

For finding unordered frequent tree patterns, most of the proposed algorithms use a canonical form and extend only candidates that are in the canonical form. In [11] and [12], Luccio et al. defined sorted pre-order string canonical form can be obtained in linear time (assuming a finite alphabet for vertex labels). Later, Asai et al. [2], Nijssen et al. [14], and Chi et al. [5] independently defined similar canonical representations.

Efficient algorithms for mining frequent graph patterns which are the general form of frequent tree patterns can be found in [9], [10] and [27].

VI. CONCLUSION AND FUTURE WORKS

In this paper, we presented novel tree encodings, and accordingly, developed a new method for finding frequent patterns from rooted unordered trees with unique labels on children of every node. Then, we generalized the method to find frequent patterns from rooted unordered trees without any restriction. We empirically evaluated the efficiency of the proposed algorithm, UITree, and showed that it provides significant improvements against well-known algorithms in the literature, over both real and synthetic datasets.

One future work is extending the proposed algorithm to find condensed representations like frequent *closed* patterns. A useful technique which is widely used (in different forms) to trim the state space in the closed pattern mining problem, is the *early termination* or *early pruning* technique [7] and [27]. This technique can be performed more efficiently using the encodings and the frequency counting method presented in this paper. The other future work, can be developing similar techniques for finding frequent patterns from more complex graphs such as planar graphs, bounded-treewidth graphs, or general graphs.

ACKNOWLEDGMENT

We would thank to Dr Yun Chi for providing the HybridTreeMiner source code. We are also thankful to Professor Mohammed Javeed Zaki for providing the CSLOGS datasets and the TreeGenerator program. This work was partially supported by the ERC Starting Grant 240186 'Mi-GrANT'.

REFERENCES

- [1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto and S. Arikawa, *Efficient Substructure Discovery from Large Semi-Structured Data*, Proc. Second SIAM Intl Conf. Data Mining, pp. 158-174, Apr. 2002.
- [2] T. Asai, H. Arimura, T. Uno and S. Nakano, *Discovering Frequent Substructures in Large Unordered Trees*, Proc. Sixth Intl Conf. Discovery Science, pp. 47-61, Oct. 2003.

- [3] J. L. Balczar, A. Bifet, A. Lozano, *Mining Frequent Closed Rooted Trees*, Machine Learning 78(1-2), 1-33 (2010).
- [4] Mostafa H. Chehreghani, Morteza H. Chehreghani, C. Lucas, M. Rahgozar, *OHInduced: An Efficient Algorithm for Mining Induced Patterns From Rooted Ordered Trees*, IEEE Transactions on Systems, Man and Cybernetics (A), 41(5), pp. 1013-1025, 2011.
- [5] Y. Chi, Y. Yang and R.R. Muntz, *Indexing and Mining Free Trees*, Proc. Third IEEE Intl Conf. Data Mining (ICDM), pp. 509-512, 2003.
- [6] Y. Chi, Y. Yang, and R.R. Muntz, *HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms*, Proc. 16th Intl Conf. Scientific and Statistical Database Management (SSDBM), pp. 11-20, 2004.
- [7] Y. Chi, Y. Yang, Y. Xia, R. R. Muntz, *CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees*, PAKDD, pp. 63-73, 2004.
- [8] F. Hadzic and H. Tan and T. S. Dillon, *UNI3-Efficient Algorithm for Mining Unordered Induced Subtrees Using TMG Candidate Generation*, In Proceedings of IEEE International Symposium on Computational Intelligence and Data Mining (CIDM 07), 2007.
- [9] A. Inokuchi, T. Washio and H. Motoda, *An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data*, 4th European Conference on Principles of Knowledge Discovery and Data Mining, pp. 13-23, September 2000.
- [10] M. Kuramochi and G. Karypis, *Frequent Subgraph Discovery*, Proceedings of the IEEE International Conference on Data Mining (ICDM01), pp. 313-320, November 2001.
- [11] F. Luccio, A. M. Enriquez, P. O. Rieumont and L. Pagli, *Exact Rooted Subtree Matching in Sublinear Time*, Technical Report TR-01-14, Universita Di Pisa, 2001.
- [12] F. Luccio, A. M. Enriquez, P. O. Rieumont and L. Pagli, *Bottom-up Subtree Isomorphism for Unordered Labeled Trees*, Technical Report TR-04-13, Universita Di Pisa, 2004.
- [13] T. Miyahara, Y. Suzuki, T. Shoudai, T. Uchida, K. Takahashi, H. Ueda, *Discovery of Maximally Frequent Tag Tree Patterns with Contractible Variables from Semistructured Documents*, PAKDD, pp. 133-144, 2004.
- [14] S. Nijssen and J. N. Kok, *Efficient Discovery of Frequent Unordered Trees*, Proc. First Intl Workshop Mining Graphs, Trees, and Sequences, pp. 55-64, 2003.
- [15] D. Shasha, J. Wang and S. Zhang, *Unordered Tree Mining with Applications to Phylogeny*, Proc. Intl Conf. Data Eng. (ICDE), pp. 708-719, 2004.
- [16] Y. Suzuki, T. Miyahara, T. Shoudai, T. Uchida, Y. Nakamura, *Discovery of Maximally Frequent Tag Tree Patterns with Height-Constrained Variables from Semistructured Web Documents*, WIRI, pp. 104-112, 2005.
- [17] H. Tan, F. Hadzic, T. S. Dillon, E. Chang, L. Feng, *Tree Model Guided Candidate Generation for Mining Frequent Subtrees from XML Documents*, ACM Transaction on Knowledge Discovery from Data (TKDD), 2(2), pp. 9-43, 2008.
- [18] S. Tatikonda, S. Parthasarathy, T. M. Kur, *TRIPS and TIDES: New Algorithms for Tree Mining*, CIKM 06, pp. 455-464, 2006.
- [19] S. Tatikonda, S. Parthasarathy, *Mining Tree-Structured Data on Multicore Systems*, PVLDB 2(1), 694-705, 2009.
- [20] A. Termier, M. C. Rousset and M. Sebag, *TreeFinder: a First Step towards XML Data Mining*, Second IEEE International Conference on Data Mining (ICDM'02), p. 450, 2002.
- [21] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang and B. Shi, *Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining*, Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining, pp. 441-451, 2004.
- [22] K. Wang and H. Liu, *Schema Discovery for Semistructured Data*, In Proc. of the 3rd. International Conference on Knowledge Discovery and Data Mining (KDD03), pp. 271-274, California, USA, August 1997.
- [23] K. Wang and H. Liu, *Discovering Typical Structures of Documents: A Road Map Approach*, In Proc. of the ACM SIGIR International Conference on Research and Development in Information Retrieval (SIGIR), pp. 146-154, Melbourne, Australia, August 1998.
- [24] K. Wang and H. Liu, *Discovering Structural Association of Semistructured Data*, IEEE Transactions on Knowledge and Data Engineering, 12(2), pp. 353-371, 2000.
- [25] Y. Xiao, J. F. Yao, Z. Li and M. H. Dunham, *Efficient Data Mining for Maximal Frequent Subtrees*, Proc. Intl Conf. Data Mining (ICDM), pp. 379-386, 2003.
- [26] Y. Xiao, J. F. Yao, G. Yang, *Discovering Frequent Embedded Subtree Patterns from Large Databases of Unordered Labeled Trees*, International Journal of Data Warehousing and Mining, 1(2), pp. 70-92, 2005.
- [27] X. Yan and J. Han, *CloseGraph: Mining Closed Frequent Graph Patterns*, ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD), pp. 286-295, August 2003.
- [28] M. J. Zaki, *Efficiently Mining Frequent Trees in a Forest*, Proc of the Int. Conf. Knowledge Discovery and Data Mining (KDD), pp. 71-80, July 2002.
- [29] M. J. Zaki, *Efficiently Mining Frequent Embedded Unordered Trees*, Fundam. Inform. 66(1-2), pp. 33-52, 2005.
- [30] M. J. Zaki, *Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications*, IEEE Transaction on Knowledge and Data Engineering, 17 (8), pp. 1021-1035, 2005.
- [31] M. J. Zaki and C. C. Aggarwal, *XRules: An Effective Structural Classifier for XML Data*, Machine Learning, 62(1-2), pp. 137-170, 2006.