HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms*

Yun Chi, Yirong Yang, Richard R. Muntz
Department of Computer Science
University of California, Los Angeles, CA 90095
{ychi,yyr,muntz}@cs.ucla.edu

Abstract

Tree structures are used extensively in domains such as computational biology, pattern recognition, XML databases, computer networks, and so on. In this paper, we present HybridTreeMiner, a computationally efficient algorithm that discovers all frequently occurring subtrees in a database of rooted unordered trees. The algorithm mines frequent subtrees by traversing an enumeration tree that systematically enumerates all subtrees. The enumeration tree is defined based on a novel canonical form for rooted unordered trees–the breadth-first canonical form (BFCF). By extending the definitions of our canonical form and enumeration tree to free trees, our algorithm can efficiently handle databases of free trees as well. We study the performance of our algorithms through extensive experiments based on both synthetic data and datasets from real applications. The experiments show that our algorithm is competitive in comparison to known rooted tree mining algorithms and is faster by one to two orders of magnitudes compared to a known algorithm for mining frequent free trees.

keywords: *frequent subtree, canonical form, tree isomorphism, enumeration tree, rooted unordered tree, free tree.*

1. Introduction

Graphs are widely used to represent data and relationships. Among all graphs, a particularly useful family is the family of trees. Trees in some applications are rooted: in the database area, XML documents are often rooted trees where vertices represent elements or attributes and edges represent element-subelement and attribute-value relationships; in web page traffic mining, access trees are used to represent the access patterns of different users [21]. Trees in other applications are unrooted, i.e., they are *free trees*: in analysis of molecular evolution, an evolutionary tree (or phylogeny), which can be either a rooted tree or a free tree, is used to describe the evolution history of certain species [8]; in computer networking, unrooted multicast trees are used for packet routing [7]. From the above examples we can also see that trees in real applications are often *labeled*, with labels attached to vertices and edges where these labels are not necessarily unique.

In this paper, we present a computationally efficient algorithm, *HybridTreeMiner*, that discovers all frequently occurring subtrees in databases of labeled rooted unordered trees and databases of labeled free trees. Mining frequent subtrees is an important problem that has many applications, such as aggregating multicast trees from different groups[7] and classifying XML documents[22]. The main contributions of this paper are: (1) We introduce a new canonical form, which is based on breadth-first traversal, to uniquely represent a rooted unordered tree. (2) In order to mine frequent rooted unordered subtrees, based on our canonical form, we define an enumeration tree to systematically enumerate all (frequent) rooted unordered trees. We use two operations, extension and join, to grow the enumeration tree. (3) Then we extend our definition of canonical form to free trees. As a result, using the same enumeration tree, with certain additional constraints, we can enumerate all (frequent) free trees efficiently. (4) We have implemented all of our algorithms and have carried out extensive experimental analysis. We use both synthetic data and real application data to evaluate the performance of our algorithms.

The rest of the paper is organized as follows. In Section 2 we give necessary backgrounds. In Section 3, we introduce our algorithm for mining frequent rooted unordered trees. In Section 4, we extend our algorithm to mining frequent free trees. In Section 5, we give experimental results. In Section 6, we review related work and give conclusions.

* This work was supported by NSF grants IIS-0086116, ANI-0085773 and EAR-9817773

2. Background

In this section, we provide the definitions of the concepts that will be used in the remainder of the paper.

A *labeled Graph* $G = [V, E, \Sigma, L]$ consists of a *vertex* set V, an *edge* set E, an *alphabet* Σ for vertex and edge labels, and a *labeling function* $L : V \cup E \rightarrow \Sigma$ that assigns labels to vertices and edges. A *free tree* is an undirected graph that is connected and acyclic. A *rooted tree* is a free tree with a distinguished vertex that is called the *root*. In a rooted tree, if vertex v is on the path from the root to vertex w then v is an *ancestor* of w and w is a *descendent* of v. If in addition v and w are adjacent, then v is the *parent* of w and w is a *child* of v. A rooted *ordered* tree is a rooted tree that has a predefined left-to-right order among children of each vertex. A labeled free tree t is a *subtree* of another labeled free tree s if t can be obtained from s by repeatedly removing vertices with degree 1. Subtrees of rooted trees are defined similarly. Two labeled free trees t and s are *isomorphic* to each other if there is a one-to-one mapping from the vertices of t to the vertices of s that preserves vertex labels, edge labels, and adjacency. Isomorphisms for rooted trees are defined similarly except that the mapping should preserve the roots as well. An *automorphism* is an isomorphism that maps from a tree to itself. A *subtree isomorphism* from t to s is an isomorphism from t to some subtree of s. For convenience, in this paper we call a tree with k vertices a k-tree.

Let D denote a database where each transaction $s \in D$ is a labeled rooted unordered tree (or D is a database of free trees). For a given pattern t, which is a rooted unordered tree (or a free tree correspondingly), we say t occurs in a transaction s (or s *supports* t) if there exists at least one subtree of s that is isomorphic to t. The *support* of a pattern t is the fraction of transactions in database D that support t. A pattern t is called *frequent* if its support is greater than or equal to a *minimum support* (*minsup*) specified by a user. The frequent subtree mining problem is to find all frequent subtrees together with their supports in a given database.

3. Mining Frequent Rooted Unordered Trees

Two categories of algorithms are used in traditional market-basket association rule mining. The first category of algorithms put all frequent itemsets in an enumeration lattice and traverse the lattice level by level. Apriori [1] is a representative algorithm of this category. The second category of algorithms put all frequent itemsets in an enumeration tree and traverse the tree either level by level or following a depth-first traversal order. The main advantage of the Apriori-like algorithms is efficient pruning: an itemset becomes a potentially-frequent candidate only if it passes the "all subsets are frequent" check. The main advantage of tree enumeration mining algorithms is their relatively small memory footprint: for Apriori-like algorithms, in order to generate candidate *(k+1)*-itemsets, all frequent *k*-itemsets are involved and hence must be in memory; in contrast, in tree enumeration mining methods, only the parent of a candidate *(k+1)*-itemset in the enumeration tree needs to be in memory.

In this section, we introduce a frequent subtree mining algorithm that combines the ideas from both the above categories of mining algorithms to take advantage of both. First, we introduce a unique way to represent a rooted unordered tree–the breadth-first canonical form; then based on the canonical form we build an enumeration tree and use two operations, extension and join, to grow the enumeration tree efficiently.

Notice that if a labeled tree is rooted, then without loss of generality we can assume that all edge labels are identical: because each edge connects a vertex with its parent, so we can consider an edge, together with its label, as a part of the child vertex. (For the root, we can assume that there is a *null* edge connecting to it from above.) So for all running examples in the following discussion, we assume that all edges in all trees have the same label or equivalently, are unlabeled, and we therefore ignore all edge labels.

3.1. The Canonical Form

From a rooted unordered tree we can derive many rooted ordered trees, as shown in Figure 1. From these rooted ordered trees we want to uniquely select one as the canonical form to represent the corresponding rooted unordered tree.

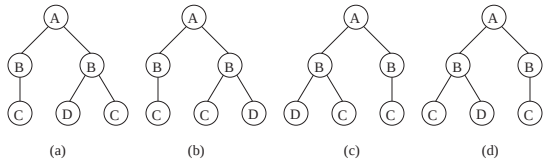


Figure 1. Four Rooted Ordered Trees Obtained from the Same Rooted Unordered Tree

We first define the *breadth-first string encoding* for a rooted ordered tree. Assume there are two special symbols, "$" and "#", which are not in the alphabet of edge labels and vertex labels. The breadth-first string encoding of a rooted ordered tree is obtained by traversing the tree in a *breadth-first* order, level by level. Following the order of traversal, we record in the string the label for each vertex. In addition, in the string we use "$" to partition the families of

siblings and use "#" to indicate the end of the string encoding. We assume that (1) there exists a total ordering among edge and vertex labels, and (2) "#" sorts greater than "$" and both sort greater than any other symbol in the alphabet of vertex and edge labels.

Now, for a rooted unordered tree, we can obtain different rooted ordered trees and corresponding breadth-first string encodings, by assigning different orders among the children of internal vertices. The *breadth-first canonical string (BFCS)* of the rooted unordered tree is defined as the minimal one among all these breadth-first string encodings, according to the lexicographical order. The corresponding orders among children of internal vertices define the *breadth-first canonical form (BFCF)*, which is a rooted ordered tree, of the rooted unordered tree. The breadth-first string encodings for each of the four trees in Figure 1 are for (a) $A\$BB\$C\$DC\#$, for (b) $A\$BB\$C\$CD\#$, for (c) $A\$BB\$DC\$C\#$, and for (d) $A\$BB\$CD\$C\#$. The breadth-first string encoding for *tree (d)* is the BFCS, and *tree (d)* is the BFCF for the corresponding labeled rooted unordered tree.

There are many other ways to define a canonical form for a labeled rooted unordered tree. In [5], we defined a canonical form based on depth-first-traversal. Independent of our work, Asai *et al.* in [4] and Nijssen *et al.* in [14] independently defined equivalent canonical forms. It will be seen later that with the new BFCF defined in this paper, we are able to extend our algorithm to handle free trees easily. We will compare our new algorithm based on the new BFCF with algorithms mentioned above in the section of experiments.

3.2. The Enumeration Tree

In this section we define an enumeration tree that enumerates all rooted unordered trees based on their BFCFs. For convenience, we call a leaf (together with the edge connecting it to its parent) at the bottom level of a BFCF tree a *leg*. Among all legs, we call the rightmost leaf at the bottom level the *last leg*. The following lemma provides the basis for our enumeration tree:

Lemma 1. [1] *Removing the last leg, i.e., the rightmost leg at the bottom level, from a rooted unordered (k+1)-tree in its BFCF will result in the BFCF for another rooted unordered k-tree.*

Based on the above lemma we can build an enumeration tree in which the nodes of the enumeration tree consist of all rooted unordered trees in their BFCFs and the parent for each rooted unordered tree is determined uniquely by removing the last leg from its BFCF. Figure 2 shows a fraction of the enumeration tree (for all rooted unordered subtrees with A as the root) for the BFCF tree at the bottom of the figure. (Ignore the dashed lines in the figure for now.)

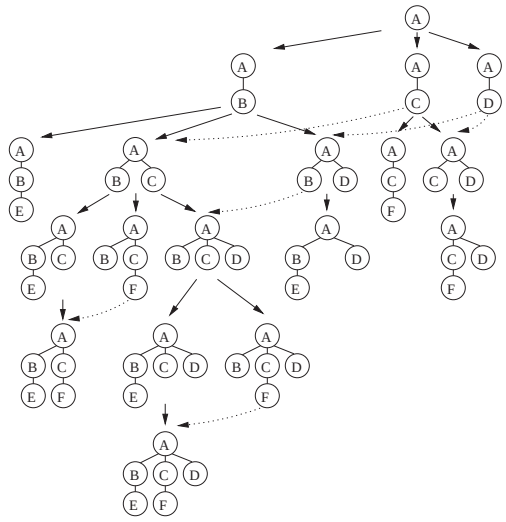


Figure 2. The Enumeration Tree for Rooted Unordered Trees in Their BFCFs

3.3. Two Operations on the Enumeration Tree

We want to grow the enumeration tree efficiently. The most straightforward method is to start from a node v of the enumeration tree, try to add all possible last legs in order to find all valid children of v. However, if we only use the extension method for enumeration tree growing, it could be inefficient because the number of potential last legs can be very large, especially when the cardinality of the alphabet for vertex labels is large. Therefore, in our algorithm we adopt an idea, which was first introduced by Huan *et al.* in [9] for frequent subgraph mining, that allows a special local join operation in addition to the extension. If we look at Figure 2 carefully we can see that all children of a node v in the enumeration tree can be obtained by either of two methods (assume the height of the BFCF corresponding to v is h): by extending a new leg at the bottom level, which gives a BFCF with height *h+1*, or by joining a pair of siblings (indicated by the dashed arrows in Figure 2), which gives a BFCF with height h. In addition, all the children of a node v in the enumeration tree are partitioned naturally into two families: those children derived from v by the extension method and those by joining v with one of its siblings. Moreover, two children of v can be further joined only if they are in the same family. With these observations in mind, we apply a hybrid enumeration tree growing al-

1 All proofs can be found in [6].

gorithm, which is based on two operations–extension and join, that takes advantage of both Apriori-like mining algorithms and vertical mining algorithms. (This is the first reason why we call our algorithm the *HybridTreeMiner* algorithm.)

3.3.1. Extension The first operation is the extension as described above. However, we use the extension only when the resulting BFCF tree has height one more than its parent, i.e., a new leg only grows from an old leg.

Definition 1 (Extension). *For a node v in the enumeration tree, we call the BFCF tree that v represents t_v and we assume the height of t_v is h. The extension operation is applied on v to obtain a new node v' in the enumeration tree that represents a new BFCF tree $t_{v'}$, where $t_{v'}$ has height h+1 and a single (new) leg. The new leg is the child of one of the legs of t_v. v' is the child of v in the enumeration tree.*

3.3.2. Join Join is an operation on a pair of sibling nodes in the enumeration tree. The resulting BFCF tree has the same height as its parent but with one more leg.

Definition 2 (Join). *Assume two sibling nodes v_1 and v_2 in the enumeration tree share the same parent v, and both the BFCF tree t_{v_1} represented by v_1 and the BFCF tree t_{v_2} represented by v_2 have height h. In addition, we assume that t_{v_1} sorts lower than (or equal to) t_{v_2}. The join operation is applied on v_1 and v_2 in the enumeration tree to obtain a new node v_1', which corresponds to a BFCF tree $t_{v_1'}$. v_1' is a child of v_1 in the enumeration tree and $t_{v_1'}$ has height h. $t_{v_1'}$ is constructed by adding the last leg of t_{v_2} to t_{v_1}. So in certain respects, the join operation is just a guided extension that grows legs on the leaves at level h-1 of t_{v_1}.*

3.4. Automorphisms

Automorphisms of a tree are the isomorphisms of the tree to itself. If the BFCF represented by the parent of a pair of sibling nodes in the enumeration tree has automorphisms, the join procedure will become more complicated. For example, the pair of BFCFs in Figure 3 create 9 candidate BFCFs because of the automorphisms of the parent shared by the pair of sibling nodes in the enumeration tree. From Figure 3 we can also see that self-join is necessary in growing an enumeration tree.

Therefore, we need an efficient scheme to record all possible automorphisms of a BFCF and consider them while growing the enumeration tree. In order to record the information on tree automorphisms, we introduce the *equivalence relation in the sense of automorphisms* among vertices of a tree in its BFCF:

Definition 3. *Vertices of a given tree in its BFCF belong to the same equivalence class if and only if*

(1). They are at the same level of the tree; and,

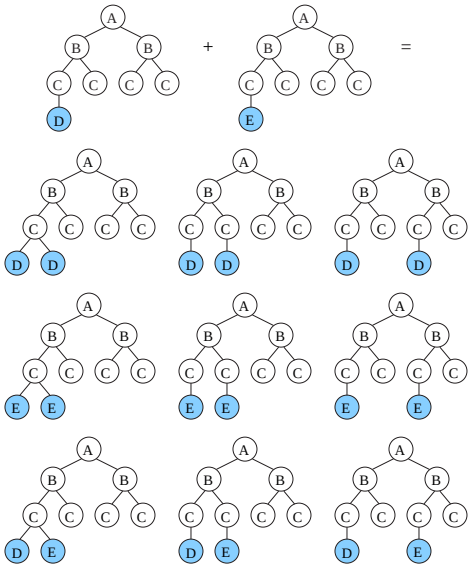


Figure 3. Automorphisms

(2). Attaching the same leaf to any of these vertices will result in a tree with the same BFCF.

As an example, Figure 4 gives a rooted unordered tree in its BFCF(left) and the equivalence classes among the vertices of the tree(right). It can be shown that we can obtain the automorphisms of a tree in time $O(ck \log k)$[6].

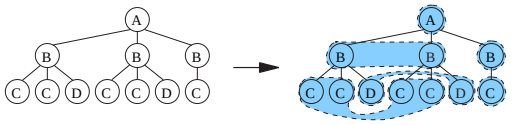


Figure 4. Obtaining Automorphisms

We use the information on automorphisms in the join operation. While joining two sibling nodes v_1 and v_2 in the enumeration tree (which share a common parent v in the enumeration tree), if the right most leg of t_{v_1} is the only child of its parent and the right most legs of both t_{v_1} and t_{v_2} have the same location, then during the join, we have to consider all the possible positions the last leg of t_{v_2} can take due to automorphisms of t_v. By doing this, all the cases given in Figure 3 will be included in the enumeration tree.

3.5. Support Counting

The *occurrence list* L_{t_v} for a rooted unordered *k*-tree t_v in its BFCF is a list that records information on each occurrence of t_v in the database. Each element $l \in L_{t_v}$ is of the

form $l = (tid, i_1, \ldots, i_k)$, where tid is the id of the transaction in the database that contains t_v and $i_1, \ldots, i_k$ represent the mapping between the vertex indices in t_v and those in the transaction. From the occurrence list L_{t_v} for t_v we can tell if t_v is frequent, because the *support* of t_v is the number of elements in L_{t_v} with distinct tid's. For the join operation, we "combine" a pair of occurrence lists L_{t_1} and L_{t_2} for two BFCFs t_1 and t_2 to get the occurrence list $L_{t_{12}}$ for a new *(k+1)*-tree t_{12}. The combination happens between any compatible pair of elements $l_1 \in L_{t_1}$ and $l_2 \in L_{t_2}$, where we define $l_1 = (tid_1, i_1, \ldots, i_k)$ and $l_2 = (tid_2, j_1, \ldots, j_k)$ to be *compatible* if $tid_1 = tid_2$ and $i_m = j_m$ for $m = 1, \ldots, k-1$. The result of combining l_1 and l_2 is $l_{12} = (tid_1, i_1, \ldots, i_k, j_k) \in L_{t_{12}}$. For the extension operation, assume l is an element in the occurrence list L_{t_v} for a *k*-tree t_v in BFCF. l can potentially be extended to an element l' in the occurrence list $L_{t_{v'}}$ for a *(k+1)*-tree $t_{v'}$ in BFCF, where $t_{v'}$ is a child of t_v in the enumeration tree and the height of tree $t_{v'}$ is one more than that of tree t_v. Assume $l = (tid, i_1, \ldots, i_k)$, then l is extended to l' if and only if (1) $l' = (tid, i_1, \ldots, i_k, i_{k+1})$ where $i_1, \ldots, i_k, i_{k+1}$ is a valid mapping between the vertex indices in $t_{v'}$ and those in the transaction identified by *tid* and (2) $i_{k+1} \neq i_m$ for $m = 1, \ldots, k$.

3.6. Putting It All Together

Figure 5 summarizes our *HybridTreeMiner* algorithm. The main step in the algorithm is the function *Enum-Grow*, which grows the whole enumeration tree. We can see that in the algorithm, two operations, join and extension, are applied separately. We have shown in [6] that the time complexity of *HybridTreeMiner* algorithm is $O(|F| \cdot (hk^2 + |D|kc))$ where F is the set of all frequent subtrees, D is the database, h is the maximal height and k is the maximal size of all frequent subtrees, and c is the maximal degree among all vertices in all transactions of the database.

4. Mining Frequent Free Trees

If the transactions in the database are free trees, then the problem becomes mining all frequent free subtrees. Because there is not a unique root, there are more ways to represent free trees compared to that of rooted trees. Therefore the problem of mining frequent free trees seems to be a much more difficult problem compared to mining frequent rooted unordered trees. However, it turns out that by extending our definition for the BFCF to free trees and by adding certain constrains to our enumeration tree, the *HybridTreeMiner* algorithm can efficiently handle free trees as well. (This is the second reason why we call our algorithm the *HybridTreeMiner* algorithm.)

Algorithm **HybridTreeMiner**(D,$minsup$)

```
1: F1, F2 ← {frequent 1, and 2-trees};
2: F ← F1 ∪ F2;
3: C ← sort(F2);
4: Enum-Grow(C,F,minsup);
5: return F;
```

Algorithm **Enum-Grow**(C,F,$minsup$)

```
1:  for i ← 1, ..., |C| do
        ▷ The Join Operation
2:      J ← ∅;
3:      for j ← i, ..., |C| do
4:          p ← join(c_i, c_j);
5:          if supp(p) ≥ minsup
6:          then J ← J ∪ p;
7:      F ← F ∪ J;
8:      Enum-Grow(J,F,minsup);
        ▷ The Extension Operation
9:      E ← ∅;
10:     for each leg l_m of c_i do
11:         for each possible new leg l_n do
12:             q ← c_i plus leg l_n at position l_m;
13:             if supp(q) ≥ minsup
14:             then E ← E ∪ q;
15:     F ← F ∪ E;
16:     Enum-Grow(E,F,minsup);
```

Figure 5. The HybridTreeMiner Algorithm

4.1. Extending the Canonical Form

Free trees do not have roots, but we can uniquely create roots for them for the purpose of constructing a unique canonical form. Starting from a free tree in each step we remove all leaf vertices (together with their incident edges), and we repeat this step until a single vertex or two adjacent vertices are left. For the first case, the free tree is called a *centered tree* and the remaining vertex is called the *center*; for the second case, the free tree is called a *bicentered tree* and the pair of remaining vertices are called the *bicenters* [2]. A free tree is either centered or bicentered. The above procedure takes $O(k)$ time where k is the number of vertices in the free tree. Figure 6 shows a centered free tree and a bicentered free tree as well as the procedure to obtain the corresponding center and bicenters.

If a free tree is centered, we can uniquely identify its center and make it the root to obtain a rooted unordered tree. Then we can normalize this rooted unordered tree to obtain the BFCF for the centered free tree as we did in previous sections. However, if a free tree is bicentered, we can only identify a pair of vertices (the bicenters). We can make each of the bicenters the root to obtain a pair of rooted un-

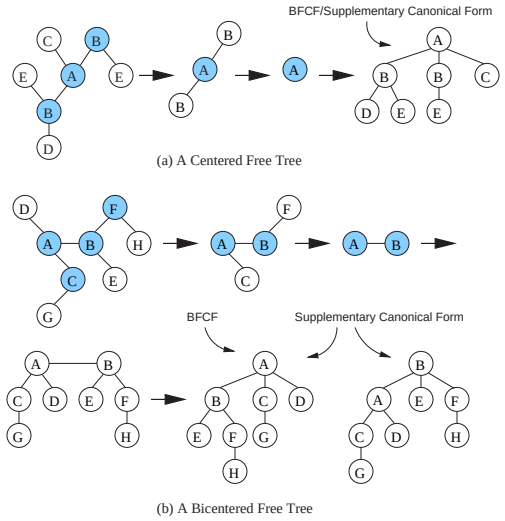


Figure 6. A Centered Free Tree (above) and A Bicentered Free Tree (below) together with Their Canonical Forms

ordered trees. We call the canonical forms (BFCFs) for the pair of rooted unordered trees the *supplementary canonical forms*. Among the string encodings of the two supplementary canonical forms, one sorts lower (or they are identical). We call the string encoding that sorts lower the BFCS and the corresponding supplementary canonical form the BFCF for the bicentered free tree. If we also call the BFCF for the centered free tree a supplementary canonical form, then we have a relation: a BFCF for a free tree is a supplementary canonical form, while a supplementary canonical form for a free tree may or may not be the BFCF for the free tree if the free tree is bicentered. In Figure 6, for the centered free tree, the right most tree is the supplementary canonical form as well as the BFCF; for the bicentered free tree, the right most pair of trees are the supplementary canonical forms and the first tree in the pair is the BFCF.

Theorem 1. *A BFCF for a rooted unordered tree with 3 or more vertices and height h is a supplementary canonical form for a free tree if and only if the following two conditions hold:*

(1). There are at least 2 children for the root; and

(2). One subtree induced by a child of the root has height h-1 and at least one subtree induced by another child of the root has height $\geq$ h-2.

4.2. Extending the Enumeration Tree

With the extended definition for the BFCF and the new concept of a supplementary canonical form for a free tree, we can build an enumeration tree for all free trees in their supplementary canonical forms. The enumeration tree for free trees is almost identical to that for rooted unordered trees, except that we replace the BFCF with the supplementary canonical form:

Corollary 1. *Removing the last leg, i.e., the rightmost leg, from a supplementary canonical form of a free tree will result in a supplementary canonical form for another free tree.*

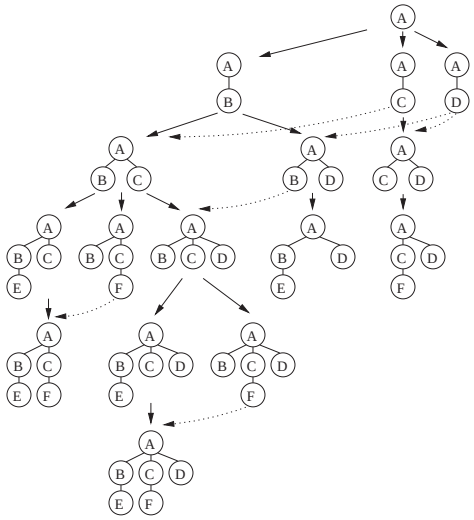


Figure 7. The Enumeration Tree for Free Trees in Their Supplementary Canonical Forms

Figure 7 is the enumeration tree for free trees with A as the root of the supplementary canonical forms. Comparing Figure 7 with Figure 2 we note that Figure 7 has two fewer nodes because these two nodes do not represent supplementary canonical forms. There is a new constraint for growing an enumeration tree for free trees: the node grown from its parent must be a supplementary canonical form as well. As we can see from Theorem 1, comparing to the BFCF for a rooted unordered tree, supplementary canonical forms for free trees have certain constraints on the heights of the subtrees induced by the children of the root. In other words, supplementary canonical forms have constraints on the "shape" of the tree. As a result, we need to revise the extension operation and the join operation for growing the enumeration tree, to ensure that the result of the operations are valid supplementary canonical forms.

For a node v in the enumeration tree, we call the supplementary canonical form that v represents t_v. We denote the root of t_v by r and assume the height of t_v is h. We also denote the children of r by $r_1, \ldots, r_m$. First, let's look at the join operation. It turns out that we do not have to make any

change to it: the join operation does not change the height of the BFCF, so it will not change the subtrees with height *h-1*, which are induced by the children of the root. It may increase the height of the subtree induced by a child of the root from *h-2* to *h-1*, but in this case by Theorem 1 the result is still a supplementary canonical form. The extension operation, however, needs to be changed. We can apply the extension operation on v to obtain a new node v' that represents a supplementary canonical form in the enumeration tree only if, among the subtrees induced by $r_1, \ldots, r_m$, at least two have height *h-1*. In other words, we only apply the extension operation to supplementary canonical forms that represent centered trees, because extending supplementary canonical forms that represent bicentered trees will not result in a supplementary canonical form.

With these two operations revised, we can systematically grow the enumeration tree for mining frequent free trees. The mining algorithm given in Figure 5 can be applied to mining free trees without any change. Notice that there exists redundancy in the enumeration tree for free trees: because a bicentered free tree has two supplementary canonical forms, it is represented twice in the enumeration tree. At the time of outputting frequent subtrees, we need check and only output the supplementary canonical form that is a real BFCF.

It is very interesting that enumeration trees can be defined for three types of trees–rooted ordered trees, rooted unordered trees, and free trees. In that order, the structures of the three types of trees become simpler and simpler; in contrast, their enumeration trees become more and more complicated (restricted). This is because for all three types of trees, their final representation (the canonical form) must be rooted ordered trees.

5. Experiments

5.1. Experiment Setup

We performed extensive experiments to evaluate the performance of the *HybridTreeMiner* algorithm on both rooted unordered trees and free trees, using both synthetic datasets and datasets from real applications. All experiments were done on a 2GHz Intel Pentium IV PC with 1GB main memory, running RedHat Linux 7.3 operating system. All algorithms are implemented in C++ using the g++ 2.96 compiler.

For mining rooted unordered trees, we compare *HybridTreeMiner* with the *Unot* algorithm given in [4] and the *uFreqt* algorithm given in [14]. However, no implementation for *Unot* was given in [4], so we have implemented *Unot* to the best of our knowledge. For mining frequent free trees, we compare *HybridTreeMiner* with our previous *FreeTreeMiner* algorithm [5], which is an Apriori-like algorithm.

5.2. Synthetic Data Generator

Table 1 provides the parameters for our synthetic data generator and their meanings. The detailed procedures that we followed to create the synthetic datasets are as follows: starting from a master graph that is generated using the universal Internet topology generator BRITE [12], we first sample a set of $|N|$ subtrees whose size are determined by $|I|$. We call this set of $|N|$ subtrees the *seed trees*. (For data of rooted unordered trees, for each seed tree we randomly select a vertex as the root.) Each seed tree is the starting point for $|D| \cdot |S|$ transactions; each of these $|D| \cdot |S|$ transactions is obtained by first randomly permuting the seed tree then adding more random vertices to increase the size of the transaction to $|T|$. After this step, more random transactions with size $|T|$ are added to the database to increase the cardinality of the database to $|D|$. The number of distinct edge and vertex labels is controlled by the parameter $|L|$.

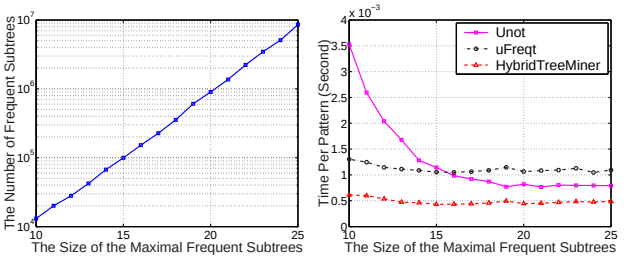
Parameter	Description
$\|D\|$	the number of transactions in the database
$\|T\|$	the size of each transaction in the database
$\|I\|$	the maximal size of frequent subtrees
$\|N\|$	the number of frequent subtrees with size $\|I\|$
$\|S\|$	the minimum support [%] for frequent subtrees
$\|L\|$	the size of the alphabet for vertex/edge labels

Table 1. Parameters for Synthetic Generators

5.3. Results for Rooted Unordered Trees

5.3.1. Synthetic Datasets In our first experiment, we want to study the effect of the size of maximal frequent subtrees on our algorithm. With all other parameters fixed ($|D|$=10000, $|T|$=50, $|N|$=100, $|S|$=1%), we increase the maximal frequent tree size $|I|$ from 10 to 25. Figure 8(A) gives the number of frequent subtrees versus size $|I|$. From the figure we can see that the number of frequent subtrees grows exponentially with the size of the maximal frequent subtrees (notice the logarithm scale of the y axis in the figure). As we know, in all

these databases, the number of maximum frequent subtrees (a frequent subtree is maximum if it is not a subtree of any other frequent subtree) is fixed to be $|N|$=100. As a result, the experiment result suggests that in some circumstances, the total number of frequent subtrees can be dramatically larger than that of maximum frequent subtrees. Figure 8(B) shows the average time to mine each frequent subtrees, using each of the three methods: *HybridTreeMiner*, *Unot*, and *uFreqt*. As can be seen, the average time for all algorithms to mine each pattern is not affected very much by $|I|$. (The curves are not smooth because of the randomness in datasets generating.) However, this average time decreases a little as the size $|I|$ increases. Our explanation for this decline is that for a node v in the enumeration tree, as the size of t_v, the tree represented by v, grows larger, v will have many children and for these children we only need to scan database once to check if they are frequent. Therefore the amortized time for each child is decreased. Also from Figure 8(B) we see that although *HybridTreeMiner* is faster than both *Unot* and *uFreqt*, because the margins are too small, we cannot draw any conclusion on which method is better. However, we want to argue that our *HybridTreeMiner* has competitive performance comparing to the other rooted unordered tree mining algorithms, while it can handle free trees as well.

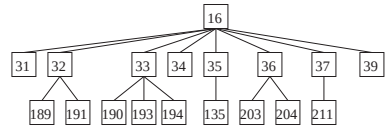


(A) (B)

Figure 8. Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees

We have performed experiments to study if and how the *HybridTreeMiner* algorithm is sensitive to other parameters, such as the size of the databases, the number of distinct labels and the shapes of trees in the datasets. We refer interested readers to [6] for a more detailed study.

5.3.2. Web Access Trees In this section, we present an application on mining frequent accessed web pages from web logs. We ran experiments on the log files at UCLA Data Mining Laboratory (http://dml.cs.ucla.edu). First, we used the WWWPal system [15] to obtain the topology of the web site and wrote a program to generate a database from the log files. Our program generated 2793 user access trees from the log files collected over year 2003 at our laboratory that touched a total of 310 web pages. In the user access trees, the vertices correspond to the web pages and the edges correspond to the links between the web pages. We take URLs as the vertex labels and each vertex has a distinct label. We do not assign labels to edges. For support equals 1%, our *HybridTreeMiner* algorithm mined 16507 frequent subtrees in less than 1 sec. Among all the frequent subtrees, the maximum subtree has 18 vertices. Figure 9 shows this maximum subtree. It turns out that this subtree is a part of web site for the $\mathrm{ESP}^2\mathrm{Net}$ (Earth Science Partners' Private Network) project. From this mining result, we can infer that many visitors to our web site are interested in details about the $\mathrm{ESP}^2\mathrm{Net}$ project.



16 http://dml.cs.ucla.edu/projects/dml_esip/
31 http://dml.cs.ucla.edu/projects/dml_esip/Objectives/objectives.html
32 http://dml.cs.ucla.edu/projects/dml_esip/Status/status.html
33 http://dml.cs.ucla.edu/projects/dml_esip/Technology/technology.html
34 http://dml.cs.ucla.edu/projects/dml_esip/Science/science.html
35 http://dml.cs.ucla.edu/projects/dml_esip/Contacts/contacts.html
36 http://dml.cs.ucla.edu/projects/dml_esip/Clustering/clustering.html
37 http://dml.cs.ucla.edu/projects/dml_esip/Links/links.html
39 http://dml.cs.ucla.edu/projects/dml_esip/index.html
189 http://dml.cs.ucla.edu/oasis/OASIS_Overview/oasis_overview.html
191 http://dml.cs.ucla.edu/~weiwang/myresearch.shtml
190 http://dml.cs.ucla.edu/projects/dml_esip/Technology/Conquest/conquest.html
193 http://dml.cs.ucla.edu/projects/dml_esip/Technology/VPN/vpn.html
194 http://dml.cs.ucla.edu/projects/dml_esip/Technology/SEML/seml.html
135 http://dml.cs.ucla.edu/people/index.html
203 http://dml.cs.ucla.edu/projects/dml_esip/Clustering/ESP2Net-CaLSIP/esp2net-calsip.html
204 http://dml.cs.ucla.edu/projects/dml_esip/Clustering/DM_Search/dm_search.html
211 http://dml.cs.ucla.edu/access/simple.html

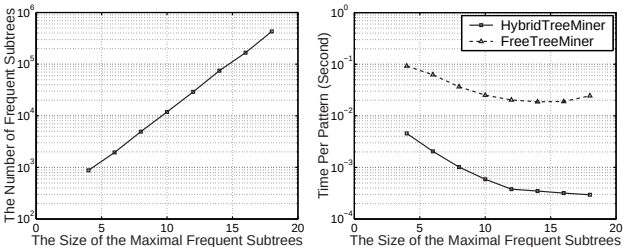
Figure 9. The Maximum Frequent Subtree Mined From Web Log Files

5.4. Results on Free Trees

In this section, we report our experiments on datasets of free trees. We compare the *HybridTreeMiner* algorithm with the *FreeTreeMiner* algorithm, an *Apriori*-like algorithm that we have developed before[5], and the only algorithm, to the best of our knowledge, that mines frequent free trees.

5.4.1. Synthetic Datasets In the first experiment, we fix all parameters other than $|I|$ ($|D|$=10000, $|T|$=30, $|N|$=100, $|S|$=1%), while changing the maximal frequent tree size $|I|$. For fair comparison, we watched the memory usage for both algorithms carefully. Because in our experiments, when $|I|$ grows larger than 18 the memory used by *FreeTreeMiner* will surpass the available memory, we decided to compare

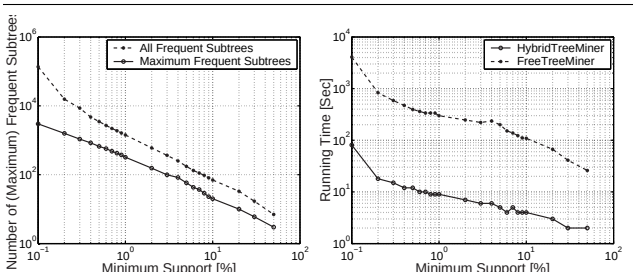
the two algorithms with $|I|$ between 4 and 18. Figure 10 gives the results. Figure 10(A) shows that, similar to that of rooted unordered trees, the number of frequent subtrees grows exponentially with the size of maximal frequent subtrees. Figure 10(B) gives the average time for *HybridTreeMiner* and *FreeTreeMiner* to mine each frequent subtree. We can see that our new algorithm *HybridTreeMiner* is faster than *FreeTreeMiner* by 1 to 2 orders of magnitudes. In addition, we have observed that the peak memory usage for *HybridTreeMiner* is 30MB and that for *FreeTreeMiner* is around 500MB. This observation indicates that our new algorithm has much smaller memory footprint compared to Apriori-like algorithms.



(A) (B)

Figure 10. Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees

5.4.2. The Chemical Compound Dataset This dataset was described in [5]. It contains 17,663 tree-structured chemical compounds sampled from a graph dataset of the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) [13]. In the tree transactions, the vertices correspond to the various atoms in the chemical compounds and the edges correspond to the bonds between the atoms. We take atom types as the vertex labels and bond types as the edge labels. There are a total of 80 distinct vertex labels and 3 distinct edge labels. We explored a wide range of the minimum support from 0.1% to 50%. Figure 11(A) gives the numbers of all frequent subtrees and maximum frequent subtrees under different supports. We can see that compared to all frequent subtrees, there are much fewer (about 10 times) maximum frequent subtrees. The numbers for both frequent subtrees and maximum frequent subtrees decrease exponentially with the support. Figure 11(B) gives the running time for the two algorithms to mine all frequent subtrees under different supports. Again, *HybridTreeMiner* outperforms *FreeTreeMiner* by 1 to 2 orders of magnitudes.



(A) (B)

Figure 11. Number of (Maximum) Frequent Subtrees and Running Time vs. Support for the Chemical Compound Dataset

6. Related Work and Conclusions

Related Work Many recent studies have focused on databases of trees partly because of the increasing popularity of XML in databases. In a recent paper, Zaki [21] presented an algorithm, TREEMINER, to discover all frequent embedded subtrees, i.e., those subtrees that preserve ancestor-descendent relationships, in a forest or a database of rooted ordered trees. In [3] Asai *et al.* presented algorithm FREQT to discover frequent rooted ordered subtrees. They used a string encoding similar to that defined by Zaki [21] and built an enumeration tree for all (frequent) rooted ordered trees. The *rightmost expansion* is used to grow the enumeration tree. Notice that the above methods mine frequent subtrees in databases of rooted *ordered* trees while our algorithm mine frequent subtrees in databases of rooted *unordered* trees. In [5] we have studied the problem of indexing and mining free trees. We defined a canonical form, which is applicable to both rooted unordered trees and free trees, and developed an Apriori-like algorithm to mine all frequent free trees. Independent of our work, Asai *et al.* in [4] and Nijssen *et al.* in [14] defined equivalent canonical forms for rooted unordered tree, which are equivalent to our canonical form in [5], and built enumeration trees for all rooted unordered trees. Again, the *rightmost expansion* is used to grow the enumeration tree. Note that all the string encodings and canonical forms in these papers [3, 4, 5, 14] are based on the depth-first traversal of a tree while our new canonical form (BFCF) is based on the breadth-first traversal of a tree, so it can easily be extended to free trees. In addition, there are other studies on mining frequent subtrees, such as those given in [18, 16], that do not guarantee completeness, i.e., some frequent subtrees may not be in the search results. There are also studies [17, 19] on mining and querying phylogenetic trees, which are rooted unordered trees with *distinct* vertex la-

bels. Moreover, closely related to mining frequent subtrees, many recent studies have focused on mining frequent subgraphs [9, 10, 11, 20], which is a much more difficult problem than mining frequent subtrees (e.g., the subgraph isomorphism is an NP-complete problem while the subtree isomorphism problem is in P).

Conclusion In this paper, we presented a new canonical form, the breadth-first canonical form (BFCF), which is defined based on the breadth-first traversal, for both rooted unordered trees and free trees. In addition, we built enumeration trees to enumerate all (frequent) rooted unordered trees in their BFCFs and all (frequent) free trees in their supplementary canonical forms. Our canonical forms make it possible to introduce the join operation in enumeration tree growing. In addition, the breadth-first canonical form can be extended easily from rooted unordered trees to free trees. Therefore our frequent subtree mining algorithm applies to both rooted unordered trees and free trees. In our construction, rooted unordered trees and free trees share similar canonical forms, similar enumeration trees, similar operations on the enumeration trees, and identical frequent subtree mining algorithms. Our algorithm is shown to be competitive in comparison to other rooted unordered subtree mining algorithms and one to two orders of magnitudes faster than a previously known free tree subtree mining algorithm.

Acknowledgement

Thanks to Siegfried Nijssen at the Leiden Institute of Advanced Computer Science for providing us the source codes for the *uFreqt* algorithm. Thanks to Professor J. Punin, M. Krishnamoorthy, and Professor Zaki at the Rensselaer Polytechnic Institute for helping us with the WWWPal system.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Intl. Conf. on Very Large Databases (VLDB'94)*, 1994.

[2] J. M. Aldous and R. J. Wilson. *Graphs and Applications, An Introductory Approach*. Springer, 2000.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *2nd SIAM Int. Conf. on Data Mining (SDM'02)*, 2002.

[4] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. Technical Report DOI-TR-CS 216, Department of Informatics, Kyushu University, June 2003.

[5] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proc. of the 2003 IEEE Intl. Conf. on Data Mining (ICDM'03)*, 2003.

[6] Y. Chi, Y. Yang, and R. R. Muntz. Mining frequent rooted trees and free trees using canonical forms. Technical Report CSD-TR No. 030043, ftp://ftp.cs.ucla.edu/tech-report/2003-reports/030043.pdf, UCLA, 2003.

[7] J. Cui, J. Kim, D. Maggiorini, K. Boussetta, and M. Gerla. Aggregated multicast–a comparative study. In *Proceedings of IFIP Networking 2002*, 2002.

[8] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71:153–169, 1996.

[9] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proc. 2003 Int. Conf. on Data Mining (ICDM'03)*, 2003.

[10] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, pages 13–23, 2000.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining (ICDM'01)*, 2001.

[12] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: Universal topology generation from a user's perspective. Technical Report BUCS-TR2001-003, Boston University, 2001.

[13] N. C. I. (NCI). DTP/2D and 3D structural information. World Wide Web, ftp://dtpsearch.ncifcrf.gov/jan03_2d.bin, 2003.

[14] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First Intl. Workshop on Mining Graphs, Trees and Sequences (MGST'03)*, 2003.

[15] J. Punin and M. Krishnamoorthy. WWWPal system–a system for analysis and synthesis of web pages. In *WebNet 98 Conference*, 1998.

[16] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.

[17] D. Shasha, J. T. L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate searching in unordered trees. In *Proc. of the 14th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'02)*, 2002.

[18] A. Termier, M.-C. Rousset, and M. Sebag. TreeFinder: a first step towards xml data mining. In *Proc. of the 2002 IEEE Intl. Conf. on Data Mining (ICDM'02)*, 2002.

[19] J. T. L. Wang, H. Shan, D. Shasha, and W. H. Piel. Treerank: A similarity measure for nearest neighbor searching in phylogenetic databases. In *Proc. of the 15th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'03)*, 2003.

[20] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Intl. Conf. on Data Mining (ICDM'02)*, 2002.

[21] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. 2002 Intl. Conf. Knowledge Discovery and Data Mining (SIGKDD'02)*, July 2002.

[22] M. J. Zaki and C. C. Aggarwal. XRules: An effective structural classifier for XML data. In *Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.