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 known algorithms for mining frequent free trees.

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 unordered 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. 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 [12]; in pattern recognition, a free tree called *shape axis tree* is used to represent shapes [16]; in computer networking, unrooted multicast trees are used for packet routing [10]. 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 study some issues in mining databases of labeled trees, where the trees can be rooted unordered trees or free trees.

1.1 Related Work.

Recently, there has been growing interest in mining databases of graphs and trees. Inokuchi *et al.* [14] presented an algorithm, AGM, and Kuramochi *et al.* [15] presented another algorithm, FSG, both for mining subgraphs in a graph database. The two algorithms used a level-wise Apriori [2] approach: the AGM algorithm extends subgraphs by adding a vertex per level and the FSG algorithm by adding an edge. Yan *et al.* [22, 23] and Huan *et al.* [13] presented subgraph mining algorithms based on traversing different enumeration trees. However, to check if a transaction supports a graph is an instance of the subgraph isomorphism problem which is NP-complete [11]; to check if two graphs are isomorphic (in order to avoid creating a candidate multiple times) is an instance of the graph isomorphism problem which is not known to be in either P or NP-complete [11]. Therefore without taking advantage of the tree structure, these graph algorithms are not likely to be efficient for the frequent tree mining problem.

Many recent studies have focused on databases of trees because the increasing popularity of XML in databases. Chen *et al.* [8] provided data structures and algorithms to accurately estimate the number of occurrences of a small tree (twig) in a large tree. Termier *et al.* [21] presented an algorithm, *TreeFinder*, which finds a subset of frequent trees in a set of tree-structured XML data. Shasha *et al.* [20] gave a detailed survey on keytree and keygraph searching in databases of trees and graphs and the survey focused mainly on approximate containment queries. The work in [21, 20], however, does not guarantee completeness, i.e., some frequent subtrees may not be in the search results. In a recent paper, Zaki [24] presented an algorithm called TREEMINER to discover all frequent embedded subtrees, i.e., those subtrees that preserve ancestor-decedent relationships, in a forest or a database of rooted ordered trees. In [5] Asai *et al.* presented algorithm FREQT to discover frequent rooted ordered subtrees. They used a string encoding similar to that defined by Zaki [24] and built an enumeration tree for all (frequent) rooted ordered trees. The *rightmost expansion* is used to grow the enumeration tree. In [9] 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.* [6] defined a canonical form for rooted unordered tree, which is equivalent to our canonical form in [9], and built an enumeration tree for all rooted unordered trees. Again, the *rightmost expansion* is used to grow the enumeration tree. However, there was no implementation given in [6]. Note that all the string encodings and canonical forms in these papers [24, 5, 9, 6] are based on the depth-first traversal of a tree.

1.2 Contributions of This Paper. 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) Finally, 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 algorithm.

2 Background.

In this section, we provide the definitions of the concepts that will be using 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 graph is *directed* (*undirected*) when each edge is an ordered (unordered) pair of vertices. A *path* is a list of vertices of the graph such that each pair of neighboring vertices in the list is an edge of the graph. A *cycle* is a path such that the first and the last vertices of the path are the same. A graph is *acyclic* if the graph contains no cycle. A graph is *connected* if there exists at least one path between any pair of vertices, *disconnected* otherwise. 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 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 [2] 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, as in [7], or following a depth-first traversal order, as in [1]. Algorithms of the latter type are usually called vertical mining algorithms. 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 vertical 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 vertical mining, 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.

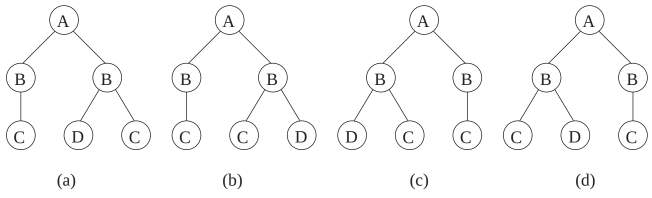


Figure 1: Four Rooted Ordered Trees Obtained from the Same Rooted Unordered Tree

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.

We now give a bottom-up procedure to construct the BFCF for a labeled rooted unordered tree. Starting from the bottom, level by level, for each vertex v at the level, because by recursion all the subtrees induced by the children of v have been in the correct forms, we can compare the string encodings of these subtrees and order them from left to right from small to large. We repeat the procedure until finally all the children of the root vertex are re-ordered . Figure 2 gives a running example on how to obtain the BFCF for a labeled rooted unordered tree. In the figure, the levels surrounded by the dashed boxes are the corresponding levels of vertices we are working on in each stage.

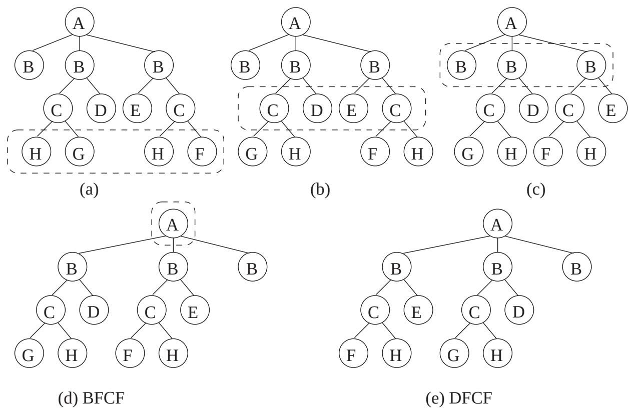


Figure 2: To Obtain the BFCF of a Labeled Rooted Unordered Tree

Theorem 3.1. *The above construction procedure gives the BFCF for a labeled rooted unordered tree.*

Proof. For a rooted unordered tree t, we denote the root of t by r, the children of r by $r_1, \ldots, r_m$, and the subtrees induced by $r_1, \ldots, r_m$ by $t_{r_1}, \ldots, t_{r_m}$. Because of the recursive construction procedure and because of the fact that a tree consisting of a single vertex is in its BFCF, we only have to prove the following statement: If all $t_{r_1}, \ldots, t_{r_m}$ are in their BFCFs and we rearrange the order among $t_{r_1}, \ldots, t_{r_m}$ from left to right in non-decreasing order (according to the lexicographical order among their BFCSs) to get the rooted ordered tree t', then t' is the BFCF for t. We prove this statement by contradiction. If, for the sake of contradiction, the BFCF of t is t'', such that there are a pair of BFCF trees among $t_{r_1}, \ldots, t_{r_m}$ (say t_{r_i} and t_{r_j}) with $BFCS(t_{r_i}) < BFCS(t_{r_j})$, but t_{r_i} is to the right of t_{r_j} in t''. We cut the BFCS of t_{r_i} into segments $s_{i1}, \ldots, s_{ih}$ where each segment s_{ip} represents the part of the BFCS of t_{r_i} at level p of the tree t''. We do the same to t_{r_j} to get $s_{j1}, \ldots, s_{jh}$. Because $BFCS(t_{r_i}) < BFCS(t_{r_j})$, there is an integer q, where $1 \leq q \leq h$, such that $s_{ip} = s_{jp}$ for $0 \leq p < q$ and $s_{iq} \neq s_{jq}$. With $s_{ip} = s_{jp}$ for $0 \leq p < q$, there are the same number of "\$" symbols in s_{iq} and s_{jq}. (Actually, for both t_{r_i} and t_{r_j}, the number of "\$" symbols at level q is the same as the number of vertices at level *q-1*.) In addition, both s_{iq} and s_{jq} end with a "\$". Therefore s_{iq} is not a prefix of s_{jq}, which implies $s_{iq} < s_{jq}$ because $BFCS(t_{r_i}) < BFCS(t_{r_j})$. As a result, if we switch the order of t_{r_i} and t_{r_j} in t'', we will get a breadth-first encoding that is smaller than the string encoding of t'', hence a contradiction. □

THEOREM 3.2. *The above BFCF construction procedure has time complexity $O(k^2 c \log c)$, where k is the number of vertices the tree has and c is the maximal degree of the vertices in the tree.*

Proof. For each vertex v, to order all its children takes $O(c \log c)$ comparisons and because the comparisons are among subtrees induced by the children of v, each comparison takes $O(k)$ time. The tree has k vertices therefore the total time complexity for normalization is $O(k^2 c \log c)$. □

THEOREM 3.3. *The length of the BFCS for a labeled rooted unordered tree is at most $3k$ where k is the number of vertices of the tree.*

Proof. The symbols in the BFCS for a labeled rooted unordered tree include vertex labels, edge labels, "\$" symbols, and a "#" symbol. The number of vertices is k and the number of edges is *k-1*, so the number of symbols in the BFCS for edge/vertex labels is *2k-1*. There is one "\$" at the root level. For all levels below, the number of "\$" symbols at each level is at most the number of vertices at the level above. In addition, the last "\$" will be replaced by a "#". So the BFCS contains one "#" and at most k "\$" symbols. Therefore the length of the BFCS is at most $3k$. □

It is interesting that there is a different way to define a canonical form. We can define a string encoding for a rooted *ordered* tree through a *depth-first* traversal and use "\$" to represent a backtrack and "#" to represent the end of the string encoding. The depth-first string encodings for each of the four trees in Figure 1 are for (a) $ABC\$\$BD\$C\#$, for (b) $ABC\$\$BC\$D\#$, for (c) $ABD\$C\$\$BC\#$, and for (d) $ABC\$D\$\$BC\#$. If we define the *depth-first canonical string (DFCS)* of the rooted unordered tree as the minimal one among all possible depth-first string encodings, then we can define the *depth-first canonical form (DFCF)* of a rooted unordered tree as the corresponding rooted ordered tree that gives the minimal DFCS. In Figure 1, the depth-first string encoding for *tree (d)* is the DFCS, and *tree (d)* is the DFCF for the corresponding labeled rooted unordered tree. We can construct the DFCF for a rooted unordered tree in $O(ck \log k)$ time, using a tree isomorphism algorithm given by Aho *et al.*[3]. The algorithm sorts the vertices of the rooted unordered tree level by level bottom-up. When sorting vertices at a given level, we first compare the labels of the vertices in that level, then the ranks (in order) of each of the children (in their own level) of these vertices. Figure 3 is a running example for the algorithm. In the figure, for each vertex, the symbols in the parentheses are first the vertex label then, in order, the ranks of its children ("#" denotes the end of the encoding); the symbol in front of the parentheses is the rank of the vertex in its level. After sorting all levels, the tree is scanned top-down level by level, starting from the root, and children of each vertex in the current level are rearranged to be in the determined order.

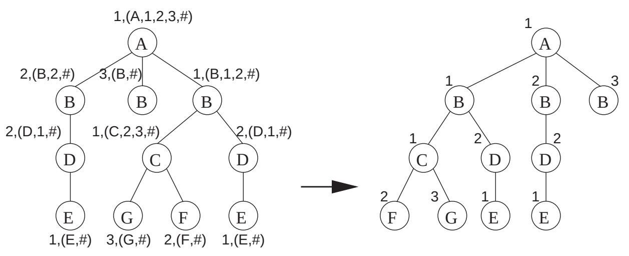


Figure 3: To Obtain the DFCF of A Rooted Unordered Tree

THEOREM 3.4. *The above construction procedure gives the DFCF for a labeled rooted unordered tree.*

Proof. For a rooted unordered tree t, we denote the root of t by r, the children of r by $r_1, \ldots, r_m$, and the subtrees induced by $r_1, \ldots, r_m$ by $t_{r_1}, \ldots, t_{r_m}$. Because of the recursive construction procedure and because of the fact that a tree consisting of a single vertex is in its DFCF, we only have to prove the following two statements: (1) If all $t_{r_1}, \ldots, t_{r_m}$ are in their DFCFs and we rearrange the order among $t_{r_1}, \ldots, t_{r_m}$ from left to right in non-decreasing order (according to the lexicographical order among their DFCSs) to get the rooted ordered tree t', then t' is the DFCF for t; (2) The rank among $r_1, \ldots, r_m$, which is given by the above construction procedure, is the same as the order of $t_{r_1}, \ldots, t_{r_m}$ according to the lexicographical order of their DFCSs. For statement (1), we note that in order to construct the string encoding of t, we combine, in order, the vertex label of the root r, the string encoding of the subtree induced by r's first child, ..., the string encoding of the subtree induced by r's last child. (Note that there are some "\$"s in between to represent backtracks.) Obviously, if all $t_{r_1}, \ldots, t_{r_m}$ are in their DFCFs, and if we order them from left to right in non-decreasing order according to the lexicographical order among their DFCSs, we will get the minimal string encoding (hence the DFCS) for t. We can prove statement (2) by recursion: it is trivially true for the bottom level; for each level above, the rank among the vertices in the level is obtained by first comparing vertex labels, then the rank (in order) of their children. From the argument in the proof of statement (1) we can see the resulting rank among the vertices at each level is the same as the order of the corresponding subtrees induced by these vertices according to the lexicographical order of their DFCSs. □

Theorem 3.5. *The above DFCF construction procedure has time complexity $O(ck \log k)$, where k is the number of vertices the tree has and c is the maximal degree of the vertices in the tree.*

Proof. Assuming there are k_h vertices in level h of the tree for $h = 0, 1, 2, \ldots$, to sort vertices at level h takes $O(k_h \log k_h)$ comparisons; the total number of comparisons for normalizing the whole tree is $\sum_h O(k_h \log k_h)$, which is $O(k \log k)$ (notice that $\sum_h (k_h \log k_h) \leq \sum_h (k_h \log k) = k \log k$); the time for each comparison is bounded by the maximal fan-out c of the tree because we can consider c as the length of the "keys" to be compared. □

Theorem 3.6. *The length of the DFCS for a labeled rooted unordered tree is at most $3k - 1$ where k is the number of vertices of the tree.*

Proof. In a DFCS, in addition to the *2k-1* symbols representing edge/vertex labels, there are "\$" symbols to represent backtracks, and a "#" symbol to represent the end of the DFCS. If we look at the procedure of depth-first traversal, we can see that each edge is visited at most twice: one forward visit and one backtrack. A backtrack will introduce a "\$" into the DFCS. In addition, there is no backtrack on the last-visited edge, but a "#" will be introduced into the DFCS instead. Because there are *k-1* edges in the tree, there are one "#" and at most *k-2* "\$" symbols in the DFCS. Therefore the length of the DFCS is at most $3k - 2$. For a special case, if the tree consists of a single vertex, the length of its DFCS is 2. So we change the bound on the DFCS to *3k-1*. □

The DFCF and the BFCF for a labeled rooted unordered tree may or may not be the same. The example in Figure 1 is a case in which they are the same and the example in Figure 2 is a case in which they are not.

Depth-first string encoding was first introduced by Zaki [24] for enumerating all rooted ordered trees. In [9], we defined the DFCF, as defined above, for both rooted unordered trees and free trees. Independent of our work in [9], Asai *et al.* [6] used a string encoding equivalent to depth-first string encoding to define the canonical form for rooted unordered tree in order to enumerate them. It will be seen later that with the BFCF, we are able to extend our algorithm to handle free trees easily. We will compare our algorithm based on the BFCF with algorithms based on the DFCF in the section on experiment results.

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 3.1. *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.*

Proof. We prove that by removing the last leg l from a BFCF, there is no change to the order between any two subtrees induced by a pair of sibling vertices. Assume t_1 and t_2 are two subtrees induced by a pair of sibling vertices and l belongs to t_2 (then l must be the last leg of t_2). In the first case if $BFCS(t_1) \leq BFCS(t_2)$, by removing the last leg, we change $BFCS(t_2)$ from "$\ldots l_2\$ \ldots \$ l_1 \#$" into "$\ldots l_2 \#$", where l_1 is the vertex label of the last leg, l_2 is the vertex label of the second-to-last vertex in the BFCF, and there are 0 or more

"\$"s in between. Therefore $BFCS(t_2)$ is increased so the original order between t_1 and t_2 still holds. In the second case if $BFCS(t_1) > BFCS(t_2)$, then because t_1 is to the right of t_2 in the BFCF, there is no leg of the BFCF belonging to t_1. So the order between t_1 and t_2 is determined at some level above the bottom level. In this case, removing a leg from the bottom level of t_2 does not change the original order between t_1 and t_2. (Notice that for the second case, in some extreme situations, removing the last leg from t_2 will result in $BFCS(t_1) = BFCS(t_2)$.) □

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 4 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.)

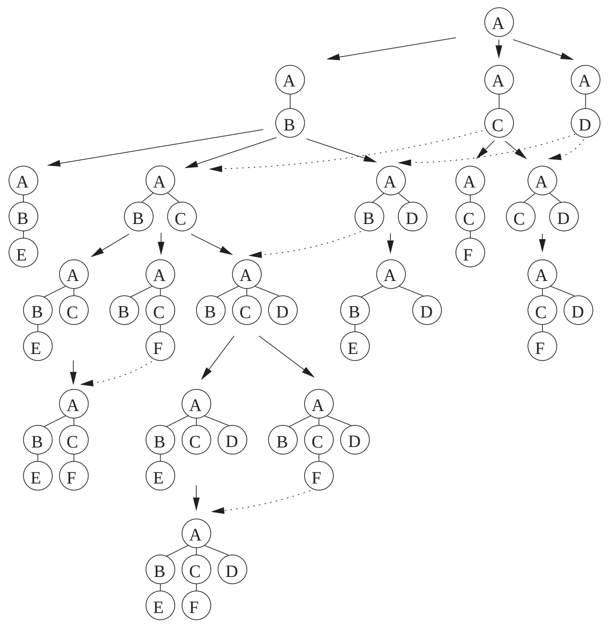


Figure 4: The Enumeration Tree for Rooted Unordered Trees in Their BFCFs

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 [13] for frequent subgraph mining, that allows a special local join operation in addition to the extension. If we look at Figure 4 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 4), 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 algorithm, which is based on two operations–extension and join, that takes advantage of both Apriori-like mining algorithms and vertical mining algorithms.

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 3.1. (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 3.2. (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 this pair of nodes 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 non-identity 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 5 create 9 candidate BFCFs because of the automorphisms of the parent shared by the pair of sibling nodes in the enumeration tree. From Figure 5 we can also see that self-join is necessary in growing an enumeration tree.

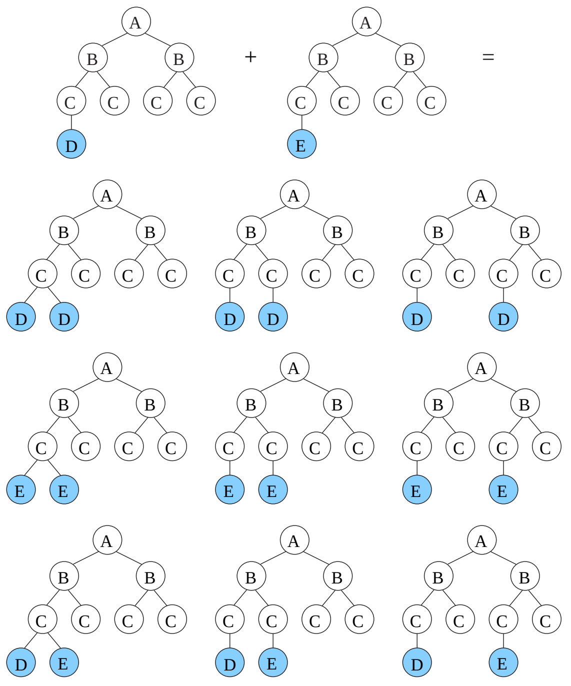


Figure 5: Automorphisms

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 3.3. *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,

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

The information on automorphisms can be obtained through the DFCF normalization procedure: after ordered vertices at all levels, we apply the following procedure top-down recursively: the root is the only member in its equivalence class; all children with the same order at a given level belong to the same equivalence class if their parents belong to the same equivalence class. Figure 6 gives a running example for obtaining automorphisms. It's obvious that this procedure of obtaining automorphisms has time complexity $O(ck \log k)$. Notice that the definition of automorphism is about equality, not about order, therefore it is independent of whether we choose the BFCF or the DFCF to uniquely represent a rooted unordered tree.

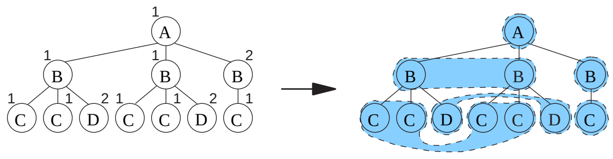


Figure 6: Obtaining Automorphisms

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 5 will be included in the enumeration tree. We will explain the details further in the next section.

3.5 Why Extension and Join Are Enough. Equipped with the two operations, extension and join, together with the handling of automorphisms, we have a systematic approach to efficiently growing the enumeration tree therefore enumerating all (frequent) rooted unordered trees. As we have seen from the previous section, when the parent node v of two sibling nodes v_1 and v_2 has automorphisms, the join operation is more complicated. In this section, we introduce the *automorphism variation* of a tree in BFCF to define more rigorously the join operation under automorphisms.

Definition 3.4. (automorphism variation) *For a tree t_v in its BFCF, the automorphism variations of t_v are those rooted ordered trees that are isomorphic to t_v and different from t_v only in the position of the last leg.*

If we take the automorphism variations of a BFCF tree into consideration while applying the join operation, our algorithm is complete, i.e., we can prove that any k-BFCF is either the result of extending a *(k-1)*-BFCF or the result of joining a pair of *(k-1)*-BFCFs (or a BFCF with an automorphism variation of a BFCF). The following lemma shows that a BFCF with 2 or more legs is guaranteed to be the result of either joining a pair of BFCFs or joining a BFCF with an automorphism variation of a BFCF.

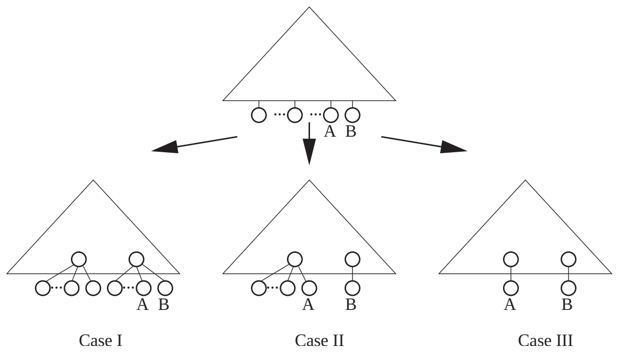


Figure 7: Three Cases for the Last Two Legs

LEMMA 3.2. *For a BFCF t_v of a rooted unordered tree with 2 or more legs, removing the second-to-last leg will result in a BFCF t_{v_2} for another rooted unordered tree or result in an automorphism variation of t_{v_2}. In addition, the BFCF t_{v_1} obtained by removing the last leg from t_v and the BFCF t_{v_2} share the same parent t_v in the enumeration tree. Furthermore, $BFCS(t_{v_1}) \leq BFCS(t'_{v_2})$ where t'_{v_2} is either t_{v_2} itself or any automorphism variation of t_{v_2}.*

Proof. In Figure 7, the tree at the top of the figure is the BFCF for a rooted unordered k-tree. Let's call the two rightmost legs A and B. Let's consider all the possible cases. For case I, A and B are siblings. In this case, it's very obvious that the *(k-1)*-tree obtained by removing A is still in its BFCF because removing A is equivalent to increasing A to B and increasing B to ∞. In case II, B does not have siblings and A has siblings. If as in this case, leg B turns out to be on the right side of leg A, then either the order of A's and B's ascendants has been resolved at higher level, or A and B have identical ascendants and A's left sibling(s) is less or equal to B. As a result, for case II, the *(k-1)*-tree obtained by removing A is still in its canonical form. In Case III, neither A nor B has siblings. Again, since A is on the left of B, either the order of A's and B's ascendants has been resolved at higher level, or A and B have identical ascendants and A is less than or equal to B. The only trouble is when A is less than B because if so, removing A will not result in a canonical form–for the canonical form, B should be moved to the old position of A. But this case only happens when A's and B's ascendants are identical and if that is so, removing A will result in an automorphism variation of a BFCF. $\square$

Because of the above lemma, a BFCF with 2 or more legs is guaranteed to be the result of joining its parent and one of the siblings (or one of its automorphism variations) of its parent in the enumeration tree. Therefore, together with the extension operation that creates all BFCFs with one leg, our algorithm enumerates all possible (frequent) rooted unordered trees in their BFCFs.

3.6 Support Counting. The *occurrence list* L 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 of L is of the form $(tid, i_1, \ldots, i_k)$, 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 of t_v we know if t_v is frequent, because the *support* of t_v is the number of elements in L with distinct tid's. For the join operation, we "combine" a pair of occurrence lists L_1 and L_2 for two BFCFs to get the occurrence list L_{12} for a new *(k+1)*-tree. The combination happens between any compatible pair of elements $l_1 \in L_1$ and $l_2 \in L_2$, 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 $l_{12} = (tid_1, i_1, \ldots, i_k, j_k) \in L_{12}$. For the extension operation, an occurrence list for a k-tree in BFCF is "expanded" to get the occurrence list for a new *(k+1)*-tree in BFCF. An element $l = (tid, i_1, \ldots, i_k)$ is expanded to $l' = (tid, i_1, \ldots, i_k, i_{k+1})$ if, in the tuple identified by tid in the database, $i_{k+1} \neq i_m$ for $m = 1, \ldots, k$ and the vertex with index i_{k+1} is a child of the vertex with index i_k.

3.7 Putting It All Together. Figure 8 summarizes our *HybridTreeMiner* algorithm. The main step in the algorithm is the function *Enum-Grow*, which grows the whole enumeration tree. Figure 9 gives the details of the function *Enum-Grow*. We can see that in the algorithm, two operations, join and extension, are applied separately.

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

```
Algorithm HybridTreeMiner($D$,$minsup$)
1:  $F_1$, $F_2$ ← {frequent 1, and 2-trees};
2:  $F \leftarrow F_1 \cup F_2$;
3:  $C \leftarrow sort(F_2)$;
4:  Enum-Grow($C$,$F$,$minsup$);
5:  return $F$;
```

Figure 8: The HybridTreeMiner Algorithm

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

Figure 9: The Enumeration Tree Growing Algorithm

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.

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* [4]. 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 10 shows a centered free tree and a bicentered free tree as well as the procedure to obtain the corresponding center and bicenters.

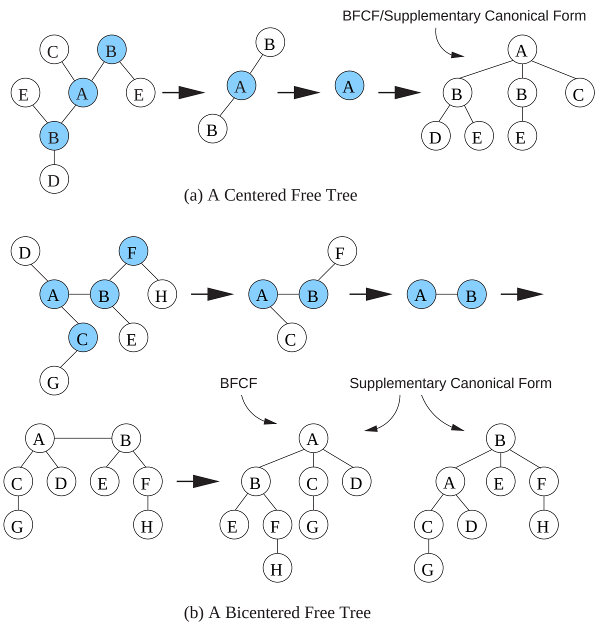

BFCF/Supplementary Canonical Form
(a) A Centered Free Tree
BFCF
Supplementary Canonical Form
(b) A Bicentered Free Tree

Figure 10: A Center Free Tree (above) and A Bicenter Free Tree (below) together with Their Canonical Forms

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 the 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 these pair of vertices the root to obtain a pair of rooted unordered trees. We call the canonical forms (BFCFs) for the pair of rooted unordered trees the *supplementary canonical forms* for the bicentered free tree. 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 that a BFCF for a free tree is a supplementary canonical form and 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 10, 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 4.1. *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 condition 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 greater than or equal to h-2.

Proof. For a tree t in its BFCF, we denote the root of t by r and the children of r by $r_1, \ldots, r_m$. The *only if* part: if t has 3 or more vertices and if r has only one child, then r will be removed in the first step of finding center/bicenter, so r cannot be the center or one of the bicenters; if r has more than one child, obviously the subtree induced by one of $r_1, \ldots, r_k$ has height *h-1*, let's assume this child is r_j. If none of the subtrees induced by other children of r has height greater than or equal to *h-2*, then r cannot be the center or one of the bicenters because the center (or the bicenters) must be some vertex (or vertices) of the subtree induced by r_j. The *if* part: W.L.O.G., we assume the subtree t_{r_1} induced by r_1 has height *h-1* and the subtree t_{r_2} induced by r_2 has height *h-1* or *h-2*, then the path from a bottom-level leaf of t_{r_1} to a bottom-level leaf of t_{r_2} has length *2h* or *2h-1* and r is the center or one of the bicenters of the path. Obviously, r is the center or one of the bicenters of the free tree. □

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 as given in Corollary 4.1 and Corollary 4.2.

COROLLARY 4.1. *Removing the last leg, i.e., the right-most leg, from a supplementary canonical form of a free tree will result in a supplementary canonical form for another free tree.*

Proof. Because a supplementary canonical form t_v of a free tree is the BFCF of a rooted unordered tree, so by Lemma 3.1 the tree t_{v_1} obtained by removing the last leg l from t_v is still the BFCF for a rooted unordered tree. Now if we can prove that t_{v_1} satisfies the two conditions in Theorem 4.1 then by that theorem t_{v_1} is a supplementary canonical form for a free tree. Assume l belongs to t_{r_1}, the subtree induced by r_1, where r_1 is a child of the root r of t_v; then the height of t_{r_1} is *h-1*. Removing l from t_{r_1} may or may not change the height of t_{r_1}. If it does not, then none of the subtrees induced by the children of the root r changes its height; since t_v is a supplementary canonical form, by Theorem 4.1 t_{v_1} is a supplementary canonical form as well. If removing l from t_{r_1} changes its height, the height of t_{r_1} is changed from *h-1* to *h-2*. Because t_v is a supplementary canonical form, by Theorem 4.1 the maximal height among all the subtrees induced by the children of r other than r_1 is either *h-1* or *h-2*. It is easy to see that in either case, the heights of the subtrees of t_{v_1} induced by the children of the root r satisfies the two conditions in Theorem 4.1 therefore t_{v_1} is a supplementary canonical form. □

COROLLARY 4.2. *For a supplementary canonical form t_v for a free tree with 2 or more legs, removing the second-to-last leg will result in a supplementary canonical form t_{v_2} for another free tree or result in an automorphism variation of t_{v_2}. In addition, the supplementary canonical form t_{v_1} obtained by removing the last leg from t_v and the supplementary canonical form t_{v_2} share the same parent t_v in the enumeration tree. Furthermore, $BFCS(t_{v_1}) \leq BFCS(t'_{v_2})$ where t'_{v_2} is either t_{v_2} itself or any automorphism variation of t_{v_2}.*

Proof. Because a supplementary canonical form for a free tree is a BFCF for a rooted unordered tree, by Lemma 3.2, we only have to prove t_{v_2} satisfies the two conditions given in Theorem 4.1. But if we study the second part of the proof of Corollary 4.1, we can see that the second part of that proof does not depend on whether l is the last leg. So we can use exactly the same proof. □

Figure 11 is the enumeration tree for free trees with A as the root of the supplementary canonical forms. Comparing Figure 11 with Figure 4 we note that Figure 11 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 4.1, 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,

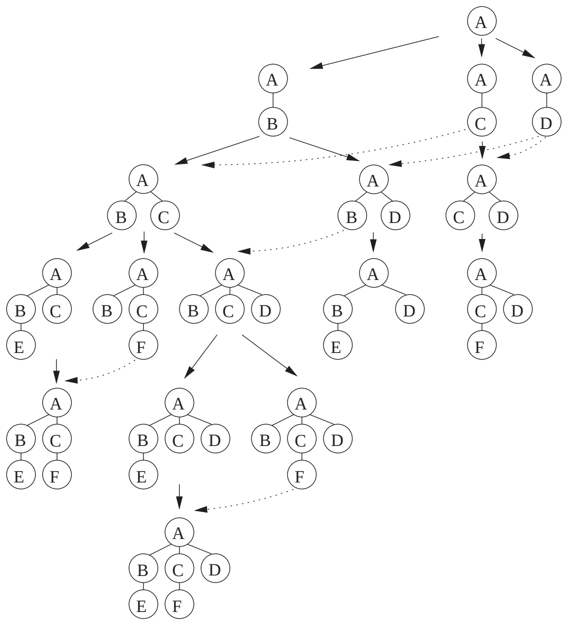


Figure 11: The Enumeration Tree for Free Trees in Their Supplementary Canonical Forms

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 4.1 the result is still a supplementary canonical form. The join 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. Now with these two operations revised, we can systematically grow the enumeration tree for mining frequent free trees. The mining algorithm given in Figure 8 and Figure 9 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.

5 Experiments.

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 Linux 7.3 operating system. All algorithms are implemented in C++ using the g++ 2.96 compiler.

5.1 Algorithms to Compare with.

We want to compare the performance of our algorithm with that of other known algorithms. To the best of our knowledge, the algorithm described in [6] is the only one that systematically mines all frequent subtrees in a database of rooted unordered trees. Unfortunately, no implementation was given in [6]. As a result, we have implemented the algorithm ourselves and call it *DFCF-extension* in the following discussion. Our implementation uses an enumeration tree based on our DFCF canonical form that is equivalent to the canonical form in [6]. To compare with the *DFCF-extension* algorithm, we have implemented two versions of the *HybridTreeMiner* algorithm: the first version, which we call *BFCF-extension*, uses extension as the only operation for growing enumeration trees; the second version, which we call *BFCF-hybrid*, uses both extension and join in growing enumeration trees. For mining frequent free trees, we compare the *BFCF-hybrid* version of our *HybridTreeMiner* algorithm with our previous *FreeTreeMiner* algorithm [9], which is an Apriori-like algorithm.

5.1.1 More Discussion.

If we consider the extension operation as the only operation in our enumeration tree growing (because as we mentioned before, the join operation is just a special guided extension), then we can compare the extension of a BFCF with the extension of a DFCF: for the BFCF, the extension operation corresponds to adding a new vertex so that the new vertex becomes the new *last leg* of the new BFCF (recall that the *last leg* of a BFCF is the rightmost leaf at the *bottom level*); for the DFCF, the extension operation corresponds to adding a new vertex so that the new vertex becomes the new *rightmost vertex* of the new DFCF (the *rightmost vertex* here means the last vertex that will be visited if we traverse a DFCF in a depth-first order). Therefore, the possible position for applying the extension operation for the BFCF is the *lower border* and for the DFCF is the *rightmost path*, as given in Figure 12.

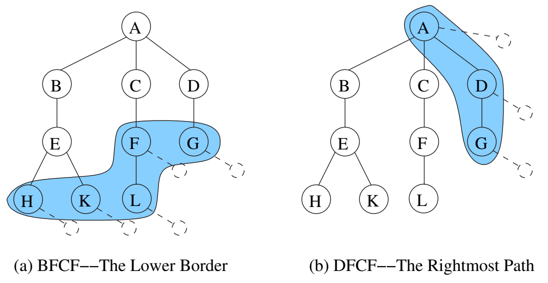


Figure 12: Extending the BFCF and the DFCF

In addition, because the enumeration tree enumerates all rooted unordered trees in their canonical forms, we never need to convert an arbitrary rooted unordered tree into its canonical form–we use the extension operation to add a new vertex to a rooted unordered tree in its canonical form so we only need to check if the resulting new tree is in the canonical form or not. As a result, the time complexities $O(k^2 c \log c)$ and $O(ck \log k)$ for normalizing a rooted unordered tree into the BFCF and the DFCF do not contribute to the complexity of our mining algorithm. Moreover, instead of extending and then checking whether the result is in the canonical form, we can compute the range of vertices that are allowable at a given position before starting the extension operation. Figure 13 gives a running example for applying the technique to the BFCF, while the same technique can be applied to the DFCF. In Figure 13, if we add a new vertex at the given position at the bottom, we may violate the BFCF by changing the order between some ancestor of the new vertex (including the vertex itself) and its immediate left sibling. So in order to determine the range of allowable vertex labels for the new vertex (so that adding the new vertex will guarantee to result in a new BFCF), we can check each vertex along the path from the new vertex to the root. In Figure 13, the result of comparison (1) is that the new vertex should have label greater than or equal to A, comparison (2) increases the label range to be greater than or equal to B, and comparison (3) increases the label range to be greater than or equal to C. As a result, before starting the extension operation, we know that adding any vertex with label greater than or equal to C at that specific position will surely result in a BFCF. Our implementation incorporates this technique.

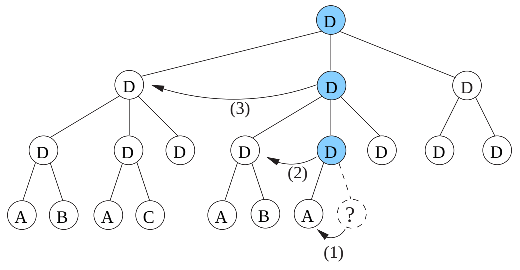


Figure 13: Computing the Range of New Legs

5.1.2 Time Complexity Analysis. From the technique introduced above we can derive an upper-bound for our frequent subtree mining algorithm. First, for the time to extend one node v in the enumeration tree (we assume that the tree t_v represented by v has k vertices and has height h): to compute the allowable vertex label range at each position at the lower border of the BFCF (or the rightmost path of the DFCF) takes $O(hk)$ time because there are at most h comparisons and each comparison takes $O(k)$ time; because the total number of positions at the lower border of the BFCF (or the rightmost path of the DFCF) is bounded by k, the total time for computing the allowable vertex labels at all possible positions is $O(hk^2)$. After computing the allowable range, we begin scanning the database; for each transaction in the database that supports t_v, we have to check for each position at the lower border of the BFCF (or the rightmost path of the DFCF) all possible new vertex the transaction can introduce; therefore the time for each transaction is $O(kc)$ where c is the maximal fan-out in the transaction. The total time for this step for the whole database is $O(|D|kc)$. So finally, the time complexity of our 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. This bound should be independent of whether we choose the BFCF or the DFCF in our enumeration tree. In addition, adding the join operation should not change this upper-bound.

5.2 Synthetic Data Generator. To generate synthetic data that reflect properties of real applications, instead of generating datasets of trees arbitrarily, we start from a graph that we call the *base graph.* To create the base graph, we use the universal Internet topology generator BRITE [17], developed by Medina et al at Boston University, that generates random graphs simulating Internet topologies with some specific network

Table 1: Parameters for Synthetic Generators

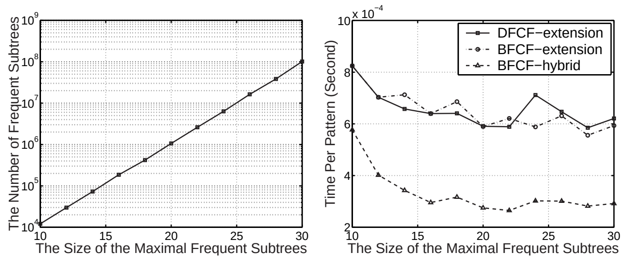
Parameter	Description
$\|D\|$	the number of transactions in the database
$\|T\|$	the size of each transaction in the database
$\|I\|$	the maximal size of frequent subtrees
$\|N\|$	the number of frequent subtrees with size $\|I\|$
$\|S\|$	the minimum support [%] for frequent subtrees
$\|L\|$	the size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

characteristics, such as the link bandwidth. We use the bandwidths of the links as the edge labels of our base graph and assign the vertex labels to the base graph uniformly. The base graph created by BRITE has the following characteristics: the number of vertex labels is 10; the number of edge labels is 10; the number of vertices is 1000; the average degree for each vertex in the base graph is 20. Starting from the base graph, we create datasets of trees with controlled parameters. Table 1 provides these parameters and their meanings. Note that the size of transactions and trees are defined in terms of the number of vertices.

The detailed procedures that we followed to create the synthetic datasets are as follows: starting from the base graph, 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 $|D|\cdot|S|\cdot 100$ transactions; each of these $|D|\cdot|S|\cdot 100$ 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 $|L|$. In particular, $|L|$ is both the number of distinct edge labels as well as the number of distinct vertex labels.

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 fre-



(A) (B)

Figure 14: Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees

quent 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 30. Figure 14(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 14(B) shows the average time to mine each frequent subtrees, using each of the three methods: *DFCF-extension*, *BFCF-extension*, and *BFCF-hybrid*. 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 14(B) we see that *DFCF-extension* has similar performance as *BFCF-extension*. This is because the enumeration trees based on the DFCF and the BFCF should have the same number of nodes. In addition, the figure shows that *BFCF-hybrid* is about 2 times faster than the other two methods, which demonstrates the effectiveness of the join operation in the growing of an enumeration tree.

Next, we want to check how sensitive the running time is to the size of the database. We created a set of databases with the same number ($|N|$ = 100) of seed trees embedded. With $|T|$=50 and $|I|$=20, we increased

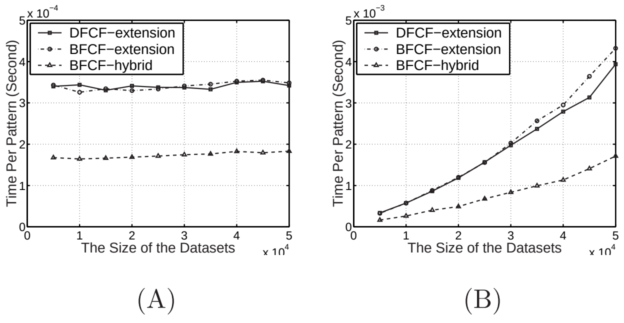


Figure 15: Running Time vs. Size of the Datasets

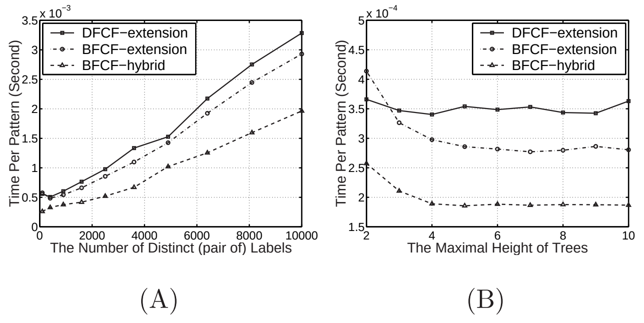
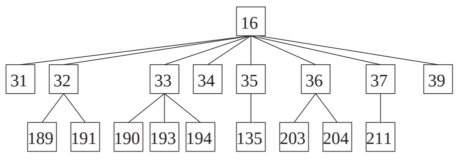


Figure 16: Running Time vs. Other Parameters

the number of transactions $|D|$ from 5000 to 50000 to get different databases. In the first test, we fix the occurrence of each seed tree to be 50. As a result, the support $|S|$ decreases from 1% to 0.1% as $|D|$ increases. The experiment result is given in Figure 15(A). As revealed by the figure, for a fixed number of maximal frequent subtrees, the running time is not sensitive to the size of the database. This result seems to contradict to the bound $O(|F| \cdot (hk^2 + |D|kc)$ given in the previous section. We believe that this is because in computing the upper-bound, we have assumed the worst case scenario in which each transaction has at least one frequent subtree embedded in it; however, if frequent subtrees occur only in some transactions, then by the occurrence list we only need to check those transactions with frequent subtrees embedded. To verify our belief, in the second test, as $|D|$ increases, we fix the support $|S|$ to be 1% (as a result, the occurrence of each frequent subtree increases proportionally to the size $|D|$ of the databases). Figure 15(B) gives the performance for this test. As we can see from the figure, the average time for mining each pattern increases with the size $|D|$, which verifies our argument.

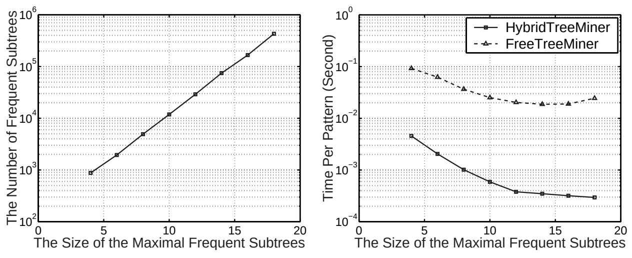
Next, we study the effects of some other parameters on the performance of our algorithm. First, in our time complexity analysis, we assume that the number of distinct labels is fixed so that it only contributes a constant factor to the time complexity. Now we check this assumption. We fixed other parameters ($|D|$=10000, $|T|$=50, $|I|$=20, $|N|$=100, $|S|$=1%) while changing $|L|$ from 10 to 100. Notice that, for example when $|L|$ is 100, there are 100 distinct vertex labels and 100 distinct edge labels, so totally there are 10000 distinct pairs of combinations. As can be seen from Figure 16(A), the running time increases linearly with the number of these pairs of combinations. Second, we study whether the "shape" of trees affects the performance of our algorithms. When generating the synthetic trees, by fixing all other parameters but increasing the heights (from 2 to 10), we get different families of trees where trees in each family change from *flat* (when the maximal height is small) to *tall* (when the maximal height is large). Figure 16(B) shows the running time for different families of trees. It is very interesting that when the trees are extremely flat (when the maximal height is 2 or 3), the performance of *BFCF-extension* and *BFCF-hybrid* deteriorates a little. This is counter-intuitive because we expected *BFCF-extension* and *BFCF-hybrid* to scan the database fewer times than *DFCF-extension* when the trees are flat. We hypothesize that the reason for this result is that although *DFCF-extension* scans the database more times, it has higher pruning power because a lot of siblings can help to determine the label range of a newly-grown vertex.

5.3.2 Web Access Trees. In this section, we present an application on mining frequent accessed webpages from web logs. We ran experiments on the log files at UCLA Data Mining Laboratory (http://dml.cs.ucla.edu). First, we used the WWW-Pal system [19] 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 webpages. 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 17 shows this maximum subtree. It turns out that this subtree is a part of web site for the ESP^{2}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 ESP^{2}Net project.



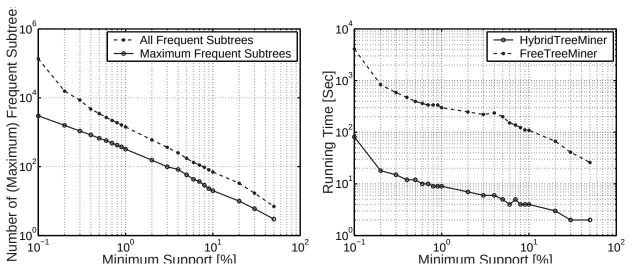
16	http://dml.cs.ucla.edu/projects/dml_esip/
31	http://dml.cs.ucla.edu/projects/dml_esip/Objectives/objectives.html
32	http://dml.cs.ucla.edu/projects/dml_esip/Status/status.html
33	http://dml.cs.ucla.edu/projects/dml_esip/Technology/technology.html
34	http://dml.cs.ucla.edu/projects/dml_esip/Science/science.html
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 17: The Maximum Frequent Subtree Mined From Web Log Files



(A) (B)

Figure 18: Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees



(A) (B)

Figure 19: Number of (Maximum) Frequent Subtrees and Running Time vs. Support for the Chemical Compound Dataset

5.4 Results on Free Trees. In this section, we report our experiments on datasets of free trees. For free trees, the enumeration tree based on the DFCF does not work. Therefore we only use the enumeration tree based on the BFCF and compare the *BFCF-hybrid* version of our *HybridTreeMiner* algorithm with the *FreeTreeMiner* algorithm, 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 18 gives the results. Figure 18(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 18(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.

5.4.2 The Chemical Compound Dataset. This dataset was described in [9]. It contains 17,663 tree-structured chemical compounds sampled from a graph dataset of the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) [18]. 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 19(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 19(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.

6 Conclusion.

In this paper, we presented a novel canonical form, the breadth-first canonical form (BFCF), 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. We also defined two operations, extension and join, to efficiently grow the enumeration 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 the previously known free tree subtree mining algorithm. The following are some additional observations.

6.1 Canonical Forms. Different canonical forms have been introduced by researchers to represent graphs. The one defined by Inokuchi *et al.* [14] is based on the lexicographical order among the adjacency matrix of a graph. The canonical form for graphs given by Yan *et al.* [22, 23] is defined based on the depth-first graph traversal while the one given by Huan *et al.* [13] is defined based on the breadth-first graph traversal. However, because it is a problem not known to be in P or NP-complete, there is no known efficient algorithm to obtain a canonical form for a graph. All of the above work has used some permutation-and-test techniques. On the other hand, we have given in this paper polynomial algorithms for normalizing trees. The canonical form for trees was discussed in [24], [6], and [9]. In all three cases the canonical form is defined by the depth-first traversal. In contrast, our canonical form for trees in this paper is defined by the breadth-first traversal. The two canonical forms are similar to each other, but the breadth-first canonical form makes it possible to introduce the join operation in enumeration tree growing, which is shown in our experiments to be very effective in improving the running time of our algorithm. 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.

6.2 Enumeration Trees. 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). In other words, the simpler the tree structure, the more difficult to represent the tree as a node in the enumeration tree. This is because for all three types of trees, the final representation (the canonical form) must be rooted ordered trees.

6.3 Future Directions. First, from the experiment results we can see that the number of subtrees grows exponentially with respect to the size of the tree. As a result, efficient algorithms to mine *maximum* frequent trees, instead of mining *all* frequent subtrees, are called for. Second, we are working on algorithms that mine frequent subgraphs with mining frequent spanning trees as the first step. Third, it is our belief that vertices in the graph do not necessarily have only a single label–each vertex can have multiple attributes. We will extend our algorithm to handle multiple-attributes labels in the future.

Acknowledgements.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116 and 0085773. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

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

[3] A. V. Aho, J. E. Hopcroft, and J. E. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[5] 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*, April 2002.

[6] 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.

[7] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of the ACM SIGMOD*, June 1998.

[8] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava.

Counting twig matches in a tree. In *ICDE*, pages 595–604, 2001.

[9] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)*, November 2003. Full version available as Technical Report CSD-TR No. 030041 at ftp://ftp.cs.ucla.edu/tech-report/2003-reports/030041.pdf.

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

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability–A Guide to the Theory of NP-Completeness.* W. H. Freeman And Company, New York, 1979.

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

[13] 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.

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

[15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, November 2001.

[16] T. Liu and D. Geiger. Approximate tree matching and shape similarity. In *International Conference on Computer Vision*, September 1999.

[17] 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.

[18] National Cancer Institute (NCI). DTP/2D and 3D structural information. World Wide Web, ftp://dtpsearch.ncifcrf.gov/jan03_2d.bin, 2003.

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

[20] 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.

[21] A. Termier, M-C. Rousset, and M. Sebag. TreeFinder: a first step towards xml data mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 450–457, 2002.

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

[23] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.

[24] M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002.