Under consideration for publication in Knowledge and Information Systems

Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining

Yun Chi, Yirong Yang, and 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 first present two canonical forms for labeled rooted unordered trees–the breadth-first canonical form (BFCF) and the depth-first canonical form (DFCF). Then the canonical forms are applied to the frequent subtree mining problem. Based on the BFCF, we develop a vertical mining algorithm, *RootedTreeMiner*, to discover all frequently occurring subtrees in a database of labeled rooted unordered trees. The *RootedTreeMiner* algorithm uses an enumeration tree to enumerate all (frequent) labeled rooted unordered subtrees. Next, we extend the definition of the DFCF to labeled free trees and present an Apriori-like algorithm, *FreeTreeMiner*, to discover all frequently occurring subtrees in a database of labeled free trees. Finally, we study the performance and the scalability of our algorithms through extensive experiments based on both synthetic data and datasets from real applications.

Keywords: canonical form; frequent subtree; labeled rooted unordered tree; labeled free tree; tree isomorphism

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

Received September 20, 2003
Revised April 9, 2004
Accepted May 8, 2004

and attribute-value relationships; in web page traffic mining, access trees are used to represent the access patterns of different users (Zaki, 2002). 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 (Hein, Jiang, Wang and Zhang, 1996); in pattern recognition, a free tree called *shape axis tree* is used to represent shapes (Liu and Geiger, 1999); in computer networking, unrooted multicast trees are used for packet routing (Cui, Kim, Maggiorini, Boussetta and Gerla, 2002). 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 either 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.* (Inokuchi, Washio and Motoda, 2000) presented an algorithm, AGM, and Kuramochi *et al.* (Kuramochi and Karypis, 2001) presented another algorithm, FSG, both for mining subgraphs in a graph database. The two algorithms used a level-wise Apriori (Agrawal and Srikant, 1994) approach: the AGM algorithm extends subgraphs by adding a vertex per level and the FSG algorithm by adding an edge. Yan *et al.* (Yan and Han, 2002; Yan and Han, 2003) and Huan *et al.* (Huan, Wang and Prins, 2003) 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 (Garey and Johnson, 1979); 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 (Garey and Johnson, 1979). 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 partially because of the increasing popularity of XML in databases. Chen *et al.* (Chen, Jagadish, Korn, Koudas, Muthukrishnan, Ng and Srivastava, 2001) provided data structures and algorithms to accurately estimate the number of occurrences of a small tree (twig) in a large tree. Termier *et al.* (Termier, Rousset and Sebag, 2002) presented an algorithm, *TreeFinder*, which finds a subset of frequent trees in a set of tree-structured XML data. Shasha *et al.* (Shasha, Wang and Giugno, 2002) 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 (Termier et al., 2002; Shasha et al., 2002), however, does not guarantee completeness, i.e., some frequent subtrees may not be in the search results. In a recent paper, Zaki (Zaki, 2002) presented an algorithm called *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 (Asai, Abe, Kawasoe, Arimura, Satamoto and Arikawa, 2002) Asai *et al.* presented algorithm FREQT to discover frequent rooted ordered subtrees. They used a string encoding similar to that defined by Zaki (Zaki, 2002) and

built an enumeration tree for all (frequent) rooted ordered trees. The *rightmost expansion* is used to grow the enumeration tree.

Asai *et al.* (Asai, Arimura, Uno and Nakano, 2003), Nijssen *et al.* (Nijssen and Kok, 2003), and Rückert *et al.* (Rückert and Kramer, 2004) independently proposed canonical forms for labeled trees that are similar to the ones that we will be describing in this paper. We introduce their results and compare them with our work in the following sections.

1.2. Contributions of This Paper

The main contributions of this paper are: (1) We introduce two new canonical forms, which are based on breadth-first traversal and depth-first traversal respectively, to uniquely represent a labeled rooted unordered tree. We give the procedures to construct the canonical forms and study some properties of the canonical forms. (2) In order to mine frequent labeled rooted unordered subtrees, based on our breadth-first canonical form, we define an enumeration tree to systematically enumerate all (frequent) labeled rooted unordered trees. (3) Then we extend our definition of canonical form to labeled free trees and introduce an Apriori-like algorithm to mine all (frequent) labeled 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 algorithms.

The rest of the paper is organized as follows. In the reminder of Section 1, we give the background that includes the concepts we will be using in the paper and the definition of the frequent subtree mining problem. Section 2 introduces the two canonical forms. Section 3 presents our algorithm, *RootedTreeMiner*, for mining frequent subtrees in a database of labeled rooted unordered trees. The algorithm is based on an enumeration tree that enumerates all (frequent) subtrees in their breadth-first canonical form. Section 4 presents our algorithm, *FreeTreeMiner*, for mining frequent subtrees in a database of labeled free trees. This algorithm is based on the extended definition of the depth-first canonical form for free trees. Section 5 includes experiments and performance analysis. Finally, Section 6 concludes our work and gives future directions.

1.3. 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. In this paper, we assume that the elements in Σ are atomic but Σ is not necessarily finite. 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

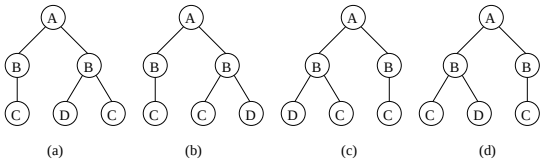


Fig. 1. Four Rooted Ordered Trees Obtained from the Same Rooted Unordered Tree

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.

2. Canonical Forms for Labeled Rooted Unordered Trees

From a labeled rooted unordered tree we can derive many labeled rooted ordered trees, as shown in Figure 1. From these labeled rooted ordered trees we want to uniquely select one as the canonical form to represent the corresponding labeled rooted unordered tree. It turns out that there are at least two ways we can define the unique representation–one based on the breadth-first traversal and the other based on the depth-first traversal of the tree. We define these two representations and study their properties in this section.

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 the 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.

2.1. The Breadth-First Canonical Form

We first define the *breadth-first string encoding* for a rooted ordered tree. Assume there are two special symbols, "$" and "#", which are not in the alphabet of edge labels and vertex labels. We also assume that (1) there exists a total ordering among edge and vertex labels, and (2) "#" sorts greater than "$" and both sort greater than any other symbol in the alphabet of vertex and edge labels. The breadth-first string encoding of a rooted ordered tree is obtained by traversing the tree in a *breadth-first* order (also called *level-order* traversal (Valiente, 2002)),

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. Now we prove the one-to-one correspondence between a labeled rooted ordered tree and its breadth-first string encoding.

Lemma 2.1. Each labeled rooted ordered tree corresponds to a unique breadth-first string encoding; each valid breadth-first string encoding corresponds to a unique labeled rooted ordered tree.

Proof. The first statement is implied by the definition of the breadth-first string encoding. We prove the second statement by induction on the number of vertices k in a labeled rooted ordered tree. For the base case, when $k = 1$, the valid breadth-first string encoding are of the form $l\#$ where l is the label of the single vertex. In this case, the corresponding labeled rooted ordered tree is a single vertex, which is unique. For the induction step, we assume that for each breadth-first string encoding S_n with $k = n$ vertices, there is a unique labeled rooted ordered tree K_n corresponds to it. We indicate the index for the parent vertex of the n-th vertex in K_n as m. A valid breadth-first string encoding S_{n+1} with $k = n + 1$ vertices is of the form $S_n\$ \ldots \$l\#$. Obviously S_n determines a unique labeled rooted ordered tree with n vertices. In addition, the last vertex (with label l) either becomes the only right sibling of vertex n, if there is no "$" between S_n and l in S_{n+1}, or the only child of a certain vertex with index i, where $m < i \leq n$ and i is uniquely determined by the number of "$" between S_n and l in S_{n+1}. In both cases, the labeled rooted ordered tree K_{n+1} corresponding to S_{n+1} is determined uniquely. □

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 lexicographic 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. Notice that the total ordering on strings and the uniqueness of the breadth-first string encoding for a labeled rooted ordered tree guarantee the uniqueness of the BFCS and the BFCF for a 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.

Theorem 2.1. The above construction procedure gives the BFCF for a labeled rooted unordered tree.

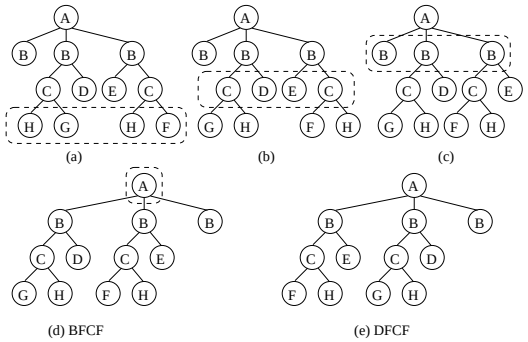


Fig. 2. To Obtain the BFCF of a Labeled Rooted Unordered Tree

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 lexicographic 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 2.2. 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 2.3. 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 $3k$. □

2.2. The Depth-First Canonical Form

Now we introduce a second way to define a canonical form. We first define the *depth-first string encoding* for a rotated ordered tree through a *depth-first* traversal (also called the *pre-order* traversal (Valiente, 2002)) and use "$" to represent a backtrack and "#" to represent the end of the string encoding. Similar to the case of the breadth-first string encoding, there is a one-to-one correspondence between a labeled rooted ordered tree and its depth-first string encoding.

Lemma 2.2. Each labeled rooted ordered tree corresponds to a unique depth-first string encoding; each valid depth-first string encoding corresponds to a unique labeled rooted ordered tree.

Proof. The first statement is implied by the definition of the depth-first string encoding. We prove the second statement by induction on the number of vertices k in a labeled rooted ordered tree. For the base case, when $k = 1$, the valid depth-first string encoding are of the form $l\#$ where l is the label of the single vertex. In this case, the corresponding labeled rooted ordered tree is a single vertex, which is unique. For the induction step, we assume that for each depth-first string encoding S_n with $k = n$ vertices, there is a unique labeled rooted ordered tree K_n corresponds to it. A valid depth-first string encoding S_{n+1} with $k = n + 1$ vertices is of the form $S_n\$ \ldots \$l\#$. Obviously S_n determines a unique labeled rooted ordered tree with n vertices. In addition, the last vertex (with label l) becomes the rightmost child of vertex n or one of the ancestors of vertex n, depending the number of "$" between S_n and l in S_{n+1}. As a result, the labeled rooted ordered tree K_{n+1} corresponding to S_{n+1} is determined uniquely. □

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. Again, the total ordering on strings and the uniqueness of the depth-first string encoding for a labeled rooted ordered tree guarantee the uniqueness of the DFCS and DFCF for a 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.* (Aho, Hopcroft and Ullman, 1974). 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

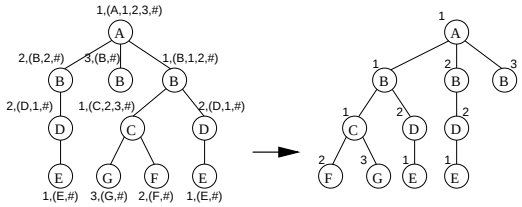


Fig. 3. To Obtain the DFCF of A Rooted Unordered Tree

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.

Theorem 2.4. 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 lexicographic 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 lexicographic 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 lexicographic 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 lexicographic order of their DFCSs. □

Theorem 2.5. 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 compar-

ison 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 2.6. 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.

2.3. Canonical Form Properties

With canonical forms, we introduce a unique representation for labeled rooted unordered trees. With such a unique representation, a traditional indexing method such as hash tables can be used on databases of labeled rooted unordered trees. In addition, we also introduce a total order among labeled rooted unordered trees. Hence we can apply traditional database indexing methods that depend on such a total order, such as B-trees, on databases of labeled rooted unordered trees. It will be shown shortly that the canonical forms, with the above desirable properties, can be extended to labeled free trees as well. In the implementations of our algorithms that will be introduced in the following sections, we have used memory-resident data structures and used the container *maps* in the C++ standard library (which are implemented as balanced binary trees) to store frequent subtrees and candidates for frequent subtrees. We have used the canonical strings of the frequent subtrees as the key for the maps. In future work, we plan to extend the algorithms to disk-resident data structures such as B-trees.

2.4. Relationship to Other Canonical Forms

In (Aho et al., 1974), Aho *et al.* discussed the tree isomorphism problem and gave a bottom-up algorithm, with linear time complexity, to check whether or not two unlabeled rooted unordered trees are isomorphic. Their algorithm also works for labeled rooted unordered trees, with the same time complexity if we assume the vertex labels to be integers in the range between 1 and n (so that a linear sorting, e.g., radix sorting, can be used). We have adopted Aho's algorithm in our DFCF normalization procedure (as shown in Figure 3). However, Aho's algorithm compared two rooted unordered trees but did not normalize a given rooted unordered tree as we did. In (Valiente, 2002), Valiente presented a very detailed discussion on the tree isomorphism problem for both rooted

ordered trees and rooted unordered trees. For a rooted unordered tree T, Valiente defined the *isomorphism code* of T as a sequence of integers in the range between 1 and $|T|$: *Code[root[T]]=[size[root[T]]];Code[*w_1*];...;Code[*w_k*]*, where vertices $w_1, \ldots, w_k$ are the children of the root of T arranged in nondecreasing lexicographic order of isomorphism code. This isomorphism code defined a unique representation for a rooted unordered tree and introduced a linear ordering among all rooted unordered trees, as we did in the canonical strings and the canonical forms. Although the isomorphism code is defined for unlabeled trees, it is straightforward to extended the definition to labeled trees. The main distinction between Valiente's isomorphism code and our canonical strings is that isomorphism code explicitly records the sizes of all (sub)trees while our canonical strings do not. Because of this distinction, we can add a new vertex to the end of a canonical string or remove a vertex from the end of a canonical string and the result is a canonical string. (In later sections we will see that these operations are very important in frequent subtree mining.) In contrast, adding a new vertex to or removing a vertex from a tree in Valiente's isomorphism code requires a global update (at least for all ancestors of the vertex) to the isomorphism code, because the sizes of all subtrees rooted at ancestors of the newly added or removed vertex will change. In (Buss, 1997), Buss studied the tree isomorphism, the tree comparison, and the tree canonization problems. The canonical string representation in (Buss, 1997) is equivalent to the isomorphism code in (Valiente, 2002). Buss showed that tree isomorphism, tree comparison and tree canonization can all be solved in alternating logarithmic time. However, all the algorithms and complexity analysis in (Buss, 1997) are oriented toward *unlabeled* trees. In other words, only the topological structures are used to distinguish the trees. In our canonical strings and canonical forms, the lexicographic order of vertex and edge labels play an equally important role in tree isomorphism, tree comparison, and tree canonization.

Independent of our work (Chi, Yang and Muntz, 2003), Asai *et al.* (Asai et al., 2003) and Nijssen *et al.* (Nijssen and Kok, 2003) defined equivalent string encodings for labeled rooted ordered trees. In their string encoding, the depth-first traversal is used and the depth of a vertex is explicitly recorded in the string encoding. According to their definitions, the string encoding for *tree (a)* in Figure 1 is $((0, A), (1, B), (2, C), (1, B), (2, D), (2, C))$, where each pair represents a vertex: the first number in the pair is the depth of the vertex, and the second symbol is the vertex label. Both Asai *et al.* (Asai et al., 2003) and Nijssen *et al.* (Nijssen and Kok, 2003) defined the canonical form for a labeled rooted unordered tree as the labeled rooted ordered tree that gives the lexicographically minimal string encoding. Their definitions can be shown to be equivalent to our DFCF.

3. Mining Frequent Labeled Rooted Unordered Trees

We now apply the canonical forms for the labeled trees to the problem of mining frequent subtrees. In this section, we will show a method to use the BFCFs for labeled rooted unordered trees to discover all frequently occurring subtrees from a database of labeled rooted unordered trees. In the next section, we will extend the DFCF from rooted trees to free (or un-rooted) trees and develop an Apriori-like algorithm for mining all frequent free subtrees from a database of free trees. We first define the frequent subtree mining problem.

3.1. The Frequent Subtree Mining Problem

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.

Closely related to the frequent subtree mining problem is the market-basket association rule mining, whose first step is to mine all frequent itemsets. Two categories of algorithms are used in frequent itemset mining. The first category of algorithms put all frequent itemsets in an enumeration lattice and traverse the lattice level by level. Apriori (Agrawal and Srikant, 1994) 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 (Bayardo, 1998), or following a depth-first traversal order, as in (Agarwal, Aggarwal and Prasad, 2001). 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 vertical mining algorithm, *RootedTreeMiner*, that uses an enumeration tree, which is based on the BFCF, to enumerate all (frequent) labeled rooted unordered subtrees. In the next section, we will give an Apriori-like algorithm, *FreeTreeMiner*, that mines all frequent labeled free subtrees, using an index based on the DFCF.

3.2. The Enumeration Tree

Enumeration trees are used extensively in frequent itemsets mining (Agarwal et al., 2001; Bayardo, 1998). The enumeration tree technique was first used for frequent subtree mining by Asai *et al.* in (Asai et al., 2002), where an enumeration tree is used to enumerate (frequent) labeled rooted ordered trees. Later, Asai *et al.* extended the enumeration tree to mining labeled rooted unordered trees (Asai et al., 2003). Independently, Nijssen *et al.* proposed a similar technique in (Nijssen and Kok, 2003). Both Asai *et al.* and Nijssen *et al.* built their enumeration trees based on canonical forms defined by the depth-first traversal. Here, we define an enumeration tree that enumerates all rooted unordered trees based on their BFCFs, which are defined by the breadth-first traversal.

To build an enumeration tree, the key issue is to define a unique parent for each pattern. 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:

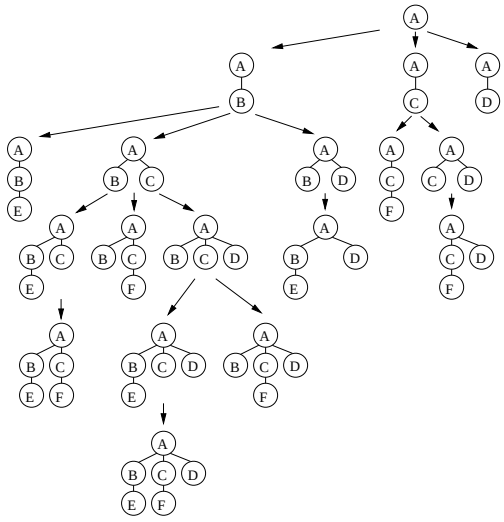


Fig. 4. The Enumeration Tree for Rooted Unordered Trees in Their BFCFs

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.

3.3. Growing the Enumeration Tree

We want to grow the enumeration tree efficiently. Starting from a node v of the enumeration tree, we need to find all valid children of v. Each child of v is

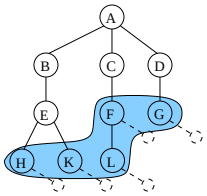


Fig. 5. Extending the BFCF

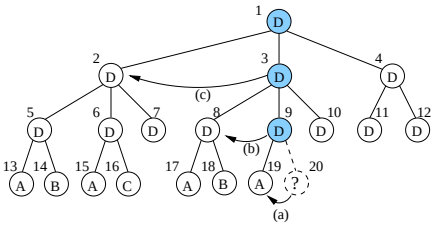


Fig. 6. Computing the Range of New Legs

obtained by adding a new vertex to v 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*). Therefore, the possible position for adding the new *last leg* to a BFCF is the *lower border* of the BFCF, as shown in Figure 5.

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–after adding a new vertex to a rooted unordered tree in its canonical form, we only need to check if the resulting new tree is in the canonical form or not. As a result, the time complexity $O(k^2 c \log c)$ for normalizing a rooted unordered tree into the BFCF does not contribute to the complexity of our mining algorithm. Moreover, instead of adding a vertex 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 adding vertices.

Figure 6 gives an example on how we compute the allowable range. In Figure 6, we want to add a new rightmost leg to the given position of the BFCF tree with 19 vertices. By adding 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. 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 check each vertex along the path from the new vertex to the root. For the convenience of discussion, we attach an index number to each vertex and use $T(k)$ to represent the maximal subtree rooted at vertex k (i.e., the vertex k plus all its descendants), and $BFCF(T(k))$ to represent the BFCF of $T(k)$.

For example, in Figure 6, the result of comparison (a) is that $BFCF(T(19)) =$ "$A\#$" while $BFCF(T(20)) =$ "?#", and we need $BFCF(T(19)) \leq BFCF(T(20))$. So the label of the new vertex must satisfy ? $\geq A$. Similarly, for comparison (b), $BFCF(T(8)) =$ "$D\$AB\#$", $BFCF(T(9)) =$ "$D\$A?\#$", and we require $BFCF(T(8)) \leq BFCF(T(9))$. So the label of the new vertex must satisfy ? $\geq B$. Finally, for comparison (c), $BFCF(T(2)) =$ "$D\$DDD\$AB\$AC\#$", $BFCF(T(3)) =$ "$D\$DDD\$AB\$A?\#$", and we require $BFCF(T(2)) \leq BFCF(T(3))$. So the label of the new vertex must satisfy ? $\geq C$.

In general, we have the following observations. The first observation is that adding a new vertex v (vertex 20 in Figure 6) to a BFCF will change the string encodings of the subtrees rooted at v's ancestors ($T(1)$, $T(3)$, $T(9)$, and $T(20)$ in Figure 6). The second observation is that because adding a new vertex decreases the order of a string encoding, we only need to compare the new subtrees rooted at v's ancestors other than the root ($T(3)$, $T(9)$, and $T(20)$ in Figure 6) with the subtrees rooted at their corresponding left siblings ($T(2)$, $T(8)$, and $T(19)$ in

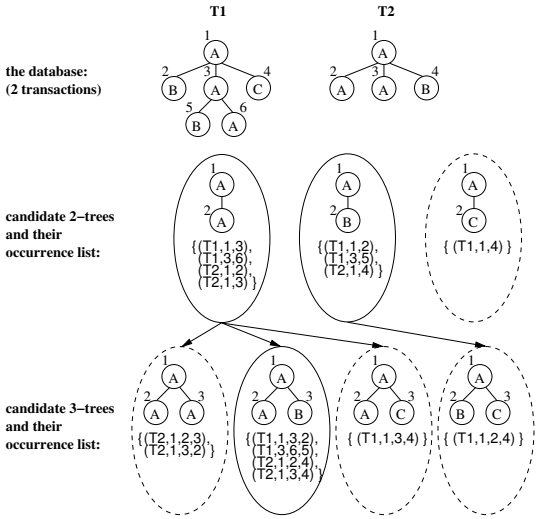


Fig. 7. The Example for Support Counting

Figure 6) in order to obtain the range of the valid labels for the new vertex. Our *RootedTreeMiner* algorithm incorporates this technique in the *Compute-Range* function given in Figure 8.

3.4. 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 adding a new leg, 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. 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 $m = 1, \ldots, k$.

Figure 7 gives an example for support counting. On the top of Figure 7 is a database consists of 2 transactions. Below the database is a portion of the enumeration tree with candidate 2-trees and candidate 3-trees (whose roots have label A) together with their occurrence list. The frequent subtrees are encircled by solid lines and the infrequent candidate trees are encircled by dashed lines (with *minsup*=2).

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

```
1:  F_1, F_2 ← {frequent 1 and 2-trees, sorted};
2:  F ← F_1 ∪ F_2;
3:  Enum-Grow(F_2,F_1,F,minsup);
4:  return F;
```

Algorithm **Enum-Grow**(C,F_1,F,$minsup$)

```
1:  for i ← 1,...,|C| do
2:      E ← ∅;
3:      for each vertex v_m on the lower border of c_i do
4:          minLabel,maxLabel ← Compute-Range(c_i, F_1, v_m);
5:          for each minLabel≤ l ≤maxLabel do
6:              e ← c_i plus a new last leg with label l and parent v_m;
7:              if support(e) ≥ minsup then E ← E ∪ e;
8:      F ← F ∪ E;
9:      Enum-Grow(E,F_1,F,minsup);
```

Algorithm **Compute-Range**(c,F_1,v_m)

```
1:  minLabel ← min(F_1);
2:  maxLabel ← max(F_1);
3:  k ← sizeof(c);
4:  c' ← c plus a new last leg v_{k+1} with label minLabel and parent v_m;
5:  if v_m has at least one child then
6:      label(v_{k+1}) ← label(v_k);
7:  for each ancestor v_a of v_m that is not the root do
8:      v_{a'} ← the left sibling of v_a in c';
9:      S(T(v_a)) ← string encoding of the subtree rooted at v_a;
10:     S(T(v_{a'})) ← string encoding of the subtree rooted at v_{a'};
11:     if S(T(v_{a'})) > S(T(v_a)) then
12:         S'(T(v_a)) ← S(T(v_a)) with label(v_{k+1}) replaced by maxLabel;
13:         if S(T(v_{a'})) > S'(T(v_a)) then
14:             minLabel ← maxLabel+1; return;
15:         else label(v_{k+1}) ← the minimum label l ∈ F_1 that
                                  makes S(T(v_{a'})) ≤ S(T(v_a));
16: return label(v_{k+1}), maxLabel;
```

Fig. 8. The RootedTreeMiner Algorithm

3.5. Putting It Together

Figure 8 summarizes our *RootedTreeMiner* algorithm. The main step in the algorithm is the function *Enum-Grow*, which grows the whole enumeration tree. *Enum-Grow* calls the function *Compute-Range* to obtain the range of valid labels for the newly added vertex at a given position.

3.6. Time Complexity Analysis

We now 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 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 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 all possible new vertices 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.

4. Mining Frequent Labeled 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 we can easily extend our definition for the canonical forms to labeled free trees. In this section, we introduce an Apriori-like algorithm that uses the canonical forms to mine all frequent free subtrees from a database of labeled free trees. We choose to use the depth-first canonical form (DFCF), because of its efficient $O(ck \log k)$ normalization algorithm.

4.1. Extending the Canonical Forms

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* (Aldous and Wilson, 2000). 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 9 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 the rooted unordered tree to obtain the DFCF for the centered free tree as we did in previous

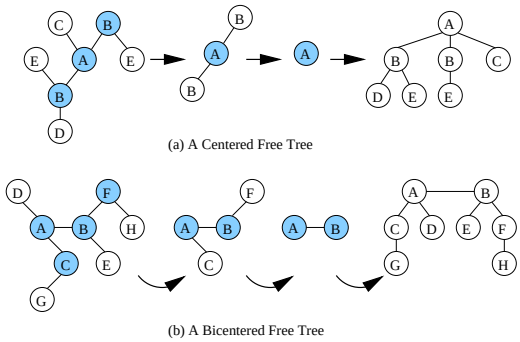


Fig. 9. A Centered Free Tree (above) and A Bicentered Free Tree (below) together with Their Canonical Forms

sections. However, if a free tree is bicentered, we can only identify a pair of vertices (the bicenters). In this case, if we relax the definition of rooted trees to allow a pair of roots (together with an edge connecting them), the from a bicentered free tree we can again obtain a rooted tree (with a pair of roots) and obtain the corresponding DFCF, as shown in Figure 9. Notice that for a bicentered tree, the order of the pair of roots is fixed in its canonical form.

Rückert *et al.* (Rückert and Kramer, 2004) have independently defined a canonical form for labeled free trees that is similar to our DFCF. In their definition, for a bicentered tree, a unique root is selected from the pair of bicenters using the following criterion: first, by removing the edge connecting the bicenters, two rooted subtrees are obtained; then by comparing the canonical representations of the two rooted subtrees, the root of the subtree with lower lexicographical order is chosen as the unique root for the free tree.

4.2. The Apriori-like Algorithm

Figure 10 gives *FreeTreeMiner*, our algorithm for solving the frequent subtree mining problem. This algorithm, like most previous studies on the frequent itemsets mining problem, is based on the bottom up *Apriori* method (Agrawal and Srikant, 1994). However, the number of patterns with 2 or fewer vertices is not very large; so for these patterns, to avoid the step of support checking which is time-consuming, we have used a brute-force method: we scan all transactions in the database to find and count all vertices as well as subtrees with 2 vertices, then remove those that do not meet the minimum support requirement.

The two main steps in the above algorithm are (1) candidate generation and (2) frequency counting. We now describe each in detail.

4.3. Candidate Generation

4.3.1. Basic Ideas

By the downward closure property, for a *(k+1)*-tree to be frequent, all its k-subtrees must be frequent. On the other hand, if we have discovered all the frequent k-trees, we can combine a pair of frequent k-trees to get a candidate for

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

```
1:  F_1, F_2 ← {frequent 1 and 2-trees};
2:  for ( k ← 3; F_{k-1} ≠ ∅; k++ ) do
3:      C_k ← candidate-generate(F_{k-1});
4:      for each transaction t ∈ D do
5:          for each candidate c ∈ C_k do
6:              if (t supports c) then c.count++;
7:      F_k ← {c ∈ C_k | c.count ≥ minsup};
8:  Answer ← Union all F_k's;
```

Fig. 10. The FreeTreeMiner Algorithm

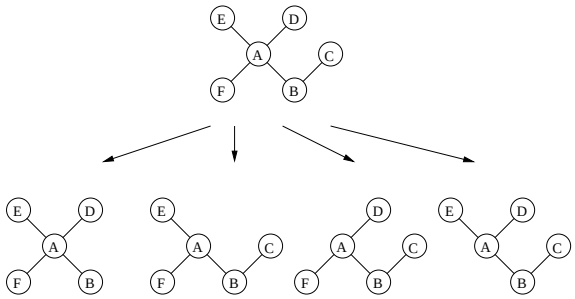


Fig. 11. A 6-tree and It's Cores

frequent *(k+1)*-trees, as long as this pair of k-trees share all structure but one leaf vertex. This method is usually called *a priori* in the data mining literature.

For a *(k+1)*-tree, how many k-subtrees does it have? For simplicity, we first assume the vertices of the *(k+1)*-tree are distinct. As we can see from the example given in Figure 11, the answer is equal to the number of leaves the *(k+1)*-tree has: to obtain a k-subtree, we need to remove one vertex (together with all edges incident with it) from the *(k+1)*-tree; the vertex to be removed can be any of the leaves but not any of the internal nodes, whose removal will make the remaining graph disconnected.

In order to create candidate *(k+1)*-trees, a self-join on the list of all the frequent k-trees is needed. During the self-join, it is time consuming to determine if two frequent k-trees share all structure other than one leaf vertex (i.e., to determine if the two frequent k-trees are joinable). Here we use a technique that is based on a container–the *maps*–in the C++ standard library to expedite the self-join step. For a frequent k-tree, we remove one of its leaves. The remaining graph is a tree with $k-1$ vertices. We call this *(k-1)*-tree a *core* of the k-tree and the removed vertex (together with the removed edge) the corresponding *limb*. The number of cores for a frequent k-tree is equal to the number of its leaves because each leaf can be a limb. A pair of frequent k-trees can be joined to obtain a candidate *(k+1)*-tree if and only if they share a core with $k-1$ vertices. For each frequent k-tree, we can remove one leaf at a time to obtain all its possible cores, then register the k-tree to all its cores where the cores are stored in *maps*. We know that *maps* are stored internally as balanced binary tree in

C++, so searching for a core in *maps* is a very efficient operation. For the *maps* container, we provide the DFCS of a core as its key for searching. Two frequent k-trees registered at the same core can be joined together to create candidate *(k+1)*-trees.

4.3.2. How Many Limbs Are Enough?

A frequent k-tree can have as many as $k-1$ cores. If each k-tree is registered to all its cores, considerable redundancy can result because there are multiple ways to create a candidate *(k+1)*-tree from frequent k-trees. For example in Figure 11, any two of the given 5-subtrees can be joined to get the 6-tree in the figure. We want to reduce the redundancy as much as possible. In other words, we want a candidate to be generated in a unique way.

We use the idea from the traditional market-basket data mining problem. In traditional market-basket problem, for example, although a 4-itemset *abcd* can be obtained in multiple ways by joining 3-itemsets, if we join a pair of 3-itemsets only if they share the prefix (after sorting by items) then the candidate *abcd* is generated in a unique way by joining *abc* and *abd*.

Following the same idea, we take advantage of the labels of the leaves of a tree, as shown in Lemma 4.1. Again we first assume the labels among the leaves of our frequent trees are distinct.

Lemma 4.1. In generating candidate *(k+1)*-trees, we have to combine two frequent k-subtrees only if they share the same core and the corresponding limbs are the *top 2 leaves* in the resulting *(k+1)*-tree, where the *top 2 leaves* are defined by the order of labels.

Proof. For a *(k+1)*-tree, we first sort all its leaves by their labels. We call the two maximal leaves (i.e., the *top 2 leaves*) *leaf A* and *leaf B*. One k-subtree can be obtained from the *(k+1)*-tree by removing *leaf A* (call it *tree1*) and another by removing *leaf B* (call it *tree2*). *Tree1* and *tree2* share the same core. The core can be obtained from *tree1* by removing *leaf B*, and from *tree2* by removing *leaf A*. A *(k+1)*-tree always has these two special k-subtrees. As a result, if we combine two frequent k-subtrees only when they share the same core and the corresponding limbs are the *top 2 leaves* in the resulting *(k+1)*-tree, we do not miss any valid candidate *(k+1)*-tree. □

As a result of the above lemma, registering a frequent k-tree to all its cores is not necessary, because not all leaves can become one of the *top 2 leaves* in the generated candidate *(k+1)*-tree. This is shown in Lemma 4.2.

Lemma 4.2. In generating candidate *(k+1)*-trees, for a frequent k-tree whose vertex labels are distinct, we only have to consider two cores created by removing each of the *top 2* leaves.

Proof. Continuing the proof of Lemma 4.1, *leaf B* is the top leaf of *tree1*, or the second top leaf if removing *leaf A* from the candidate *(k+1)*-tree exposes a new leaf with higher order than *leaf B*; *leaf A* is the top or the second top leaf of *tree2*. Consequently for *tree1*, we only need to remove the top 2 leaves to guarantee *leaf B* is one of the limbs. Similarly for *tree2*, we only need to remove the top 2 leaves to guarantee *leaf A* is one of the limbs. As a result, a frequent k-tree only has to register to the two cores created by removing each of the *top 2 leaves*. □

In the above discussions, we have assumed that the leaf labels are distinct.

If this is not the case, Lemma 4.3 extends the above results. In the lemma, we define *leaves with top 2 labels* as the leaves whose label ranks either the first or the second among all leaf labels. For example, if the leaf labels of a frequent k-tree are $\{C, C, B, B, B, A\}$, then the leaves with top 2 labels are limbs $\{C, C\}$; if the leaf labels are $\{C, B, B, B, A\}$, then the leaves with top 2 labels are limbs $\{C, B, B, B\}$.

Lemma 4.3. For trees with leaf labels that are not necessarily distinct, we need to change "top 2 leaves" in Lemma 4.1 to "leaves with top 2 labels" and to change "two cores created by removing each of the *top 2 leaves*" in Lemma 4.2 to "all cores created by removing each of the leaves with top 2 leaves".

Proof. For a *(k+1)*-tree t, assume that the leaves of t with top 2 leaf labels are $\{l_1, \ldots, l_n\}$, where $n \geq 2$. Removing the leaves in $\{l_1, \ldots, l_n\}$ one at a time will give us a set of k-subtrees $\{t_{l_1}, \ldots, t_{l_n}\}$. Combining any pair, t_{l_i} and t_{l_j} for $1 \leq i, j \leq n$ and $i \neq j$, of these k-subtrees will give us t. To extend Lemma 4.1, we notice that there are at least one pair of such k-subtrees, t_{l_i} and t_{l_j} (for $1 \leq i, j \leq n$ and $i \neq j$), combining whom will give us t, while the corresponding leaves l_i and l_j are among the leaves with top 2 labels in t. To extend Lemma 4.2, we notice that when we combine two k-subtrees, say t_1 and t_2, that share the same core in order to get a *(k+1)*-tree, if the limb l of t_1 is not among the leaves with top 2 labels in t_1, then l cannot be among the leaves with top 2 labels in t either. □

4.3.3. Tree Automorphisms

Automorphisms of a tree are the isomorphisms of the tree to itself. If the core of a tree has automorphisms, then the join procedure becomes more complicated. For example, the two trees in Figure 12 create 9 candidate trees because of the automorphisms of the core shared by the two trees. From Figure 12 we can also see that in creating candidate *(k+1)*-trees, joining a frequent k-tree with itself is necessary.

Therefore, we need an efficient scheme to record all possible automorphisms of a DFCF 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 DFCF:

Definition 4.1 (Equivalence Classes in the Sense of Automorphisms). Vertices of a given tree in its DFCF 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 DFCF.

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 13 gives a running example for obtaining automorphisms. It's obvious that this procedure of obtaining automorphisms has time complexity $O(ck \log k)$.

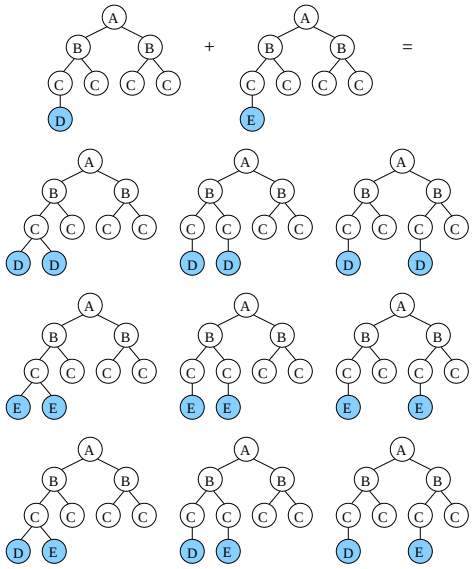


Fig. 12. Automorphisms

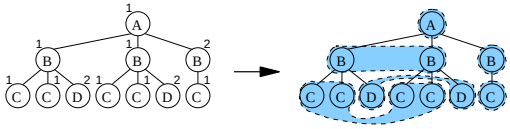


Fig. 13. Obtaining Automorphisms

We can use the equivalence classes defined above to explore all the automorphisms of a core in the joining procedure: a limb attaches to its core through a vertex; if the vertex belongs to an equivalence class with multiple elements, then in the joining procedure, we consider all the combinations that are resulted from attaching the limb to each element in the equivalence class.

Notice that automorphisms affect the resulting patterns of the join operation. In the *RootedTreeMiner* algorithm that is introduced in the previous section, automorphisms are not an explicit issue, because they do not affect the BFCF and the enumeration tree–a tree with automorphisms still has a unique BFCF and its parent in the numeration tree is still uniquely determined by removing the last leg. However, the occurrence list for a tree with automorphisms is more complicated because it must record all the possible permutations (due to automorphisms) of the mapping between the tree indices and those of the transactions. Fortunately, this is handled automatically by the incremental fashion of occurrence list building. For example, the first candidate 3-tree in Figure 7 has an occurrence list $L = \{(T2, 1, 2, 3), (T2, 1, 3, 2)\}$, which implicitly stores the automorphisms for the corresponding candidate 3-tree.

Algorithm **candidate-generate**(F_k)

```
1:  C_{k+1} ← ∅, CL ← ∅;
2:  for each tree f ∈ F_k do
3:      for each leaf l among top 2 leaves of f do
4:          cl ← remove l from f;
5:          if cl ∉ CL then CL ← cl ∪ CL;
6:          register l to cl in CL;
7:  for each core cl ∈ CL do
8:      for each limb pair (l_1, l_2) of cl do
9:          for each automorphism of cl related to l_1,l_2 do
10:             c ← attach l_1 and l_2 to cl;
11:             if downward-check(c,F_k) = success then
12:                 C_{k+1} ← c ∪ C_{k+1};
13: return C_{k+1};
```

Fig. 14. The candidate-generate Algorithm

4.3.4. Downward Closure Checking

In the last step of candidate generation we use the downward closure checking to filter out those candidates that cannot be frequent. The downward closure property says that in order for a candidate *(k+1)*-tree to be frequent, each of its k-subtrees must be frequent. As a result, after all candidate *(k+1)*-trees have been created, we check the downward closure property for each candidate by removing a leaf at a time from the candidate and checking if the remaining k-subtrees are all frequent. If any of its k-subtree fails to be frequent, a candidate *(k+1)*-tree will fail the downward closure checking and therefore can be eliminated.

4.3.5. Putting It Together

To summarize, Figure 14 gives the candidate generation procedure in our *FreeTreeMiner* algorithm. In the figure, F_k represents the list of frequent k-trees, CL is the core list, and C_{k+1} is the list of candidate *(k+1)*-trees. To guarantee a candidate is created only once, in step 5 and step 11 of the algorithm, we use our indexing technique, i.e., the depth-first canonical form (DFCF) for labeled free trees, to index all cores in CL and all candidate *(k+1)*-trees in C_{k+1}.

4.4. Frequency Counting

In the frequency counting step, we verify if a candidate tree is frequent or not by checking its support in the database. Because of the large memory footprint of the Apriori-like algorithms, it is impractical to use the same occurrence list method that we have used in mining frequent rooted unordered subtrees. Here we introduce a disk-resident method. The key work for frequency counting is, for each transaction t in the database and each candidate c, we want to check if t supports c. That is, we want to detect if c is embedded in t. This is a subtree isomorphism problem. We have implemented, with some variations, the $O(k^{1.5}n)$ algorithm described in (Chung, 1987) (where n is the number of vertices in t and

k is the number of vertices in c). The main idea of the algorithm is to first fix a root r for t (we call the resulting rooted tree t^r) then test for each vertex v of c if the rooted tree c^v with v as the root is isomorphic to some subtree of t^r. The test is done on each subtree of t^r in a post-order and is reduced to maximum bipartite matching problems. For the maximum bipartite matching problem we have adopted the algorithms described in (Setubal, 1996).

5. Experiments

We performed extensive experiments to evaluate the performance of the *RootedTreeMiner* algorithm and the *FreeTreeMiner* algorithm, 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 the RedHat Linux 7.3 operating system. All algorithms are implemented in C++ and compiled using the g++ 2.96 compiler.

5.1. 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 (Medina, Lakhina, Matta and Byers, 2001), developed by Medina et al at Boston University, that generates random graphs simulating Internet topologies with some specific network 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 $|D| \cdot |S|$ 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 $|L|$. In particular, $|L|$ is both the number of distinct edge labels as well as the number of distinct vertex labels.

Table 1. Parameters for Synthetic Generators

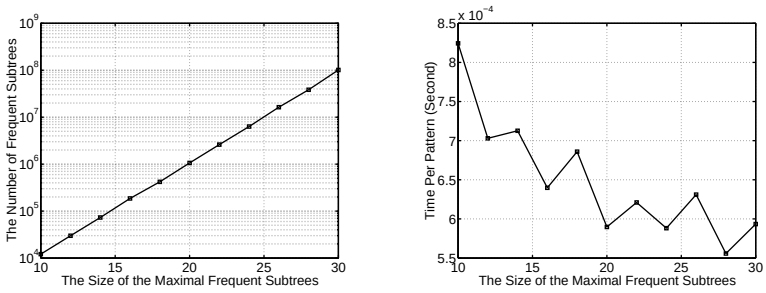
Parameter	Description
$\|D\|$	the number of transactions in the database
$\|T\|$	the size of each transaction in the database
$\|I\|$	the maximal size of frequent subtrees
$\|N\|$	the number of frequent subtrees with size $\|I\|$
$\|S\|$	the minimum support [%] for frequent subtrees
$\|L\|$	the size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

5.2. Results for Labeled Rooted Unordered Trees

5.2.1. Synthetic Datasets

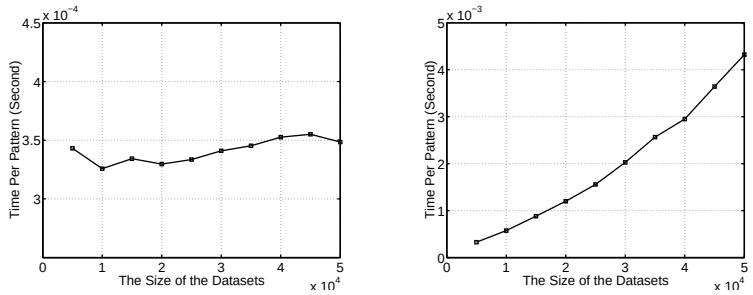
In our first experiment, we want to study the effect of the size of maximal frequent subtrees on our algorithm. With all other parameters fixed ($|D|$=10000, $|T|$=50, $|N|$=100, $|S|$=1%), we increase the maximal frequent tree size $|I|$ from 10 to 30. Figure 15(a) gives the number of frequent subtrees 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 15(b) shows the average time to mine each frequent subtrees. As can be seen, the average time for our algorithm 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.

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 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 16(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



(a) (b)

Fig. 15. Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees

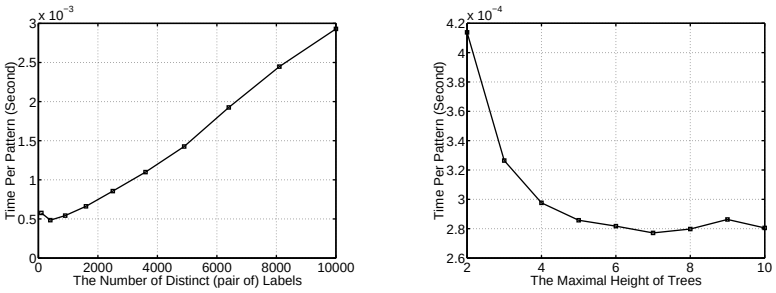


(a) (b)

Fig. 16. Running Time vs. Size of the Datasets

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 16(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 17(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 17(b) shows the running time for different families of trees. It is



(a) (b)

Fig. 17. Running Time vs. Other Parameters

very interesting that when the trees are extremely flat (when the maximal height is 2 or 3), the performance of the algorithm deteriorates a little.

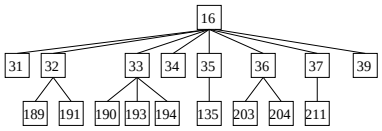
5.2.2. Web Access Trees

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 (Punin and Krishnamoorthy, 1998) 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 *RootedTreeMiner* algorithm mined 16507 frequent subtrees in less than 2 sec. Among all the frequent subtrees, the maximum subtree has 18 vertices. Figure 18 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.

5.2.3. Comparison with TreeMiner

In this sub section, we compare our *RootedTreeMiner* algorithm with the *TreeMiner* algorithm given in (Zaki, 2002). However, there are a few differences between the two algorithms. First, *TreeMiner* is an algorithm that mines frequent rooted ordered trees while *RootedTreeMiner* is an algorithm that mines frequent rooted unordered trees. Because *TreeMiner* mines frequent rooted ordered trees, the order of vertices in the transactions provides information which can be used in the candidate generation step, as the *scope-list join* algorithm did in (Zaki, 2002). In contrast, *RootedTreeMiner*, which mines frequent rooted unordered tree, does not have this advantage. Notice that normalizing a transaction does not help, as illustrated in Figure 19. In Figure 19, although the transaction on the left is in the BFCF, an algorithm that mines rooted ordered trees still cannot discover the candidate tree on the right, which is also in the BFCF.

The second difference between the two algorithms is that *TreeMiner* discov-



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

Fig. 18. The Maximum Frequent Subtree Mined From Web Log Files

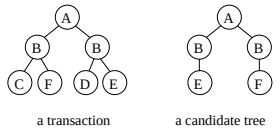


Fig. 19. Normalizing Transactions

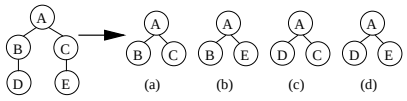


Fig. 20. Embedded and Induced Subtrees

ers all frequent *embedded* subtrees while *RootedTreeMiner* discovers all frequent *induced* subtrees. An induced subtree t_i of a tree t must preserve the parent-child relationship in t while an embedded subtree t_e only has to preserve the ancestor-descendent relationship. Figure 20 illustrates the difference between the two definitions: while all the four 3-trees on the right are embedded subtrees (with A as the root and with the root having two children) of the tree on the left, only *tree (a)* is an induced subtree.

Despite these differences, it is helpful for us to compare the experimental results of the two algorithms, in order to compare their performance and the frequent subtrees they discover. We have created the *T1M* dataset in (Zaki, 2002) using the generator provided by the author of (Zaki, 2002), and run the two algorithms on the dataset. Figure 21 shows the number of frequent subtrees and the running time for each frequent subtree versus the minimum support (with the minimum support less than 0.02%, *TreeMiner* exhausted all available memory). From Figure 21(a) we can see that the number of frequent induced subtrees grows exponentially as the minimum support decreases. In comparison, the number of frequent embedded subtrees is even 1 to 2 order of magnitudes higher. We believe that the large number of frequent subtrees is not necessarily a benefit, because it will be difficult for end users to get useful information from such large number of frequent subtrees. Figure 21(b) shows the average running time for the two algorithms to discover each frequent subtree. From the figure

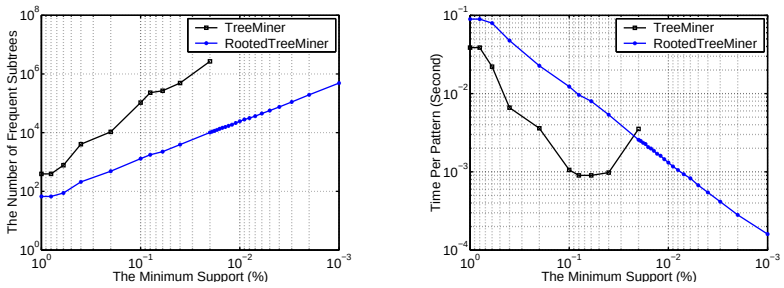


Fig. 21. Number of Frequent Subtrees and Running Time vs. Support for the T1M Dataset

we can see that *TreeMiner* is more efficient than *RootedTreeMiner* when the minimum support is large. However, as the minimum support decreases further, the average running time for *RootedTreeMiner* decreased dramatically. This is because the total running time of *RootedTreeMiner* is very small–from 6 seconds when the minimum support is 1% to 78 seconds when the minimum support is 0.001%–therefore the overheads (such as scanning the whole database to get the frequent vertices and frequent 2-trees) dominates in the support range in Figure 21. As shown in Figure 21, the incremental cost per frequent subtree is smaller for *RootedTreeMiner* than for *TreeMiner* and therefore as the support decreases and more frequent trees are found, *RootedTreeMiner* performs better than *TreeMiner* in terms of time per frequent subtree.

5.3. Results for Labeled Free Trees

In this section, we report our experiments on datasets of labeled free trees.

5.3.1. Synthetic Datasets

In the first experiment, we study the performance of *FreeTreeMiner* on the number of transactions. We use datasets of $|D| = 10000$, 20000, 40000, 60000, 80000, and 100000. The other parameters are fixed: $|S| = 1\%, |N| = 10, |I| = 10, |L| = 10$, and $|T|$ ranges within $\{10, 15, 20, 30\}$. (When $|D| = 100000$ and $|T| = 30$, *FreeTreeMiner* exhausted all available memory.) These results are shown in Figure 22. As we can see from the figure, the total running time of *FreeTreeMiner* scales linearly with the number of transactions.

In the second experiment, we fix all parameters other than $|I|$ ($|D|$=10000, $|T|$=30, $|N|$=10, $|S|$=1%), while changing the maximal frequent tree size $|I|$ from 4 to 20. Figure 23 gives the results. Figure 23(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 23(b) gives the average time for *FreeTreeMiner* to mine each frequent subtree. The average time for *FreeTreeMiner* to mine each frequent subtree decreases a little as the maximal frequent tree size $|I|$ increases. Our explanation to this decrease is that as $|I|$ increases, the number of automorphisms decreases and the number of false positives (those join attempts that do not result in *top 2* new limbs) decreases. Both contribute to the decrease of average time to mine a frequent subtree.

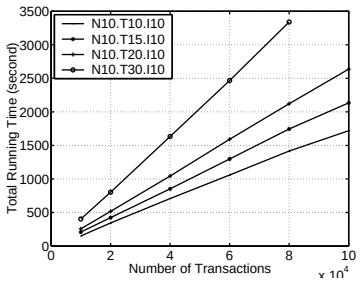
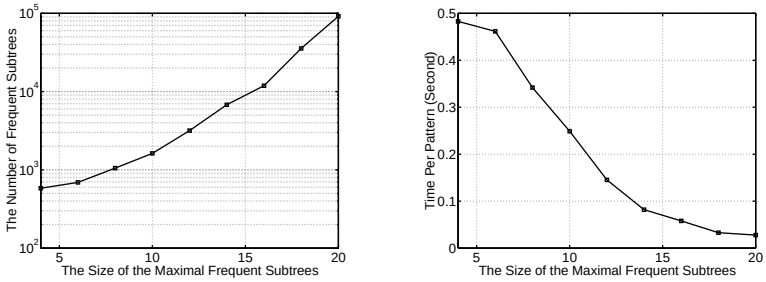


Fig. 22. Scalability on the Number of Transactions



(a) (b)

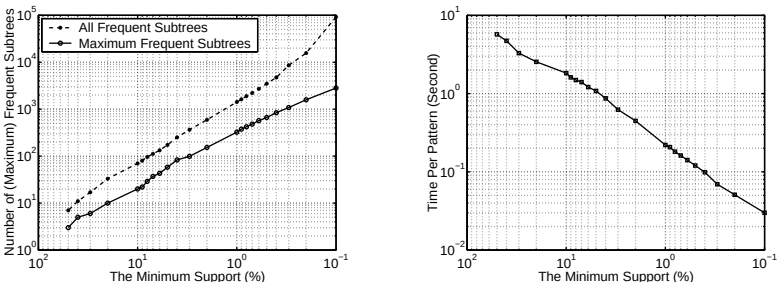
Fig. 23. Number of Frequent Subtrees and Running Time vs. Size of Maximal Frequent Trees

5.3.2. The Chemical Compound Dataset

We apply our *FreeTreeMiner* on a real chemical compound dataset. The dataset contains 17,663 tree-structured chemical compounds sampled from a graph dataset of the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) ((NCI), 2003). 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 24(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 24(b) gives the running time for the *FreeTreeMiner* algorithm to mine all frequent subtrees under different supports.

6. Conclusion and Future Work

In this paper, we presented two novel canonical forms, the breadth-first canonical form (BFCF) and the depth-first canonical form (DFCF), for both labeled rooted unordered trees and labeled free trees. In addition, we built an enumeration tree to enumerate all (frequent) rooted unordered trees in their BFCFs and developed



(a) (b)

Fig. 24. Number of (Maximum) Frequent Subtrees and Running Time vs. Support for the Chemical Compound Dataset

the *RootedTreeMiner* algorithm that uses this enumeration tree to discover all frequent subtrees in a database of labeled rooted unordered trees. Moreover, we extended one of our canonical forms, the DFCF, to represent labeled free trees and presented an efficient algorithm, *FreeTreeMiner*, to discover all frequent subtrees in a database of labeled free trees. We used both synthetic and real application datasets to study the performance of our algorithms.

We plan to extend our work in several directions in the future. 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. In this direction, we have had some preliminary results (Chi, Yang, Xia and Muntz, 2004). Second, enumeration tree based methods usually have much better running time performance than Apriori-like methods. We are working on applying the enumeration tree techniques to mining frequent free trees and have some preliminary results as reported in (Chi, Yang and Muntz, 2004). Third, we are working on algorithms that mine frequent subgraphs with mining frequent spanning trees as the first step. Forth, it is our belief that vertex labels in the trees are not necessarily atomic. For example, for a web access tree, we have chosen the URL as the vertex labels. However, each web page file can have multiple attributes, such as the size of the file, the time of creation, the type (html, jpg, etc.) of the file, etc.; each access to an URL can have its own attributes also, such as the domain (.edu, .com, etc.) of the visitor, the duration of stay, etc. As a result, it is possible for the vertex labels of a frequent subtree to be defined over any subset of all these attributes. One example of such frequent tree is a tree that "has a root of html type, has two children both built before January 1, 2000, and is visited by a visitor from .edu domain". To mine such frequent subtrees with general vertex labels efficiently is a challenge that we plan to work on in the future.

Acknowledgements. We thank the anonymous reviewers for their very useful comments and suggestions. Thanks to Professor Mohammed J. Zaki at the Rensselaer Polytechnic Institute for providing us the source codes for the *TreeMiner* algorithm and helping us with the WWWPal system.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116, 0085773, and 9817773. 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

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

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

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

Aldous, J. M. and Wilson, R. J. (2000), *Graphs and Applications, An Introductory Approach*, Springer.

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

Asai, T., Arimura, H., Uno, T. and Nakano, S. (2003), Discovering frequent substructures in large unordered trees, *in* 'The 6th International Conference on Discovery Science'.

Bayardo, Jr, R. J. (1998), Efficiently mining long patterns from databases, *in* 'Proceedings of the ACM SIGMOD'.

Buss, S. R. (1997), Alogtime algorithms for tree isomorphism, comparison, and canonization, *in* 'Computational Logic and Proof Theory, 5th Kurt Gödel Colloquium (KGC'97)', Vol. 1289 of *Lecture Notes in Computer Science*, Springer, pp. 18–33.

Chen, Z., Jagadish, H. V., Korn, F., Koudas, N., Muthukrishnan, S., Ng, R. T. and Srivastava, D. (2001), Counting twig matches in a tree, *in* 'ICDE'01', pp. 595–604.

Chi, Y., Yang, Y. and Muntz, R. R. (2003), Indexing and mining free trees, *in* 'Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)'.

Chi, Y., Yang, Y. and Muntz, R. R. (2004), HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms, *in* 'The 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)'.

Chi, Y., Yang, Y., Xia, Y. and Muntz, R. R. (2004), CMTreeMiner: Mining both closed and maximal frequent subtrees, *in* 'The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)'.

Chung, M. J. (1987), '$O(n^{2.5})$ time algorithm for subgraph homeomorphism problem on trees', *Journal of Algorithms* **8**, 106–112.

Cui, J., Kim, J., Maggiorini, D., Boussetta, K. and Gerla, M. (2002), Aggregated multicast–a comparative study, *in* 'Proceedings of IFIP Networking 2002'.

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

Hein, J., Jiang, T., Wang, L. and Zhang, K. (1996), 'On the complexity of comparing evolutionary trees', *Discrete Applied Mathematics* **71**, 153–169.

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

Inokuchi, A., Washio, T. and Motoda, H. (2000), 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)', pp. 13–23.

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

Liu, T. and Geiger, D. (1999), Approximate tree matching and shape similarity, *in* 'International Conference on Computer Vision'.

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

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

Nijssen, S. and Kok, J. N. (2003), Efficient discovery of frequent unordered trees, *in* 'First International Workshop on Mining Graphs, Trees and Sequences'.

Punin, J. and Krishnamoorthy, M. (1998), WWWPal system–a system for analysis and synthesis of web pages, *in* 'WebNet 98 Conference'.

Rückert, U. and Kramer, S. (2004), Frequent free tree discovery in graph data, *in* 'Special Track on Data Mining, ACM Symposium on Applied Computing (SAC'04)'.

Setubal, J. C. (1996), Sequential and parallel experimental results with bipartite matching algorithms, Technical Report IC-96-09, Institute of Computing, State University of Campinas (Brazil).

Shasha, D., Wang, J. T. L. and Giugno, R. (2002), Algorithmics and applications of tree and graph searching, *in* 'Symposium on Principles of Database Systems', pp. 39–52.
Termier, A., Rousset, M.-C. and Sebag, M. (2002), TreeFinder: a first step towards xml data mining, *in* 'Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)', pp. 450–457.
Valiente, G. (2002), *Algorithms on Trees and Graphs*, Springer.
Yan, X. and Han, J. (2002), gSpan: Graph-based substructure pattern mining, *in* 'Proc. 2002 Int. Conf. on Data Mining (ICDM'02)'.
Yan, X. and Han, J. (2003), CloseGraph: Mining closed frequent graph patterns, *in* 'Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)'.
Zaki, M. J. (2002), Efficiently mining frequent trees in a forest, *in* '8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining'.

Correspondence and offprint requests to: Yun Chi, Department of Computer Science, University of California, Los Angels, CA 90095, USA. Email: ychi@cs.ucla.edu