Indexing and Mining Free Trees

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

Abstract

Tree structures are used extensively in domains such as computational biology, pattern recognition, computer networks, and so on. In this paper, we present an indexing technique for free trees and apply this indexing technique to the problem of mining frequent subtrees. We first define a novel representation, the canonical form, for rooted trees and extend the definition to free trees. We also introduce another concept, the canonical string, as a simpler representation for free trees in their canonical forms. We then apply our tree indexing technique to the frequent subtree mining problem and present *FreeTreeMiner*, a computationally efficient algorithm that discovers all frequently occurring subtrees in a database of free trees. Our mining algorithm is a variation of the traditional *a priori* method for mining frequent itemsets. We study the performance and the scalability of our algorithms through extensive experiments based on both synthetic data and datasets from two real applications: a dataset of chemical compounds and a dataset of Internet multicast trees. The experiments show that our algorithm scales linearly in the cardinality of the database.

1 Introduction

Graphs are used extensively in various areas such as computational biology, chemistry, pattern recognition, computer networks, etc. Among all graphs, a particularly useful family is the family of *free trees*–the connected, acyclic and undirected graphs. Here are some examples: 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 [10]; in pattern recognition, a free tree called *shape axis tree* is used to represent shapes [15]; in computer networking, multicast trees are used for packet routing [8]. In addition to being unrooted, trees in these 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 databases of labeled free trees.

In the above applications, two problems are important from the database point of view. The first one is how to index trees. For example, in the pattern recognition example [15], the first step of recognizing a shape might be to look in a database of available objects (in the format of shape axis trees) to find objects similar to the given shape. The second problem in these applications is how to efficiently discover *interesting* patterns. One type of interesting patterns consists of those patterns that are embedded in a lot of transactions in a database. In the multicast example [8], the networking engineers are interested in aggregating parts of multicast trees that are

shared by different groups, so they want to know which parts are more beneficial to aggregate, i.e., which parts occur most frequently. In this paper, we present our approaches to solving these two important problems. Some of the main contributions of our work are: (1) We introduce a unique representation, the canonical form, for a free tree and give an efficient algorithm to convert a free tree to its canonical form. An equivalent representation, the canonical string, is also introduced to simplify certain operations such as comparing or searching free trees. With canonical forms, free trees can be indexed using traditional indexing techniques such as B-tree and hashing. (2) We apply the canonical form of free trees to the frequent subtree mining problem. In mining procedures, canonical forms are used to index frequent trees and candidate trees; they are also used to speed up the join step. (3) We have implemented all of our algorithms and have extensive experiments analysis. We use both synthetic data and real application data to evaluate the performance of our algorithm.

The rest of this paper is organized as follows. Section 2 introduces the canonical form and the corresponding canonical string for free trees. Section 3 presents our algorithm for mining frequent subtrees. Section 4 includes experiments and performance analysis. Section 5 discusses other related work. Finally, section 6 concludes our work and gives future directions.

2 Canonical Form for Labeled Free Trees

In this section, we give a unique representation for labeled free trees, i.e. a canonical form that represents labeled free trees in the same equivalence class where the relation defining the equivalence classes is an isomorphism. Two labeled trees T_1 and T_2 are *isomorphic* to each other if there is a one-to-one mapping from the vertices of T_1 to the vertices of T_2 that preserves vertex labels, edges labels, and adjacency.

2.1 Labeled, Rooted, Ordered Trees

A *rooted tree* is a tree in which one vertex is singled out as the root. We say that a rooted tree is *ordered* if the set of children of each vertex in the tree is ordered. If a labeled rooted tree is ordered, then there are ways to represent the tree in a unique form. [21] is one example in which a depth-first traversal is used to obtain unique string representations for rooted ordered trees. Notice that when a rooted tree is ordered, the isomorphism does not really apply–two rooted ordered trees are either the same or not.

2.2 Labeled, Rooted, Unordered Trees

We assume that for the trees in our databases, both vertices and edges are labeled and there exist total orders among vertex labels and edge labels. We define the total order among trees based on the ordering for labels.

Let's first assume that the trees are rooted. (Later, we will extend our definition to free trees.) If a tree is rooted, without loss of generality we can assume that all edge labels are identical, because each edge connects a vertex with its parent and 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.)

There are two concepts we want to define: the *canonical form* for a rooted tree and the *order* among rooted trees. These two concepts are defined mutually recursively (notice that in the definition below we assume that all edge labels are identical):

Definition 1 (Canonical Form and Order for Labeled, Rooted, Unordered Trees). *For labeled rooted trees with height 0 (i.e., trees consisting of a single vertex), the canonical forms are the vertices themselves and the order among such trees is defined by the order of the vertex labels.*

For a labeled rooted tree with height h where $h > 0$, the canonical form is obtained by first normalizing all subtrees of the root then rearranging the subtrees in increasing order (from the left to the right in illustrating examples).

For a pair of labeled rooted trees (in their canonical forms) with heights less than or equal to h where $h > 0$, their order is defined by first comparing the labels of their roots then comparing their corresponding subtrees from the left to the right until their relative order is resolved.

Essentially, after normalizing rooted trees into their canonical forms, we can compare two trees; the normalization of a tree with height $h > 0$ depends on the order among the subtrees (whose heights are less than h) of the root. It's very easy to see that the above definition introduces a total order among all rooted trees. Figure 1 gives a rooted tree and its canonical form. Notice two things in the example: first, an edge connects a child vertex to its parent and the edge label is considered as a part of the vertex label of the child–this is why the branch "2,D" is *less* than branch "3,C" at the leaf level; second, in comparing two (sub)trees, if root nodes have different number of subtrees, then we need to conceptually pad the smaller set of subtrees with subtrees having the largest possible label–this is why we switch the two subtrees of the root to get the canonical form in our example.

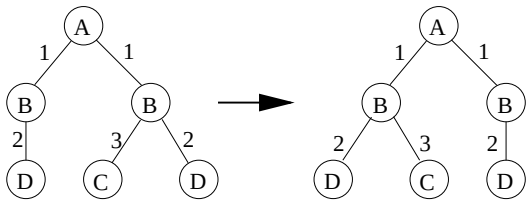

Figure 1: A Labeled, Rooted, Unordered Tree (left) and Its Canonical Form (right)

2.3 Labeled Free Trees

Free trees do not have roots, but we can uniquely create roots for them for the purpose of constructing a unique canonical form for each free tree. Starting from a free tree in each step we remove all leaf vertices (together with their incident edges), and we repeat the step until a single vertex or two adjacent vertices are left. For the first case, the tree is called a *central tree* and the remaining vertex is called the *centre*; for the second case, the tree is called a *bicentral tree* and the pair of remaining vertices are called the *bicentre* [3]. The procedure takes $O(k)$ time where k is the number of vertices in the free tree. Figure 2 shows a central tree and a bicentral tree as well as the procedure to obtain the corresponding centre and bicentre.

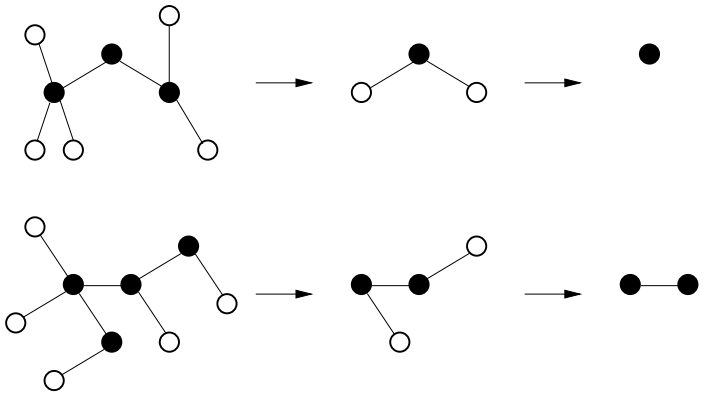

Figure 2: A Central Tree (above) and A Bicentral Tree (below)

If we relax the definition of rooted trees to allow a pair of roots (together with an edge connecting them) then from an arbitrary free tree we can obtain a rooted tree. After obtaining such a rooted tree with a root or a pair of roots, we can extend the definition of canonical form to an arbitrary labeled free tree. Notice that for a bicentral tree, the order of the pair of roots is fixed in its canonical form.

2.4 Some Practical Issues in Canonical Form

In this section, we give a method to transform a labeled rooted tree (which can be obtained from a free tree using the above procedure) into its canonical form. Then we describe how to convert canonical forms into an equivalent but simpler representation–the canonical string. We also describe how to convert from canonical strings back to trees in their canonical forms.

2.4.1 Normalizing Rooted Trees

We now show a bottom-up procedure to normalize labeled rooted trees. The procedure is based on a tree isomorphism algorithm given in [2]. For the reason explained earlier, without loss of generality, we assume that all edge labels are identical. Figure 3 gives a running example on how to obtain the canonical form for a labeled rooted tree. In the figure, we start from the original tree, normalize level by level bottom-up using the orders among subtrees at each level, until finally we obtain the canonical form. Notice that we have used (acyclic, directed) multigraph to represent trees in intermediate steps in order to combine subtrees that are *equal*. It is straightforward to extend this procedure to rooted trees obtained from bicentral free trees.

The key step in each level of the procedure is sorting. With appropriate data structures, the running time for the normalization is $O(c \cdot k \log k)$, where c is the maximal fan-out of the tree and k is the number of vertices in the tree: assuming there are k_h vertices in each level of the tree for $h = 0, 1, 2, \ldots$, to sort vertices at level h, it takes $O(k_h \log k_h)$ comparisons; the total number of comparison for normalizing the whole tree is $\sum_h O(k_h \log k_h)$ which is $O(k \log k)$ (notice that $\sum_h (k_h \log k_h) \leq \sum_h (k_h \log k) = k \log k$); the time for each comparison is bounded by the maximal fan-out c of the tree because we can consider c as the length of the "keys" to be compared.

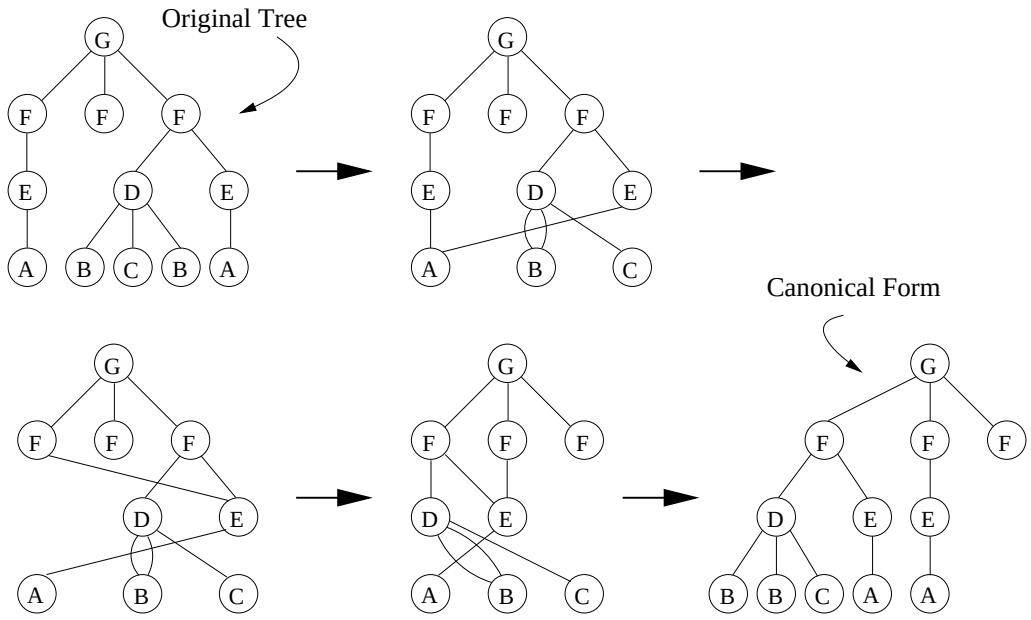

Figure 3: To Obtain the Canonical Form of A Rooted Tree

2.4.2 Converting to Canonical Strings

A canonical string representation for labeled trees is equivalent to, but simpler than, canonical forms. There are two ways to define a canonical string–one based on depth-first tree traversal and the other based on breadth-first tree traversal. Assume two special symbols, "#" and "$", are not in the alphabet of edge labels and vertex labels. To get the first canonical string, we traverse a tree in canonical form in depth-first fashion, using "$" to represent a backtrack and "#" to represent the end of the string. The depth-first canonical string for the example in Figure 3 is $G1F1D1B\$1B\$1C\$\$1E1A\$\$\$1F1E1A\$\$\$1F\#$, assuming all edges have label "1". If we assume "#" is greater than "$" and both are greater than any other symbols in the alphabet of vertex labels and edge labels, then the alphabetical order among depth-first canonical strings turns out to be the same as the order among trees in canonical forms. In other words, we could have defined the canonical form for a rooted unordered tree as the ordered tree derived from the unordered tree that gives the minimum depth-first string encoding.

We obtain the second type of canonical string by scanning a tree in canonical form top-down level by level in a breadth-first fashion: we use "$" to partition the families of siblings and use "#" to indicate the end of the canonical string. The canonical string for the example in Figure 3 is $G\$1F1F1F\$1D1E\$1E\$\$1B1B1C\$1A\$1A\#$, assuming all edges have label "1". The order among breadth-first canonical strings is a total order, although it is not the same total order as the order among trees in canonical form. Because we use canonical form for indexing and any total order will work for this purpose, we can use either the depth-first canonical string or the breadth-first canonical string.

It is easy to see that the procedures for obtaining both types of canonical strings take $O(k)$ time where k is the number of vertices in the original tree. In addition, it is easy to prove that

the length of the two types of canonical strings are both bounded by $3k$. In contrast, adjacency list representation will take *5k-3* space–k for vertex labels, k for head pointers for adjacency lists, for each of the $k-1$ nodes in adjacency lists: spaces for edge label, vertex connects to, and next pointer. In the actual implementation we have chosen the breadth-first canonical string and put some additional information, such as the number of roots and the number of vertices of a tree, in the canonical strings to expedite other operations on them.

2.4.3 Obtaining Trees from Breadth-first Canonical Strings

Converting from breadth-first canonical strings to trees in their canonical forms is also very straightforward: the order that vertices appear in the canonical string is the same order as they appear in the tree, from the left to the right level by level top-down; the vertices (together with the corresponding edges) between the i-th and the *(i+1)*-th symbol \$'s are the children of the i-th vertex of the tree. The time complexity of this procedure is also $O(k)$ where k is the number of vertices in the original tree.

2.5 Indexing Labeled Free Trees

With canonical forms (or canonical strings), we introduce a unique representation for labeled free trees. With such a unique representation, a traditional indexing method such as hashing can be used on databases of free trees. In addition, we also introduce a total order among labeled free trees. Hence we can apply traditional database indexing methods that depend on such a total order, such as B-trees, on databases of free trees. Notice that our canonical form applies to rooted, unordered trees as well. Therefore, it can be used in many applications related to rooted trees such as indexing XML documents.

3 Mining Frequent Subtrees

As we have mentioned before, one important problem in databases of free trees is to find patterns that are embedded in a lot of transactions. In this section, we apply our tree indexing technique to the frequent subtree mining problem. First, let's define the problem.

Frequent Subtree Mining Problem Let D denote a database where each transaction $t \in D$ is a labeled free tree. For a given pattern s (which is a free tree) we say s occurs in a transaction t (or t *supports* s) if there exists a subtree of t that is isomorphic to s. The *support* of a pattern s is the fraction of transactions in database D that supports s. A pattern s 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 patterns in a given database.

Figure 4 gives *FreeTreeMiner*, our algorithm for solving the frequent subtree mining problem. In the algorithm (and in our future discussion as well), we call a tree with k vertices a k-tree. Our algorithm, like most previous studies on the frequent itemsets mining problem, is based on the bottom up *Apriori* method [1]. However, the number of patterns with 4 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 subtrees with 2, 3, and 4 vertices then remove those that do not meet the minimum support requirement.

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

1: F_2, F_3, $F_4 \leftarrow$ {frequent 2, 3, and 4-trees};
2: **for** ($k \leftarrow 5$; $F_{k-1} \neq \emptyset$; k++) **do**
3: $C_k \leftarrow$ candidate-generate(F_{k-1});
4: **for each** transaction $t \in D$ **do**
5: **for each** candidate $c \in C_k$ **do**
6: **if** (t supports c) **then** $c.count$++;
7: $F_k \leftarrow \{c \in C_k | c.count \geq minsup\}$;
8: *Answer* $\leftarrow$ Union all F_k's;

Figure 4: The FreeTreeMiner Algorithm

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

3.1 Candidate Generation

3.1.1 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 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-subtree 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 5, 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 the indexing technique that we introduced previously 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 indexed using our free tree indexing technique. Two frequent k-trees registered at the same core can be joined together to create candidate *(k+1)*-trees.

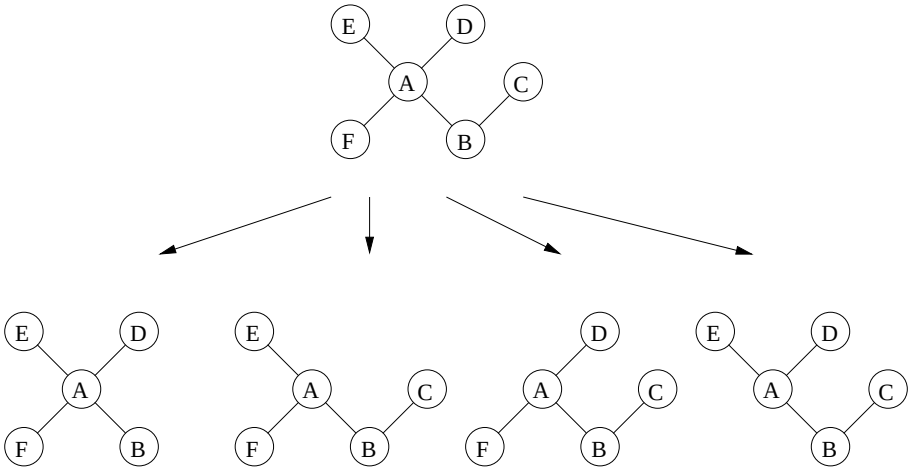

Figure 5: A 6-tree and It's Cores

3.1.2 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 5, 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. Again we first assume the labels among the leaves of our frequent trees are distinct. For a *(k+1)*-tree, we first sort all its leaves by their labels. Let's call the two maximal leaves *leaf A* and *leaf B*. One k-subtree can be obtained from the *(k+1)*-tree by removing *leaf A* (let's call it *tree1*) and another by removing *leaf B* (let's call it *tree2*). *Tree1* and *tree2* share the same core. The core can be obtained from *tree1* by removing *leaf B*, *tree2* by removing *leaf A*. A *(k+1)*-tree always has such two special k-subtrees. As a result, in order to generate a candidate *(k+1)*-tree in a unique way, we combine two frequent k-trees only if they share the same core and the corresponding limbs are the top 2 leaves in the resulting *(k+1)*-tree. The following lemma is very obvious:

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

As a result of Lemma 1, registering a frequent k-tree to all its cores is not necessary, because not all leaves can become top 2 in the generated candidate *(k+1)*-tree. Continuing the previous example, *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*. As a result, a frequent k-tree only has to register to two cores, as given in the following lemma:

Lemma 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.*

In the discussions above, we have assumed that the leaf labels are distinct. If this is not the case, the following lemma extends the above results:

Lemma 3. *For trees with leaf labels that are not necessarily distinct, Lemma 2 still holds, provided we define "top 2" as all leaves with "top 2" label(s).*

For example, if the leaf labels of a frequent k-tree are $\{C, C, B, B, B, A\}$, then we have to register the tree to cores with limbs $\{C, C\}$; if the leaf labels are $\{C, B, B, B, A\}$, then we have to register the tree to cores with limbs $\{C, B, B, B\}$.

3.1.3 Tree Automorphisms

By *automorphisms* of a tree we mean non-identity 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 6 create 9 candidate trees because of the automorphisms of the core shared by the two trees. From Figure 6 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 tree and consider all of them when generating candidates. In order to record the information on tree automorphisms, we introduce a partition among vertices of a tree in its canonical form:

Definition 2 (Equivalence Classes Defined by Automorphisms). *Vertices of a given tree in its canonical form belong to the same equivalence class 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 canonical form.

We can create the partition for the vertices of a tree into equivalence classes at the last step of normalization. In the last step of normalization, a multi-graph is unfolded into a tree from top to bottom and a partition can be created at the same time, following two rules: (1) A single edge will introduce an equivalence class with a single vertex; a multi-edge will introduce an equivalence class whose number of elements is equal to the number of edges. (2) The corresponding children of vertices in the same equivalence class belong to the same equivalence class. Figure 7 gives an example that shows how to obtain the partition of vertices into equivalence classes in the unfolding step by following the above two rules.

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.

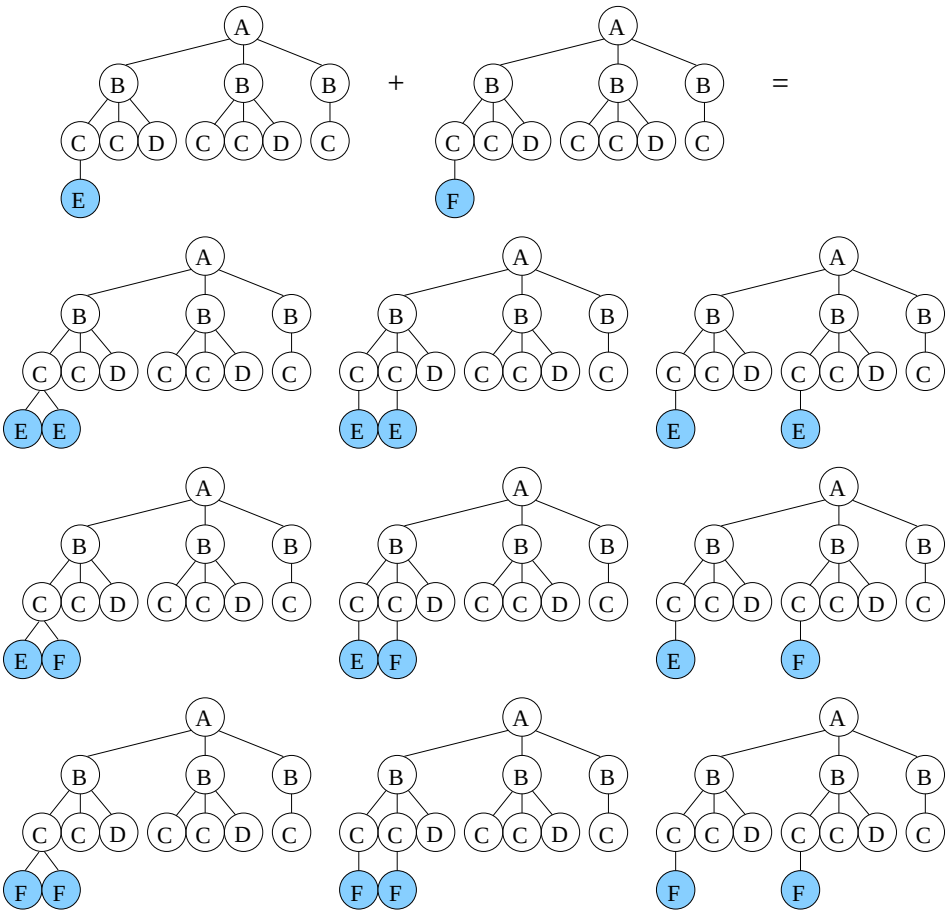

Figure 6: Different Joins for Trees Whose Core Has Automorphisms

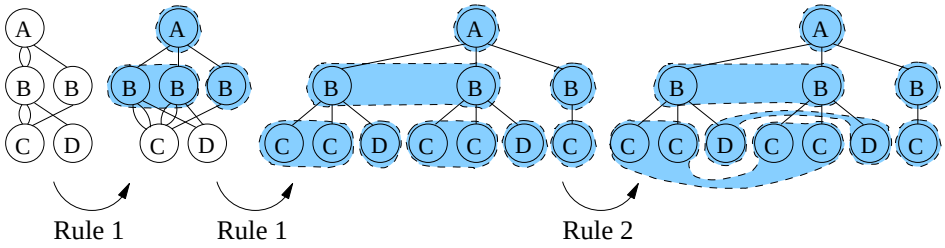

Figure 7: A Partition of the Vertices of a Tree

3.1.4 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.

3.1.5 Putting Together

To summarize, Figure 8 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 12 of the algorithm, we use our indexing technique to index all candidate *(k+1)*-trees in C_{k+1}. Due to space limitations, the implementation details for the downward closure checking are skipped.

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};
```

Figure 8: The candidate-generate Algorithm

3.2 Frequency Counting

In the frequency counting step, we verify if a candidate tree is frequent or not by checking its support in the database. The key work 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 [7] (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 postorder and is reduced to maximum bipartite matching problems. For the maximum bipartite matching problem we have adopted the algorithms described in [18]. Besides, in order to speed up the frequency counting step, we attached a *Tid-List* to each candidate to record the transactions that potentially support the candidate.

4 Experimental Results

We performed three sets of experiments to evaluate the performance of the *FreeTreeMiner* algorithm. In the first set of experiments we used various synthetic datasets, and in the last two sets of experiments we used real application data of chemical compounds and multicast trees. All experiments were done on a 2GHz Intel Pentium IV PC with 2GB main memory, running Linux 7.3 operating system. All algorithms are implemented in C++ using g++ 2.96 compiler.

4.1 Synthetic Dataset

In order to study the performance of *FreeTreeMiner* on various datasets with different characteristics, we developed a synthetic dataset generator which is controlled by a set of parameters shown in Table 1. In this section, we first describe the synthetic tree generator followed by a detailed experimental evaluation of *FreeTreeMiner* on the synthetic datasets.

Table 1: Synthetic dataset parameters

Notation	Parameter
$\|D\|$	The total number of transactions
$\|T\|$	The size of each transaction (in terms of the number of vertices)
$\|I\|$	The maximum size of frequent subtrees (in terms of the number of vertices)
$\|N\|$	The number of frequent subtrees with size $\|I\|$
$\|S\|$	Minimum Support [%] for frequent subtrees
$\|L\|$	The maximum number of distinct edge/vertex labels in the dataset
$\|F\|$	Maximum vertex degrees in the dataset
$\|H\|$	Maximum diameter of trees in the dataset

4.1.1 Synthetic Tree Generator

In order to study the performance of *FreeTreeMiner* on datasets with different characteristics, we want to generate datasets which have some fixed properties and a certain number of varying properties, and also reflect real application data. Therefore, we want to take subtrees of a base graph as seeds to generate tree transactions. To create the base graph, we used the universal Internet topology generator BRITE [16], developed by Medina et al at Boston University. BRITE can generate

random graphs that simulate Internet topologies with some specific network characteristics, such as the link bandwidth. We use the bandwidths of the links as edge labels of our base graph and assign vertex labels uniformly. The base graph created by BRITE has the following characteristics: number of vertex labels is 10; number of edge labels is 10; number of vertices is 1000; average degree for each vertex in the base graph is 20.

From the base graph, we first generate a set of $|N|$ subtrees whose size is determined by $|I|$. We call this set of $|N|$ subtrees *seed trees*. Each seed tree is the starting point for $|D| \cdot |S| \cdot 100$ transactions; each of these $|D| \cdot |S| \cdot 100$ transactions is obtained by first randomly permuting the seed tree then adding more random vertices to increase the size of the transaction to $|T|$. After this step, more random transactions with size $|T|$ are added to the database to increase the cardinality of the database to $|D|$. The number of distinct edge and vertex labels is controlled by the parameter $|L|$. In particular, $|L|$ is both the number of distinct edge labels as well as the number of distinct vertex labels.

4.1.2 Experiments on Synthetic Datasets

We first study the performance of *FreeTreeMiner* as a function of different structural characteristics of the trees. Using the synthetic tree generator we obtained a number of different tree datasets by using different combinations of $|T|, |I|, |N|, |L|, |F|, |H|$, while keeping $|D|$ and $|S|$ fixed. In the following experiments, we use the combinations of the parameters shown in Table 2. When the average transaction size $|T|$ is smaller than the maximum size of the frequent subtrees $|I|$, no datasets are generated.

Table 2: Parameter settings

Parameter	Values
$\|D\|$	10000
$\|T\|$	10, 15, 20, 30
$\|I\|$	6, 8, 10, 12, 14
$\|N\|$	5, 10, 20, 40
$\|S\|$	1%
$\|L\|$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
$\|F\|$	2, 3, 4, 5
$\|H\|$	1, 2, 3, 4, 5, 6, 7, 8

Table 3 shows the total running time and the total number of discovered frequent subtrees for various datasets with $|L| = 10$ and without restriction on maximum vertex degrees and maximum tree diameter, using a support of 1%.

From Table 3, we can observe a number of interesting points regarding the performance of *FreeTreeMiner* on different dataset structural characteristics. First, as the average size of transactions $|T|$ increases the total running time increases as well while $|T|$ does not affect the number of frequent trees very much. This increase in running time is because of that subtree isomorphism checking time is proportional to the transaction size. The relative increase is slightly affected by $|I|$ and $|N|$. Second, as $|I|$ increases the total running time also increases. This increase is in nonlinear fashion

Table 3: Running time and number of frequent subtrees for synthetic datasets. The other parameters are: $|D| = 10000, |S| = 1\%, |L| = 10$, no restriction on vertex degrees and tree diameter.

$\|N\|$	$\|I\|$	$\|T\|$	RunTime [sec]	Number of Frequent Subtrees
5	6	10	19	537
		15	39	620
		20	62	628
		30	126	623
5	8	10	24	694
		15	50	789
		20	83	774
		30	165	809
5	10	10	32	987
		15	59	1145
		20	102	1068
		30	206	1053
5	12	15	78	1906
		20	130	2072
		30	260	2579
5	14	15	105	3844
		20	162	3595
		30	324	3954

$\|N\|$	$\|I\|$	$\|T\|$	RunTime [sec]	Number of Frequent Subtrees
10	6	10	18	609
		15	39	707
		20	63	704
		30	126	687
10	8	10	24	894
		15	50	943
		20	83	989
		30	165	1006
10	10	10	34	1807
		15	65	1792
		20	105	1630
		30	213	1700
10	12	15	93	4012
		20	140	3573
		30	272	3342
10	14	15	138	6977
		20	214	7906
		30	400	8684

$\|N\|$	$\|I\|$	$\|T\|$	RunTime [sec]	Number of Frequent Subtrees
20	6	10	19	749
		15	39	861
		20	63	827
		30	127	848
20	8	10	26	1280
		15	51	1402
		20	83	1397
		30	169	1461
20	10	10	38	2809
		15	70	2765
		20	116	3078
		30	221	2865
20	12	15	123	6826
		20	175	6548
		30	326	7007
20	14	15	215	13988
		20	325	16428
		30	478	13681

$\|N\|$	$\|I\|$	$\|T\|$	RunTime [sec]	Number of Frequent Subtrees
40	6	10	19	1045
		15	38	1085
		20	64	1144
		30	125	1152
40	8	10	27	2244
		15	53	2336
		20	88	2311
		30	173	2251
40	10	10	52	5115
		15	86	4976
		20	131	5080
		30	248	5450
40	12	15	185	13624
		20	263	12328
		30	538	11245
40	14	15	376	29075
		20	473	28929
		30	771	31479

because firstly the time to check subtree isomorphism is asymptotically proportional to $|I|^{1.5}$ (as we have mentioned in Section 3.2) and secondly the number of frequent subtrees increases as $|I|$ increases. The relative increase is higher for larger $|N|$. Third, when $|N|$ increases, the number of frequent subtrees increases, and thus the total running time increases as well.

We also did experiments on various datasets with different number of vertex labels, vertex degrees and tree diameters, while fixing other parameters. From the experimental results (which are not included due to space limitations), we observe that the shape of trees (the maximum vertex degrees $|F|$ and the maximum tree diameter $|H|$) do not have much effect on the total running time. However, as the number of edge/vertex labels $|L|$ decreases, the total running time increases, because there are more automorphisms and subtree isomorphisms, and thus there are longer generating time and checking time.

Finally, we did experiments to study the performance of *FreeTreeMiner* on the number of transactions. We used datasets of $|D| = 10000$, 20000, 40000 and 80000. The other parameters are: $|S| = 1\%, |N| = 10, |I| = 10, |L| = 10$, no restriction on vertex degrees and tree diameter, and $|T|$ ranges within $\{10, 15, 20, 30\}$. These results are shown in Figure 9. As we can see from the figure, the total running time scales linearly with the number of transactions.

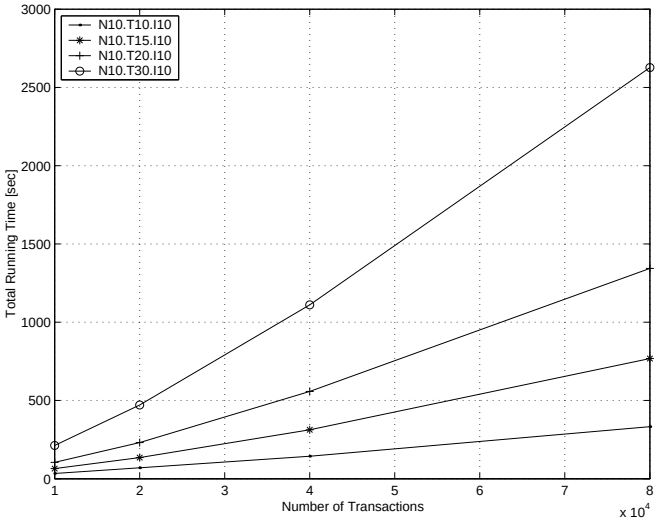

Figure 9: Scalability on the number of transactions

4.2 Chemical Compound Dataset

Our first application data contains 17,663 tree-structured chemical compounds sampled from a graph dataset of the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) [17]. 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 the various atom types as vertex labels and the various types of bonds as edge labels. There are a total of 80 distinct vertex labels and 3 distinct edge labels. Figure 10 and Figure 11 show the experimental results by *FreeTreeMiner* for finding frequent subtrees. We explored a wide range of the minimum support from 0.1% to 50%.

Figure 10 shows the total running time required for different values of support threshold. The total running time includes the time to check subtree isomorphism in the dataset and the time to generate candidate subtrees. Figure 11 displays the total number of discovered frequent subtrees, as well as the total number of maximum frequent subtrees, on those support levels. The running time and the number of frequent subtrees increase exponentially as $|S|$ decreases. With $|S| = 0.1\%$, the largest frequent subtree discovered has 43 vertices. Figure 12 shows the number of frequent subtrees with respect to tree size for some of the support threshold values. There are not many frequent subtrees with small size or large size. These experiments on the chemical compound dataset show that our *FreeTreeMiner* algorithm can handle large real application data well with a large range of support thresholds.

4.3 Multicast Trees Dataset

Our second application dataset is a dataset of IP multicast trees. IP multicast is an efficient way to send messages to a group of users. Usually on the Internet there are multiple multicast groups running at the same time, where typically each group is set up for an event such as a meeting,

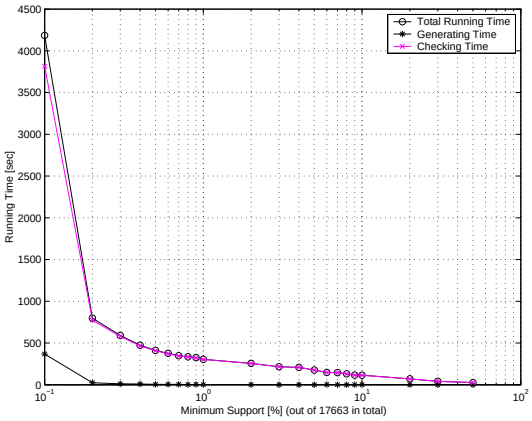

Figure 10: Total running time vs. support threshold.

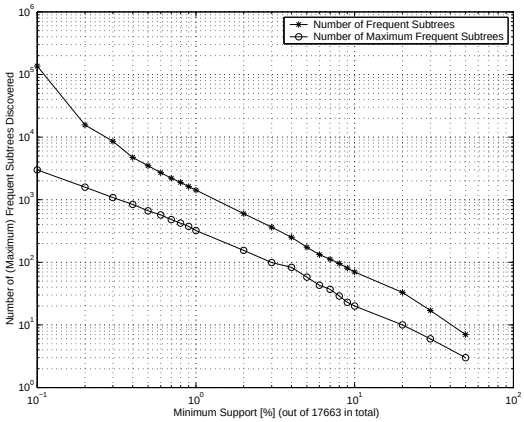

Figure 11: Total number of (maximum) frequent subtrees vs. support threshold.

a seminar, an online game, etc. One of the key issues in IP multicast is the scalability to large numbers of groups because the size of the forwarding state table at each router is proportional to the number of groups passing by the router. One solution to the scalability issue in IP multicast is to aggregate parts or all of multicast trees from different groups that are running at the same time [8]. There are several factors that affect the performance of aggregating multicast trees: Which groups to aggregate? Which parts of multicast trees to aggregate from different groups? Do we have to change the aggregation scheme very often? Our frequent tree mining algorithm can provide data to help answer these questions. By applying our *FreeTreeMiner* algorithm to a family of mutlicast trees where each tree has its own group ID, we can obtain the maximum frequent mutlicast subtrees and their supports. The size of maximum frequent multicast subtrees provides information on the reduction of the forwarding state tables using aggregation scheme. The corresponding supports give information on the number of groups that can be aggregated based on the given maximum frequent multicast subtrees.

To address the question of how often we should change the aggregation scheme, we apply our *FreeTreeMiner* algorithm to data measured from a single multicast group. For a single multicast group, we are interested in some dynamic characteristics such as which part of the multicast tree does not change very much during the whole event and what the temporal coherence for the change of the multicast tree is. We have used the MBONE multicast data provided by Chalmers and Almeroth from University of Santa Barbara ([5, 6]). The data were measured during the NASA shuttle launch between 14th and 21st in the February of 1999. It has 333 vertices (each vertex takes the IP address as its label) and 397 edges. Since we do not have information on edges, we assume a single label for all the edges. Therefore the multicast trees in this dataset are special in that each vertex in a multicast tree have a distinct label and all the edges have the same label.

First we sampled the data from this NASA dataset with 10 minutes sampling interval and got a dataset with 1,000 transactions. The transactions are the multicast trees for the same NASA event at different time. Our experimental results show that using very high support threshold, there are very few maximum frequent subtrees with more than 5 vertices. From the experiment with support threshold 80%, there are 20,765 frequent subtrees discovered with 6 maximum frequent

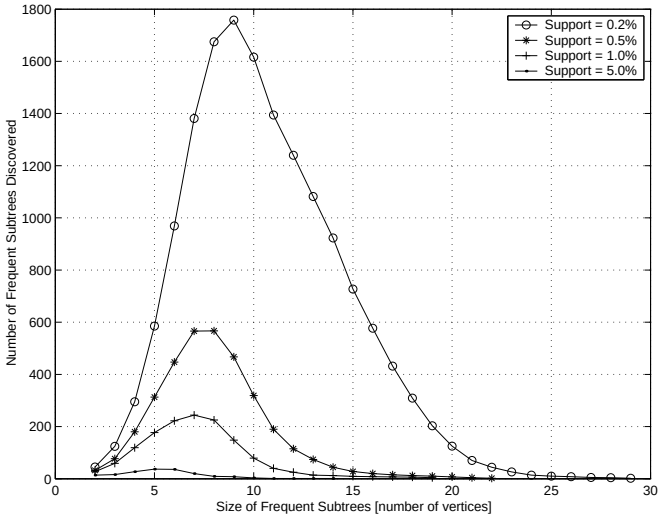

Figure 12: The number of frequent subtrees vs. frequent tree size.

subtrees, and the maximum size of discovered frequent subtrees is 29 (comparing to the average size 164 of all trees in the dataset). The result indicates that a relatively large part of the multicast tree does not change during 80% of the span of the event. We also did experiments on a wide range of sampling interval from 10 minutes up to 500 minutes with fixed support threshold of 80%. It turns out that the discovery results (number of frequent subtrees, maximum frequent subtrees and maximum size of frequent subtrees) are the same. However, as the sampling interval increases, the running time for discovering all these frequent subtrees decreases (from 5 hours to 10 minutes), because the increase of sampling interval results in decrease of number of transactions. This information provides multicast protocol designers with the approximate measure on the rate of change of (the main part of) the multicast tree. Unfortunately, our algorithm cannot handle the cases of high sampling rate with lower support thresholds in reasonably short time, because the large number of frequent subtrees exhausts the main memory. This reveals some weak points of the *FreeTreeMiner* algorithm: first, it is not quite scalable to the size of maximum frequent trees because of the combinational explosion–there are too many subtrees for a tree with very large size; second, in our candidate generating, we have recorded the *Tid-List* of each candidate to speed up the support counting step, but when the support is very large (around 80% in our case), almost every transaction supports every candidate, then the *Tid-List* does not help very much.

5 Related Work

In a recent paper, Zaki [21] presented an algorithm called TreeMiner to discover all frequent subtrees in a forest or a database of trees. In TreeMiner, Zaki also used string encoding to represent trees, but in contrast to our breadth-first traversal he used depth-first traversal for string encoding. We choose to use breadth-first traversal because of our definition of order and canonical form. In [4] Asai *et al.* modelled semi-structure data such as webpages and XML data using labeled ordered trees and presented algorithm FREQT to discover frequent subtrees. In [20], Wang

et al. also used tree structures to express semi-structured documents and presented algorithm to discovery frequent patterns. All the above algorithms focused on databases of rooted trees (either ordered or unordered) while we focus on free trees. On the one hand, in the middle steps of our algorithm, we convert free trees into rooted trees for indexing, but vertices in our transactions and subtrees do not have inherent descendent/anscestor relationships. This conversion to rooted trees therefore has to be done in each level of mining and consequently the descendent/ancestor relationships are potentially changing at each level. Consequently, the above algorithms for rooted trees are not applicable to the free tree mining problem. On the other hand, we can restrict our canonical form to rooted trees, i.e., we can skip the first step of creating root(s), therefore our algorithm can be used for applications with rooted trees with little change.

Inokuchi *et al.* [12] presented an algorithm, AGM, for mining frequent *induced* subgraphs, which are not necessarily connected, in a graph database, where an induced subgraph G_i of a graph G has vertex set V_i a subset of vertices of G and edge set E_i consisting of all those edges of G incident with two elements of V_i. Kuramochi *et al.* [13] presented another algorithm, FSG, for mining general subgraphs in a graph database. Both methods used a level-wise Apriori [1] approach: the AGM algorithm extends subgraphs by adding a vertex per level; the FSG extends by adding an edge. To check if a transaction supports a graph is an instance of the subgraph isomorphism problem which is NP-complete [9]. 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 [9]. Therefore without taking advantage of the restriction to tree-structured transactions these algorithms are not likely to be efficient for the frequent tree mining problem.

6 Conclusions and Future Work

In this paper we introduced a novel indexing technique for databases of labeled free trees. Our technique is based on a unique representation, the canonical form, for free trees that represents a free tree by its isomorphism family. With the canonical form and its equivalent representation the canonical string, we assigned a total order among all labeled free trees and therefore we can apply traditional indexing techniques to databases of free trees. Our indexing technique can be extended to unlabeled trees or unordered rooted trees because they are just special case of labeled free trees. We also defined the frequent subtree mining problem and presented an efficient algorithm, which is based on our indexing technique, to discover all frequent subtrees in a database. We used both synthetic and real application datasets to study the performance of our algorithm. The experiments showed that our algorithm is scalable with the cardinality of databases and the average size of transactions in databases.

We plan to extend our work in several directions in the future. First, the bottleneck of our algorithm is the subtree isomorphism checking, i.e., to verify if a transaction supports a candidate. In our implementation, the subtree isomorphism checking in each level is done independent of that of previous levels. We have taken this approach because to record the exact locations of subtree embedding in previous levels will require large amount of memory. Our next implementation will use incremental checking to speed up this procedure by memorizing all or parts of locations of embedding in previous levels. Second, in many applications, such as chemical compounds, there are 2D or 3D coordinates for vertices of trees. In the future, we will include this geometric information in our implementation to see if such information will improve the performance of our algorithm.

Acknowledgement

Thanks to Jun-Hong Cui in the CS Department at UCLA for providing us with the multicast dataset and for the stimulating discussions. Thanks to Michihiro Kuramochi in the CS Department at the University of Minnesota for pointing us to the chemical compound dataset. This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116 and 0085773. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

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

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

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

[4] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *2nd SIAM Int. Conf. on Data Mining*, April 2002.

[5] R. Chalmers and K. Almeroth. Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proceedings of the IEEE INFOCOM'2001*, April 2001.

[6] R. Chalmers and K. Almeroth. On the topology of multicast trees. UCSB Technical Report, March 2002.

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

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

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

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

[11] J. E. Hopcroft and R. M. Karp. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

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

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

[14] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 258–265, December 2002.

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

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

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

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

[19] R. Shamir and D. Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33:267–280, 1999.

[20] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 146–154, 1998.

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