Indexing and Mining Free Trees

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

Abstract

Tree structures are used extensively in domains such as computational biology, pattern recognition, 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. 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.

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. Some real applications using free trees include the evolutionary tree (or phylogeny) for analysis of molecular evolution, the *shape axis tree* for representing shapes in pattern recognition, and the multicast trees in computer networking. 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. The second 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 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. (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 experimental analysis. We use both synthetic data and real application data to evaluate the performance of our algorithm.

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. [5] 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 is 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.

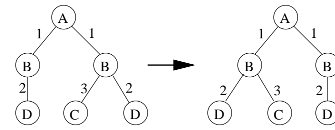

Figure 1. A Labeled, Rooted, Unordered Tree (left) and Its Canonical Form (right)

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*. 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.

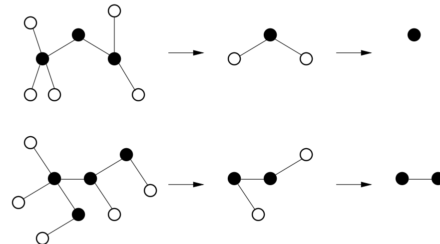

Figure 2. A Central Tree (above) and A Bicentral Tree (below)

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 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]. 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.

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

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.

2.5 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 lexicographical 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 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.

2.6 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

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. Our algorithm, like many other studies on the frequent itemsets mining problem, is based on the *Apriori* method [1]. The two main steps in the above algorithm are (1) candidate generation and (2) frequency counting. We now describe each in detail.

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

For candidate generation, 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. Therefore, a self-join on the list of all the frequent k-trees is needed. In our algorithm, 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.

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 [4] (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.

4 Experimental Results

We performed three sets of experiments to evaluate the performance of the *FreeTreeMiner* algorithm: a group of synthetic datasets, a chemical compounds dataset, and a multicast trees dataset. The main results are (1) The running time of our *FreeTreeMiner* algorithm scales linearly with the number of transactions in a database. (2) The running time scales with the size of the frequent trees in a nonlinear fashion because of the subtree isomorphism checking algorithm. (3) The number of intermediate frequent subtrees increases exponentially with the size of the maximal frequent subtree.

5 Conclusions

In this paper we introduced a novel indexing technique for databases of labeled free trees, which 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. 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.

We refer interested readers to the full version of this paper [3] for more detailed description.

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 *VLDB 94*, September 1994.

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

[3] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *ICDM 2003*, November 2003. Full version available as Technical Report CSD-TR No. 030041 at ftp://ftp.cs.ucla.edu/tech-report/2003-reports/030041.pdf.

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

[5] M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD*, July 2002.