

CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees^{*}

Yun Chi, Yirong Yang, Yi Xia, and Richard R. Muntz

University of California, Los Angeles, CA 90095, USA
{ychi,yyr,xiayi,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. One important problem in mining databases of trees is to find frequently occurring subtrees. However, because of the combinatorial explosion, the number of frequent subtrees usually grows exponentially with the size of the subtrees. In this paper, we present *CMTreeMiner*, a computationally efficient algorithm that discovers all closed and maximal frequent subtrees in a database of rooted unordered trees. The algorithm mines both closed and maximal frequent subtrees by traversing an enumeration tree that systematically enumerates all subtrees, while using an enumeration DAG to prune the branches of the enumeration tree that do not correspond to closed or maximal frequent subtrees. The enumeration tree and the enumeration DAG are defined based on a canonical form for rooted unordered trees—the depth-first canonical form (DFCF). We compare the performance of our algorithm with that of *PathJoin*, a recently published algorithm that mines maximal frequent subtrees.

keywords: *frequent subtree, closed subtree, maximal subtree, enumeration tree, rooted unordered tree.*

1 Introduction

Tree structures are used extensively in domains such as computational biology, pattern recognition, XML databases, computer networks, and so on. 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 one important issue in mining databases of labeled rooted unordered trees—finding frequently occurring subtrees, which has much practical importance [5]. However, as we have discovered in our previous study [5], because of the combinatorial explosion, the number of frequent subtrees usually grows exponentially with the tree size. This is the case especially when the transactions in the database are strongly correlated. This phenomenon has two effects: first, there are too many frequent subtrees for users to manage and use, and second, an algorithm that discovers all frequent subtrees is not able to handle frequent subtrees with large size. To solve this problem, in this paper, we propose *CMTreeMiner*, an efficient algorithm

^{*} This work was supported by NSF under Grant Nos. 0086116, 0085773, and 9817773.

that, instead of looking for all frequent subtrees, only discovers both closed and maximal frequent subtrees in a database of labeled rooted unordered trees.

Related Work Recently, there has been growing interest in mining databases of labeled trees, partly due to the increasing popularity of XML in databases. In [9], Zaki presented an algorithm, TREEMINER, to discover all frequent embedded subtrees, i.e., those subtrees that preserve ancestor-descendant relationships, in a forest or a database of rooted ordered trees. The algorithm was extended further in [10] to build a structural classifier for XML data. In [2] Asai *et al.* presented an algorithm, FREQT, to discover frequent rooted ordered subtrees. For mining rooted unordered subtrees, Asai *et al.* in [3] and we in [5] both proposed algorithms based on enumeration tree growing. Because there could be multiple ordered trees corresponding to the same unordered tree, similar canonical forms for rooted unordered trees are defined in both studies. In [4] we have studied the problem of indexing and mining free trees and developed an Apriori-like algorithm, *FreeTreeMiner*, to mine all frequent free subtrees. In [8], Xiao *et al.* presented an algorithm called *PathJoin* that rewrites a database into a compact in-memory data structure—*FST-Forest*, for the mining purpose. In addition, to the best of our knowledge, *PathJoin* is the only algorithm that mines *maximal* frequent subtrees.

Our Contributions The main contributions of this paper are: (1) We introduce the concept of *closed frequent subtrees* and study its properties and its relationship with *maximal frequent subtrees*. (2) In order to mine both closed and maximal frequent rooted unordered subtrees, we present an algorithm—*CMTreeMiner*, which is based on the canonical form and the enumeration tree that we have introduced in [5]. We develop new pruning techniques based on an enumeration DAG. (3) Finally, we have implemented our algorithm and have carried out experimental study to compare the performance of our algorithm with that of *PathJoin* [8].

The rest of the paper is organized as follows. In section 2, we give the background concepts. In section 3, we present our *CMTreeMiner* algorithm. In section 4, we show experiment results. Finally, in Section 5, we give the conclusion.

2 Background

2.1 Basic Concepts

In this section, we provide the definitions of the concepts that will be used in the remainder of the paper. We assumed that the readers are familiar with the notions such as *rooted unordered tree*, *ancestor/descendant*, *parent/child*, *leaf*, etc. In addition, a rooted tree t is a (proper) *subtree* of another rooted tree s if the vertices and edges of t are (proper) subsets of those of s . If t is a (proper) subtree of s , we say s is a (proper) *supertree* of t . Two labeled rooted unordered 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, adjacency, and the root. 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 rooted tree with k vertices a k -tree.

Let D denote a database where each transaction $s \in D$ is a labeled rooted unordered tree. For a given pattern t , which is a rooted unordered tree, we say t *occurs* in a transaction s if there exists at least one subtree of s that is isomorphic to t . The *occurrence* $\delta_t(s)$ of t in s is the number of distinct subtrees of s that are isomorphic to t . Let $\sigma_t(s) = 1$ if $\delta_t(s) > 0$, and 0 otherwise. We say s *supports* pattern t if $\sigma_t(s)$ is 1 and we define the *support* of a pattern t as $\text{supp}(t) = \sum_{s \in D} \sigma_t(s)$. 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.

One nice property of frequent trees is the *a priori* property, as given below:

Property 1. Any subtree of a frequent tree is also frequent and any supertree of an infrequent tree is also infrequent.

We define a frequent tree t to be *maximal* if none of t 's proper supertrees is frequent, and *closed* if none of t 's proper supertrees has the same support that t has. For a subtree t , we define the *blanket* of t as the set of subtrees $B_t = \{t' | \text{removing a leaf or the root from } t' \text{ can result in } t\}$. In other words, the blanket B_t of t is the set of all supertrees of t that have one more vertex than t . With the definition of blanket, we can define maximal and closed frequent subtrees in another equivalent way:

Property 2. A frequent subtree t is maximal iff for every $t' \in B_t$, $\text{supp}(t') < \text{minsup}$; a frequent subtree t is closed iff for every $t' \in B_t$, $\text{supp}(t') < \text{supp}(t)$.

For a subtree t and one of its supertrees $t' \in B_t$, we define the *difference* between t' and t ($t' \setminus t$ in short) as the additional vertex of t' that is not in t . We say $t' \in B_t$ and t are *occurrence-matched* if for each occurrence of t in (a transaction of) the database, there is at least one corresponding occurrence of t' ; we say $t' \in B_t$ and t are *support-matched* if for each transaction $s \in D$ such that $\sigma_t(s) = 1$, we have $\sigma_{t'}(s) = 1$. It is obvious that if t' and t are occurrence-matched, it implies that they are support-matched.

2.2 Properties of Closed and Maximal Frequent Subtrees

The set of all frequent subtrees, the set of closed frequent subtrees and the set of maximal frequent subtrees have the following relationship.

Property 3. For a database D and a given *minsup*, let \mathcal{F} be the set of all frequent subtrees, \mathcal{C} be the set of closed frequent subtrees, and \mathcal{M} be the set of maximal frequent subtrees, then $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$.

The reason why we want to mine closed and maximal frequent subtrees instead of all frequent subtrees is that usually, there are much fewer closed or maximal frequent subtrees compared to the total number of frequent subtrees [7]. In addition, by mining only closed and maximal frequent subtrees, we do not

lose much information because the set of closed frequent subtrees maintains the same information (including support) as the set of all frequent subtrees and the set of maximal frequent subtrees subsumes all frequent subtrees:

Property 4. We can obtain all frequent subtrees from the set of maximal frequent subtrees because any frequent subtree is a subtree of one (or more) maximal frequent subtree(s); similarly, we can obtain all frequent subtrees with their supports from the set of closed frequent subtrees with their supports, because for a frequent subtree t that is not closed, $supp(t) = \max_{t'}\{supp(t')\}$ where t' is a supertree of t that is closed.

2.3 The Canonical Form for Rooted Labeled Unordered Trees

From a rooted unordered tree we can derive many rooted ordered trees, as shown in Figure 1. From these rooted ordered trees we want to uniquely select one as the canonical form to represent the corresponding rooted unordered tree. 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. So for all running examples in the following discussion, we assume that all edges in all trees have the same label or equivalently, are unlabeled, and we therefore ignore all edge labels.

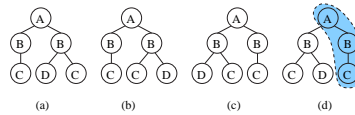


Fig. 1. Four Rooted Ordered Trees Obtained from the Same Rooted Unordered Tree

Without loss of generality, we assume that there are two special symbols, “\$” and “#”, which are not in the alphabet of edge labels and vertex labels. In addition, we assume that (1) there exists a total ordering among edge and vertex labels, and (2) “#” sorts greater than “\$” and both sort greater than any other symbol in the alphabet of vertex and edge labels. We first define the *depth-first string encoding* for a rooted ordered tree through a *depth-first* traversal and use “\$” to represent a backtrack and “#” to represent the end of the string encoding. The depth-first string encodings for each of the four trees in Figure 1 are for (a) $ABC\$BD\$C\#$, for (b) $ABC\$BC\$D\#$, for (c) $ABD\$C\$BC\#$, and for (d) $ABC\$D\$BC\#$. With the string encoding, we define the *depth-first canonical string (DFCS)* of the rooted unordered tree as the minimal one among all possible depth-first string encodings, and we 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. Using a tree isomorphism algorithm given by Aho *et al.* [1, 4, 5], we can construct the DFCF for a rooted unordered tree in $O(ck \log k)$ time, where k is the number of vertices the tree has and c is the maximal degree of the vertices in the tree.

For a rooted unordered tree in its DFCF, we define the *rightmost leaf* as the last vertex according to the depth-first traversal order, and *rightmost path* as the path from the root to the rightmost leaf. The rightmost path for the DFCF of the above example (Figure 1(d)) is the path in the shaded area and the rightmost leaf is the vertex with label C in the shaded area.

3 Mining Closed and Maximal Frequent Subtrees

Now, we describe our *CMTreeMiner* algorithm that mines both closed and maximal frequent subtrees from a database of labeled rooted unordered trees.

3.1 The Enumeration DAG and the Enumeration Tree

We first define an enumeration DAG that enumerates all rooted unordered trees in their DFCFs. The nodes of the enumeration DAG consist of all rooted unordered trees in their DFCFs and the edges consist of all ordered pairs (t, t') of rooted unordered trees such that $t' \in B_t$. Figure 2 shows a fraction of the enumeration DAG. (For simplicity, we have only shown those trees with A as the root.)

Next, we define a unique enumeration tree based on the enumeration DAG. The enumeration tree is a spanning tree of the enumeration DAG so the two have the same set of the nodes. The following theorem is key to the definition of the enumeration tree.

Theorem 1. ¹ *Removing the rightmost leaf from a rooted unordered $(k+1)$ -tree in its DFCF will result in the DFCF for another rooted unordered k -tree.*

Based on the above theorem we can build an enumeration tree such that the parent for each rooted unordered tree is determined uniquely by removing the rightmost leaf from its DFCF. Figure 3 shows a fraction of the enumeration tree for the enumeration DAG in Figure 2. In order to grow the enumeration tree, starting from a node v of the enumeration tree, we need to find all valid children of v . Each child of v is obtained by adding a new vertex to v so that the new vertex becomes the new rightmost leaf of the new DFCF. Therefore, the possible positions for adding the new rightmost leaf to a DFCF are the vertices on the rightmost path of the DFCF.

3.2 The CMTreeMiner Algorithm

In the previous section, we have used the enumeration tree to enumerate all (frequent) subtrees in their DFCFs. However, the final goal of our algorithm is to find all closed and maximal frequent subtrees. As a result, it is not necessary to grow the complete enumeration tree, because under certain conditions, some branches of the enumeration tree are guaranteed to produce no closed or maximal

¹ Proofs for all the theorems in this paper are available in [6].

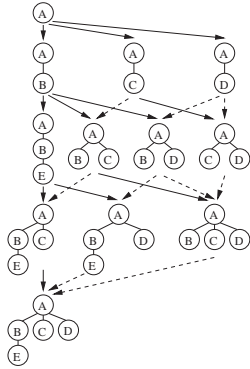


Fig. 2. The Enumeration DAG for Rooted Unordered Trees in DFCFs

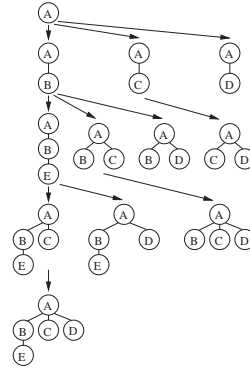


Fig. 3. The Enumeration Tree for Rooted Unordered Trees in DFCFs

frequent subtrees and therefore can be pruned. In this section, we introduce techniques that prune the unwanted branches with the help of the enumeration DAG (more specifically, the blankets).

Let us look at a node v_t in the enumeration tree. We assume that v_t corresponds to a frequent k -subtree t and denote the blanket of t as B_t . In addition, we define three subsets of B_t :

$$\begin{aligned}
 B_t^F &= \{t' \in B_t \mid t' \text{ is frequent}\} \\
 B_t^{SM} &= \{t' \in B_t \mid t' \text{ and } t \text{ are support-matched}\} \\
 B_t^{OM} &= \{t' \in B_t \mid t' \text{ and } t \text{ are occurrence-matched}\}
 \end{aligned}$$

From Property 2 we know that t is closed iff $B_t^{SM} = \emptyset$, that t is maximal iff $B_t^F = \emptyset$, and that $B_t^{OM} \subseteq B_t^{SM}$. Therefore by constructing B_t^{SM} and B_t^F for t , we can know if t is closed and if t is maximal. However, there are two problems. First, knowing that t is not closed does not automatically allow us to prune v_t from the enumeration tree, because some descendants of v_t in the enumeration tree might be closed. Second, computing B_t^F is time and space consuming, because we have to record all members of B_t and their support. So we want to avoid computing B_t^F whenever we can. In contrast, computing B_t^{SM} and B_t^{OM} is not that difficult, because we only need to record the intersections of all occurrences.

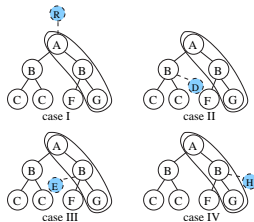


Fig. 4. Locations for an Additional Vertex to Be Added To a Subtree

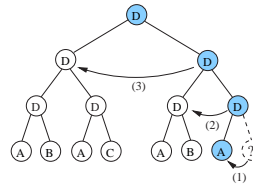


Fig. 5. Computing the Range of a New Vertex for Extending a Subtree

To solve the first problem mentioned above, we use B_t^{OM} , instead of B_t^{SM} , to check if v_t can be pruned from the enumeration tree. For a $t^o \in B_t^{OM}$ (i.e., t^o and t are occurrence-matched), the new vertex $t^o \setminus t$ can occur at different locations, as shown in Figure 4. In Case I of Figure 4, $t^o \setminus t$ is the root of t^o ; in Case II $t^o \setminus t$ is attached to a vertex of t that is not on the rightmost path; in Case III and case IV, $t^o \setminus t$ is attached to a vertex on the rightmost path. The difference between Case III and Case IV is whether or not $t^o \setminus t$ can be the new rightmost vertex of t^o .

To distinguish Case III and Case IV in Figure 4, we compute the range of vertex labels that could possibly be the new rightmost vertex of a supertree in B_t . Notice that this information is also important when we extend v_t in the enumeration tree—we have to know what are the valid children of v_t . Figure 5 gives an example for computing the range of valid vertex labels at a given position on the rightmost path. In the figure, if we add a new vertex at the given position, we may violate the DFCE by changing the order between some ancestor of the new vertex (including the vertex itself) and its immediate left sibling. So in order to determine the range of allowable vertex labels for the new vertex (so that adding the new vertex will guarantee to result in a new DFCE), we can check each vertex along the path from the new vertex to the root. In Figure 5, the result of comparison (1) is that the new vertex should have label greater than or equal to A , comparison (2) increases the label range to be greater than or equal to B , and comparison (3) increases the label range to be greater than or equal to C . As a result, before start adding new vertices, we know that adding any vertex with label greater than or equal to C at that specific position will surely result in a DFCE. Therefore, at this given location, adding a new vertex with label greater than or equal to C will result in case IV (and therefore the new vertex becomes the new rightmost vertex), and adding a new vertex with label less than C will result in case III in Figure 4.

Now we propose a pruning technique based on B_t^{OM} , as given in the following theorem.

Theorem 2. *For a node v_t in the enumeration tree and the corresponding subtree t , assume that t is frequent and $B_t^{OM} \neq \emptyset$. If there exists a $t^o \in B_t^{OM}$ such that $t^o \setminus t$ is at location of Case I, II, or III in Figure 4, then neither v_t nor any descendant of v_t in the enumeration tree correspond to closed or maximal frequent subtrees, therefore v_t (together with all of its descendants) can be pruned from the enumeration tree.*

For the second problem mentioned above, in order to avoid computing B_t^F as much as possible, we compute B_t^{OM} and B_t^{SM} first. If some $t^o \in B_t^{OM}$ is of Case I, II, or III in Figure 4, we are lucky because v_t (and all its descendants) can be pruned completely from the enumeration tree. Even if this is not the case, as long as $B_t^{SM} \neq \emptyset$, we only have to do the regular extension to the enumeration tree, with the knowledge that t cannot be maximal. To extend v_t , we find all potential children of v_t by checking the potential new rightmost leaves within the range that we have computed as described above. Only when $B_t^{SM} = \emptyset$, before doing the regular extension to the enumeration tree, do we have to compute B_t^F

to check if t is maximal. Putting all the above discussion together, Figure 6 gives our *CMTreeMiner* algorithm.

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

- 1: $CL \leftarrow \emptyset, MX \leftarrow \emptyset$;
- 2: $C \leftarrow$ frequent 1-trees;
- 3: CM-Grow($C, CL, MX, minsup$);
- 4: **return** CL, MX ;

Algorithm **CM-Grow**($C, CL, MX, minsup$)

- 1: **for** $i \leftarrow 1, \dots, |C|$ **do**
- 2: $E \leftarrow \emptyset$;
- 3: compute $B_{c_i}^{OM}, B_{c_i}^{SM}$;
- 4: **if** $\exists c^o \in B_{c_i}^{OM}$ that is of case I,II, or III **then continue**;
- 5: **if** $B_{c_i}^{SM} = \emptyset$ **then**
- 6: $CL \leftarrow CL \cup c_i$;
- 7: compute $B_{c_i}^F$;
- 8: **if** $B_{c_i}^F = \emptyset$ **then** $MX \leftarrow MX \cup c_i$;
- 9: **for each** vertex v_n on the rightmost path of c_i **do**
- 10: **for each** valid new rightmost vertex v_m of c_i **do**
- 11: $e \leftarrow c_i$ plus vertex v_m , with v_n as v_m 's parent;
- 12: **if** $supp(e) \geq minsup$ **then** $E \leftarrow E \cup e$;
- 13: **if** $E \neq \emptyset$ **then** CM-Grow($E, CL, MX, minsup$);
- 14: **return**;

Fig. 6. The *CMTreeMiner* Algorithm

We want to point out two possible variations to the *CMTreeMiner* algorithm. First, the algorithm mines both closed frequent subtrees and maximal frequent subtrees at the same time. However, the algorithm can be easily changed to mine only closed frequent subtrees or only maximal frequent subtrees. For mining only closed frequent subtrees, we just skip the step of computing B_t^F . For mining only maximal frequent subtrees, we just skip computing B_t^{SM} and use B_t^{OM} to prune the subtrees that are not maximal. Notice that this pruning is indirect: B_t^{OM} only prunes the subtrees that are not closed, but if a subtree is not closed then it cannot be maximal. If $B_t^{OM} = \emptyset$, for better pruning effects, we can still compute B_t^{SM} to determine if we want to compute B_t^F . If this is the case, although we only want the maximal frequent subtrees, the closed frequent subtrees are the byproducts of the algorithm. For the second variation, although our enumeration tree is built for enumerating all rooted *unordered* subtrees, it can be changed easily to enumerate all rooted *ordered* subtrees—the rightmost expansion is still valid for rooted ordered subtrees and we only have to remove the canonical form restriction. Therefore, our *CMTreeMiner* algorithm can handle databases of rooted ordered trees as well.

4 Experiments

We performed extensive experiments to evaluate the performance of the *CMTreeMiner* algorithm using both synthetic datasets and datasets from real applica-

tions. Due to the space limitation, here we only report the results for a synthetic dataset. We refer interested readers to [6] for other results. All experiments were done on a 2GHz Intel Pentium IV PC with 1GB main memory, running Red-Hat Linux 7.3 operating system. All algorithms were implemented in C++ and compiled using the g++ 2.96 compiler.

As far as we know, *PathJoin* [8] is the only algorithm for mining maximal frequent subtrees. *PathJoin* uses a subsequent pruning that, after obtaining all frequent subtrees, prunes those frequent subtrees that are not maximal. Because *PathJoin* uses the paths from roots to leaves to help subtree mining, it does not allow any siblings in a tree to have the same labels. Therefore, we have generated a dataset that meets this special requirement. We have used the data generator given in [5] to generate synthetic data. The detailed procedure for generating the dataset is described in [5] and here we give a very brief description. A set of $|N|$ subtrees are sampled from a large base (labeled) graph. We call this set of $|N|$ subtrees the *seed trees*. Each seed tree is the starting point for $|D| \cdot |S|$ transactions where $|D|$ is the number of transactions in the database and $|S|$ is the minimum support. 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 vertex labels is controlled by the parameter $|L|$. The parameters for the dataset used in this experiment are: $|D|=100000$, $|N|=90$, $|L|=1000$, $|S|=1\%$, $|T|=|I|$, and $|I|$ varies from 5 to 50. (For $|I| > 25$, *PathJoin* exhausts all available memory.)

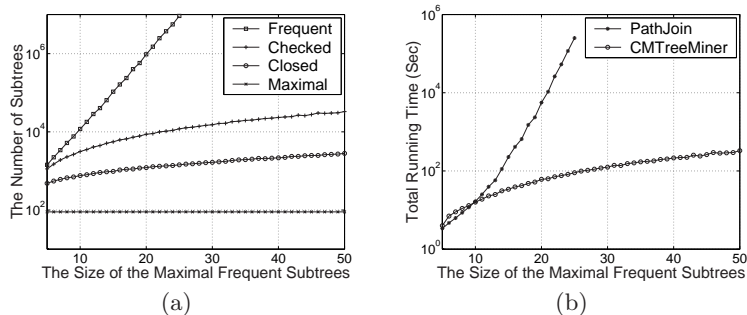


Fig. 7. CMTreeMiner vs. PathJoin

Figure 7 compares the performance of *PathJoin* with that of *CMTreeMiner*. Figure 7(a) gives the number of all frequent subtrees obtained by *PathJoin*, the number of subtrees checked by *CMTreeMiner*, the number of closed frequent subtrees, and the number of maximal frequent subtrees. As the figure shows, the number of subtrees checked by *CMTreeMiner* and the number of closed subtrees grow in polynomial fashion. In contrast, the total number of all frequent subtrees (which is a lower bound of the number of subtrees checked by *PathJoin*) grows exponentially. As a result, as demonstrated in Figure 7(b), although *PathJoin* is very efficient for datasets with small tree sizes, as tree sizes increases beyond some reasonably large value (say 10), it becomes obvious that *PathJoin* suffers from

exponential explosion while *CMTreeMiner* does not. (Notice the logarithmic scale of the figure.) For example, with the size of the maximal frequent subtrees to be 25 in the dataset, it took *PathJoin* around 3 days to find all maximal frequent subtrees while it took *CMTreeMiner* only 90 seconds!

5 Conclusion

In this paper, we have studied the issue of mining frequent subtrees from databases of labeled rooted unordered trees. We have presented a new efficient algorithm that mines both closed and maximal frequent subtrees. The algorithm is built based on a canonical form that we have defined in our previous work. Based on the canonical form, an enumeration tree is defined to enumerate all subtrees and an enumeration DAG is used for pruning branches of the enumeration tree that will not result in closed or maximal frequent subtrees. The experiments showed that our new algorithm performs in polynomial fashion instead of the exponential growth shown by other algorithms.

Acknowledgements

Thanks to Professor Y. Xiao at the Georgia College and State University for providing the *PathJoin* source codes and offering a lot of help.

References

1. A. V. Aho, J. E. Hopcroft, and J. E. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. of the 2nd SIAM Int. Conf. on Data Mining*, 2002.
3. T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *Proc. of the 6th Intl. Conf. on Discovery Science*, 2003.
4. Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proc. of the 2003 IEEE Int. Conf. on Data Mining*, 2003.
5. Y. Chi, Y. Yang, and R. R. Muntz. Mining frequent rooted trees and free trees using canonical forms. Technical Report CSD-TR No. 030043, <ftp://ftp.cs.ucla.edu/tech-report/2003-reports/030043.pdf>, UCLA, 2003.
6. Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. CMTreeMiner: Mining both closed and maximal frequent subtrees. Technical Report CSD-TR No. 030053, <ftp://ftp.cs.ucla.edu/tech-report/2003-reports/030053.pdf>, UCLA, 2003.
7. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *Lecture Notes in Computer Science*, 1540:398–416, 1999.
8. Y. Xiao, J-F Yao, Z. Li, and M. Dunham. Efficient data mining for maximal frequent subtrees. In *Proc. of the 2003 IEEE Int. Conf. on Data Mining*, 2003.
9. M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.
10. M. J. Zaki and C. C. Aggarwal. XRules: An effective structural classifier for XML data. In *Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining*, 2003.