

# Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees\*

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

May 17, 2004

## 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. Because of the combinatorial explosion, the number of frequent subtrees usually grows exponentially with the size of frequent subtrees, and therefore, mining *all* frequent subtrees becomes infeasible for large tree sizes. In this paper, we present *CMTreeMiner*, a computationally efficient algorithm that discovers only closed and maximal frequent subtrees in a database of labeled rooted trees, where the rooted trees can be either ordered or unordered. The algorithm mines both closed and maximal frequent subtrees by traversing an enumeration tree that systematically enumerates all frequent subtrees. Several pruning techniques are proposed to prune the branches of the enumeration tree that do not correspond to closed or maximal frequent subtrees. Heuristic techniques are used to arrange the order of computation so that relatively expensive computation is avoided as much as possible. These techniques are based on a novel concept, the *blanket* of a frequent subtree, in the POSET structure for all frequent subtrees. We study the performance of our algorithm through extensive experiments using both synthetic data and datasets from real applications. The experimental results show that our algorithm is very efficient in reducing the search space and quickly discovers all closed and maximal frequent subtrees.

**keywords:** Trees, Graph algorithms, Data mining, Mining methods and algorithms, frequent subtree, closed frequent subtree, maximal frequent subtree

## 1 Introduction

### 1.1 Motivation

Graphs are widely used to represent data and relationships. Among all graphs, a particularly useful family is the family of rooted trees: in the database area, XML documents are often rooted trees where vertices represent elements or attributes and edges represent element-subelement and attribute-value relationships; in Web traffic mining, access trees are used to represent the access patterns of different users; in analysis of molecular evolution, an evolutionary tree (or phylogeny) is used to describe the evolution history of certain species [12]; in computer networking, multicast trees are used for packet routing [10]. From these examples we can also see that trees in real

---

\*Part of this technical report has been reported in a previous one—CSD-TR No. 030053

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 trees—finding frequently occurring subtrees. This issue has practical importance, as shown in the following examples.

**Gaining general information of data sources** When a user initially studies a new dataset, he or she may not know the characteristics of the dataset. The frequent substructures of the dataset will often help the user to understand the dataset and guide the user to query the details about the dataset. For example, Wang et al. [25] applied their frequent subtree mining algorithm to an Internet movie database and discovered the common structures of the movie documents.

**Directly using the discovered frequent substructures** Cui et al. [10] showed a potential application of discovering frequent subtrees in network multicast routing. When multiple events are running in the network, for a router, building and storing routing tables for all the events requires too many resources (both time and space). Therefore it is more practical to partition the events into groups and to build a routing table for each group of events. Here frequent subtrees among the multicast trees of different events hint at the events to be put into one group.

**Association rule mining** To a commercial online book seller, the information on its web site structure is very important. One sample association rule that may be of interest to the online book seller is “According to the web logs, before buying a specific book, 90% buyers visited the customer evaluation section, the book description section, and the table of contents of the book (which is a subsection of the book description section).” Such an association rule can provide the book seller with insights concerning buyers’ navigation patterns and therefore help improve the web site design.

**Classification and clustering** Classification and clustering techniques classify or group data points according to their similarities. Here, a data point can be a labeled tree. For example, from the web logs of a web site we can obtain the access patterns (access trees) of the visitors. We can use the access trees to classify different types of users (casual vs. serious customers, normal visitors vs. web crawlers, etc.). In another example, Zaki et al. [31] presented an algorithm to classify XML documents according to their subtree structures.

**Helping standard database indexing and access methods design** Frequent subtrees in a database of labeled trees can provide us with information on how to build efficient indexing structures for the databases and how to design efficient access methods for different types of queries. For example, Yang et al. [29] presented algorithms for mining frequent query patterns from the logs of historic queries on an XML document. Answers to historic frequent queries can be stored and indexed for future efficient query answering.

However, as we have observed in our previous studies [6,8], 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. Two consequences follow from this exponential growth. First, the end-users will be overwhelmed by the huge number

of frequent subtrees presented to them and therefore have difficulty in gaining insights from the frequent subtrees. Second, mining algorithms may become intractable due to the exponential number of frequent subtrees. The algorithms presented by Wang et al. [25] and Xiao et al. [26] tried to alleviate the first problem by finding and presenting to end-users only the *maximal* frequent subtrees. A *maximal* frequent subtree is a frequent subtree none of whose proper supertrees is frequent. Usually, the number of *maximal* frequent subtrees is much smaller than the total number of *all* frequent subtrees. Nevertheless, because both of these algorithms use post-processing techniques that prune away non-maximal frequent subtrees after discovering all the frequent subtrees, they do not solve the second problem mentioned above. To avoid creating all the frequent subtrees in the first place, hence solving both the problems mentioned above, in this paper, we propose an efficient algorithm—*CMTreeMiner*. Instead of looking for all frequent subtrees in a database of rooted trees, the algorithm directly aims at closed and maximal frequent subtrees only, where a *closed* frequent subtree is a frequent subtree  $t$  that every  $t$ 's proper supertree occurs in fewer transactions than  $t$  does. (We will give more rigorous definitions in the following sections.) In the proposed algorithm, we use several pruning and heuristic techniques. With the pruning techniques, a large part of the search space that does not correspond to closed or maximal frequent subtrees is determined in the early stages of the algorithm and is pruned from the search space; with the heuristic techniques, the order of the tests are organized in such a way that the pruning takes place as early as possible and the relatively expensive computation is eliminated whenever possible. As the search space greatly reduced and the relatively expensive computation greatly eliminated, the running time of our algorithm is much reduced compared to algorithms that search the entire space for all the frequent subtrees. In addition, our algorithm handles databases of rooted ordered trees and databases of rooted unordered trees equally well.

## 1.2 Related Work

Recently, there has been growing interest in mining databases of labeled trees, in part due to the increasing popularity of XML in databases. The proposed frequent subtree mining algorithms can be broadly classified into three categories. The first category are Apriori-like algorithms that systematically traverse a lattice structure of all frequent subtrees through either a depth-first or a breadth-first traversal order. In [30], Zaki presented such 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 in [31] to build a structural classifier for XML data. In [7], Chi et al. have studied the problem of indexing and mining free trees and developed an Apriori-like algorithm, *FreeTreeMiner*, to mine all frequent free subtrees. In Apriori-like algorithms, a candidate subtree is generated by joining two frequent subtrees with smaller size. The second category of algorithms are based on enumeration trees. In these algorithms, a candidate subtree is generated by extending its unique parent, which is a frequent subtree of smaller size, in the enumeration tree. In [2] Asai et al. presented such an algorithm, FREQT, to discover frequent rooted ordered subtrees. For mining rooted unordered subtrees, Asai et al. [3], Nijssen et al. [19], and Chi et al. [6, 8, 9] all 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 all these studies. The third category of algorithms for mining frequent subtrees adopt the idea of the *FP*-tree [11] in frequent itemsets mining and construct a concise in-memory data structure that preserves all

necessary information. This data structure is then used for mining frequent subtrees. In [26], Xiao et al. presented such an algorithm called *PathJoin*. In addition, *PathJoin* allows the user to keep only *maximal* frequent subtrees by using a post-processing pruning that, after obtaining all frequent subtrees, eliminates those that are not maximal.

In addition to the work we have mentioned above, there are other studies on mining frequent subtrees, such as those given in [22, 24], that do not guarantee completeness, i.e., some frequent subtrees may not be in the search results. Moreover, closely related to mining frequent subtrees, many recent studies have focused on mining frequent subgraphs and closed frequent subgraphs [13–15, 27, 28], which are much more difficult problems than mining frequent subtrees (e.g., the subgraph isomorphism is an *NP*-complete problem while the subtree isomorphism problem is in *P*).

### 1.3 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) We study the POSET (partial order set) structure of all frequent subtrees and represent it using a special data structure—the enumeration DAG (directed acyclic graph). We study the relationship between the enumeration DAG and the enumeration tree, a commonly used data structure in various frequent subtree mining algorithms. We introduce a novel concept, the *blanket* of a frequent subtree in the enumeration DAG. (3) In order to mine both closed and maximal frequent rooted ordered subtrees, we present *CMTreeMiner*, a computationally efficient algorithm that discovers all closed frequent subtrees and maximal frequent subtrees by traversing an enumeration tree. In the algorithm, the blanket is used to determine the closedness and maximality of a frequent subtree. In addition, we propose efficient techniques that, in the early stages of the enumeration tree traversal, prune branches in the enumeration tree that will not give closed or maximal frequent subtrees. We also propose heuristic techniques for organizing the order of tests so that the pruning occurs early and relative expensive computation is eliminated whenever possible. (4) By using a canonical form representation for labeled rooted unordered trees, we extend our *CMTreeMiner* algorithm to mining frequent unordered subtrees. We show that with certain variations, all the techniques that we have developed for mining frequent ordered subtrees can be applied to mining frequent unordered trees as well. (5) Finally, we have implemented our algorithm and have carried out extensive experimental analysis. We use both synthetic data and real application data to evaluate the performance of our algorithm and compare the performance of our algorithm with those of several existing algorithms.

To the best of our knowledge, *CMTreeMiner* is the first algorithm that, instead of using post-processing procedures, directly mines only closed and maximal frequent subtrees.

The rest of the paper is organized as follows. In section 2, we give necessary background in graph theory, the problem definition, and various related concepts. In section 3, we look in detail at the *CMTreeMiner* algorithm for mining closed and maximal frequent subtrees from databases of labeled rooted ordered trees. In section 4, we extend the *CMTreeMiner* algorithm to handle databases of labeled rooted unordered trees. In section 5, we show experiment results. Finally, in Section 6, we offer conclusions and future research directions.

## 2 Background

In this section, we provide the definitions for some general concepts that will be used in the remainder of the paper. More specific concepts will be introduced in later sections.

### 2.1 Background in Graph Theory

A *labeled Graph*  $G = [V, E, \Sigma, L]$  consists of a *vertex* set  $V$ , an *edge* set  $E$ , an *alphabet*  $\Sigma$  for vertex and edge labels, and a *labeling function*  $L : V \cup E \rightarrow \Sigma$  that assigns labels to vertices and edges. A graph is *directed* (*undirected*) if each edge is an ordered (unordered) pair of vertices. A *path* is a list of vertices of the graph such that each pair of neighboring vertices in the list is an edge of the graph. A *cycle* is a path such that the first and the last vertices of the path are the same. A graph is *acyclic* if the graph contains no cycle. A *rooted unordered tree* is a directed acyclic graph satisfying (1) there is a distinguished vertex call the *root* that has no entering edges, (2) every other vertex has exactly one entering edge, and (3) there is a unique path from the root to any other vertex. A *rooted ordered tree* is a rooted tree that has a predefined left-to-right ordering among the children of each vertex. The *size* of a rooted tree is defined as its number of vertices. For convenience, in this paper we call a rooted tree of size  $k$  a  $k$ -tree. In a rooted tree, if vertex  $v$  is on the path from the root to vertex  $w$  then  $v$  is an *ancestor* of  $w$  and  $w$  is a *descendant* of  $v$ . If in addition  $v$  and  $w$  are adjacent, then  $v$  is the *parent* of  $w$  and  $w$  is a *child* of  $v$ . If a vertex  $w$  has no child, then  $w$  is called a *leaf*. A rooted tree  $t$  is called a *subtree* of another rooted tree  $s$  if and only if  $t$  can be obtained by repeatedly removing a leaf or the root from  $s$  (the latter case is allowed only when removing the root does not result in  $s$  becoming disconnected). If in addition, the size of  $t$  is strictly less than the size of  $s$ , then  $t$  is called a *proper subtree* of  $s$ . If  $t$  is a (proper) subtree of  $s$ , then  $s$  is called a (proper) *supertree* of  $t$ . Notice that the definitions of (proper) subtree and supertree apply to both rooted ordered trees and rooted unordered trees. Obviously, for a rooted ordered tree  $s$  and a subtree  $t$  of  $s$ , the left-to-right ordering among the children of each vertex in  $t$  is a sub ordering of that in  $s$ .

The subtree defined above is also called *induced subtree*. In [30], Zaki defined a different type of subtree called *embedded subtree*: For a tree  $s$  with vertex set  $V_s$  and edge set  $E_s$ , and a tree  $t$  with vertex set  $V_t$  and edge set  $E_t$ ,  $t$  is said to be an *embedded subtree* of  $s$  if and only if (1)  $V_t \subseteq V_s$ , and (2)  $(v_1, v_2) \in E_t$  (here  $v_1$  is the parent of  $v_2$  in  $t$ ) only if  $v_1$  is an ancestor of  $v_2$  in  $s$ . Intuitively, as an embedded subtree,  $t$  must not break the ancestor-descendant relationship among the vertices of  $s$ . In this paper, unless otherwise specified, *subtree* means *induced subtree*.

### 2.2 The Frequent Subtree Mining Problem

Let  $D$  denote a database where each transaction  $s \in D$  is a labeled rooted tree. ( $D$  can be either a database of labeled rooted ordered trees or a database of labeled rooted unordered trees.) For a given pattern  $t$  (where  $t$  is a rooted tree and whether  $t$  is ordered or unordered depends on  $D$ ), we say  $t$  *occurs* in a transaction  $s$  if  $t$  is a subtree of  $s$ . Let  $\sigma_t(s) = 1$  if  $t$  is a subtree of  $s$ , and 0 otherwise. We say  $s$  *supports* pattern  $t$  if  $\sigma_t(s)$  is 1 and we define the *support* of a pattern  $t$  in the database  $D$  as  $support(t) = \sum_{s \in D} \sigma_t(s)$ . 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.

A similar frequent subtree mining problem can be defined in the context of *embedded subtrees*, as done by Zaki in [30].

One nice property of frequent subtrees is the *apriori* property, as given in the following:

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

*Proof.* If a tree occurs in a transaction, then all its subtrees also occur in the transaction; if a tree does not occur in a transaction, then none of its supertrees can occur in the transaction.  $\square$

### 2.3 Closed and Maximal Frequent Subtrees

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.

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

**Lemma 2.** *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  $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$ .*

*Proof.*  $\mathcal{C} \subseteq \mathcal{F}$  is implied by the definition of  $\mathcal{C}$ . For  $\mathcal{M} \subseteq \mathcal{C}$ , we notice that for a tree  $t \in \mathcal{M}$ , since  $t$  is frequent and none of  $t$ 's proper supertrees is frequent, so none of  $t$ 's proper supertrees has the same support as  $t$  has, and therefore  $t \in \mathcal{C}$ .  $\square$

We are interested in mining closed and maximal frequent subtrees, instead of mining all frequent subtrees, because generally, there are much fewer closed or maximal frequent subtrees compared to the total number of frequent subtrees [20]. In addition, by mining only closed and maximal frequent subtrees, we do not lose much information, if any at all, 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:

**Lemma 3.** *We can obtain all frequent subtrees from the set of maximal frequent subtrees; similarly, we can obtain all frequent subtrees with their supports from the set of closed frequent subtrees with their supports.*

*Proof.* For the first statement, we notice that any frequent subtree is a subtree of one (or more) maximal frequent subtree(s); for the second statement, we notice that for a frequent subtree  $t$  that is not closed,  $support(t) = \max_{t'}\{support(t')\}$  where  $t'$  is a supertree of  $t$  that is closed.  $\square$

## 3 Mining Closed and Maximal Frequent Rooted Ordered Subtrees

In this section, we describe our *CMTreeMiner* algorithm that mines both closed and maximal frequent subtrees from a database of labeled rooted ordered trees. In the next section, we will extend the algorithm to labeled rooted unordered trees.

In the following discussion, we use the database given in Figure 1 as a running example. The database consists of 3 transactions and the *minsup* is set to 2. Each transaction in the database has a unique transaction id, i.e., *tid1*, *tid2*, and *tid3*, respectively. For the rest of the paper, for



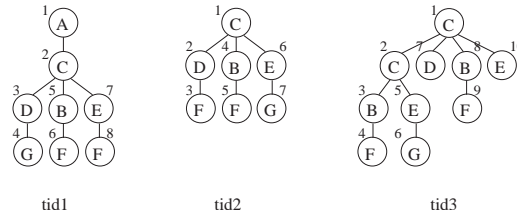


Figure 1: A Database with Three Transactions

convenience, we interchangeably talk about a transaction and its transaction id. In addition, in the database, a unique index number is assigned to each vertex in a transaction, as shown in Figure 1.

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 examples in the following discussion, we assume that all edges in all trees have the same label or equivalently, are unlabeled, and we therefore ignore all edge labels.

### 3.1 The Enumeration DAG and Its Relationship with the Enumeration Trees

We first introduce a data structure, called the *enumeration DAG*, to represent the partial order set (POSET) of all frequent subtrees. As is well known, the subtree/supertree relationship defines a partial order on the set of all frequent subtrees. This POSET is not a lattice structure, as opposed to the lattice of frequent *itemsets*, because the *least upper bound* and the *greatest lower bound* are not well defined. However, we still can put all frequent subtrees in a data structure of multiple levels, where each level consists of the frequent subtrees with size equal to the level number. In addition, directed edges are added between neighboring levels: each edge represents a subtree/supertree relationship between two frequent subtrees in neighboring levels and points from the subtree to the supertree. The result is an enumeration DAG (directed acyclic graph) whose transitive closure is the partial order defined on the POSET of all frequent subtrees. Figure 2 gives the enumeration DAG of all frequent subtrees for the database shown in Figure 1.

*Enumeration trees*, which are commonly used for frequent subtree mining, are actually spanning trees of the above enumeration DAG. Enumeration trees were first introduced by Asai et al. [2] for systematically enumerating all frequent rooted ordered trees. In Asai et al.'s algorithm, each candidate subtree is generated at most once (and therefore redundancies are avoided) from its unique parent in the enumeration tree. The parent of a subtree is uniquely determined by removing the *rightmost* vertex of the subtree according to the depth-first traversal order. Figure 3 shows the enumeration tree, which is a spanning tree of the enumeration DAG in Figure 2. In Figure 3, for each frequent subtree, we use a bounding box to emphasize its parent in the enumeration tree.

As observed by Asai et al., to add a new vertex  $w$  to a frequent subtree  $t$ , in order for  $w$  to be the rightmost vertex of the new subtree,  $w$ 's parent  $v$  must be a vertex on the *rightmost path* of  $t$ , and  $w$  must be  $v$ 's rightmost child. The rightmost path of  $t$  is the path from the root to the rightmost vertex of  $t$ . For example, for the root tree shown in Figure 4, vertex  $G$  is the rightmost vertex and the rightmost path is the path in the shaded area. A frequent subtree mining algorithm therefore systematically grows the enumeration tree by extending each frequent subtree in the enumeration tree. When extending a frequent subtree  $t$ , an additional vertex  $w$  is added as the rightmost child

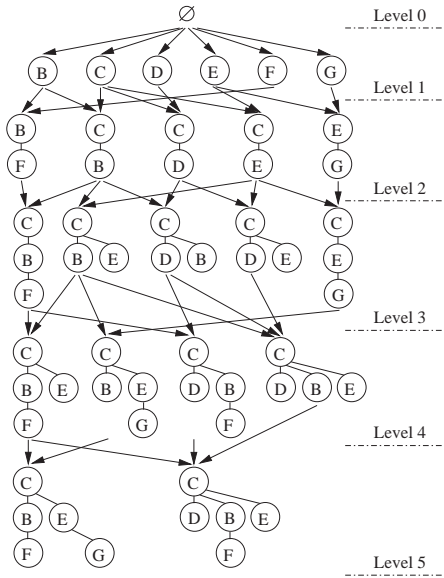


Figure 2: The Enumeration DAG

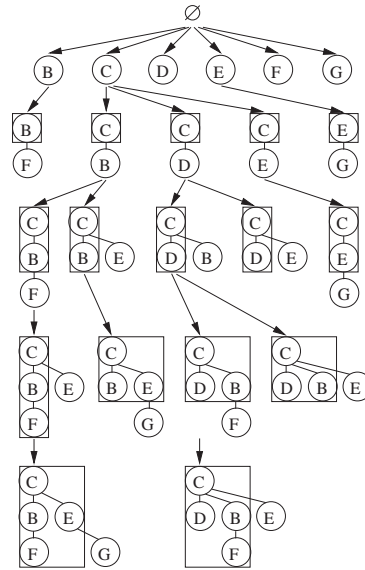


Figure 3: The Enumeration Tree

of one vertex on the rightmost path of  $t$  (and therefore  $w$  becomes the new rightmost vertex); if the resulting  $t'$  is frequent,  $t'$  becomes a child of  $t$  in the enumeration tree.

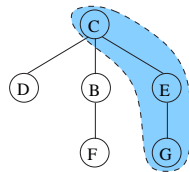


Figure 4: The Rightmost Path and the Rightmost Vertex

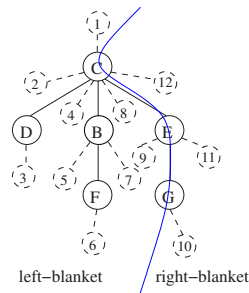


Figure 5: The Left Blanket and the Right Blanket

Our *CMTreeMiner* algorithm follows Asai et al.’s enumeration tree idea. However, the enumeration DAG (more specifically, the *blanket*) is used to determine the closedness and the maximality of frequent subtrees. In addition, the enumeration DAG (the *blanket*) is also used to prune the branches of the enumeration tree that do not correspond to closed or maximal frequent subtrees.

### 3.2 The Blanket, the Closedness and the Maximality Checking

For a frequent subtree  $t$ , we define the *blanket* of  $t$ , denoted by  $B_t$ , as the set of *immediate supertrees* of  $t$  that are frequent, where an immediate supertree of  $t$  is a supertree  $t'$  of  $t$  that has one more vertex than  $t$ . Figure 6 shows a rooted tree  $t$  and some supertrees that are potentially in  $B_t$  ( $tree(a)$  is not frequent so actually it is not in  $B_t$ ). For a node  $t$  (which represents a frequent subtree) in the



enumeration DAG of all frequent subtrees, the blanket  $B_t$  of  $t$  consists of the nodes reachable from  $t$  in one step. The definition of blanket applies to both rooted ordered trees and rooted unordered trees.

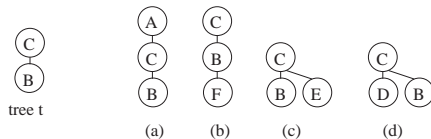


Figure 6: A Rooted Ordered Tree  $t$  (left) and Some Potential Supertrees in Its Blanket  $B_t$  (right)

For a subtree  $t$  and one of its immediate supertrees  $t' \in B_t$ , we use  $t' \setminus t$  to represent the additional vertex  $w$  in  $t'$  that is not in  $t$ . Please notice that  $t' \setminus t$  represents not only the vertex label of  $w$ , but also its position. In addition,  $t' \setminus t$  might be the root of  $t'$ , as shown in case(a) in Figure 6.

Using the definition of blanket, we can define maximal and closed frequent subtrees in an equivalent manner which turns out to be quite useful:

**Lemma 4.** *A frequent subtree  $t$  is maximal iff  $B_t = \emptyset$ ; a frequent subtree  $t$  is closed iff for each  $t' \in B_t$ ,  $support(t') < support(t)$ .*

*Proof.* Directly from the definition of the closed frequent subtrees and that of maximal frequent subtrees. □

Therefore, instead of a post-processing step, we can tell the closedness and the maximality of a frequent subtree  $t$  by checking the supports of its supertrees in  $B_t$ . However, this checking by itself does not help us too much, for we still have to traverse the whole enumeration tree. In the next section, we will see how the blanket helps us to prune the branches in the enumeration tree that do not correspond to closed or maximal frequent subtrees.

### 3.3 The Pruning Techniques

In the previous section, we have used the enumeration tree to enumerate all frequent subtrees. However, the final goal of our algorithm is to find only closed and maximal frequent subtrees. Therefore, 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 frequent subtrees and therefore can be pruned from the search space. In this section, we introduce techniques that prune the unwanted branches with the help of the enumeration DAG (more specifically, the blankets).

#### 3.3.1 Occurrence-Matching and Transaction-Matching

If a subtree  $t$  occurs in a transaction  $s$ , it can occur more than once. For example, the 2-tree with  $C$  as the root and  $B$  as  $C$ 's only child (i.e.,  $t_8$  in Figure 7) occurs once in  $tid1$ , once in  $tid2$ , and twice in  $tid3$ . We call each of them an *occurrence* of  $t$  in the database. For a  $t' \in B_t$ , we say that  $t'$  and  $t$  are *occurrence-matched* if for each occurrence of  $t$  in (a transaction of) the database, there is at least one (there could be more than one) corresponding occurrence of  $t'$ ; we say that  $t'$  and  $t$  are *transaction-matched* if for each transaction  $s \in D$  such that  $\sigma_t(s) = 1$ , we have  $\sigma_{t'}(s) = 1$ .

**Lemma 5.** *If  $t'$  and  $t$  are occurrence-matched, then they are transaction-matched.*

*Proof.* Directly implied by the two definitions. □

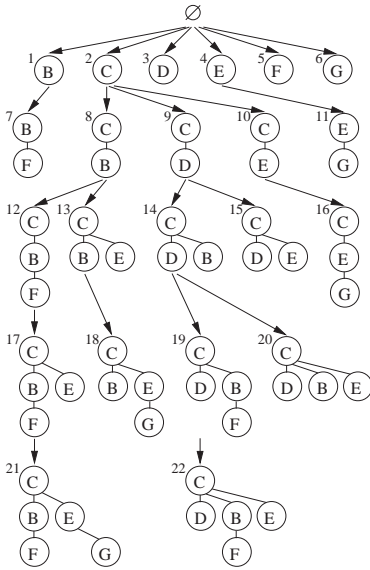


Figure 7: The Enumeration Tree

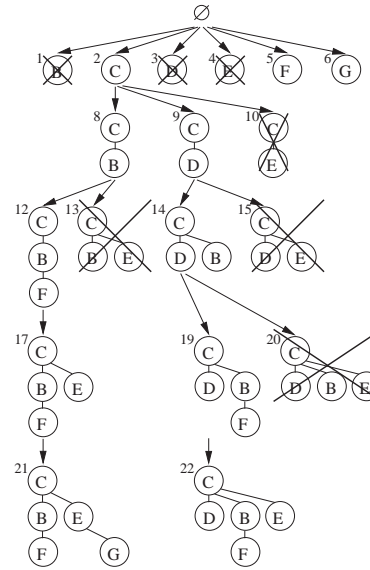


Figure 8: The Enumeration Tree after Pruning

For example,  $t_{13}$  and  $t_{10}$  in Figure 7 are occurrence-matched. (Figure 7 is a redrawn enumeration tree for our database, with indices added for convenience of discussion.) This is because (1)  $t_{13} \in B_{t_{10}}$  and (2) whenever  $t_{10}$  occurs in any position of any transaction in the database, there is a vertex  $B$  ( $t_{13} \setminus t_{10}$ ) as the left sibling of the vertex  $E$  in  $t_{10}$ . On the other hand,  $t_{14}$  and  $t_8$  are transaction-matched but not occurrence-matched: although  $t_{14}$  occurs in all transactions where  $t_8$  occurs ( $tid1$ ,  $tid2$ , and  $tid3$ ),  $t_{14}$  does not occur in *all* position that  $t_8$  occurs. In  $tid3$ ,  $t_8$  occurs twice but  $t_{14}$  only occurs once.

### 3.3.2 The Left-Blanket Pruning

For a frequent subtree  $t$  and one of its supertrees  $t' \in B_t$ , the vertex  $t' \setminus t$  can be at different locations, such as position 1 through position 12 shown in Figure 5. According to whether  $t' \setminus t$  is the rightmost vertex of  $t'$ , we divide  $B_t$  into two parts: the *left-blanket*  $B_{t\_left}$  and the *right-blanket*  $B_{t\_right}$ , as shown in Figure 5. For every  $t' \in B_{t\_right}$ ,  $t' \setminus t$  is the rightmost vertex of  $t'$  (in Figure 5, this corresponds to  $t' \setminus t$  to be at position 10, 11, or 12); for every  $t' \in B_{t\_left}$ ,  $t' \setminus t$  is not the rightmost vertex of  $t'$  (in Figure 5, this corresponds to  $t' \setminus t$  to be at position 1 through 9). An equivalent definition for the left-blanket and right-blanket is that  $B_{t\_right} = \{t' \in B_t \mid t' \text{ is a child of } t \text{ in the enumeration tree}\}$  and  $B_{t\_left} = B_t \setminus B_{t\_right}$ . (Here  $B_t \setminus B_{t\_right}$  means  $B_t \cap \overline{B_{t\_right}}$ .)

Now we are ready to introduce the first pruning technique—the *left-blanket pruning*.

**Lemma 6.** *For a frequent subtree  $t$  in the enumeration tree, if there exists a  $t' \in B_{t\_left}$  such that  $t'$  and  $t$  are occurrence-matched, then (1)  $t$  is not closed (and therefore not maximal) and (2) for each child  $t''$  of  $t$  in the enumeration tree (i.e., for each  $t'' \in B_{t\_right}$ ), there exists at least one supertree  $t''' \in B_{t''\_left}$  so that  $t'''$  and  $t''$  are occurrence-matched.*

*Proof.* The first statement is true because  $support(t) = support(t')$ . For the second statement, we notice that  $t' \setminus t$  occurs at each occurrence of  $t$  so it occurs at each occurrence of  $t''$ ; in addition,  $t' \setminus t$  is on the left of the rightmost path of  $t$  so  $t' \setminus t$  will never occur in  $t''$ . Therefore, we can obtain the  $t'''$  that satisfies the requirement by adding  $t' \setminus t$  to  $t''$ .  $\square$

**Theorem 1.** *For a frequent subtree  $t$  in the enumeration tree, if there exists a  $t' \in B_{t\_left}$  such that  $t'$  and  $t$  are occurrence-matched, then neither  $t$  nor any  $t$ 's descendants in the enumeration tree can be closed (or maximal), and therefore  $t$  (together with all  $t$ 's descendants) can be pruned from the enumeration tree.*

*Proof.* By inductively applying Lemma 6 to  $t$  and its children in the enumeration tree.  $\square$

In the example shown in Figure 8,  $t_1$ ,  $t_3$ ,  $t_4$ ,  $t_{10}$  and  $t_{15}$  are pruned from the enumeration tree by using the left-blanket pruning.

In the theorem we have used *occurrence-matching*, because a transaction-matching element in  $B_{t\_left}$  does not allow us to prune  $t$ . For example,  $t_{14} \in B_{t_8\_left}$  and  $t_{14}$  and  $t_8$  are transaction-matched (for they both occur in every transaction in the database given in Figure 1). But if we prune  $t_8$  and its descendants from the enumeration tree, we will miss  $t_{22}$ , which is a closed (and maximal) frequent subtree.

### 3.3.3 The Right-Blanket Pruning

If an occurrence-matching  $t'$  occurs in the right blanket  $B_{t\_right}$  of  $t$ , we cannot prune  $t$  from the enumeration tree. However, depending on the location of  $t' \setminus t$ , we still can possibly prune some children of  $t$  in the enumeration tree. For example, assume that the rooted tree  $t$  in Figure 4 is frequent and we want to extend  $t$  by finding all  $t$ 's children in the enumeration tree. To find  $t$ 's children, we add a vertex at position 10, 11, or 12 in Figure 5 to see if the resulting supertree  $t'$  of  $t$  is frequent. However, if there is an element  $t' \in B_{t\_right}$  such that  $t' \setminus t$  is at the location of 10 in Figure 5 and if  $t'$  and  $t$  are occurrence-matched, then although we still have to explore  $t'$  in the enumeration tree, after doing it, we do not have to further extend  $t$  by adding a new rightmost vertex to position 11 or position 12 in Figure 5. In other words, we can prune some children of  $t$  in  $B_{t\_right}$  by skipping the exploration of the ancestors of vertex  $G$  on the rightmost path of  $t$ .

**Theorem 2.** *For a frequent subtree  $t$  in the enumeration tree, if there exists  $t' \in B_{t\_right}$  such that  $t'$  and  $t$  are occurrence-matched, and the parent of  $t' \setminus t$  is  $v$  (where  $v$  is a vertex on the rightmost path of  $t$ ), then we do not have to extend  $t$  by adding new rightmost vertices to any proper ancestor of  $v$ .*

*Proof.* Assume that there is a child  $t''$  of  $t$  in the enumeration tree that is obtained by adding a new rightmost vertex  $t'' \setminus t$  to  $t$ , where the parent  $v'$  of  $t'' \setminus t$  is a proper ancestor of  $v$ . Then  $v$  is on the left of the rightmost path of  $t''$  and therefore so is  $t' \setminus t$  (because  $t' \setminus t$  is a child of  $v$ ). As a result we can construct a  $t''' \in B_{t''\_left}$  by adding  $t' \setminus t$  to  $t''$  and obviously  $t'''$  and  $t''$  are occurrence-matched.

Therefore by Theorem 1, neither  $t''$  nor its descendants in the enumeration tree can be closed (or maximal).  $\square$

In the example shown in Figure 8,  $t_{13}$  and  $t_{20}$  are pruned from the enumeration tree by using the right-blanket pruning. Actually, these two subtrees will not appear in the enumeration tree at all, i.e., because of the right-blanket pruning of  $t_8$  and  $t_{14}$  respectively,  $t_{13}$  and  $t_{20}$  will not be generated in the first place.

### 3.4 The Order of Computation—A Heuristic Technique

To extend a frequent subtree  $t$  in the enumeration tree, supertrees in  $B_t$  and their supports are needed. However, in some cases, only a subset of  $B_t$  is needed (e.g., when  $t$  can be pruned using the left-blanket pruning). In this section, we show that different subsets of  $B_t$  have different computational costs. We also organize the order of computation for these subsets to avoid, whenever possible, computing the subset with the highest computational cost.

#### 3.4.1 The Subsets to Be Computed

To extend a frequent subtree  $t$  in the enumeration tree, depending on if  $t$  can be pruned from the enumeration tree and if  $t$  is closed/maximal, we will have to compute one or more of the following mutually exclusive subsets of  $B_t$ :

$$\begin{aligned} B_t^{OM} &= \{t' \in B_t \mid t' \text{ and } t \text{ are occurrence-matched}\} \\ B_t^{TM} &= \{t' \in B_t \setminus B_t^{OM} \mid t' \text{ and } t \text{ are transaction-matched}\} \\ B_t^F &= \{t' \in B_t \setminus (B_t^{OM} \cup B_t^{TM}) \mid t' \text{ is frequent}\} \end{aligned}$$

Each of these three subsets can be further partitioned into the left part and the right part, by projecting them on  $B_{t\_left}$  and  $B_{t\_right}$ . (For example,  $B_t^{OM} = B_{t\_left}^{OM} \cup B_{t\_right}^{OM}$ .) If there exists  $t' \in B_{t\_left}^{OM}$ , then neither  $t$  nor any of  $t$ 's descendants in the enumeration tree can possibly be closed or maximal, so we can safely prune  $t$  and all  $t$ 's descendants from the enumeration tree. If  $B_{t\_left}^{OM} = \emptyset$  but there exists  $t' \in B_{t\_right}^{OM}$ , then  $t$  still cannot be closed or maximal. In this case, although we cannot prune  $t$  from the enumeration tree, we can possibly apply Theorem 2 to prune some children of  $t$  in the enumeration tree. If  $B_t^{OM} = \emptyset$ , then no pruning is possible and we have to compute  $B_t^{TM}$  to determine if  $t$  is closed. If  $B_t^{TM} \neq \emptyset$ , then  $t$  is not closed and therefore not maximal; if  $B_t^{TM} = \emptyset$ , then  $t$  is closed. In this case,  $t$  can also be maximal and whether or not  $B_t^F = \emptyset$  determines whether or not  $t$  is maximal.

#### 3.4.2 How to Compute the Subsets

Now we study in detail how to compute each of these three subsets and show that their computational costs are different.

For a frequent subtree  $t$ , each occurrence of  $t$  has some candidate supertrees for  $B_t^{OM}$ . To finally determine  $B_t^{OM}$ , we compute the *intersection* of these candidate supertrees from each occurrence of  $t$ , because for a supertree  $t' \in B_t^{OM}$ ,  $t' \setminus t$  should occur with *each* occurrence of  $t$ . For example, assume that we want to compute  $B_{t_8}^{OM}$  for  $t_8$  in Figure 7. As we can see,  $t_8$  occurs once in *tid1*, once in *tid2*, and twice in *tid3*. We first look at the first occurrence of  $t_8$  for candidate supertrees

in  $B_{t_8}^{OM}$ . The first occurrence happens in *tid1* and there are three candidate supertree  $t'_8$ 's (here we only show  $t'_8 \setminus t_8$ ):  $D$  (as the left sibling of  $B$ ),  $E$  (as the right sibling of  $B$ ), and  $F$  (as the child of  $B$ ). (We ignore  $A$  because it is not frequent.) Next, we look at all the remaining occurrences of  $t_8$  to see if any of these three candidate supertrees occur in all of them. It turns out that only two,  $E$  and  $F$ , are in all occurrences, and therefore  $B_{t_8}^{OM}$  consists of these two supertrees. From this example we see that computing  $B_t^{OM}$  is not an expensive operation:  $B_t^{OM}$  is just the intersection of the candidate supertrees from all occurrences of  $t$ .

Computing  $B_t^{TM}$  is similar to computing  $B_t^{OM}$ , except that each *transaction*, instead of each *occurrence*, has its candidate supertrees for  $B_t^{TM}$ . If there are multiple occurrences of  $t$  in the same transaction, then the *union* of the candidate supertrees from these occurrences is taken as the candidate supertrees proposed by the transaction. For example, assume that we want to compute  $B_{t_{17}}^{TM}$  for  $t_{17}$  in Figure 7. The occurrence of  $t_{17}$  in *tid1* has two candidate supertrees (here we only show  $t'_{17} \setminus t_{17}$ ): a  $D$  (as the left sibling of  $B$ ) and an  $F$  (as the child of  $E$ ); the occurrence of  $t_{17}$  in *tid2* has two candidate supertrees: a  $D$  (as the left sibling of  $B$ ) and a  $G$  (as the child of  $E$ ). However,  $t_{17}$  occurs twice in *tid3*, so we compute the union of all the candidate supertrees from both occurrences, which results in 4 candidate supertrees: a  $C$  (as the left sibling of  $B$ ), a  $D$  (as the left sibling of  $B$ ), a  $G$  (as the child of  $E$ ), and another  $C$  (as the parent of  $C$ ). The intersection of these candidate supertrees in each transaction gives us a single element in  $B_{t_{17}}^{TM}$ —a  $D$  as the left sibling of  $B$ . Again, we can see that computing  $B_t^{TM}$  is not an expensive operation, although it is slightly more expensive than computing  $B_t^{OM}$ . In addition, we can see that the computation of  $B_t^{OM}$  and  $B_t^{TM}$  does not involve storing any support, and the computation may be terminated before visiting all the occurrences of  $t$ : whenever the intersection of candidate supertrees becomes empty, the rest of the occurrences can be skipped.

To compute  $B_t^F$ , however, is relatively expensive, for the following reason. Again, each transaction has candidate supertrees for  $B_t^F$ , and if there are multiple occurrences of  $t$  in a transaction, the union of the candidate supertrees from each of these occurrences is taken as the candidate supertrees from the transaction. This time, however, instead of a simple intersection, the *union* of all the candidate supertrees from each transaction is used. Furthermore, the support of each candidate supertree must be stored and updated in the process, although a large part of the candidate supertrees may turn out to be infrequent at the end. For example, assume that we want to compute  $B_{t_{22}}^F$  for  $t_{22}$  in Figure 7.  $t_{22}$  occurs once in each of *tid1*, *tid2*, and *tid3*. For the occurrence in *tid1*, two candidate supertrees need to be recorded (here we only show  $t'_{22} \setminus t_{22}$ ): a  $G$  (as the child of  $D$ ) and an  $F$  (as the child of  $E$ ); for the occurrence in *tid2*, two candidate supertrees need to be recorded: an  $F$  (as the child of  $D$ ) and a  $G$  (as the child of  $E$ ); for the occurrence in *tid3*, one candidate supertree needs to be recorded: a  $C$  (as the left sibling of  $D$ ). Although finally none of these candidate supertrees turns out to be frequent (they all occur less than the minimum support 2), during the process, we have to store all of them and update their supports whenever necessary, because we do not know if any of them will become frequent at the end. This also implies that the memory usage for computing  $B_t^F$  is not bounded by the size of any transaction. Therefore, computing  $B_t^F$  is relatively expensive.

### 3.4.3 The Order of Computation

Fortunately, the subset of  $B_t$  that has the most pruning power, i.e.,  $B_t^{OM}$ , costs the least to compute. In order to avoid computing  $B_t^F$  as much as possible, we adopt the following order in our algorithm

to compute the three subsets:

- (1) Compute  $B_t^{OM}$ . If  $\exists t' \in B_{t\_left}^{OM}$ , prune  $t$ ; else if  $\exists t' \in B_{t\_right}^{OM}$ , apply Theorem 2. In either case, there is no need to compute  $B_t^{TM}$  or  $B_t^F$ , because  $t$  is neither closed nor maximal.
- (2) Compute  $B_t^{TM}$  if  $B_t^{OM} = \emptyset$ . If  $B_t^{TM} \neq \emptyset$ , there is no need to compute  $B_t^F$ , because  $t$  is neither closed nor maximal; else,  $t$  is closed.
- (3) Explore  $t$  in the enumeration tree. If any children  $t'$  of  $t$  turns out to be frequent, then there is no need to compute  $B_t^F$ , because  $t$  is not maximal.
- (4) Compute  $B_{t\_left}^F$  if  $B_t^{OM} = \emptyset$ ,  $B_t^{TM} = \emptyset$ , and none of  $t$ 's children in the enumeration tree is frequent. If  $B_{t\_left}^F = \emptyset$ , then  $t$  is both closed and maximal, otherwise,  $t$  is closed but not maximal.

### 3.5 Putting It All Together—The CMTreeMiner Algorithm

Figure 9 and Figure 10 summarize the final *CMTreeMiner* algorithm. The inputs of the algorithm are the database and the user-defined minimum support, and the outputs are the closed frequent subtrees and the maximal frequent subtrees. Applying the algorithm to the database shown in Figure 1, the outputs will be three closed frequent subtrees ( $t_6$ ,  $t_{21}$ , and  $t_{22}$  in Figure 8), among which two ( $t_{21}$  and  $t_{22}$ ) are maximal frequent subtrees.

---

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

---

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

---

Figure 9: The CMTreeMiner Algorithm

### 3.6 Possible Variations

It is worthwhile to point out some possible variations to the *CMTreeMiner* algorithm. 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. To mine only closed frequent subtrees, the algorithm is modified to skip the step of computing  $B_{t\_left}^F$  (i.e., lines 15, 16, and 17). For mining only maximal frequent subtrees, the algorithm modification is to skip computing  $B_t^{TM}$  and use  $B_t^{OM}$  to prune the subtrees that are not maximal. 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  $B_t^{TM}$  to determine if we want to compute  $B_{t\_left}^F$ . In this case, although we only want the maximal frequent subtrees, the closed frequent subtrees are the byproducts of the algorithm.



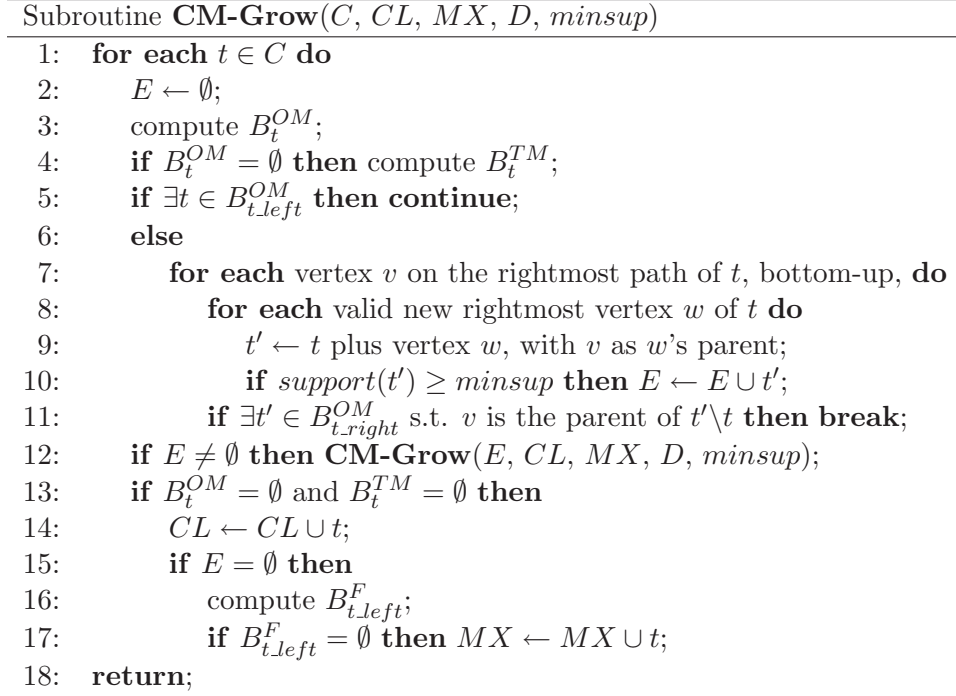


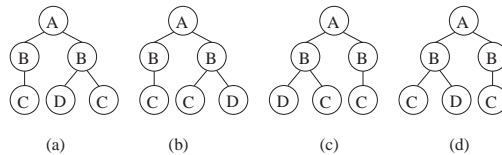
Figure 10: The CM-Grow Subroutine

## 4 Mining Closed and Maximal Frequent Rooted Unordered Subtrees

In this section, we extend the *CMTreeMiner* algorithm to handle databases of labeled rooted unordered trees. By defining a canonical form for labeled rooted unordered trees, we extend the concepts that are introduced in the previous section, such as the enumeration DAG, the enumeration tree, and the blanket. In addition, we show that all the pruning and heuristic techniques in the *CMTreeMiner* algorithm are applicable to mining labeled rooted unordered trees.

### 4.1 The Canonical Form for Rooted Labeled Unordered Trees

From a rooted unordered tree we can derive many rooted ordered trees, as shown in Figure 11. From these rooted ordered trees we want to uniquely select one as the canonical form to represent the corresponding rooted unordered tree.

Figure 11: 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 as the string of labels obtained through a *depth-first* traversal of the tree, where a “\$” represents a backtrack and a “#” represents the end of the string encoding. The depth-first string encodings for each of the four trees in Figure 11 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 form (DFCF)* for a labeled rooted unordered tree using the following recursive definition:

- (1) For a labeled rooted unordered tree with a single vertex, it is trivially in DFCF;
- (2) For a labeled rooted unordered tree  $t$  with more than one vertex, its DFCF is obtained by the following recursive procedure: assuming that the root  $r$  of  $t$  has  $K$  children denoted by  $r_1, \dots, r_K$ , we first determine the DFCFs for the subtrees  $t_{r_1}, \dots, t_{r_K}$  rooted at  $r_1, \dots, r_K$ , then reorder these DFCFs (which are labeled rooted ordered trees) from left to right in the increasing lexicographical order of their depth-first string encodings.

In Figure 11, *tree (d)* is the DFCF for the corresponding labeled rooted unordered tree. It can be shown [6] that the DFCF can be defined in another equivalent way: the DFCF of a labeled rooted unordered tree  $t$  is the labeled rooted ordered tree, among all the labeled rooted ordered trees that can be derived from  $t$ , that gives the smallest depth-first string encoding.

Using a tree isomorphism algorithm given by Aho et al. [1, 17], we can construct the DFCF for a rooted unordered tree in linear time with respect to the size of the tree.

Note that the *depth-first string encoding* we use here is equivalent to the string encoding that was first proposed by Zaki [30] to represent labeled rooted ordered trees. Independent of our work [7], Asai et al. [3] and Nijssen et al. [19] proposed similar canonical forms, which are equivalent to our DFCF, for labeled rooted unordered trees.

## 4.2 Extending the Enumeration DAG and the Enumeration Tree

Similar to the case of rooted ordered trees, we can also build an enumeration DAG to represent the POSET of all frequent rooted unordered trees. In the enumeration DAG, each node represents a frequent rooted unordered tree in its canonical form (DFCF). Figure 12 shows the enumeration DAG of all the frequent rooted unordered trees in the database given in Figure 1. An enumeration tree, which is a spanning tree of the enumeration DAG, can be obtained by uniquely determining the parent of each frequent subtree in the enumeration DAG. In our algorithm, we again adopt the enumeration tree defined by Asai et al. [3], in which the parent of a frequent subtree  $t$  is determined by removing the rightmost vertex from the DFCF of  $t$ . Figure 12 shows such an enumeration tree (by ignoring the arrows with dashed lines). In the Figure,  $t_6$ ,  $t_{21}$ , and  $t_{22}$  are closed, and among them,  $t_{21}$  and  $t_{22}$  are maximal.

## 4.3 Extending the Blanket and Other Concepts

The concepts of the blanket, the occurrence-matching, and the transaction-matching all apply to rooted unordered trees as well. The enumeration tree growing method is similar as well: for a

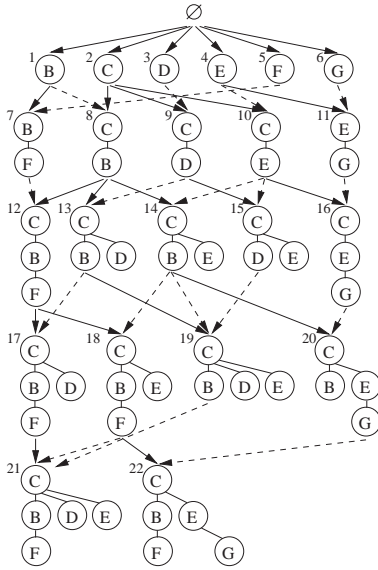


Figure 12: The Enumeration DAG and Enumeration Tree

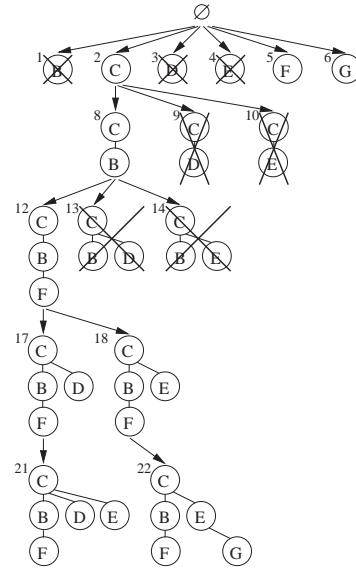


Figure 13: The Enumeration Tree after Pruning

frequent unordered subtree  $t$  in the enumeration tree (without loss of generality, when we talk about a subtree  $t$  in the enumeration tree, we assume that  $t$  is in its DFCF), to create a candidate child for  $t$ , a new vertex  $w$  is added as a child of some vertex  $v$  on the rightmost path of  $t$ . However, even if the resulting  $t'$  is frequent, in order for  $t'$  (which is in DFCF) to be the child of  $t$  in the enumeration tree,  $w$  must be the rightmost vertex of  $t'$ . This requirement restricts the range of labels  $w$  can take, as will become evident from the following discussion.

Again, we define the left and the right blankets for a frequent subtree  $t$  in the enumeration tree as  $B_{t\_right} = \{t' \in B_t | t' \text{ is a child of } t \text{ in the enumeration tree}\}$  and  $B_{t\_left} = B_t \setminus B_{t\_right}$ . However, for a  $t' \in B_t$ , whether  $t'$  is in  $B_{t\_left}$  or in  $B_{t\_right}$  not only depends on the location of  $t' \setminus t$ , it may also depend on the vertex label of  $t' \setminus t$ . For example, in Figure 14, if the parent of  $t' \setminus t$  is not a vertex on the rightmost path of  $t$  (as shown in case I and case II), then  $t' \in B_{t\_left}$ ; if the parent of  $t' \setminus t$  is a vertex on the rightmost path of  $t$ , then depending on the vertex label,  $t' \setminus t$  may (as shown in case IV) or may not (as shown in case III) be the rightmost vertex of the DFCF for  $t'$ . In the former case,  $t' \in B_{t\_right}$ , because  $t'$  is a child of  $t$  in the enumeration tree (i.e., removing the rightmost vertex from  $t'$  will result in  $t$ ); in the latter case,  $t' \in B_{t\_left}$ .

To distinguish case III and case IV in Figure 14, for each vertex  $v$  on the rightmost path of  $t$ , we compute the ranges of labels that the new vertex  $w$  can take in each case. For case IV, the label for  $w$  should be in such a range that adding  $w$  as the child of  $v$  will give us a supertree  $t' \in B_{t\_right}$  with  $w$  as the rightmost vertex;  $w$  with other labels will give us case III. It is worth mentioning that this information is also important when we extend  $t$ , because it tells us what are the valid children of  $t$  in the enumeration tree. Figure 15 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 DFCF 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

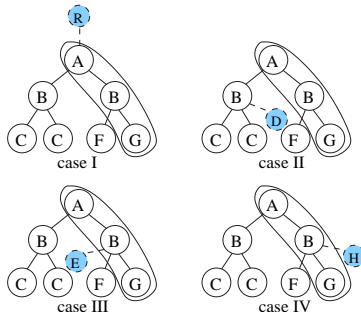


Figure 14: Locations for an Additional Vertex to Be Added To a Subtree

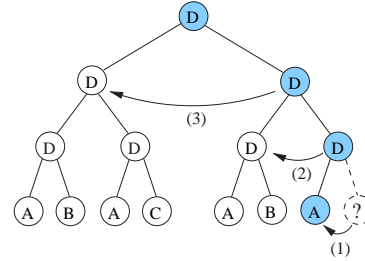


Figure 15: Computing the Range of a New Vertex for Extending a Subtree

range of allowable vertex labels for the new vertex (so that adding the new vertex will guarantee to result in a new DFCF), we can check each vertex along the path from the new vertex to the root. In Figure 15, 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, we know that at this given location, adding a new vertex with label greater than or equal to  $C$  will result in a  $t'$  similar to case IV in Figure 14 (and therefore the resulting  $t'$  is in  $B_{t\_right}$ ), and adding a new vertex with label less than  $C$  will result in a  $t'$  similar to case III in Figure 14 (and therefore the resulting  $t'$  is in  $B_{t\_left}$ ).

#### 4.4 Extending the Pruning and the Heuristic Techniques

With well-defined left-blanket and right-blanket, the pruning techniques developed in the previous section for labeled rooted ordered trees can be applied to labeled rooted unordered trees as well. This is because Lemma 6, Theorem 1, and Theorem 2 are still valid for the enumeration tree of unordered frequent subtrees. For example, in Figure 13,  $t_1$ ,  $t_3$ , and  $t_4$  are pruned by the left-blanket pruning;  $t_{13}$  and  $t_{14}$  are pruned by the right-blanket pruning.

Moreover, the right-blanket pruning for unordered trees can be more powerful than that for ordered trees:

**Theorem 3.** *For a frequent subtree  $t$  in the enumeration tree of frequent rooted unordered trees, if there exists  $t' \in B_{t\_right}$  such that  $t'$  and  $t$  are occurrence-matched, and the parent of  $t' \setminus t$  is  $v$  (where  $v$  is a vertex on the rightmost path of  $t$ ), then (1) we do not have to extend  $t$  by adding new rightmost vertices to any proper ancestors of  $v$ , and (2) we do not have to extend  $t$  by adding to  $v$  a new rightmost vertex with vertex label lexicographically greater than  $t' \setminus t$ .*

*Proof.* The first statement can be proved in the same way as Theorem 2. For the second statement, we notice that for any  $t'' \in B_{t\_right}$  such that the parent of  $t'' \setminus t$  is  $v$  and  $t'' \setminus t$  is lexicographically greater than  $t' \setminus t$ , we have that  $t' \setminus t$  is the left sibling of  $t'' \setminus t$  in *all* occurrences of  $t''$ , and therefore neither  $t''$  nor any of its descendants can be closed or maximal.  $\square$

For example, in Figure 13  $t_9$  and  $t_{10}$  are pruned by the right-blanket pruning, because that  $t_8$  and  $t_2$  are occurrence-matched. Here, although  $t_9 \setminus t_2$  and  $t_{10} \setminus t_2$  share the same parent (vertex  $C$ ) with  $t_8 \setminus t_2$ , they are still pruned because  $t_8 \setminus t_2 < t_9 \setminus t_2$  ( $B < D$ ) and  $t_8 \setminus t_2 < t_{10} \setminus t_2$  ( $B < E$ ).

Finally, the heuristic order of computation for reducing relatively expensive computation can be used for extending the enumeration tree of unordered frequent subtrees as well.

To summarize, with the definition of the canonical form (DFCF) and with only minor changes (e.g., the procedure to determine left and right blankets), the *CMTreeMiner* algorithm developed in the previous section can be used for mining frequent subtrees from databases of labeled rooted unordered trees.

## 5 Experiments

We performed extensive experiments to evaluate the performance of the *CMTreeMiner* algorithm using both synthetic datasets and datasets from real applications. All experiments were done on a 2GHz Intel Pentium IV PC with 1GB main memory, running RedHat Linux 7.3 operating system. All algorithms were implemented in C++ and compiled using the g++ 2.96 compiler.

### 5.1 Experiments on Rooted Ordered Trees

Since to our knowledge, there is no existing work on mining closed or maximal frequent ordered subtrees, we compare *CMTreeMiner* with the class of algorithms that use post-processing techniques to get closed or maximal frequent ordered trees, and use the cost of *FREQT*, which is developed by Asai et al. [2], as a lower bound for this class of algorithms. In the following experiments, we only consider the cost of *FREQT* for discovering all frequent ordered subtrees, and zero cost in the post-processing stage to obtain the closed or maximal ordered subtrees.

#### 5.1.1 The Synthetic Dataset vs. the Real Dataset

The synthetic dataset T1M is generated by the tree generation program provided by Zaki [30]. In brief, a mother tree is generated first with the following parameters: the number of distinct node labels  $N = 100$ , the total number of nodes in the tree  $M = 10,000$ , the maximal depth of the tree  $D = 10$  and the maximum fanout  $F = 10$ . The dataset is then generated by creating subtrees of the mother tree. In our experiments, we set the total number of trees in the dataset to be  $T = 1,000,000$ . The average number of nodes in each tree is 6.94.

The real dataset CSLOGS is provided by Zaki [30] and is composed of users' access trees to the CS department website at RPI. An access tree not only records the sequence of web pages that have been visited in a user session; but also integrates the website's topological information. It is possible that the labels in an access tree are not unique. That means the same page is visited multiple times in a user's access sequence. CSLOGS contains 59,691 trees that cover a total of 13,361 unique web pages. The average size of each tree is 12.

#### 5.1.2 Performance Comparison

Figure 16(a) shows the time spent by *CMTreeMiner* and *FREQT* under different support thresholds for the synthetic dataset T1M. The running time increase for *CMTreeMiner* is much slower than that for *FREQT* as the support threshold decreases. And when the support threshold is low, the trends of *CMTreeMiner* and *FREQT* become more distinct. The significant increase in the running time of *FREQT* at low support thresholds can be explained by the jump in the maximum

frequent subtree size in Figure 16(b). The increase of the maximum frequent subtree size leads to an exponential increase in the number of frequent subtrees, and finally the exponential increase in the running time. Since *CMTreeMiner* does not directly mine frequent subtrees, but the closed and maximal frequent subtrees, it is less affected than *FREQT*.

The above trends for *CMTreeMiner* and *FREQT* can also be observed in the real dataset CSLOGS (see Figure 17(a), 17(b)), only more pronounced. The maximum frequent subtree size increases dramatically from 28 to 204 when the support threshold drops from 0.2% to 0.1%, and it becomes difficult for *FREQT* to find all frequent subtrees at support threshold 0.17% or lower. On the other hand, *CMTreeMiner* can still handle such dramatic changes.

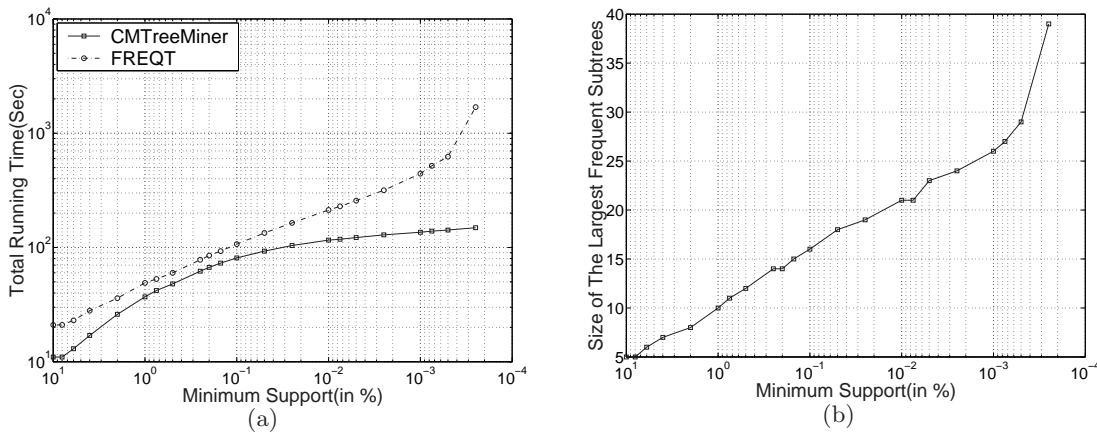


Figure 16: Support Thresholds vs. Running Time and the Maximum Size of Frequent Subtrees for T1M

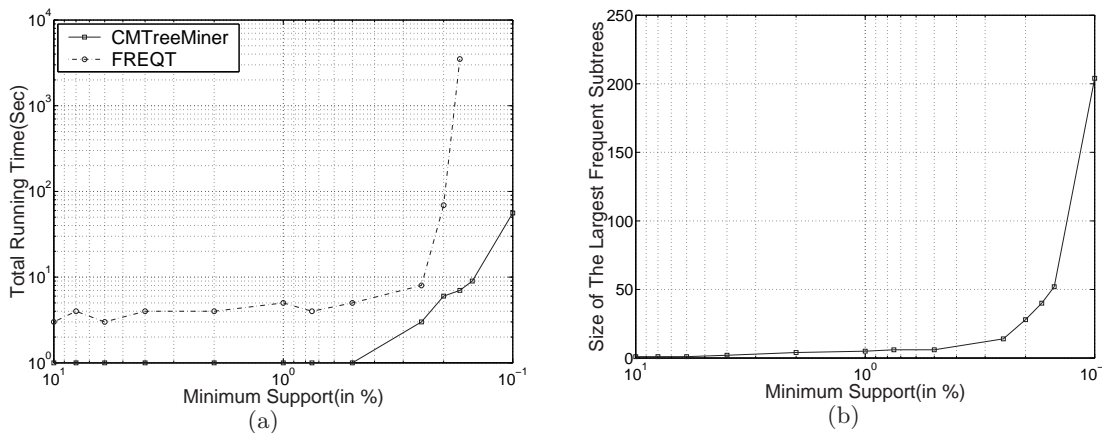


Figure 17: Support Thresholds vs. Running Time and the Maximum Size of Frequent Subtrees for CSLOGS

The time savings of *CMTreeMiner* come from the difference between the number of frequent subtrees and closed frequent subtrees. The larger the difference is, the more significant are the savings realized by *CMTreeMiner*. In fact, the savings are decided by the correlations among the frequent



subtrees. Strong correlations imply that the same set of frequent subtrees can be represented by a relatively small number of closed frequent subtrees. In real applications, strong correlations do exist, and it is a motivation of the frequent subtree mining algorithms to discover these strong correlations. Figure 18(a) and 18(b) show the number of frequent subtrees, the number of trees that have been checked by *CMTreeMiner*, the number of closed frequent subtrees, and the number of maximal frequent subtrees in dataset T1M and CSLOGS respectively, under different support thresholds. It can be easily seen that in both datasets, the number of closed frequent subtrees is significantly less than the total number of frequent subtrees, especially when the support threshold is low. The number of subtrees that have been checked by *CMTreeMiner* is also significantly less than the number of frequent subtrees at low support thresholds.

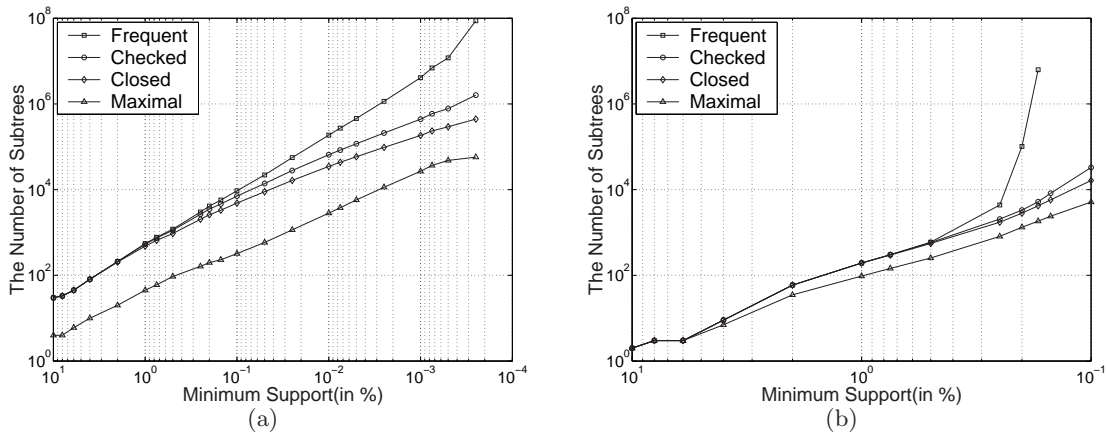


Figure 18: Support Thresholds vs. the Number of Frequent Subtrees for T1M and CSLOGS Respectively

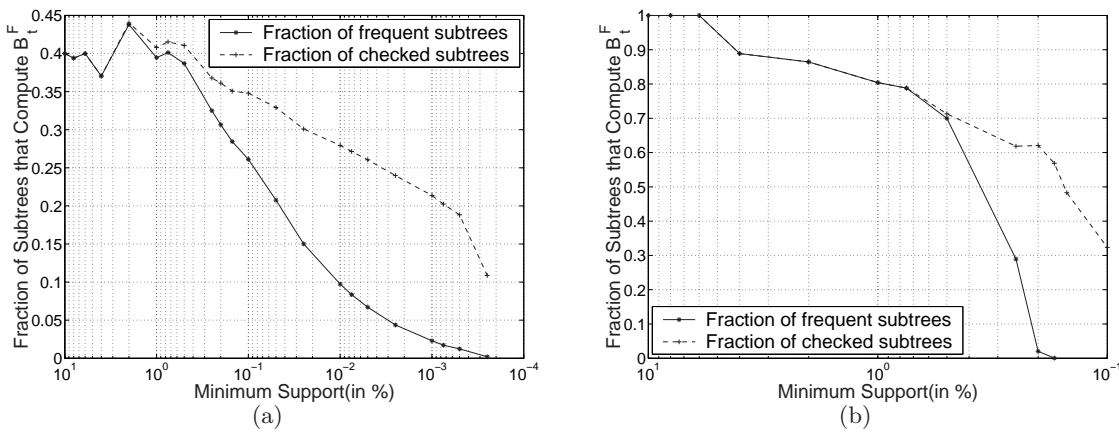


Figure 19: Support Thresholds vs. the Fraction of Subtrees that Compute  $B_t^F$  for T1M and CSLOGS Respectively

In *CMTreeMiner*, the most expensive operation is to compute the subset  $B_t^F$  of the blanket  $B_t$  for a frequent subtree  $t$ . Figure 19(a) shows the fraction of frequent subtrees for which this

operation is actually performed, and the fraction of closed subtrees in *CMTreeMiner* for which  $B_t^F$  is computed. As we can see, as the support threshold decreases, the fraction decreases also. Our understanding of this phenomena is that the pruning of non-closed subtrees occurs at places near the root of the enumeration tree. A similar trend can be observed in the dataset CSLOGS (see Figure 19(b)).

We also studied the memory usage of both *CMTreeMiner* and *FREQT*, and found that they are comparable. At the lowest support threshold 0.00025% for dataset T1M, the memory used by *CMTreeMiner* is 256M, while that by *FREQT* is around 254M. As for CSLOGS, at the support threshold 0.17%, the memory usages of *CMTreeMiner* and *FREQT* are 159M and 176M respectively.

## 5.2 Experiments on Rooted Unordered Trees

In this section, we study the performance of *CMTreeMiner* on mining frequent unordered subtrees. We first compare the performance of *CMTreeMiner* with that of *HybridTreeMiner*, an algorithm that we have previously developed for mining *all* frequent unordered subtrees [8]. For this comparison, we use the dataset that has been used in [8]. Second, we compare the performance of *CMTreeMiner* with that of *PathJoin* [26], a recently proposed algorithm that mines maximal frequent unordered subtrees. For this comparison, we use both a synthetic dataset and a real application dataset—the MBONE multicast data provided in [4, 5].

### 5.2.1 Comparison with HybridTreeMiner

*HybridTreeMiner* is an algorithm that we have previously developed for mining all frequent subtrees from a database of rooted unordered trees [8]. Here we use one of the datasets given in [8] to compare the performance of *CMTreeMiner* with that of *HybridTreeMiner*. The detailed procedure for generating the dataset is described in [8] and here we give a very brief description. A set of  $|N|$  ( $=100$ ) 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  $|D|$  ( $=10,000$ ) is the number of transactions in the database and  $|S|$  ( $=1\%$ ) 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  $|T|$  ( $=50$ ). 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  $|L|$  ( $=10$ ), which is both the number of distinct edge labels and the number of distinct vertex labels. The size of the seed trees  $|I|$  increases from 10 to 30. As can be seen, data generated in this way are strongly correlated. An advantage of this generator is that the size of maximal frequent subtrees is a parameter that can be controlled, so we can clearly observe the running time of various algorithms as a function of the size of maximal frequent subtrees.

Figure 20 shows the experimental results. Figure 20(a) gives the total number of frequent subtrees obtained by *HybridTreeMiner*, the number of subtrees checked by *CMTreeMiner*, the number of closed frequent subtrees, and the number of maximal frequent subtrees. As we can see from the figure, the total number of all frequent subtrees grows exponentially but the number of closed subtrees and maximal subtrees do not. The number of subtrees that are checked by *CMTreeMiner* grows in polynomial fashion. Therefore, as shown in Figure 20(b), as the size of

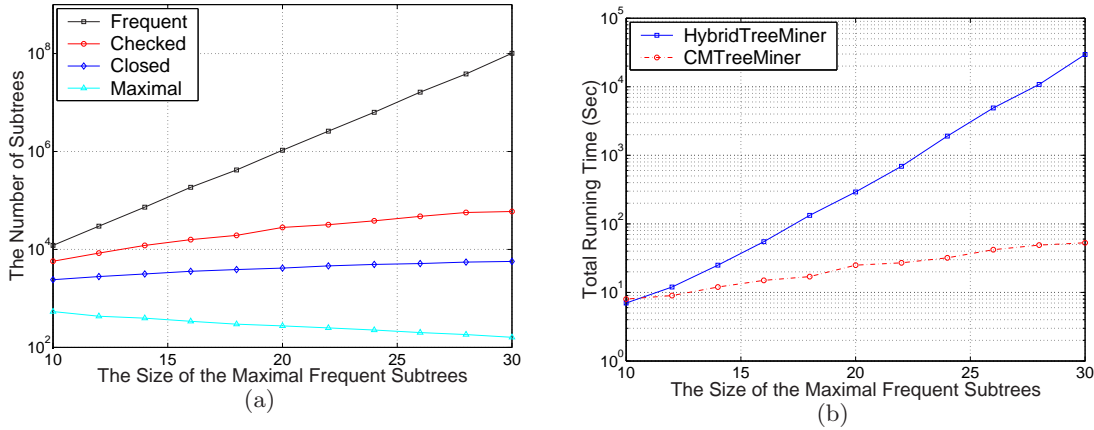


Figure 20: CMTreeMiner vs. HybridTreeMiner on Synthetic Datasets

maximal frequent subtrees increases, the total running time of *CMTreeMiner* grows much more slowly than that of *HybridTreeMiner*.

### 5.2.2 Comparison with PathJoin using Synthetic Data

*PathJoin* [26] is a recently proposed algorithm that mines maximal frequent subtrees from a database of labeled rooted unordered trees. However, 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. In addition, *PathJoin* assumes no edge labels. Therefore, we have changed the synthetic generator in [8] to generate a dataset that meets these requirements. The parameters for the dataset are:  $|D|=100,000$ ,  $|N|=90$ ,  $|L|=1,000$ ,  $|S|=1\%$ ,  $|T|=|I|$ , and  $|I|$  varies from 5 to 50. (For  $|I| > 25$ , *PathJoin* exhausts all available memory.)

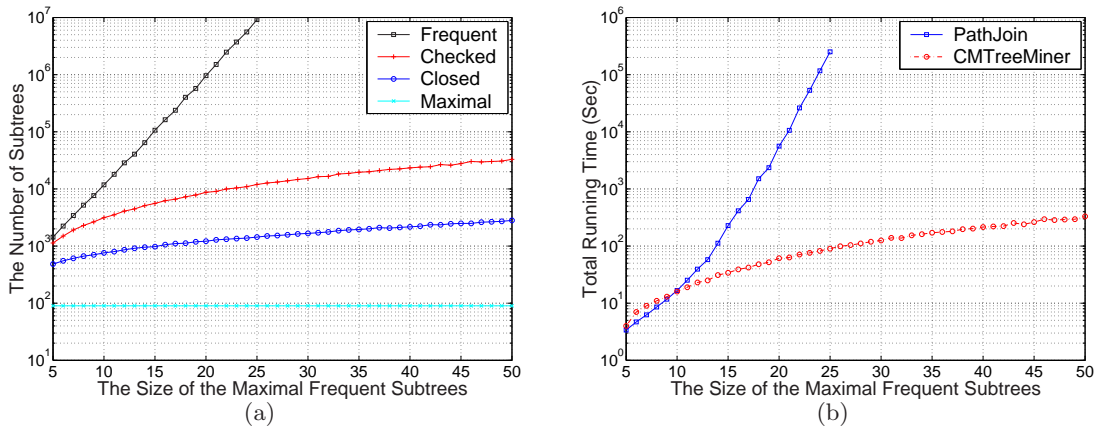


Figure 21: CMTreeMiner vs. PathJoin on Synthetic Datasets

Figure 21 compares the performance of *PathJoin* with that of *CMTreeMiner* on this dataset. Figure 21(a) gives the total number of 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 we can see from the figure, the number of subtrees that are 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 21(b), although *PathJoin* is very efficient for datasets with small tree sizes, as tree sizes increase beyond some value (say 10), it becomes obvious that *PathJoin* suffers from the exponential growth of computation time while *CMTreeMiner* does not. Note that the  $Y$ -axis is drawn on a logarithmic scale, and therefore the improvement correspond to orders of magnitude. For example, with the size of maximal frequent subtrees to be 25 in the dataset, it took *PathJoin* nearly 3 days to find all maximal frequent subtrees while it took *CMTreeMiner* only 90 seconds!

The memory usage of *CMTreeMiner* is also smaller than that of *PathJoin*. When *PathJoin* exhausts the 1GB available memory at  $|I| = 26$ , *CMTreeMiner* uses about 302 MB memory; when  $|I| = 50$ , *CMTreeMiner* uses about 567 MB memory.

### 5.2.3 Comparison with PathJoin using Data from Real Application

We also compared the performance of *CMTreeMiner* with that of *PathJoin* using a dataset of IP multicast trees. IP multicast is an efficient way to send messages to a group of users. In our experiment we have used the MBONE multicast data provided in [4, 5]. The data were measured during the NASA shuttle launch between 14th and 21st of February, 1999. It has 333 vertices where each vertex takes an IP address as its label. We sampled the data from this NASA dataset with 10 minutes sampling interval and got a dataset with 1,000 transactions. Therefore the transactions are the multicast trees for the same NASA event at different times.

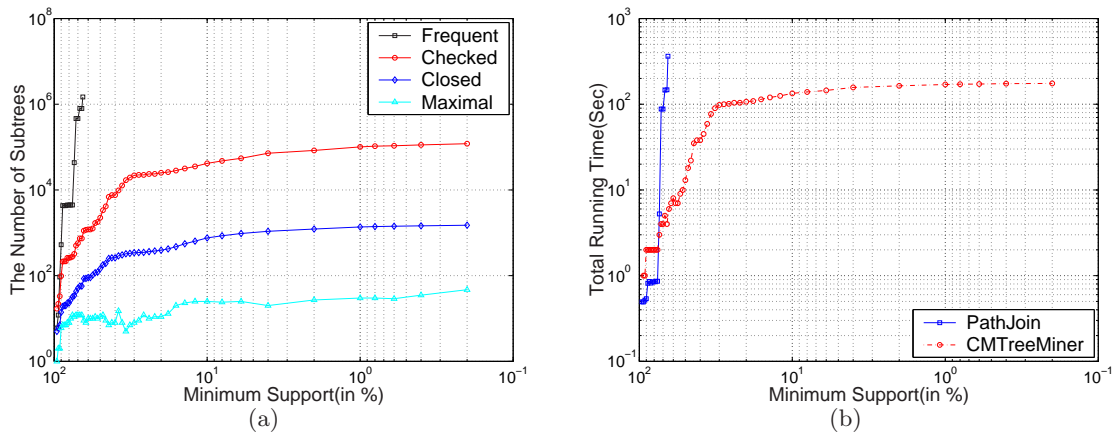


Figure 22: CMTreeMiner vs. PathJoin on the Multicast Dataset

Figure 22 compares the performance of *PathJoin* with that of *CMTreeMiner* on the multicast dataset. As shown in Figure 22(a), this dataset is extremely dense: as the support decreases, the total number of frequent subtrees dramatically increases. However, the number of subtrees checked by *CMTreeMiner*, the number of closed frequent subtrees, and the number of maximal frequent subtrees grow relatively slowly. Therefore, as we can see from Figure 22(b), even before *PathJoin*

uses up the memory, its running time blows up quickly. In contrast, *CMTreeMiner* can handle very low supports (e.g., 0.2%, which corresponds to a subtree occurring in only 2 transactions) within 180 seconds.

### 5.3 Observations Obtained from Mining Real Application Data

In this section, we study two datasets from real applications: the first one is the dataset of multicast trees that was introduced in the previous section; the second one is a dataset of Web access trees obtained from the log files at UCLA Data Mining Laboratory. Instead of focusing on the performance of algorithms, we study the maximal frequent trees mined from the datasets and look for previously unknown information from these frequent trees.

#### 5.3.1 The Dataset of Multicast Trees

For the multicast dataset that was introduced in the previous section, Figure 23 gives the size of the largest maximal frequent subtree as a function of the minimum support. Our previous attempts to mine all frequent subtrees from this dataset failed with support less than 80% [7] and now we know why. Because this dataset of NASA multicast trees has very strong correlation among its transactions (the difference between two consecutive transactions may result from a couple of users who left the event and a couple of users who just joined the event), the sizes of maximal frequent subtrees grow very large even with a very high support threshold. In addition, from Figure 23 we can see that as the minimum support decreases to values less than 30%, the size of the largest maximal frequent subtree does not change very much as the support threshold is decreased further. This suggests that first, not all the 333 participants of the NASA shuttle launch event participated at the same time (during the peak time, there were only about 240 participants), and second, a large portion of the whole multicast tree participated at least 30% of the period during the event.

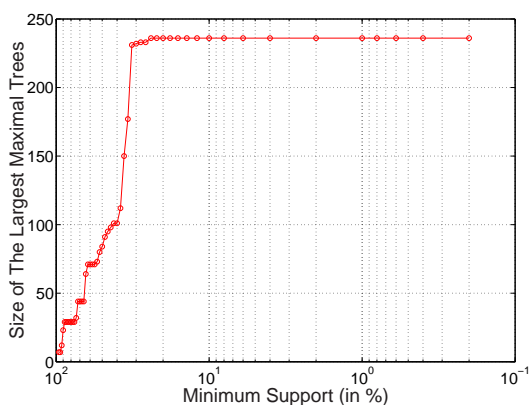


Figure 23: Multicast Dataset Results

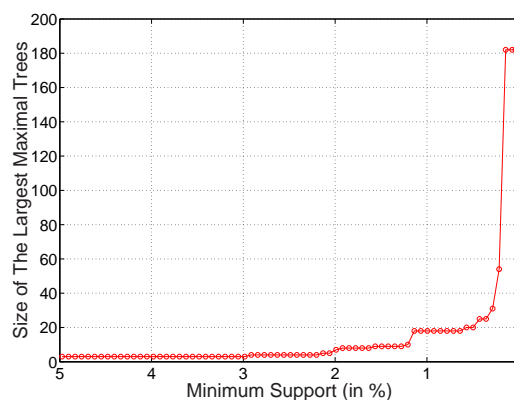


Figure 24: WebLog Dataset Results

#### 5.3.2 The Dataset of Web Access Trees

The second dataset is obtained from the log files at UCLA Data Mining Laboratory (<http://dml.cs.ucla.edu>). The transactions in this dataset do not reflect the navigation pattern of each user, as opposed to

the CSLOGS dataset; instead, each transaction records the fraction of the web site that is visited during a single session in the web log. First, we used the WWWPal system [21] to obtain the topology of the web site. Then we wrote a program that, for each session in the log file, generates a spanning tree that contains all the webpages visited in the session. Our program generated 2,814 user access trees from the log files collected over year 2003 at our laboratory that touched a total of 310 web pages. In the user access trees, the vertices correspond to the web pages and the edges correspond to the links between the webpages. Similar to CSLOGS, URLs of the webpages are taken as the vertex labels and therefore each vertex has a distinct label. We do not assign labels to edges.

In Figure 24 we have shown the growth of the size of the largest maximal frequent subtree as we reduce the support. As we can see, with larger support (from 1% to 5%), there are no large frequent subtrees. However, as the support decreases to extremely low value (lower than 0.2%), the size of the largest maximal frequent subtree jumps dramatically to very large values. Obviously, these large frequent subtrees with low support were created by web crawlers or robots as opposed of human web surfers. Therefore, Figure 24 tells us that among the visitors to our lab website, around 0.2% are web crawlers.

## 6 Conclusion and Future Directions

In this paper, we have introduced the problem of mining closed and maximal frequent subtrees from databases of rooted labeled trees. We presented a novel algorithm, *CMTreeMiner*, that efficiently mines closed and maximal frequent subtrees without first generating *all* frequent subtrees. Various pruning and heuristic techniques are used in the algorithm to reduce the search space and to improve the computational efficiency. Our algorithm can be used for mining both labeled rooted ordered trees and labeled rooted unordered trees. Extensive experiments showed that our algorithm outperformed all existing frequent subtree mining algorithms. The discovered closed and maximal frequent subtrees also revealed some interesting knowledge about the datasets that we have used.

We plan to extend our work in the following directions. First, our algorithm discovers frequent *induced* subtrees only. In real applications, frequent *embedded* subtrees are equally important. For example, in XML documents, not only the parent/child relationships, but also the ancestor/descendant relationships reveal important patterns in the documents, where the latter relationships can only be expressed using *embedded* subtrees. Therefore, mining closed and maximal frequent *embedded* subtrees is one of our future directions. Second, some trees in real applications are *unrooted*, i.e., they are *free trees*. For example, in pattern recognition, a free tree called *shape axis tree* is used to represent shapes [16]. Extending our algorithm to mining closed and maximal frequent free trees is another challenge. Third, frequent itemset mining has been extended to sequential pattern mining [23] and episode mining [18]. Many databases of labeled trees (e.g., the multicast trees and the web access trees used in our study) also have time-stamps for each transaction tree. Mining closed and maximal sequential patterns and episode trees from such databases is one of our future research topics.



## Acknowledgements

Thanks to Professor Mohammed J. Zaki at the Rensselaer Polytechnic Institute for providing us the CSLOGS dataset and the synthetic data generator. Thanks to Professor Yongqiao Xiao at the Georgia College and State University for providing the *PathJoin* source codes and offering a lot of help. Thanks to Professor Jun-Hong Cui at the University of Connecticut for providing the NASA multicast event data and offering many helpful suggestions. Thanks to Professor John R. Punin and Mukkai S. Krishnamoorthy at the Rensselaer Polytechnic Institute for helping us with the WWWPal system.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116, 0085773, and 9817773. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

- [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 *2nd SIAM Int. Conf. on Data Mining*, April 2002.
- [3] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *The 6th International Conference on Discovery Science*, October 2003.
- [4] 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.
- [5] R. Chalmers and K. Almeroth. On the topology of multicast trees. Technical Report, UCSB, March 2002.
- [6] Y. Chi, Y. Yang, and R. R. Muntz. Canonical forms for labeled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*. (to appear).
- [7] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)*, November 2003.
- [8] Y. Chi, Y. Yang, and R. R. Muntz. HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In *The 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, June 2004.
- [9] Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. CMTreeMiner: Mining both closed and maximal frequent subtrees. In *The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, May 2004.
- [10] J. Cui, J. Kim, D. Maggiorini, K. Boussetta, and M. Gerla. Aggregated multicast—a comparative study. In *Proceedings of IFIP Networking 2002*, May 2002.

- [11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, May 2000.
- [12] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71:153–169, 1996.
- [13] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proc. 2003 Int. Conf. on Data Mining (ICDM'03)*, 2003.
- [14] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, pages 13–23, September 2000.
- [15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, November 2001.
- [16] T. Liu and D. Geiger. Approximate tree matching and shape similarity. In *International Conference on Computer Vision*, September 1999.
- [17] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli. Exact rooted subtree matching in sublinear time. Technical Report TR-01-14, Universita Di Pisa, 2001.
- [18] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 1997.
- [19] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, 2003.
- [20] 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.
- [21] J. Punin and M. Krishnamoorthy. WWWPal system—a system for analysis and synthesis of web pages. In *WebNet 98 Conference*, November 1998.
- [22] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.
- [23] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology, EDBT'96*, 1996.
- [24] A. Termier, M-C. Rousset, and M. Sebag. TreeFinder: a first step towards xml data mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 450–457, 2002.
- [25] 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.
- [26] Y. Xiao, J-F Yao, Z. Li, and M. Dunham. Efficient data mining for maximal frequent subtrees. In *Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)*, November 2003.

- [27] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, 2002.
- [28] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.
- [29] L. H. Yang, M. L. Lee, W. Hsu, and S. Achary. Mining frequent quer patterns from xml queries. In *Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, 2003.
- [30] M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002.
- [31] 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 (SIGKDD'03)*, 2003.