Efficient Data Mining for Maximal Frequent Subtrees

Yongqiao Xiao, Jenq-Foung Yao
Dept. of Math & Computer Science
Georgia College and State University
Milledgeville, GA 31061
{yongqiao.xiao, jf.yao}@gcsu.edu

Zhigang Li, Margaret H. Dunham *
Dept. of Computer Science & Engineering
Southern Methodist University
Dallas, TX 75275
{zgli,mhd}@engr.smu.edu

Abstract

A new type of tree mining is defined in this paper, which uncovers maximal frequent induced subtrees from a database of unordered labeled trees. A novel algorithm, PathJoin, is proposed. The algorithm uses a compact data structure, FST-Forest, which compresses the trees and still keeps the original tree structure. PathJoin generates candidate subtrees by joining the frequent paths in FST-Forest. Such candidate subtree generation is localized and thus substantially reduces the number of candidate subtrees. Experiments with synthetic data sets show that the algorithm is effective and efficient.

1 Introduction

Data mining has evolved from association rule mining [1], sequence mining [2, 4, 10], to tree mining [12, 5] and graph mining [11, 8, 7, 9]. Association rule mining and sequence mining are one-dimensional structure mining, and tree mining and graph mining are two-dimensional or higher structure mining. The applications of tree mining arise from Web usage mining, mining semi-structured data, and bioinformatics, etc.

The focus of this paper is on a new type of tree mining. As a motivating example for this new type of tree mining, consider mining the Web logs at a particular Web site. Several types of traversal patterns have been proposed to analyze the browsing behavior of the user[4, 10]. One drawback of such one-dimensional traversal patterns for the Web logs is that the document structure of the Web site, which is is essentially hierarchical (a tree) or a graph, is not well captured.

In this paper, we uncover the maximal frequent subtree structures from the access sessions. The access sessions are regarded as trees instead of sequences. The trees are unordered, and the frequent subtrees are induced subtrees and maximal. Other contributions of the paper include: a compact data structure is used to compress the trees in the database, and at the same time the original tree structure is kept; and the proposed algorithm, PathJoin, uses a new candidate subtree generation method, which is localized to the children of a node in a tree and thus substantially reduces the number of candidate subtrees.

The rest of the paper is organized as follows. In Section 2 the tree mining problem is formally defined, and then the related work is described and compared to our work. Section 3 describes the compact data structure, and the PathJoin algorithm. Section 4 reports the experimental results. The last Section concludes the paper and points out the future work.

2 Problem Statement and Related Work

2.1 Problem Statement

A $tree$ is an acyclic connected directed graph. Formally, we denote a tree as $T =< N, B, r, L >$, where N is the set of nodes, B is the set of branches (directed edges), $r \in N$ is the root of the tree, and L is the set of labels on the nodes. For each branch $b =< n_1, n_2 >\in B$, where $n_1, n_2 \in N$, we call n_1 the parent of n_2, and n_2 a child of n_1. If the children of each node are orderd, the tree is called an *ordered tree*, otherwise, it is an *unordered tree*. On each node $n_i \in N$, there is a label $l_i \in L$. The labels in a tree could be unique, or duplicate labels are allowed for different nodes. Without loss of generality, the labels are represented by positive integers.

Paths and Root Paths A $path$ is a sequence of connected branches, i.e., $p =<< n_1, n_2 >, < n_2, n_3 >, \cdots, < n_{k-1}, n_k >>$, where $n_i \in N(1 \leq i \leq k)$, and k is the number of nodes on the path. For short, we represent the path just by the nodes on the path $< n_1, n_2, \cdots, n_k >$. A node x is called an ancestor of another node y if there exists

*This material is based upon work supported by the National Science Foundation under Grant No. IIS-9820841 and IIS-02808741.

a path starting from x to y, and accordingly y is called a descendent of x. A path starting from the root node is called a *root path*. Since there is only one root path to any node in the tree, each root path in a tree can be uniquely identified by the last node on the path. In Figure 1, node $n3$ represents the root path $< n1, n2, n3 >$, and the labels on the path are $< 1, 2, 3 >$.

Subtrees and Root Subtrees A tree $T' = < N', B', r', L' >$ is said to be a $subtree$ of another tree $T = < N, B, r, L >$, if and only if there exists a mapping $\theta : N' \rightarrow N$ such that (1) for each node $x \in N'$, $l(x) = l(\theta(x))$, where l is the labeling function, and (2) for each branch $b =< x, y > \in B'$, $< \theta(x), \theta(y) > \in B$. Such subtrees are called *induced subtrees* in [12]. The embedded subtrees defined in [12] allow the two nodes after mapping to be on the same path (ancestor/descendent relationship), that is, $< \theta(x), \cdots, \theta(y) >$ is a path in T. In this paper, a subtree is referred to as an induced subtree, unless otherwise indicated explicitly. If a tree T' is a subtree of another tree T, we also say that tree T contains T' or T' occurs in T. If a subtree has the same root as the tree, i.e., $r' = r$, the subtree is called a *root subtree*. Figure 1 shows a root subtree S of tree T.

Itemset Representation for Root Subtrees A root subtree T' of tree T can be uniquely identified by the corresponding nodes in T of the leaf nodes in T', i.e., a root subtree T' with k leaf nodes $\{y_1, y_2, \cdots, y_k\}$ can be represented by the set of nodes $\{x_1, x_2, \cdots, x_k\}$ in T, where $x_i = \theta(y_i)(1 \leq i \leq k)$. Such representation for root subtrees is called *itemset representation*, since the set of representative nodes for the subtree is similar to a k-itemset, where a k-itemset consists of k items as for association rules [1], and an item here corresponds to a representative node in the tree (i.e., the root path ending at the node). For the root subtree S of tree T in Figure 1, the itemset representation is $\{n3, n5, n7\}$.

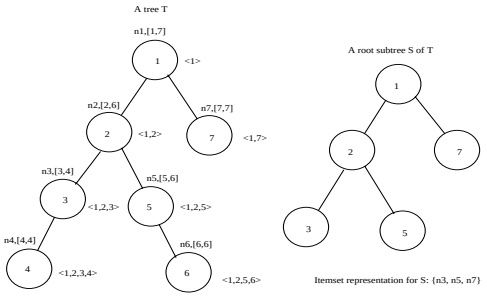


Figure 1. A Tree Example

Support and Maximal Frequent Subtrees Given a database of trees D, and a subtree S, the frequency of S in D, $freq_D(S)$, is the total number of occurrences of S in D, i.e., $freq_D(S) = \sum_{T \in D} freq_T(S)$, where $freq_T(S)$ is 1 if S occurs in T, otherwise 0. The $support$ of S in D, $sup_D(S)$, is the percentage of trees in D that contain S, i.e., $sup_D(S) = \frac{freq_D(S)}{|D|}$, where $|D|$ is the number of trees in D. Such definition excludes multiple occurrences of the subtree in a tree, thus it is called *unweighted support*. If the frequency of S includes every occurrence of S in every tree T in D, i.e., $FREQ_T(S)$ is n if S occurs n times in T, otherwise 0, the support can be defined as the ratio of the total frequency to the total size of the database (total number of nodes in all trees), i.e., $SUP(S) = \frac{\sum_{T \in D} FREQ_T(S)}{\sum_{T \in D} |T|}$, where $|T|$ is the number of nodes in $T (i.e., |N|)$. Such support is called *weighted support*, and it is similar to that in [10]. Weighted support and frequency are shown in upper case letters to distinguish from unweighted support and frequency. Given some support threshold s_{min} for unweighted or S_{min} for weighted, a subtree is said to be frequent, if the support for the subtree is not less than the threshold, i.e., $sup(X) \geq s_{min}$ for unweighted support or $SUP(X) \geq S_{min}$ for weighted support. A frequent subtree is maximal if it is not a subtree of another frequent subtree.

The Frequent Subtree Mining Problem Given a database of trees D, and some support threshold s_{min} or S_{min}, the objective is to find all maximal frequent subtrees in the database. The trees in D are assumed to be unordered trees, and the nodes in a tree could have duplicate labels, however, the labels for the children of every node are assumed to be unique. The support could be weighted or unweighted support.

2.2 Related Work

In [12], Zaki presented two algorithms, TreeMiner and PatternMatcher, for mining embedded subtrees from ordered labeled trees. PatternMatcher is a level-wise algorithm similar to Apriori [1] for mining association rules. TreeMiner performs a depth-first search for frequent subtrees, and uses the novel scope-list (a vertical representation for the trees in the database) for fast support counting. FREQT was proposed in [3] for mining labeled ordered trees. FREQT uses the notion of *rightmost expansion* to generate candidate trees by attaching new nodes only to the rightmost branch of a frequent subtree. The problem of discovering frequent substructures from hierarchical semi-structured data was proposed in [5]. It assumes that the hierarchical semi-structured objects are of the same type, e.g., XML documents of the same DTD schema. Thus it is not a general-purpose subtree mining problem.

Other recent work related to frequent subtree mining include mining frequent graph patterns [7, 8, 11]. Such graph

mining algorithms are likely to be too general for tree mining as pointed out in [12]. The one-dimensional traversal patterns for Web usage mining include [4, 10]. These one-dimensional traversal patterns do not capture well the document structure of the Web site.

The maximal frequent subtree mining problem proposed by us is different from others in that: (1) the uncovered subtrees are induced subtrees of the unordered labeled trees in the database; (2) the subtrees are maximal. The trees are assumed to be unordered, because we think when analyzing the user's browsing from a Web page (e.g., home page) on a Web server, it would be more interesting to know which pages the user follows from the starting page, regardless of the order of the access; The subtrees are induced subtrees, as argued in [4, 10] that such contiguity can help analyze the user's browsing behavior for Web usage mining. The maximality of subtrees can reduce the number of meaningful patterns. Notice that we do not intend to imply that other types of frequent subtrees are not or less important.

3 Algorithm PathJoin

3.1 Outline

We propose an efficient algorithm, PathJoin, for mining the maximal frequent subtrees. For easier presentation, it is initially assumed that there are no duplicate labels in the tree. Some extensions for handling duplicate labels are described in the last subsection.

The main idea of algorithm PathJoin is as follows: first all maximal frequent paths are found, then the frequent subtrees are mined by joining the frequent paths. These maximal frequent paths are special frequent subtrees (or 1-itemsets), and joining k maximal frequent paths results in subtrees with k leaf nodes (or k-itemsets). After all frequent subtrees are found by joining the maximal frequent paths, the frequent subtrees that are not maximal are pruned, so that we have the set of all maximal frequent subtrees.

One of the features of the PathJoin algorithm is the use of a new compact data structure, *FST-Forest (Frequent SubTree-Forest)*, to find the maximal frequent paths. FST-Forest consists of compressed trees, representing the trees in the database (with infrequent 1-itemsets pruned). The idea of compressing the database was inspired by the earlier work [6] in mining association rules. In this paper, the compact structure is used to facilitate finding the maximal frequent paths, and with additional features it is also used in algorithm PathJoin for mining frequent subtrees. FST-Forest reduces the overall space requirements significantly (in most cases) due to the overlap among trees.

The PathJoin algorithm is outlined in Algorithm 1. There are only two database scans. In the first scan, the frequent size 1 subtrees (with one node) are found. In the second scan, the trees in the database are trimmed with only frequent nodes left, and then merged into the compact structure, FST-Forest, which is done by function constructCompressedForest. After compressing all trees in the database to FST-Forest, the maximal frequent (root) paths are mined in each tree in FST-Forest, and then the frequent (root) subtrees are mined by joining the frequent paths. Finally, the subtrees that not maximal are pruned.

Algorithm 1 *PathJoin*

Input:
D*: database of trees.*
s_{min} *or* S_{min}*: unweighted/weighted support threshold.*
Output:
$MFST$*: all maximal frequent subtrees*
Method:
// First database scan to find frequent 1-itemsets
(1) $minsup = |D| * s_{min}$
or $minsup = (\sum_{T \in D} |T|) * S_{min}$
(2) $F_1 = \{$ *frequent size 1 subtrees*$\}$*;*
// Second database scan to trim trees and create FST-Forest
(3) $Forest = constructCompressedForest(D, F_1)$*;*
(4) $FST = \emptyset$*;*
(5) **for** *each tree* $T \in Forest.trees$ **do begin**
// Find the maximal frequent root paths in T
(6) $computeMFP(T, Forest, minsup)$*;*
// Find the frequent root subtrees in T
(7) $computeFST(\{T.root\}, \emptyset, minsup, FST)$*;*
(8) **endfor**
// Find the maximal frequent subtrees
(9) $MFST = maximize(Forest, FST)$*;*
(10) return MFST;

3.2 Compressed Tree Construction

The compact structure, FST-Forest(or $Forest$ for short), consists of compressed trees. The construction of Forest is a three step process: identify frequent subtrees with only one node, trim the original trees in the database by removing the infrequent nodes, create Forest by appropriately merging these trimmed trees. Prior to explaining Forest creation process, we examine the Forest data structure in more detail.

The compressed trees in Forest are indexed by the root label of each tree. For each compressed tree, there is a nodelist for each label, which links together all nodes in the tree with the same label. Notice that nodelist is called header table in [6], and header table links all nodes in the entire forest, while in our structure, the nodelist is distributed to each tree in Forest to reduce overall main memory requirement during subtree mining. Figure 2 shows the FST-

Forest structure. Each node in the forest is stored using the basic Node structure shown.

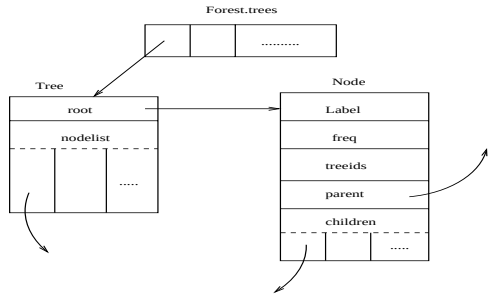


Figure 2. FST-Forest Data Structure

The treeids field in the node structure is new, and is required to reconstruct the original tree structure after compression. For a node in Forest, the treeids keeps the ids of all original trees (before being compressed to Forest) which have a root path ending at the node. To reduce the main memory requirement, the tree ids only need be saved on the leaf nodes of the original trimmed tree, since the tree ids can be merged upward(i.e., to ancestors) later in mining frequent subtrees.

To construct the Forest, each tree string (i.e., the labels by an depth-first traversal over the tree with -1 for each backard traversal as in [12]) in the database is scanned to create a tree in main memory. Then the infrequent nodes (those not in F_1) are removed from the tree yielding the trimmed trees. When creating the trimmed database, a tree in the original database may become be disconnected (like a forest). If so, each connected subtree is treated as an independent tree (trimmed tree) but with the same tree id as the original tree. For the example database and an unweighted support of 50% there are only two subtrees of size one which are small: 8 and 9. When these are removed from the database we obtain the trimmed database. Figure 3 shows the original trees (upper part) and the trimmed trees (lower part). There are five trimmed trees. Two of them are the same as original trees: $T1$ and $T4$. Three are modified from the original ones: $T2a$, $T2b$, and $T3a$.

The creation of the Forest is performed by either inserting (if no such tree with the same root label exists in Forest) or merging (a tree with the same root label already exists in Forest) these trimmed trees into Forest. In the first case, the new nodes from the trimmed tree are inserted directly to Forest with the tree structure kept the same, and the nodelist updated to include the new nodes. In the second case, when a trimmed tree has common nodes with an existing tree in Forest, the corresponding freq fields in Forest are incremented, and the new tree id is appended to the treeids if the node in the subtree is a leaf. After merging all the trimmed trees, the resulting Forest is shown in Figure 4.

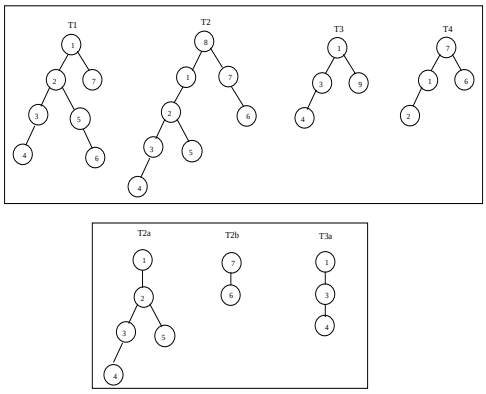


Figure 3. Original Trees vs. Trimmed Trees

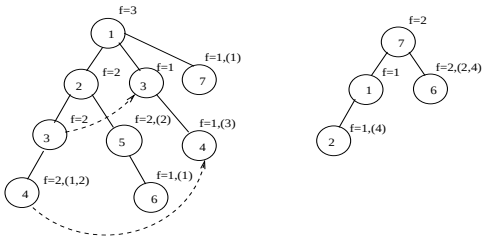


Figure 4. Compressed FST-Forest

Since the ids of the trees in the database are numbered sequentially, the treeids field on each node will be ordered after Forest is constructed. This automatic ordering is very useful in subtree expanding described in the following subsection. The treeids on the nodes are shown in the parentheses in Figure 4.

For each tree string in the database, the time for constructing the corresponding tree in main memory is linear to the string length. Determining connected subtrees with only frequent nodes can be accomplished by a breath-first traversal of the tree, which requires $O(n)$ time. Merging a subtree into Forest is like a depth-first traversal, which also requires $O(n)$ time. Overall, constructing Forest with all trees in the database needs time linear to the total number of nodes in all trees.

The space requirement for the compact structure depends on the structures of the trees in the database. In the worst case when there are no common nodes among the trees, i.e., each node is unique, there is no compression in Forest and the required memory is as large as the database. Fortunately,

there are usually a lot of common nodes among the trees (the common structures of the trees are what we are trying to discover!), and thus it results in a compact Forest as shown by the experiments. Since the tree ids are stored only in the leaf nodes of the original tree (may not be leaf nodes in Forest due to merging), the memory for storing the tree ids is reduced, especially for deep trees.

3.3 Maximal Frequent Path Mining

The constructed Forest contains all paths found in the database which could possibly be frequent. However, not every path exists as a root path in Forest. For example, the path $< 2, 3 >$ is found in Figure 4, but it is not a root path. To faciliate the counting of all paths, we expand Forest to ensure that all paths from the original database are now root paths.

For those non-root paths with the same starting label X as the root label of a tree in Forest, we need to expand them to the tree. Actually all such non-root paths are linked by $nodelist[X]$ in each tree. Each node in the nodelist can be viewed as a subtree rooted at the node, and the paths starting from the node (non-root) in the subtree should be merged with the root paths of tree with root label X. The treeids and frequency of these subtrees can be directly merged to the tree, because (1) the subtrees are disjoint (i.e., they occur in different original trees in the database); (2) for a tree rooted at X in Forest, a subtree of X rooted at node Y, and a non-root path starting from Y to node Z, the frequency of the path $< Y, \cdots, Z >$ in the tree rooted at X is the same as that of the root path $< X, \cdots, Y, \cdots, Z >$. Since tree ids are automatically ordered during Forest construction, the merging of two sets of tree ids can be done in time linear to the sum of the length of the two sets.

After expanding the all subtrees rooted at nodes with label 1, the tree with root label 1 is shown in Figure 5 (left part). Other trees in the Forest are not shown since they are not relevant while mining the tree.

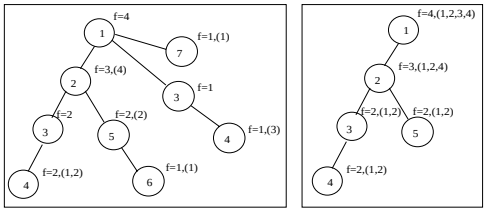


Figure 5. Subtree Expansion and MFP

As a result of expanding non-root paths (subtrees) from other trees, the tree that is being mined in Forest has all paths in the original database which start with the root node. Then the maximal frequent root paths can be found by a depth-first traversal of the tree. During the traversal for mining the maximal frequent paths, the infrequent nodes and their descendents are all removed from the tree after their treeids are merged to the closest ancestor nodes that are frequent. Such removal of infrequent nodes is valid because of the downward closure property of subtrees. That is, all subtrees of frequent trees are frequent, or equivalently, all supertrees of an infrequent tree are infrequent.

At the end of such depth-first traversal of the tree, each leaf node left in the tree corresponds to a maximal frequent path. To facilitate the subsequent frequent subtree mining, the treeids are merged upward (from descendents to ancestors) by a post-order traversal of the tree. The resulting Forest after computing the maximal frequent paths in the tree with root label 1 is shown in Figure 5 (right part). The root paths in the tree with root label 1 are all frequent. The paths terminating at the leaf nodes are maximal.

The time complexity for computing the maximal frequent paths in a tree is $O(mn + ln)$, where m and l are the total number of nodes in the subtrees that are expanded, and the number of nodes in the tree that is being mined, respectively, and n is the maximum number of tree ids on the nodes in the tree after expanding.

3.4 Frequent SubTree Mining

After expanding all subtrees with the same root label, all frequent root subtrees can be mined from the tree as shown by Theorem 1.

Theorem 1 *Given a tree T with all frequent root paths, let the label of the root node be R. All frequent subtrees with root label R are root subtrees of T.*

Proof. Straightforward using the downward closure property of frequent subtrees.

The itemset representation for root subtrees is used in the following description, i.e., each root subtree S of tree T is represented by the representative nodes in T of the leaf nodes of S. A root subtree with k leaf nodes(or k root paths) is a k-itemset, and each root path is a 1-itemset.

The main idea is to construct candidate k-itemsets by joining k 1-itemsets. For example, in Figure 5b), by joining two 1-itemsets, $n3$ and $n5$, we have a candidate 2-itemset $\{n3, n5\}$. The frequency of a candidate k-itemset is the number of common tree ids on the k 1-itemsets. The set of common tree ids of k 1-itemsets is the intersection of the tree ids of the k 1-itemset. Such k-way intersection of tree ids can be done in linear time, since the treeids are ordered on each node.

To reduce the number of candidate itemsets, the downward closure property is used to generate candidate k-

itemsets from frequent $(k-1)$-itemsets, that is, a candidate k-itemset is generated only if all its subsets ($(k-1)$-itemsets) are frequent. The candidate generation is done by function $FST\_PathJoin$. Function $FST\_PathJoin$ is similar to function $apriori\_gen$ in [1]. For two itemsets $\{r_1, r_2, \cdots, r_{k-1}\}$ and $\{s_1, s_2, \cdots, s_{k-1}\}$, a candidate k-itemset $\{r_1, r_2, \cdots, r_{k-2}, r_{k-1}, s_{k-1}\}$ is generated from them if $r_i = s_i (1 \leq i \leq k-2)$ and all subsets of it are frequent. Notice that function $FST\_PathJoin$ is applied to the child nodes of a node in the tree recursively, thus it is localized, while $apriori\_gen$ is applied to all frequent itemsets.

Function 1 *computeFST($isetRecur, isetFixed, s, FST$)*

$isetRecur$: itemset called recursively on its child nodes,
$isetFixed$: itemset fixed during recursion,
s: minimum support count,
FST: the resulting set of frequent subtrees,
$output$: the set of frequent child nodes of $itemsetRecur$.

(1) $itemset = itemsetRecur \cup itemsetFixed$;
// Frequency of an itemset is the number of common tree ids
(2) $itemset.treeids = \cap_{X \in itemset} X.treeids$;
(3) **if** *itemset is not frequent* **then**
(4) **return** $\emptyset$;
(5) $FST = FST \cup \{itemset\}$;
// Find the frequent child nodes in $itemsetRecur$
(6) $FST_1 = \emptyset$;
(7) **for** *each child C of $X \in itemsetRecur$* **do begin**
(8) $fx = itemset - \{X\}$;
(9) $re = \{C\}$;
(10) $RES = computeFST(re, fx, s, FST)$;
(11) **if** $RES \neq \emptyset$ **then**;
(12) $FST_1 = FST_1 \cup \{C\}$;
(13) **endfor**
// Generate candidates with two frequent child nodes
(14) $CFT_2 = FST\_PathJoin(FST_1)$;
(15) **for** $(k = 2; CFT_k \neq \emptyset; k++)$ **do begin**
(16) $FST_k = \emptyset$;
(17) **for** *each* $X \in CFT_k$ **do begin**
(18) $fx = itemset - \{Y.parent | Y \in X\}$;
(19) $RES = computeFST(X, fx, s, FST)$;
(20) **if** $RES \neq \emptyset$ **then**;
(21) $FST_k = FST_k \cup \{X\}$;
(22) **endfor**
// Generate candidates with $k+1$ frequent child nodes
(23) $CFT_{k+1} = FST\_PathJoin(FST_k)$;
(24) **endfor**
(25) **return** FST_1;

The time complexity of the function depends on the number of frequent subtrees. For each candidate k-itemset, its frequency checking is done in time $O(kn)$, where n is the maximum number of tree ids on the nodes. The cost of function $FST\_PathJoin$ is kept minimal, since it is applied to the child nodes of a node in the tree only, that is, it is localized to a node. Such localization reduces the number of candidate subtrees substantially as shown in the experiments. There is no extra memory requirement for the frequency checking of a candidate itemset besides $1n$ for the k-way intersection.

3.5 Maximizing

There are two steps in finding the maximal frequent subtrees from the set of frequent subtrees: (1) local maximization in the tree and (2) global maximization in Forest. The local maximization is done on each tree for the frequent subtrees with the same root label. Global maximization filters those that are subtrees (non-root) of another frequent subtree. The two types of maximizations are relatively straightforward, thus the details are omitted.

After local maximization, we have maximal frequent subtrees with the same root label. Such maximal frequent subtrees within a tree in Forest could be interesting themselves. After global maximization, we have the set of all maximal frequent subtrees. For Web usage mining, these maximal frequent subtrees provide a global view of the entire Web site.

3.6 Handling Duplicate Labels

So far we have assumed that there are no duplicate labels in the trees of the database. The PathJoin algorithm can be modified a little bit to support duplicate labels.

Recall that when compressing a tree in the database into Forest, we first get the connected subtrees of the tree with only frequent nodes and each connected subtree has the same tree id as the original tree. With duplicate labels, each connected subtree with the same root label is assigned a pair of tree ids: one is the original tree id and the other is a new tree id. The pair of tree ids are added to all nodes of each connected subtree with the same root label. Notice that for these connected subtrees with unique root labels, only the original tree id is necessary (or the new tree id is the same as the original tree id). When merging a connected subtree into Forest, the pair of tree ids are also merged. For the non-root nodes with duplicate labels in a connected subtree, a pair of tree ids (original plus new) are also added the nodes and all their descendents. These pairs of tree ids are merged when expanding subtrees in function $computeMFP$.

The new tree ids are needed to prevent invalid subtree generation. For the trees in Figure 3, if $T2$ is replaced by $T2'$ in Figure 6, the two connected subtrees have the same root label 2. In Figure 6, the pair of tree ids are shown as

$x : y$ in parentheses on each node, where x and y are the original and new tree ids respectively. After merging, the root node has 3 children. Without the new tree ids, it could generate an invalid subtree, e.g., 2 1 -1 3 -1, since children with labels 1 and 3 do not appear to be the children of the node with label 2 at the same time. By checking the new tree ids, such invalid subtrees are avoided, since the two children have different new tree ids.

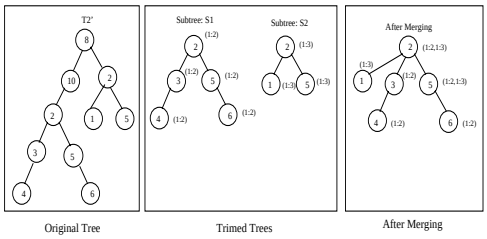


Figure 6. Handling Duplicate Labels

The frequency counting is changed as follows: for weighted support, the frequency at each node is the number of new tree ids, and for unweighted support, the frequency is the number of original tree ids.

4 Experimental Results

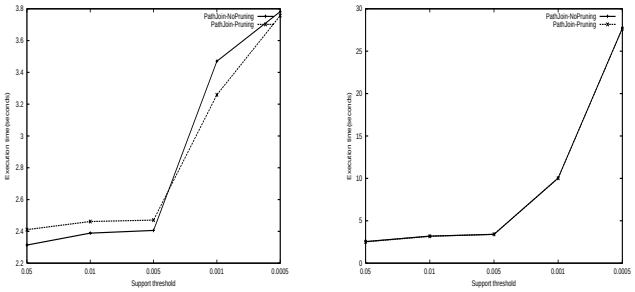
The performance of PathJoin was examined through a series of simulation experiments. All experiments were conducted on a Sun Blade 1000 with 1GB main memory and running Sun OS 5.8. The algorithm was implemented in C++ using Standard Template Library.

Three synthetic data sets, D10, F5 and T1M, were tested. These data sets were generated using the method in [12]. The synthetic data generation mimicks the Web site browsing behavior of the user. The parameters used in the data generation include the number of labels $N = 100$, the number of nodes in the master tree $M = 10000$, the maximum fanout of a node in the master tree $F = 10$($F = 5$ for data set F5), and the maximum depth of the master tree $D = 10$, and the total number of trees in the data set $T = 100000$($T = 1000000$ for T1M).

Two variations of the PathJoin algorithm were compared to examine the effect of pruning in candidate subtree generation: one uses pruning in candidate subtree generation (i.e., function $FST\_PathJoin$ checks whether the subsets of a candidate itemset are frequent) and the other does not. We did not compare our algorithm to others because of the difference of the problem defined by us from others.

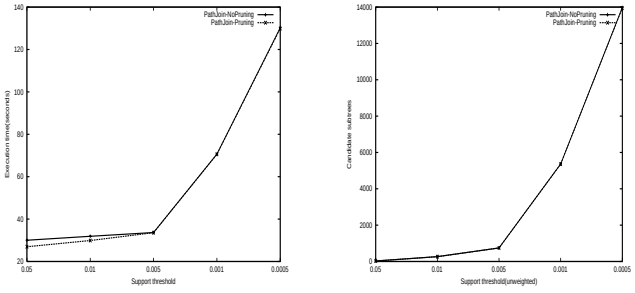
4.1 Execution Time and Number of Candidate Subtrees

The execution time for the three data sets with varying minimum support is shown in Figures 7 and 8a). The execution time increases as the minimum support decreases, since there are more frequent subtrees with smaller minimum support. It can be also seen that the two variations of the algorithm (with or without pruning in candidate subtree generation) have no big difference in execution time. This is because the pruning does not prune away many candidate subtrees as shown in Figure 8b). Thus the overhead for checking the subsets of the candidate subtrees offsets the saved time for frequency counting of the pruned candidate subtrees. Similar results were obtained for the other two data sets, which are not shown due to space limit. The reason that the pruning is not very effective is that the candidate subtree generation in function $computeFST$ is limited to the children of a node in a tree in Forest, i.e., it is localized. Such localization reduces the number of candidate subtrees compared to apriori_gen in [1], which is applied to all the frequent itemsets found in a pass.



a)Data set F5 b)Data set D10

Figure 7. Execution Time

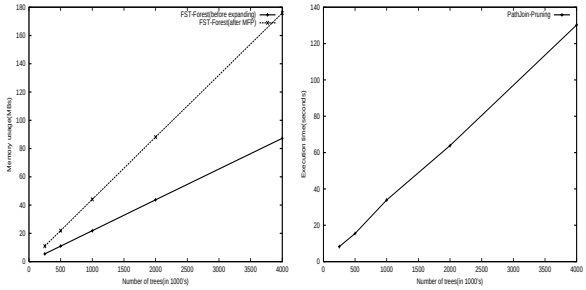


a)Execution Time b)Number of Candidate Subtrees

Figure 8. Data set T1M

4.2 Memory Usage and Scaleup

The memory usage for the compact data structure, FST-Forest, has two parts: one for the compressed tree structure, the other for the tree ids. The memory for the compressed tree structure is fixed for a data set given some minimum support threshold, while the memory for the tree ids will grow as the number of trees in the data set increases. In the experiments, the data sets with different number of trees were generated with the same parameters as for data set T1M. The minimum support was fixed to 0.5%. Figure 9a) shows the memory usage at two stages of the algorithm: the lower curve shows the the memory before expanding all subtrees with the same label, and the upper curve after computing the maximal frequent paths and merging the treeids upward. It can be seen that the memory usage is about doubled after expansion and merging. Figure 9 also shows that both the memory usage and the execution time scales linearly with respect to the change of the number of trees in the data set.



a) Memory usage b) Scaleup

Figure 9. Memory Usage and Scaleup

5 Conclusion

A new type of tree mining (maximal induced subtrees in unordered trees) is defined in the paper. A novel algorithm, PathJoin, is proposed to discover all maximal frequent subtrees given some minimum support threshold. The algorithm uses a compact data structure, FST-Forest, to compress the trees in the database and at the same time still keeps the original tree structure. A localized candidate subtree generation method is used in the algorithm, which reduces the number of candidate subtrees substantially. The algorithm is evaluated with synthetic data sets.

The future work includes: (1) earlier identification of maximal frequent subtrees, which could potentially save a lot of time for computing the non-maximal frequent subtrees. (2) extension of FST-Forest for mining maximal frequent embedded subtrees, which allow ancestor/descendent relationship.

Acknowledgement

We would like to thank Prof. Mohammed J. Zaki for sending us the source code for the tree generation program.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, Mar. 1995. IEEE Computer Society Press.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficiently substructure discovery from large semi-structured data. In *Proceedings of the 2nd SIAM Int'l Conference on Data Mining*, april 2002.

[4] M.-S. Chen, J. S. Park, and P. S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.

[5] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proceedings of the 2nd SIAM Int'l Conference on Data Mining*, Arlington, VA, april 2002.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD Conference*, 2000.

[7] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Knowledge Discovery and Data Mining*, sep 2000.

[8] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 1st IEEE Int'l Conference on Data Mining*, nov 2001.

[9] D. Shasha, J. Wang, and R. Giugno. Algorithms and applications of tree and graph searching. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 39–52, Madison, Wisconsin, june 2002.

[10] Y. Xiao and M. H. Dunham. Efficient mining of traversal patterns. *Data and Knowledge Engineering*, 39:191–214, 2001.

[11] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, pages 721–724. IEEE Computer Society, 2002.

[12] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, Edmonton, Canada, jul 2002.