Efficiently Mining Frequent Trees in a Forest

Mohammed J. Zaki [*]
Computer Science Department, Rensselaer Polytechnic Institute, Troy NY 12180
zaki@cs.rpi.edu, http://www.cs.rpi.edu/~zaki

ABSTRACT

Mining frequent trees is very useful in domains like bioinformatics, web mining, mining semi-structured data, and so on. We formulate the problem of mining (embedded) subtrees in a forest of rooted, labeled, and ordered trees. We present TreeMiner, a novel algorithm to discover all frequent subtrees in a forest, using a new data structure called scope-list. We contrast TreeMiner with a pattern matching tree mining algorithm (PatternMatcher). We conduct detailed experiments to test the performance and scalability of these methods. We find that TreeMiner outperforms the pattern matching approach by a factor of 4 to 20, and has good scaleup properties. We also present an application of tree mining to analyze real web logs for usage patterns.

1. INTRODUCTION

Frequent Structure Mining (FSM) refers to an important class of exploratory mining tasks, namely those dealing with extracting patterns in massive databases representing complex interactions between entities. FSM not only encompasses mining techniques like associations [3] and sequences [4], but it also generalizes to more complex patterns like frequent trees and graphs [12, 14]. Such patterns typically arise in applications like bioinformatics, web mining, mining semi-structured documents, and so on. As one increases the complexity of the structures to be discovered, one extracts more informative patterns; we are specifically interested in mining tree-like patterns.

As a motivating example for tree mining, consider the web usage mining [17] problem. Given a database of web access logs at a popular site, one can perform several mining tasks. The simplest is to ignore all link information from the logs, and to mine only the frequent sets of pages accessed by users. The next step can be to form for each user the sequence of links they followed, and to mine the most frequent user access paths. It is also possible to look at the entire forward accesses of a user, and to mine the most frequently accessed subtrees at that site.

[*]This work was supported in part by NSF CAREER Award IIS-0092978, and NSF Next Generation Software Program grant EIA-0103708.

Appeared in Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD), Edmonton, Canada, July 2002.

In recent years, XML has become a popular way of storing many data sets because the semi-structured nature of XML allows the modeling of a wide variety of databases as XML documents. XML data thus forms an important data mining domain, and it is valuable to develop techniques that can extract patterns from such data. Tree structured XML documents are the most widely occurring in real applications. Given a set of such XML documents, one would like to discover the commonly occurring subtrees that appear in the collection.

Tree patterns also arise in Bioinformatics. For example, researchers have collected vast amounts of RNA structures, which are essentially trees. To get information about a newly sequenced RNA, they compare it with known RNA structures, looking for common topological patterns, which provide important clues to the function of the RNA [19].

In this paper we introduce TreeMiner, an efficient algorithm for the problem of mining frequent subtrees in a forest (the database). The key contributions of our work are as follows: 1) We introduce the problem of mining *embedded* subtrees in a collection of rooted, ordered, and labeled trees. 2) We use the notion of a *scope* for a node in a tree. We show how any tree can be represented as a list of its node scopes, in a novel vertical format called *scope-list*. 3) We develop a framework for non-redundant candidate subtree generation, i.e., we propose a systematic search of the possibly frequent subtrees, such that no pattern is generated more than once. 4) We show how one can efficiently compute the frequency of a candidate tree by joining the scope-lists of its subtrees. 5) Our formulation allows one to discover all subtrees in a forest, as well as all subtrees in a single large tree. Furthermore, simple modifications also allow us to mine unlabeled subtrees, unordered subtrees and also frequent sub-forests (i.e., disconnected subtrees).

We contrast TreeMiner with a base tree mining algorithm based on pattern matching, PatternMatcher. Our experiments on several synthetic and one real dataset show that TreeMiner outperforms PatternMatcher by a factor of 4 to 20. Both algorithms exhibit linear scaleup with increasing number of trees in the database. We also present an application study of tree mining in web usage mining. The input data is in the form of XML documents that represent user-session extracted from raw web logs. We show that the mined tree patterns indeed do capture more interesting relationships than frequent sets or sequences.

2. PROBLEM STATEMENT

A *tree* is an acyclic connected graph, and a *forest* is an acyclic graph. A forest is thus a collection of trees, where each tree is a connected component of the forest. A *rooted tree* is a tree in which one of the vertices is distinguished from others, and called the *root*. We refer to a vertex of a rooted tree as a *node* of the tree. An *ordered tree* is a rooted tree in which the children of each node are ordered, i.e., if a node has k children, then we can designate them as the first child, second child, and so on up to the kth child. A *labeled tree* is a tree where each node of the tree is associated with a label. In this paper, all trees we consider are ordered, labeled, and rooted trees. We choose to focus on labeled rooted trees, since those are the types of datasets that are most common in a data mining setting, i.e., datasets represent relationships between items or attributes that are named, and there is a top root element (e.g., the main web page on a site). In fact, if we treat each node as having the same label, we can mine all ordered, unlabeled subtrees as well!

Ancestors and Descendants Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an *ancestor* of x, and is denoted as $y \leq_l x$, where l is the length of the path from y to x. If y is an ancestor of x, then x is a *descendant* of y. (Every node is both an ancestor and descendant of itself). If $y \leq_1 x$ (i.e., y is an immediate ancestor), then y is called the *parent* of x, and x the *child* of y. We say that nodes x and y are *siblings* if they have the same parent, and we say they are *embedded siblings* if they have some common ancestor.

Node Numbers and Labels We denote a tree as $T = (N, B)$, where N is the set of labeled nodes, and B the set of *branches*. The *size* of T, denoted $|T|$, is the number of nodes in T. Each node has a well-defined *number*, i, according to its position in a depth-first (or pre-order) traversal of the tree. We use the notation n_i to refer to the ith node according to the numbering scheme ($i = 0 \ldots |T| - 1$). The *label* (also referred to as an *item*) of each node is taken from a set of labels $L = \{0, 1, 2, 3, ..., m-1\}$, and we allow different nodes to have the same label, i.e., the label of node number i is given by a function, $l : N \rightarrow L$, which maps n_i to some label $l(n_i) = y \in L$. Each node in T is thus identified by its number and its label. Each branch, $b = (n_x, n_y) \in B$, is an ordered pair of nodes, where n_x is the parent of n_y.

Subtrees We say that a tree $S = (N_s, B_s)$ is an *embedded subtree* of $T = (N, B)$, denoted as $S \preceq T$, provided i) $N_s \subseteq N$, ii) $b = (n_x, n_y) \in B_s$ if and only if $n_y \leq_l n_x$, i.e., n_x is an ancestor of n_y in T. In other words, we require that a branch appears in S if and only if the two vertices are on the same path from the root to a leaf in T. If $S \preceq T$, we also say that T *contains* S. A (sub)tree of size k is also called a k-(sub)tree. Note that in the traditional definition of an *induced* subtree, for each branch $b = (n_x, n_y) \in B_s$, n_x must be a parent of n_y in T. Embedded subtrees are thus a generalization of induced subtrees; they allow not only direct parent-child branches, but also ancestor-descendant branches. As such embedded subtrees are able to extract patterns "hidden" (or embedded) deep within large trees which might be missed by the traditional definition. Henceforth, a reference to subtree should be taken to mean an embedded subtree, unless indicated otherwise. By definition, a subtree must be connected. A disconnected pattern is a *sub-forest* of T. Our main focus is on mining subtrees, although a simple modification of our enumeration scheme also produces sub-forests.

Scope Let $T(n_l)$ refer to the subtree rooted at node n_l, and let n_r be the right-most leaf node in $T(n_l)$. The *scope* of node n_l is given as the interval $[l, r]$, i.e., the lower bound is the position (l) of node n_l, and the upper bound is the position (r) of node n_r. The concept of scope will play an important part in counting subtree frequency.

Tree Mining Problem Let D denote a database of trees (i.e., a forest), and let subtree $S \preceq T$ for some $T \in D$. Each occurrence of S can be identified by its *match label*, which is given as the set of matching positions (in T) for nodes in S. More formally, let $\{t_1, t_2, \ldots, t_n\}$ be the nodes in T, with $|T| = n$, and let $\{s_1, s_2, \ldots, s_m\}$ be the nodes in S, with $|S| = m$. Then S has a match label $\{t_{i_1}, t_{i_2}, \ldots t_{i_m}\}$,if and only if: 1) $l(s_k) = l(t_{i_k})$ for all $k = 1, \ldots m$, and 2) branch $b(s_j, s_k)$ in S iff t_{i_j} is an ancestor of t_{i_k} in T. Condition 1) indicates that all node labels in S have a match in T, while 2) indicates that the tree topology of the matching nodes in T is the same as S. A match label is unique for each occurrence of S in T.
Let $\delta_T(S)$ denote the number of occurrences of the subtree S in a tree T. Let $d_T(S) = 1$ if $\delta_T(S) > 0$ and $d_T(S) = 0$ if $\delta_T(S) = 0$. The *support* of a subtree S in the database is defined as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e., the number of trees in D that contain at least one occurrence of S. The *weighted support* of S is defined as $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., total number of occurrences of S over all trees in D. Typically, support is given as a percentage of the total number of trees in D. A subtree S is *frequent* if its support is more than or equal to a user-specified *minimum support* (*minsup*) value. We denote by F_k the set of all frequent subtrees of size k. Given a user specified *minsup* value our goal is to efficiently enumerate all frequent subtrees in D. In some domains one might be interested in using weighted support, instead of support. Both of them are supported by our mining approach, but we focus mainly on support.

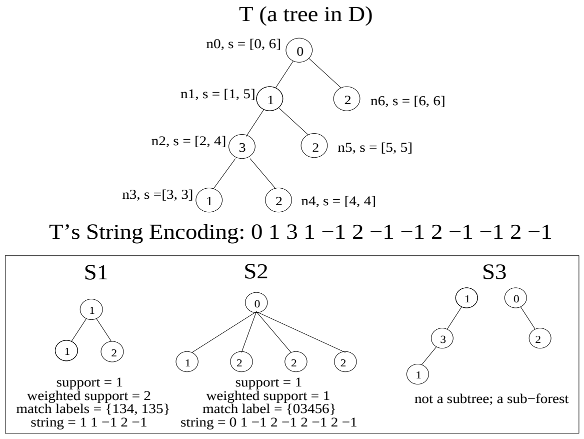


Figure 1: An Example Tree with Subtrees

EXAMPLE 1. *Consider Figure 1, which shows an example tree T with node labels drawn from the set $L = \{0, 1, 2, 3\}$. The figure shows for each node, its label (circled), its number according to depth-first numbering, and its scope. For example, the root occurs at position $n = 0$, its label $l(n_0) = 0$, and since the right-most leaf under the root occurs at position 6, the scope of the root is $s = [0, 6]$. Tree $S1$ is a subtree of T; it has a support of 1, but its weighted support is 2, since node n_2 in $S1$ occurs at positions 4 and 5 in T, both of which support $S1$, i.e., there are two match labels for $S1$, namely 134 and 135 (we omit set notation for convenience). $S2$ is also a valid subtree. $S3$ is not a (sub)tree since it is disconnected; it is a sub-forest.* ■

3. GENERATING CANDIDATE TREES

There are two mains steps for enumerating frequent subtrees in D. First, we need a systematic way of generating *candidate* subtrees whose frequency is to be computed. The candidate set should be non-redundant, i.e., each subtree should be generated as most once. Second, we need efficient ways of counting the number of occurrences of each candidate in the database D, and to determine which candidates pass the *minsup* threshold. The latter step is data structure dependent, and will be treated later. Here we are concerned with the problem of non-redundant pattern generation. We describe below our tree representation and candidate generation procedure.

Representing Trees as Strings Standard ways of representing a labeled tree are via an adjacency matrix or adjacency list. For a tree with n nodes and m branches (note: $m = n - 1$ for trees), adjacency matrix representation requires $n + fn = n(f+1)$ space (f is the maximum fanout; n term is for storing labels and fn term for storing adjacency information), while adjacency lists require $2n + 2m = 4n - 2$ space ($2n$ term is for storing labels and header pointers for adjacency lists, $2m$ is for storing label and next pointer per list node). Since f can possibly be large, we expect adjacency lists to be more space-efficient. If we directly store a labeled tree node as a (label, child pointer, sibling pointer) triplet, we would require $3n$ space.
For efficient subtree counting and manipulation we adopt a string representation of a tree. We use the following procedure to generate the *string encoding*, denoted $\mathcal{T}$, of a tree T. Initially we set $\mathcal{T} = \emptyset$. We then perform a depth-first preorder search starting at the root, adding the current node's label x to $\mathcal{T}$. Whenever we backtrack from a child to its parent we add a unique symbol -1 to the string (we assume that $-1 \notin L$). This format allows us to conveniently represent trees with arbitrary number of children for each node. Since each branch must be traversed in both forward and backward direction, the space usage to store a tree as a string is exactly $2m + 1 = 2n - 1$. Thus our string encoding is more space-efficient than other representations. Moreover, it is simpler to manipulate strings rather than adjacency lists or trees for pattern counting. We use the notation $l(T)$ to refer to the *label sequence* of T, which consists of the node labels of T in depth-first ordering (without backtrack symbol -1), i.e., label sequence ignores tree topology.

Example 2. *In Figure 1, we show the string encodings for the tree T as well as each of its subtrees. For example, subtree $S1$ is encoded by the string $1\ 1\ -1\ 2\ -1$. That is, we start at the root of $S1$ and add 1 to the string. The next node in preorder traversal is labeled 1, which is added to the encoding. We then backtrack to the root (adding -1) and follow down to the next node, adding 2 to the encoding. Finally we backtrack to the root adding -1 to the string. Note that the label sequence of $S1$ is given as 112.* ■

3.1 Candidate Subtree Generation

We use the anti-monotone property of frequent patterns for efficient candidate generation, namely that the frequency of a super-pattern is less than or equal to the frequency of a sub-pattern. Thus, we consider only a known frequent pattern for extension. Past experience also suggests that an extension by a single item at a time is likely to be more efficient. Thus we use information from frequent k-subtrees to generate candidate $(k+1)$-subtrees.

Equivalence Classes We say that two k-subtrees X, Y are in the same *prefix equivalence class* iff they share a common prefix up to the $(k-1)$th node. Formally, let $\mathcal{X}, \mathcal{Y}$ be the string encodings of two trees, and let function $p(\mathcal{X}, i)$ return the prefix up to the ith node. X, Y are in the same class iff $p(\mathcal{X}, k-1) = p(\mathcal{Y}, k-1)$. Thus any two members of an equivalence class differ only in the position of the last node.

Example 3. *Consider Figure 2, which shows a class template for subtrees of size 5 with the same prefix subtree P of size 4, with string encoding $\mathcal{P} = 3\ 4\ 2\ -1\ 1$. Here x denotes an arbitrary label from L. The valid positions where the last node with label x may be attached to the prefix are n_0, n_1 and n_3, since in each of these cases the subtree obtained by adding x to P has the same prefix. Note that a node attached to position $n2$ cannot be a valid member of class $\mathcal{P}$, since it would yield a different prefix, given as $3\ 4\ 2\ x$.*
The figure also shows the actual format we use to store an equivalence class; it consists of the class prefix string, and a list of elements. Each element is given as a (x, p) pair, where x is the label of the last node, and p specifies the depth-first position of the node in P to which x is attached. For example $(x, 1)$ refers to the case where x is attached to node n_1 at position 1. The figure shows the encoding of the subtrees corresponding to each class element. Note how each of them shares the same prefix up to the $(k-1)$th node. These subtrees are shown only for illustration purposes; we only store the element list in a class. ■

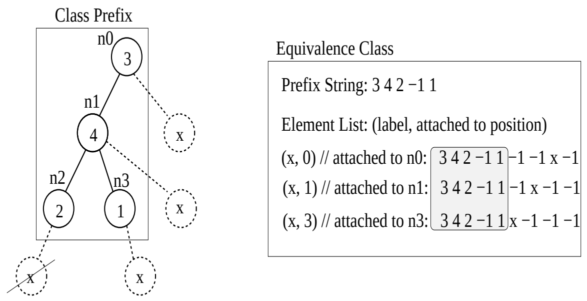

Class Prefix
n0
3
n1
4
n2
n3
2
1
x
x
x
x
Equivalence Class
Prefix String: 3 4 2 −1 1
Element List: (label, attached to position)
(x, 0) // attached to n0: 3 4 2 −1 1 −1 −1 x −1
(x, 1) // attached to n1: 3 4 2 −1 1 −1 x −1 −1
(x, 3) // attached to n3: 3 4 2 −1 1 x −1 −1 −1

Figure 2: Prefix Equivalence Class

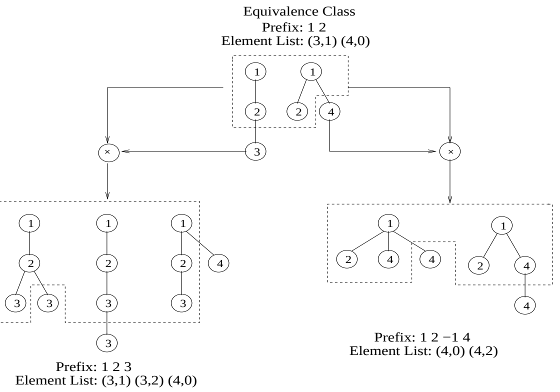
Let P be prefix subtree of size $k-1$; we use the notation $[P]_{k-1}$ to refer to its class (we omit the subscript when there is no ambiguity). If (x, i) is an element of the class, we write it as $(x, i) \in [P]$. Each (x, i) pair corresponds to a subtree of size k, sharing P as the prefix, with the last node labeled x, attached to node n_i in P. We use the notation P_x to refer to the new prefix subtree formed by adding (x, i) to P.

Lemma 1. *Let P be a class prefix subtree and let n_r be the right-most leaf node in P, whose scope is given as $[r, r]$. Let $(x, i) \in [P]$. Then the set of valid node positions in P to which x can be attached is given by $\{i : n_i$ has scope $[i, r]\}$, where n_i is the ith node in P.*

This lemma states that a valid element x may be attached to only those nodes that lie on the path from the root to the right-most leaf n_r in P. It is easy to see that if x is attached to any other position the resulting prefix would be different, since x would then be before n_r in depth-first numbering.

Candidate Generation Given an equivalence class of k-subtrees, how do we obtain candidate $(k+1)$-subtrees? First, we assume (without loss of generality) that the elements, (x, p), in each class are kept sorted by node label as the primary key and position as the secondary key. Given a sorted element list, the candidate generation procedure we describe below outputs a new class list that respects that order, without explicit sorting. The main idea is to consider each ordered pair of elements in the class for extension, including self extension. There can be up to two candidates from each pair of elements to be joined. The next theorem formalizes this notion.

Theorem 1 (Class Extension). *Let P be a prefix class with encoding $\mathcal{P}$, and let (x, i) and (y, j) denote any two elements in the class. Let P_x denote the class representing extensions of element (x, i). Define a* join operator $\otimes$ *on the two elements, denoted $(x, i) \otimes (y, j)$, as follows:*



Equivalence Class
Prefix: 1 2
Element List: (3,1) (4,0)
Prefix: 1 2 3
Element List: (3,1) (3,2) (4,0)
Prefix: 1 2 −1 4
Element List: (4,0) (4,2)

Figure 3: Candidate Generation

case I – $(i = j)$:
a) If $\mathcal{P} \neq \emptyset$, add (y, j) and (y, n_i) to the new class $[P_x]$, where n_i is the depth-first number for node (x, i) in P, given as $n_i = |P| - 1$. [1]
b) If $\mathcal{P} = \emptyset$, add $(y, j + 1)$ to $[P_x]$.
case II – $(i > j)$: *add (y, j) to class $[P_x]$.*
case III – $(i < j)$: *no new candidate is possible in this case.*

Then all possible $(k + 1)$-subtrees with the prefix P of size $k - 1$ will be enumerated by applying the join operator to each ordered pair of elements (x, i) and (y, j).
PROOF: *Omitted due to lack of space.* ■

EXAMPLE 4. *Consider Figure 3, showing the prefix class $\mathcal{P} = (1\ 2)$, which contains 2 elements, $(3, 1)$ and $(4, 0)$. The first step is to perform a self join $(3, 1)\otimes(3, 1)$. By case I a) this produces candidate elements $(3, 1)$ and $(3, 2)$ for the new class $\mathcal{P}_3 = (1\ 2\ 3)$. That is, a self join on $(3, 1)$ produces two possible candidate subtrees, one where the last node is a sibling of $(3, 1)$ and another where it is a child of $(3, 1)$. The left-most two subtrees in the figure illustrate these cases. When we join $(3, 1)\otimes(4, 0)$ case II applies, i.e., the second element is joined to some ancestor of the first one, thus $i > j$. The only possible candidate element is $(4, 0)$, since 4 remains attached to node n_0 even after the join (see the third subtree in the left hand class in Figure 3). We thus add $(4, 0)$ to class $[P_3]$. We now move to the class on the right with prefix $\mathcal{P}_4 = (1\ 2\ -1\ 4)$. When we try to join $(4, 0)\otimes(3, 1)$, case III applies, and no new candidate is generated. Actually, if we do merge these two subtrees, we obtain the new subtree $1\ 2\ 3\ -1\ -1\ 4$, which has a different prefix, and was already added to the class $[P_3]$. Finally we perform a self-join $(4, 0)\otimes(4, 0)$ adding elements $(4, 0)$ and $(4, 2)$ to the class $[P_4]$ shown on the right hand side.* [2] ■

Case I b) applies only when we join single items to produce candidate 2-subtrees, i.e., we are given a prefix class $[\emptyset] = \{(x_i, -1), i = 1, \ldots, m\}$, where each x_i is a label, and -1 indicates that it is not attached to any node. If we join $(x_i, -1)\otimes(x_j, -1)$, since we want only (connected) 2-subtrees, we insert the element $(x_j, 0)$ to the class of x_i. This corresponds to the case where x_j is a child of x_i. If we want to generate sub-forests as well, all we have to do is to insert $(x_j, -1)$ in the class of x_i. In this case x_j would be a sibling of x_i, but since they are not connected, they would be roots of two trees in a sub-forest. If we allow such class elements then one can show that the class extension theorem would produce all possible candidate sub-forests. However, in this paper we will focus only on subtrees.

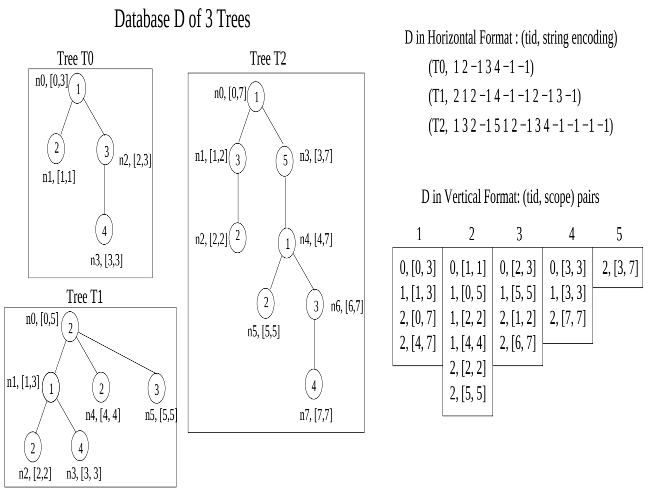
[1]Note that there was an error in the original conference version of this paper, where the second element was wrongly noted as $(y, j + 1)$.
[2]In the original conference version of the paper there was an error, which noted $(4, 1)$ as the second element.

COROLLARY 1 (AUTOMATIC ORDERING). *Let $[P]_{k-1}$ be an prefix class with elements sorted according to the total ordering $<$ given as follows: $(x, i) < (y, j)$ if and only if $x < y$ or ($x = y$ and $i < j$). Then the class extension method generates candidate classes $[P]_k$ with sorted elements.*

COROLLARY 2 (CORRECTNESS). *The class extension - method correctly generates all possible candidate subtrees, and each candidate is generated at most once.*

4. TREEMINER ALGORITHM

TREEMINER performs depth-first search (DFS) for frequent subtrees, using a novel tree representation called *scope-list* for fast support counting, as discussed below.



1	2	3	4	5
0, [0, 3]	0, [1, 1]	0, [2, 3]	0, [3, 3]	2, [3, 7]
1, [1, 3]	1, [0, 5]	1, [5, 5]	1, [3, 3]	
2, [0, 7]	1, [2, 2]	2, [1, 2]	2, [7, 7]	
2, [4, 7]	1, [4, 4]	2, [6, 7]		
	2, [2, 2]			
	2, [5, 5]			

Figure 4: Scope-Lists

Scope-List Representation Let X be a k-subtree of a tree T. Let x_k refer to the last node of X. We use the notation $\mathcal{L}(X)$ to refer to the *scope-list* of X. Each element of the scope-list is a triple (t, m, s), where t is a tree id (tid) in which X occurs, m is a match label of the $(k - 1)$ length prefix of X, and s is the scope of the last item x_k. Recall that the prefix match label gives the positions of nodes in T that match the prefix. Since a given prefix can occur multiple times in a tree, X can be associated with multiple match labels as well as multiple scopes. The initial scope-lists are created for single items (i.e., labels) i that occur in a tree T. Since a single item has an empty prefix, we don't have to store the prefix match label m for single items. We will show later how to compute pattern frequency via joins on scope-lists.

EXAMPLE 5. *Figure 4 shows a database of 3 trees, along with the horizontal format for each tree, and the vertical scope-lists format for each item. Consider item 1; since it occurs at node position 0 with scope $[0, 3]$ in tree T_0, we add $(0, [0, 3])$ to its scope list. Item 1 also occurs in T_1 at position n_1 with scope $[1, 3]$, so we add $(1, [1, 3])$ to $\mathcal{L}(1)$. Finally, 1 occurs with scope $[0, 7]$ and $[4, 7]$ in tree T_2, so we add $(2, [0, 7])$ and $(2, [4, 7])$ to its scope-list. In a similar manner, the scope lists for other items are created.*

4.1 Frequent Subtree Enumeration

Figure 5 shows the high level structure of TREEMINER. The main steps include the computation of the frequent items and 2-subtrees, and the enumeration of all other frequent subtrees via DFS search within each class $[P]_1 \in F_2$. We will now describe each step in some more detail.

```
TREEMINER (D, minsup):
    $F_1$ = { frequent 1-subtrees };
    $F_2$ = { classes $[P]_1$ of frequent 2-subtrees };
    for all $[P]_1 \in E$ do Enumerate-Frequent-Subtrees($[P]_1$);

ENUMERATE-FREQUENT-SUBTREES($[P]$):
    for each element $(x, i) \in [P]$ do
        $[P_x] = \emptyset$;
        for each element $(y, j) \in [P]$ do
            $\mathbf{R} = \{(x, i) \otimes (y, j)\}$;
            $\mathcal{L}(\mathbf{R}) = \{\mathcal{L}(x) \cap_{\otimes} \mathcal{L}(y)\}$;
            if for any $R \in \mathbf{R}$, $R$ is frequent then
                $[P_x] = [P_x] \cup \{R\}$;
        Enumerate-Frequent-Subtrees($[P_x]$);
```

Figure 5: TREEMINER **Algorithm**

Computing F_1 and F_2: TREEMINER assumes that the initial database is in the horizontal string encoded format. To compute F_1, for each item $i \in \mathcal{T}$, the string encoding of tree T, we increment i's count in a 1D array. This step also computes other database statistics such as the number of trees, maximum number of labels, and so on. All labels in F_1 belong to the class with empty prefix, given as $[P]_0 = [\emptyset] = \{(i, -1), i \in F_1\}$, and the position -1 indicates that i is not attached to any node. Total time for this step is $O(n)$ per tree, where $n = |T|$.
By Theorem 1 each candidate class $[P]_1 = [i]$ (with $i \in F_1$) consists of elements of the form $(j, 0)$, where $j \geq i$. For efficient F_2 counting we compute the supports of each candidate by using a 2D integer array of size $F_1 \times F_1$, where $cnt[i][j]$ gives the count of candidate subtree with encoding $(i\ j\ {-1})$. Total time for this step is $O(n^2)$ per tree. While computing F_2 we also create the vertical scope-list representation for each frequent item $i \in F_1$.

Computing $F_k (k \geq 3)$: Figure 5 shows the pseudo-code for the depth-first search for frequent subtrees (ENUMERATE-FREQUENT-SUBTREES). The input to the procedure is a set of elements of a class $[P]$, along with their scope-lists. Frequent subtrees are generated by joining the scope-lists of all pairs of elements (including self-joins). Before joining the scope-lists a pruning step can be inserted to ensure that subtrees of the resulting tree are frequent. If this is true, then we can go ahead with the scope-list join, otherwise we can avoid the join. For convenience, we use the set $\mathbf{R}$ to denote the up to 2 possible candidate subtrees that may result from $(x, i) \otimes (y, j)$, according to the class extension theorem, and we use $\mathcal{L}(\mathbf{R})$ to denote their respective scope-lists. The subtrees found to be frequent at the current level form the elements of classes for the next level. This recursive process is repeated until all frequent subtrees have been enumerated. If $[P]$ has n elements, the total cost is given as $O(ln^2)$, where l is the cost of a scope-list join (given later). In terms of memory management it is easy to see that we need memory to store classes along a path in DFS search. At the very least we need to store intermediate scope-lists for two classes, i.e., the current class $[P]$, and a new candidate class $[P_x]$. Thus the memory footprint of TREEMINER is not much.

4.2 Scope-List Joins ($\mathcal{L}(x) \cap_{\otimes} \mathcal{L}(y)$)

Scope-list join for any two subtrees in a class $[P]$ is based on interval algebra on their scope lists. Let $s_x = [l_x, u_x]$ be a scope for node x, and $s_y = [l_y, u_y]$ a scope for y. We say that s_x is *strictly less* than s_y, denoted $s_x < s_y$, if and only if $u_x < l_y$, i.e., the interval s_x has no overlap with s_y, and it occurs before s_y. We say that s_x *contains* s_y, denoted $s_x \supset s_y$, if and only if $l_x \leq l_y$ and $u_x \geq u_y$, i.e., the interval s_y is a proper subset of s_x. The use of scopes allows us to compute in constant time whether y is a descendant of x or y is a embedded sibling of x. Recall from the candidate extension theorem 1 that when we join elements $(x, i) \otimes (y, j)$ there can be at most two possible outcomes, i.e., we either add $(y, j+1)$ or (y, j) to the class $[P_x]$.

In-Scope Test The first candidate $(y, j+1)$ is added to $[P_x]$ only when $i = j$, and thus refers to the candidate subtree with y as a child of node x. In other words, $(y, j+1)$ represents the subtree with encoding $(\mathcal{P}_x\ y)$. To check if this subtree occurs in an input tree T with tid t, we search if there exists triples $(t_y, s_y, m_y) \in \mathcal{L}(y)$ and $(t_x, s_x, m_x) \in \mathcal{L}(x)$, such that:
1) $t_y = t_x = t$, i.e., the triples both occur in the same tree, with tid t.
2) $m_y = m_x = m$, i.e., x and y are both extensions of the same prefix occurrence, with match label m.
3) $s_y \subset s_x$, i.e., y lies within the scope of x.
If the three conditions are satisfied, we have found an instance where y is a descendant of x in some input tree T. We next extend the match label m_y of the old prefix P, to get the match label for the new prefix P_x (given as $m_y \cup l_x$), and add the triple $(t_y, s_y, \{m_y \cup l_x\})$ to the scope-list of $(y, j+1)$ in $[P_x]$. We refer to this case as an *in-scope* test.

Out-Scope Test The second candidate (y, j) represents the case when y is a embedded sibling of x, i.e., both x and y are descendants of some node at position j in the prefix P, and the scope of x is strictly less than the scope of y. The element (y, j), when added to $[P_x]$ represents the pattern $(\mathcal{P}_x\ {-1}\ \ldots\ {-1}\ y)$ with the number of -1's depending on path length from j to x. To check if (y, j) occurs in some tree T with tid t, we need to check if there exists triples $(t_y, s_y, m_y) \in \mathcal{L}(y)$ and $(t_x, s_x, m_x) \in \mathcal{L}(x)$, such that:
1) $t_y = t_x = t$, i.e., the triples both occur in the same tree, with tid t.
2) $m_y = m_x = m$, i.e., x and y are both extensions of the same prefix occurrence, with match label m.
3) $s_x < s_y$, i.e., x comes before y in depth-first ordering, and their scopes do not overlap.
If these conditions are satisfied, we add the triple $(t_y, s_y, \{m_y \cup l_x\})$ to the scope-list of (y, j) in $[P_x]$. We refer to this case as an *out-scope* test. Note that if we just check whether s_x and s_y are disjoint (with identical tids and prefix match labels), i.e., either $s_x < s_y$ or $s_x > s_y$, then the support can be counted for unordered subtrees!
Each application of in-scope or out-scope test takes $O(1)$ time. Let a and b be the distinct (t, m) pairs in $\mathcal{L}(x, i)$ and $\mathcal{L}(y, j)$, respectively. Let α denote the average number of scopes with a match label. Then the time to perform scope-list joins is given as $O(\alpha^2(a+b))$, which reduces to $O(a+b)$ if α is a small constant.

EXAMPLE 6. *Figure 6 shows an example of how scope-list joins work, using the database D from Figure 4, with* minsup= *100%, i.e., we want to mine subtrees that occur in all 3 trees in D. The initial class with empty prefix consists of four frequent items (1,2,3, and 4), with their scope-lists. All pairs of elements are considered for extension, including self-join. Consider the extensions from item 1, which produces the new class* [1] *with two frequent subtrees:* $(1\ 2\ {-1})$ *and* $(1\ 4\ {-1})$. *The infrequent subtrees are listed at the bottom of the class.*
While computing the new scope-list for the subtree $(1\ 2\ {-1})$ *from* $\mathcal{L}(1) \cap_{\otimes} \mathcal{L}(2)$*, we have to perform only in-scope tests,*

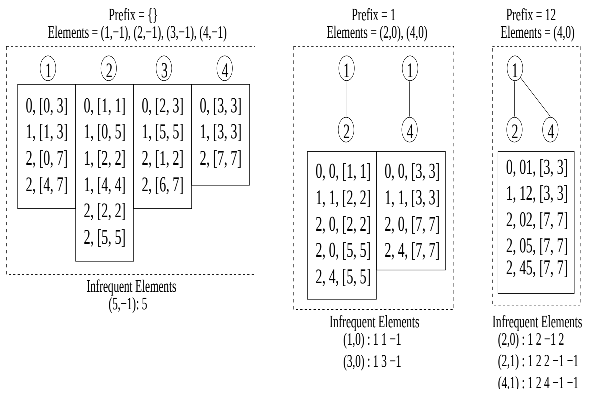


Figure 6: Scope-list joins: *minsup*= 100%

since we want to find those occurrences of 2 that are within some scope of 1 (i.e., under a subtree rooted at 1). Let s_i denote a scope for item i. For tree T_0 we find that $s_2 = [1,1] \subset s_1 = [0,3]$. Thus we add the triple $(0,0,[1,1])$ to the new scope list. In like manner, we test the other occurrences of 2 under 1 in trees T_1 and T_2. Note that for T_2 there are three instances of the candidate pattern: $s_2 = [2,2] \subset s_1 = [0,7]$, $s_2 = [5,5] \subset s_1 = [0,7]$, and $s_2 = [5,5] \subset s_1 = [4,7]$. If a new scope-list occurs in at least minsup *tids, the pattern is considered frequent.*
Consider the result of extending class $[1]$. The only frequent pattern is $(1\ 2\ -1\ 4\ -1)$, whose scope-list is obtained from $\mathcal{L}(2,0) \cap_\otimes \mathcal{L}(4,0)$, by applications of out-scope test. We need to test for disjoint scopes, with $s_2 < s_4$, which have the same match label. For example we find that $s_2 = [1,1]$ and $s_4 = [3,3]$ satisfy these condition. Thus we add the triple $(0,01,[1,1])$ to $\mathcal{L}(4,0)$ in class $[1\ 2]$. Notice that the new prefix match label (01) is obtained by adding to the old prefix match label (0), the position where 2 occurs (1). The final scope list for the new candidate has 3 distinct tids, and is thus frequent. There are no more frequent patterns at minsup= *100%.* ■

Reducing Space Requirements Generally speaking the most important elements of the in-scope and out-scope tests is to make sure that $s_y \subset s_x$ and $s_x < s_y$, respectively. Whenever the test is true we add $(t, s_y, \{m_y \cup l_x\})$ to the candidate's scope-list. However, the match labels are only useful for resolving the prefix context when an item occurs more than once in a tree. Using this observation it is possible to reduce the space requirements for the scope-lists. We add l_x to the match label m_y if and only if x occurs more than once in a subtree with tid t. Thus, if most items occur only once in the same tree, this optimization drastically cuts down the match label size, since the only match labels kept refer to items with more than one occurrence. In the special case that all items in a tree are distinct, the match label is always empty, and each element of a scope-list reduces to a $(tid, scope)$ pair.

EXAMPLE 7. *Consider the scope-list of $(4,0)$ in class $[12]$ in Figure 6. Since 4 occurs only once in T_0 and T_1 we can omit the match label from the first two entries altogether, i.e., the triple $(0,01,[3,3])$ becomes a pair $(0,[3,3])$, and the triple $(1,12,[3,3])$ becomes $(1,[3,3])$.* ■

Opportunistic Candidate Pruning We mentioned above that before generating a candidate k-subtree, S, we perform a pruning test to check if its $(k-1)$-subtrees are frequent. While this is easily done in a BFS pattern search method like PATTERNMATCHER(see next section), in a DFS search we may not have all the information available for pruning, since some classes at level $(k-1)$ would not have been counted yet. TREEMINER uses an opportunistic pruning scheme whereby it first determines if a $(k-1)$-subtree would already have been counted. If it had been counted but is not found in F_{k-1}, we can safely prune S. How do we know if a subtree was counted? For this we need to impose an ordering on the candidate generation, so that we can efficiently perform the subtree pruning test. Fortunately, our candidate extension method has the automatic ordering property (see Corollary 1). Thus we know the exact order in which patterns will be enumerated. To apply pruning test for a candidate S, we generate each subtree X, and test if $X < S$ according to the candidate ordering property. If yes, we can apply the pruning test; if not, we test the next subtree. If S is not pruned, we perform scope-list join to get its exact frequency.

5. PATTERNMATCHER ALGORITHM

PATTERNMATCHER serves as a base pattern matching algorithm to compare TREEMINER against. PATTERNMATCHER employs a breadth-first iterative search for frequent subtrees. Its high-level structure, as shown in Figure 7, is similar to Apriori [3]. However, there are significant differences in how we count the number of subtree matches against an input tree T. For instance, we make use of equivalence classes throughout, and we use a prefix-tree data structure to index them, as opposed to hash-trees. The details of pattern matching are also completely different. PATTERNMATCHER assumes that each tree T in D is stored in its string encoding (horizontal) format (see Figure 4). F_1 and F_2 are computed as in TREEMINER. Due to lack of space we describe only the main features of PATTERNMATCHER; see [22] for details.

```
PATTERNMATCHER (D, minsup):
1. $F_1 = \{$ frequent 1-subtrees $\}$;
2. $F_2 = \{$ classes of frequent 2-subtrees $\}$;
3. for $(k = 3; F_{k-1} \neq \emptyset; k = k+1)$ do
4.       $C_k = \{$ classes $[P]_{k-1}$ of candidate $k$-subtrees $\}$;
5.       for all trees $T$ in $D$ do
6.             Increment count of all $S \preceq T$, $S \in [P]_{k-1}$
7.       $C_k = \{$ classes of frequent $k$-subtrees $\}$;
8.       $F_k = \{$ hash table of frequent subtrees in $C_k \}$;
9. Set of all frequent subtrees $= \bigcup_k F_k$;
```

Figure 7: PATTERNMATCHER **Algorithm**

Pattern Pruning Before adding each candidate k-subtree to a class in C_k we make sure that all its $(k-1)$-subtrees are also frequent. To efficiently perform this step, during creation of F_{k-1} (line 8), we add each individual frequent subtree into a hash table. Thus it takes $O(1)$ time to check each subtree of a candidate, and since there can be k subtrees of length $k-1$, it takes $O(k)$ time to perform the pruning check for each candidate.

Prefix Tree Data Structure Once a new candidate set has been generated, for each tree in D we need to efficiently find matching candidates. We use a prefix tree data structure to index the candidates (C_k) to facilitate fast support counting. Furthermore, instead of adding individual subtrees to the prefix tree, we index an entire class using the class prefix. Thus if the prefix does not match the input tree T, then none of the class elements would match either. This allows us to rapidly focus on the candidates that are likely to be contained in T. Let $[P]$ be a class in C_k. An internal node of the prefix tree at depth d refers to the dth node in P's label sequence. An internal node at depth d points to a

leaf node or an internal node at depth $d+1$. A leaf node of the prefix tree consists of a list of classes with the same label sequence, thus a leaf can contain multiple classes. For example, classes with prefix encodings (1 2 -1 4 3), (1 2 4 3), (1 2 4 -1 -1 3), etc., all have the same label sequence 1243, and thus belong to the same leaf.
Storing equivalence classes in the prefix tree as opposed to individual patterns results in considerable efficiency improvements while pattern matching. For a tree T, we can ignore all classes $[P]_{k-1}$ where $P \not\preceq T$. Only when the prefix has a match in T do we look at individual elements. Support counting consists of three main steps: 1) to find a leaf containing classes that may potentially match T, 2) to check if a given class prefix P exactly matches T, and 3) to check which elements of $[P]$ are contained in T.

Finding Potential Matching Leafs Let $l(T)$ be the label sequence for a tree T in the database. To locate matching leafs, we traverse the prefix tree from the root, following child pointers based on the different items in $l(T)$, until we reach a leaf. This identifies classes whose prefixes have the same label sequence as a subsequence of $l(T)$. This process focuses the search to some leafs of C_k, but the subtree topology for the leaf classes may be completely different. We now have to perform an exact prefix match. In the worst case there may be $\binom{n}{k} \approx n^k$ subsequences of $l(T)$ that lead to different leafs. However, in practice it is much smaller, since only a small fraction of the leafs match the label sequences, especially as pattern length increases. The time to traverse from the root to a leaf is $O(k \log m)$, where m is the average number of distinct labels at an internal node. Total cost of this step is thus $O(kn^k \log m)$.

Prefix Matching Matching the prefix P of a class in a leaf against the tree T is the main step in support counting. Let $X[i]$ denote the ith node of subtree X, and let $X[i, \dots, j]$ denote the nodes from positions i to j, with $j \geq i$. We use a recursive routine to test prefix matching. At the rth recursive call we maintain the invariant that all nodes in $P[0, 1, ..., r]$ have been matched by nodes in $T[i_0, i_1, ..., i_r]$, i.e., prefix node $P[0]$ matches $T[i_0]$, $P[1]$ matches $T[i_1]$, and so on, and finally $P[r]$ matches $T[i_r]$. Note that while nodes in P are traversed consecutively, the matching nodes in T can be far apart. We thus have to maintain a stack of node scopes, consisting of the scope of all nodes from the root i_0 to the current right-most leaf i_r in T. If i_r occurs at depth d, then the scope stack has size $d+1$.
Assume that we have matched all nodes up to the rth node in P. If the next node $P[r+1]$ to be matched is the child of $P[r]$, we likewise search for $P[r+1]$ under the subtree rooted at $T[i_r]$. If a match is found at position i_{r+1} in T we push i_{r+1} onto the scope stack. On the other hand, if the next node $P[r+1]$ is outside the scope of $P[r]$, and is instead attached to position l (where $0 \leq l < r$), then we pop from the scope stack all nodes i_k, where $l < k \leq r$, and search for $P[r+1]$ under the subtree rooted at $T[i_l]$. This process is repeated until all nodes in P have been matched. This step takes $O(kn)$ time in the worst case. If each item occurs once it takes $O(k+n)$ time.

Element Matching If $P \preceq T$, we search for a match in T for each element $(x, k) \in [P]$, by searching for x starting at the subtree $T[i_{k-1}]$. (x, k) is either a descendant or embedded sibling of $P[k-1]$. Either check takes $O(1)$ time. If a match is found the support of the element (x, k) is incremented by one. If we are interested in support (at least one occurrence in T), the count is incremented only once per tree, or else, if we are interested in weighted support (all occurrences in T), we continue the recursive process until all matches have been found.

6. EXPERIMENTAL RESULTS

All experiments were performed on a 500MHz Pentium PC with 512MB memory running RedHat Linux 6.0. Timings are based on total wall-clock time, and include preprocessing costs (such as creating scope-lists for TreeMiner).

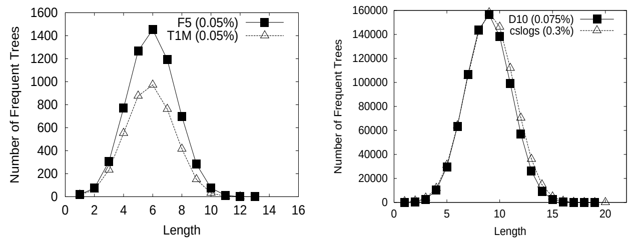


Figure 8: Distribution of Frequent Trees by Length

Synthetic Datasets We wrote a synthetic data generation program mimicking website browsing behavior. The program first constructs a master website browsing tree, $\mathcal{W}$, based on parameters supplied by the user. These parameters include the maximum fanout F of a node, the maximum depth D of the tree, the total number of nodes M in the tree, and the number of node labels N. We allow multiple nodes in the master tree to have the same label. The master tree is generated using the following recursive process. At a given node in the tree $\mathcal{W}$, we decide how many children to generate. The number of children is sampled uniformly at random from the range 0 to F. Before processing children nodes, we assign random probabilities to each branch, including an option of backtracking to the node's parent. The sum of all the probabilities for a given node is 1. The probability associated with a branch $b = (x, y)$, indicates how likely is a visitor at x to follow the link to y. As long as tree depth is less than or equal to maximum depth D this process continues recursively.
Once the master tree has been created we create as many subtrees of $\mathcal{W}$ as specified by the parameter T. To generate a subtree we repeat the following recursive process starting at the root: generate a random number between 0 and 1 to decide which child to follow, or to backtrack. If a branch has already been visited, we select one of the other unvisited branches, or backtrack. We used the following default values for the parameters: the number of labels $N = 100$, the number of nodes in the master tree $M = 10,000$, the maximum depth $D = 10$, the maximum fanout $F = 10$ and total number of subtrees $T = 100,000$. We use three synthetic datasets: $D10$ dataset had all default values, $F5$ had all values set to default, except for fanout $F = 5$, and for $T1M$ we set $T = 1,000,000$, with remaining default values.

CSLOGS Dataset consists of web logs files collected over 1 month at the CS department. The logs touched 13361 unique web pages within our department's web site. After processing the raw logs we obtained 59691 user browsing subtrees of the CS department website. The average string encoding length for a user subtree was 23.3.
Figure 8 shows the distribution of the frequent subtrees by length for the different datasets used in our experiments; all of them exhibit a symmetric distribution. For the lowest minimum support used, the longest frequent subtree in $F5$ and $T1M$ had 12 and 11 nodes, respectively. For cslogs and $D10$ datasets the longest subtree had 18 and 19 nodes.

Performance Comparison Figure 9 shows the performance of PatternMatcher versus TreeMiner. On the real cslogs

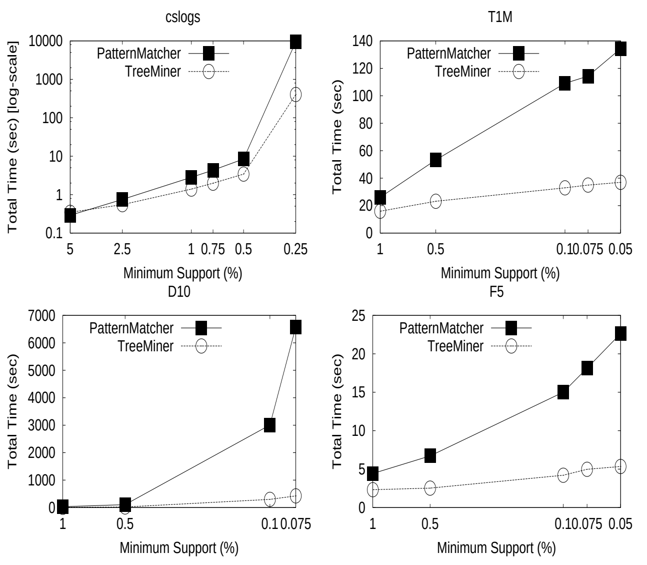


Figure 9: Performance Comparison

dataset, we find that TREEMINER is about 2 times faster than PATTERNMATCHER until support 0.5%. At 0.25% support TREEMINER outperforms PATTERNMATCHER by more than a factor of 20! The reason is that cslogs had a maximum pattern length of 7 at 0.5% support. The level-wise pattern matching used in PATTERNMATCHER is able to easily handle such short patterns. However, at 0.25% support the maximum pattern length suddenly jumped to 19, and PATTERNMATCHER is unable to efficiently deal with such long patterns. Exactly the same thing happens for $D10$ as well. For supports lower than 0.5% TREEMINER outperforms PATTERNMATCHER by a wide margin. At the lowest support the difference is a factor of 15. Both $T1M$ and $F5$ have relatively short frequent subtrees. Here too TREEMINER outperforms PATTERNMATCHER, but for the lowest support shown, the difference is only a factor of 4. These experiments clearly indicate the superiority of scope-list based-method over the pattern matching method, especially as patterns become long.

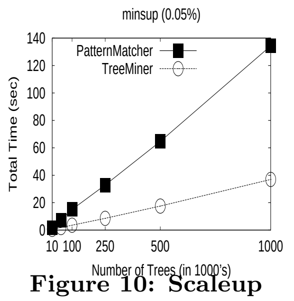


Figure 10: Scaleup

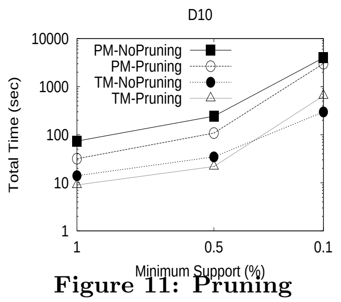


Figure 11: Pruning

Scaleup Comparison Figure 10 shows how the algorithms scale with increasing number of trees in the database D, from 10,000 to 1 million trees. At a given level of support, we find a linear increase in the running time with increasing number of transactions for both algorithms, though TREEMINER continues to be 4 times faster than PATTERNMATCHER.

Effect of Pruning In Figure 11 we evaluated the effect of candidate pruning on the performance of PATTERNMATCHER and TREEMINER. We find that PATTERNMATCHER (denoted PM in the graph) always benefits from pruning, since the fewer the number of candidates, the lesser the cost of support counting via pattern matching. On the other hand TREEMINER (labeled TM in the graph) does not always benefit from its opportunistic pruning scheme. While pruning tends to benefit it at higher supports, for lower supports its performance actually degrades by using candidate pruning. TREEMINER with pruning at 0.1% support on $D10$ is 2 times slower than TREEMINER with no pruning. There are two main reasons for this. First, to perform pruning, we need to store F_k in a hash table, and we need to pay the cost of generating the $(k-1)$ subtrees of each new k-pattern. This adds significant overhead, especially for lower supports when there are many frequent patterns. Second, the vertical representation is extremely efficient; it is actually faster to perform scope-list joins than to perform pruning test.

minsup	No Pruning	Full Pruning	Opportunistic
1%	14595	2775	3505
0.5%	70250	10673	13736
0.1%	3555612	481234	536496

Figure 12: Full vs. Opportunistic Pruning

Table 12 shows the number of candidates generated on the $D10$ dataset with no pruning, full pruning (in PATTERNMATCHER), and with opportunistic pruning (in TREEMINER). Both full pruning and opportunistic pruning are extremely effective in reducing the number of candidate patterns, and opportunistic pruning is almost as good as full pruning (within a factor of 1.3). Full pruning cuts down the number of candidates by a factor of 5 to 7! Pruning is essential thus for pattern matching methods, and may benefit scope-list method in some cases (for high support).

7. APPLICATION: WEB/XML MINING

To demonstrate the usefulness of mining complex patterns, we present below a detailed application study on mining usage patterns in web logs. Mining data that has been collected from web server log files, is not only useful for studying customer choices, but also helps to better organize web pages. This is accomplished by knowing which web pages are most frequently accessed by the web surfers.

We use LOGML [16], a publicly available XML application, to describe log reports of web servers. LOGML provides a XML vocabulary to structurally express the contents of the log file information in a compact manner. LOGML documents have three parts: a web graph induced by the source-target page pairs in the raw logs, a summary of statistics (such as top hosts, domains, keywords, number of bytes accessed, etc.), and a list of user-sessions (subgraphs of the web graph) extracted from the logs.

There are two inputs to our web mining system: 1) web site to be analyzed, and 2) raw log files spanning many days, or extended periods of time. The web site is used to populate a web graph with the help of a web crawler. The raw logs are processed by the LOGML generator and turned into a LOGML document that contains all the information we need to perform various mining tasks. We use the web graph to obtain the page URLs and their node identifiers.

For enabling web mining we make use of user sessions within the LOGML document. User sessions are expressed as subgraphs of the web graph, and contain complete history of the user clicks. Each user session has a session id (IP or host name), a list of edges (`uedges`) giving source and target node pairs, and the time (`utime`) when a link is traversed. An example user session is shown below:

```
<userSession name="ppp0-69.ank2.isbank.net.tr" ...>
<uedge source="5938" target="16470" utime="7:53:46"/>
<uedge source="16470" target="24754" utime="7:56:13"/>
<uedge source="16470" target="24755" utime="7:56:36"/>
<uedge source="24755" target="47387" utime="7:57:14"/>
<uedge source="24755" target="47397" utime="7:57:28"/>
```

```
<uedge source="16470" target="24756" utime="7:58:30"/>
```

Itemset Mining To discover frequent sets of pages accessed we ignore all link information and note down the unique nodes visited in a user session. The user session above produces a user "transaction" containing the user name, and the node set, as follows: (ppp0-69.ank2.isbank.net.tr, 5938 16470 24754 24755 47387 47397 24756).
After creating transactions for all user sessions we obtain a database that is ready to be used for frequent set mining. We applied an association mining algorithm to a real LOGML document from CS web site (one day's logs). There were 200 user sessions with an average of 56 distinct nodes in each session. An example frequent set found is shown below. The pattern refers to a popular Turkish poetry site maintained by one of our department members. The user appears to be interested in the poet Akgun Akova.

```
Let Path=http://www.cs.rpi.edu/∼name/poetry
FREQUENCY=16, NODE IDS = 16395 38699 38700 38698 5938
        Path/poems/akgun_akova/index.html
        Path/poems/akgun_akova/picture.html
        Path/poems/akgun_akova/biyografi.html
        Path/poems/akgun_akova/contents.html
        Path/sair_listesi.html
```

Sequence Mining If our task is to perform sequence mining, we look for the longest forward links [6] in a user session, and generate a new sequence each time a back edge is traversed. We applied sequence mining to the LOGML document from the CS web site. From the 200 user sessions, we obtain 8208 maximal forward sequences, with an average sequence size of 2.8. An example frequent sequence (shown below) indicates in what sequence the user accessed some of the pages related to Akgun Akova. The starting page `sair_listesi` contains a list of poets.

```
Let Path=http://www.cs.rpi.edu/∼name/poetry
FREQUENCY = 20, NODE IDS =  5938 -> 16395 -> 38698
   Path/sair_listesi.html ->
   Path/poems/akgun_akova/index.html ->
   Path/poems/akgun_akova/contents.html
```

Tree Mining For frequent tree mining, we can easily extract the forward edges from the user session (avoiding cycles or multiple parents) to obtain the subtree corresponding to each user. For our example user-session we get the tree: (ppp0-69.ank2.isbank.net.tr, 5938 16470 24754 -1 24755 47387 -1 47397 -1 -1 24756 -1 -1)
We applied the TreeMiner algorithm to the CS logs. From the 200 user sessions, we obtain 1009 subtrees (a single user session can lead to multiple trees if there are multiple roots in the user graph), with an average record length of 84.3 (including the back edges, -1). An example frequent subtree found is shown below. Notice, how the subtree encompasses all the partial information of the sequence and the unordered information of the itemset relating to Akgun Akova. The mined subtree is clearly more informative, highlighting the usefulness of mining complex patterns.

```
Let Path=http://www.cs.rpi.edu/~name/poetry
Let Akova = Path/poems/akgun_akova
FREQUENCY=59, NODES = 5938 16395 38699 -1 38698 -1 38700
                   Path/sair_listesi.html
                             |
          Path/poems/akgun_akova/index.html
         /                   |                \
Akova/picture.html Akova/contents.html Akova/biyografi.html
```

We also ran detailed experiments on logs files collected over 1 month at the CS department, that touched a total of 27343 web pages. After processing the LOGML database had 34838 user graphs. We do not have space to shows the results here (we refer the reader to [16] for details), but these results lead to interesting observations that support the mining of complex patterns from web logs. For example, itemset mining discovers many long patterns. Sequence mining takes longer time but the patterns are more useful, since they contain path information. Tree mining, tough it takes more time than sequence mining, produces very informative patterns beyond those obtained from set and sequence mining.

8. RELATED WORK

Tree mining, being an instance of frequent structure mining, has obvious relation to association [3] and sequence [4] mining. Frequent tree mining is also related to tree isomorphism [18] and tree pattern matching [8]. Given a pattern tree P and a target tree T, with $|P| \leq |T|$, the subtree isomorphism problem is to decide whether P is isomorphic to any subtree of T, i.e., there is a one-to-one mapping from P to a subtree of T, that preserves the node adjacency relations. In tree pattern matching the pattern and target trees are labeled and ordered. We say that P matches T at node v if there exists a one-to-one mapping from nodes of P to nodes of T such that: a) the root of P maps to v, b) if x maps to y, then x and y have the same labels, and c) if x maps to y and x is not a leaf, then the ith child of x maps to the ith child of y. Both subtree isomorphism and pattern matching deal with induced subtrees, while we mine embedded subtrees. Further we are interested in enumerating all common subtrees in a collection of trees. The tree inclusion problem was studied in [13], i.e., given labeled trees P and T, can P be obtained from T by deleting nodes? This problem is equivalent to checking if P is embedded in T. The paper presents a dynamic programming algorithm for solving ordered tree inclusion, which could potentially be substituted for the pattern matching step in PatternMatcher. However, PatternMatcher utilizes prefix information for fast subtree checking, and its three step pattern matching is very efficient over a sequence of such operations.
There has been very little previous work in mining all frequent subtrees. In a recent paper, Asai et al. [5] presented FREQT, an apriori-like algorithm for mining labeled ordered trees; they independently proposed a candidate generation scheme similar to ours. Wang and Liu [20] developed an algorithm to mine frequently occurring subtrees in XML documents. Their algorithm is also reminiscent of the level-wise Apriori [3] approach, and they mine induced subtrees only. A related problem of accurately estimating the number of matches of a small node-labeled tree in a large labeled tree, in the context of querying XML data, was presented in [7]. They compute a summary data structure, and then give frequency estimates based on this summary, rather than using the database for exact answers. In contrast we are interested in exact frequency of subtrees. Furthermore, their work deals with traditional (induced) subtrees, while we mine embedded subtrees.
With the advent of XML as a data representation and exchange standard, there has been active work in indexing and querying XML documents [15, 23, 2, 11], which are mainly tree (or graph) structured. To efficiently answer ancestor-descendant queries various node numbering schemes similar to ours have been proposed [15, 23, 1]. Other work has looked at path query evaluation that uses local knowledge within data graph based on path constraints [2] or graph schemas [11]. The major difference between these works and ours is that instead of answering user-specified queries based on regular path expressions, we are interested in finding all frequent tree patterns among the documents.
There has been recent work in mining frequent graph patterns. The AGM algorithm [12] discovers induced (possibly disconnected) subgraphs. The FSM algorithm [14] improves upon AGM, and mines only the connected subgraphs. Both methods follow an Apriori-style level-wise approach. If one

were to use AGM or FSM for tree mining one would discover only induced subtrees. In contrast we discover embedded subtrees. Also there are important differences in graph mining and tree mining. Our trees are rooted, and thus have a unique ordering of the nodes based on depth-first traversal. In contrast graphs do not have a root, and allow cycles. For mining graphs the methods above first apply an expensive *canonization* step to transform graphs into a uniform representation. This step is unnecessary for tree mining. Graph mining algorithms are likely to be overly general (thus not efficient) for tree mining. Our approach utilizes the tree structure for efficient enumeration.

The work by Dehaspe et al [10] describes a level-wise Inductive Logic Programming based technique to mine frequent substructures (subgraphs) describing the carcinogenesis of chemical compounds. They reported that mining beyond 6 predicates was unfeasible due to the complexity of the subgraph patterns. The SUBDUE system [9] also discovers graph patterns using the Minimum Description Length principle. An approach termed Graph-Based Induction (GBI) was proposed in [21], which uses beam search for mining subgraphs. However, both SUBDUE and GBI may miss some significant patterns, since they perform a heuristic search. We perform a complete (but not exhaustive) search, which guarantees that all patterns are found. In contrast to these approaches, we are interested in developing efficient algorithms for tree patterns.

9. CONCLUSIONS

In this paper we introduced the notion of mining embedded subtrees in a (forest) database of trees. Among our novel contributions is the procedure for systematic candidate subtree generation, i.e., no subtree is generated more than once. We utilized a string encoding of the tree that is space-efficient to store the horizontal dataset, and we use the notion of a node's scope to develop a novel vertical representation of a tree called scope-lists. Our formalization of the problem is flexible enough to handle several variations. For instance, if we assume the label on each node to be the same, our approach mines all unlabeled trees. A simple change in the candidate tree extension procedure allows us to discover sub-forests (disconnected patterns). Our formulation can find frequent trees in a forest of many trees or all the frequent subtrees in a single large tree. Finally, it is relatively easy to extend our techniques to find unordered trees (by modifying the out-scope test) or to use the traditional definition of a subtree. To summarize, this paper proposes a framework for tree mining which can easily encompass most variants of the problem that may arise in different domains. We introduced a novel algorithm, TREEMINER, for tree mining. TREEMINER uses depth-first search; it also uses the novel scope-list vertical representation of trees to quickly compute the candidate tree frequencies via scope-list joins based on interval algebra. We compared its performance against a base algorithm, PATTERNMATCHER. Experiments on real and synthetic data confirmed that TREEMINER outperforms PATTERNMATCHER from a factor of 4 to 20, and scales linearly in the number of trees in the forest. We studied an application of TREEMINER in web usage mining.

For future work we plan to extend our tree mining framework to incorporate user-specified constraints. Given that tree mining, though able to extract informative patterns, is an expensive task, performing general unconstrained mining can be too expensive and is also likely to produce many patterns that may not be relevant to a give user. Incorporating constraints is one way to focus the search and to allow interactivity. We also plan to develop efficient algorithms to mine maximal frequent subtrees from dense datasets which may have very large subtrees. Finally, we plan to apply our tree mining techniques to other compelling applications, such as finding common tree patterns in RNA structures within bioinformatics, as well as the extraction of structure from XML documents and their use in classification, clustering, and so on.

10. REFERENCES

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *ACM Symp. on Discrete Algorithms*, January 2001.

[2] S. Abiteboul and V. Vianu. Regular path expressions with constraints. In *ACM Int'l Conf. on Principles of Database Systems*, May 1997.

[3] R. Agrawal et al. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Intl. Conf. on Data Engg.*, 1995.

[5] 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'l Conference on Data Mining*, April 2002.

[6] M.S. Chen, J.S. Park, and P.S. Yu. Data mining for path traversal patterns in a web environment. In *International Conference on Distributed Computing Systems*, 1996.

[7] Z. Chen et al. Counting twig matches in a tree. In *17th Intl. Conf. on Data Engineering*, 2001.

[8] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $o(n \log^3 n)$-time. In *10th Symposium on Discrete Algorithms*, 1999.

[9] D. Cook and L. Holder. Substructure discovery using minimal description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[10] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *4th Intl. Conf. Knowledge Discovery and Data Mining*, August 1998.

[11] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *IEEE Int'l Conf. on Data Engineering*, February 1998.

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

[13] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. of Computing*, 24(2):340-356, 1995.

[14] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.

[15] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *27th Int'l Conf. on Very Large Data Bases*, 2001.

[16] J. Punin, M. Krishnamoorthy, and M. Zaki. LOGML: Log markup language for web usage mining. In *ACM SIGKDD Workshop on Mining Log Data Across All Customer TouchPoints*, August 2001.

[17] R. Cooley, B. Mobasher, and J. Srivastava. Web Mining: Information and Pattern Discovery on the World Wide Web. In *8th IEEE Intl. Conf. on Tools with AI*, 1997.

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

[19] B. Shapiro and K. Zhang. Comparing multiple rna secondary strutures using tree comparisons. *Computer Applications in Biosciences*, 6(4):309-318, 1990.

[20] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *ACM SIGIR Conference on Information Retrieval*, 1998.

[21] K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, 1995.

[22] M. J. Zaki. Efficiently mining trees in a forest. Tech. Report 01-7, CS Dept., RPI, July 2001.

[23] C. Zhang et al. On supporting containment queries in relational database managment systems. In *ACM Int'l Conf. on Management of Data*, May 2001.