IOS Press

Efficiently Mining Frequent Embedded Unordered Trees

Mohammed J. Zaki *

Computer Science Department

Rensselaer Polytechnic Institute

Troy NY 12180

zaki@cs.rpi.edu

Abstract. Mining frequent trees is very useful in domains like bioinformatics, web mining, mining semi-structured data, and so on. In this paper we introduce SLEUTH, an efficient algorithm for mining frequent, unordered, embedded subtrees in a database of labeled trees. The key contributions of our work are as follows: We give the first algorithm that enumerates all embedded, unordered trees. We propose a new equivalence class extension scheme to generate all candidate trees. We extend the notion of scope-list joins to compute frequency of unordered trees. We conduct performance evaluation on several synthetic and real datasets to show that SLEUTH is an efficient algorithm, which has performance comparable to TreeMiner, that mines only ordered trees.

Keywords: Tree Mining, Embedded Trees, Unordered Trees

1. Introduction

Tree patterns typically arise in applications like bioinformatics, web mining, mining semi-structured documents, and so on. For example, given a database of XML documents, one might like to mine the commonly occurring "structural" patterns, i.e., subtrees, that appear in the collection. As another example, given several phylogenies (i.e., evolutionary trees) from the Tree of Life [16], indicating evolutionary history of several organisms, one might be interested in discovering if there are common subtree patterns.

Whereas itemset mining [1] and sequence mining [2] have been studied extensively in the past, recently there has been tremendous interest in mining increasingly complex pattern types such as trees [3–7, 17, 21, 25] and graphs [12, 15, 22]. For example, several algorithms for tree mining have been proposed

*This work was supported in part by NSF CAREER Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, and NSF grant EIA-0103708.
Address for correspondence: Lally 307, CSCI, RPI, 110 8th St, Troy NY 12180, USA

recently, which include TreeMiner [25], which mines embedded, ordered trees; FreqT [3], which mines induced ordered trees, FreeTreeMiner [5] which mines induced, unordered, free trees (i.e., there is no distinct root); TreeFinder [19], which mines embedded, unordered trees (but it may miss some patterns; it is not complete); and PathJoin [21], uFreqt [17], uNot [4], CMTreeMiner [7] and HybridTreeMiner [6] which mine induced, unordered trees. Our focus in this paper is on a complete and efficient algorithm for mining frequent, labeled, rooted, unordered, and embedded subtrees.

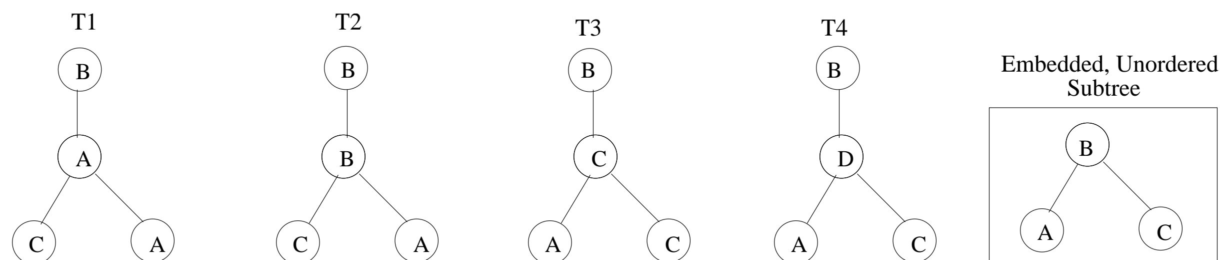

Figure 1. Embedded Unordered Subtree

We choose to look at 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., root element in an XML document, the main web page on a site). We consider unordered trees, since we might miss some potentially frequent patterns if we restrict ourselves to ordered trees. Further, many datasets are inherently unordered (like phylogenies, semi-structured data). Finally, we consider embedded subtrees, which are 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 induced definition. As an example, consider Figure 1, which shows four labeled trees. Let's assume we want to mine subtrees that are common to all four trees (i.e., 100% frequency). If we mine induced trees only, then there are no frequent trees of size more than one. If we mine embedded, but ordered trees, once again no frequent subtrees of size more than one are found. On the other hand, if we mine embedded, unordered subtrees, then the tree shown in the box is a frequent pattern appearing in all four trees; it is obtained by skipping the "middle" node in each tree. This example shows why unordered, and embedded trees are of interest.

In this paper we introduce SLEUTH [1], an efficient algorithm for the problem of mining frequent, unordered, embedded subtrees in a database of trees. The key contributions of our work are as follows: 1) We give the first algorithm that enumerates all embedded, unordered trees. 2) We propose a new self-contained equivalence class extension scheme to generate all candidate trees. Only potentially frequent extensions are considered, but some redundancy is allowed in the candidate generation to make each class self contained. 3) We extend the notion of scope-list joins (first proposed in [25]) for fast frequency computation for unordered trees. We conduct performance evaluation on several synthetic dataset and a real weblog dataset to show that SLEUTH is an efficient algorithm, which has performance comparable to TreeMiner [25] that mines only ordered trees.

[1] SLEUTH is an anagram of the bold letters in the phrase: **L**isting "**H**idden" or **E**mbedded **U**nordered **S**ub**T**rees

2. Preliminaries

Trees A *rooted, labeled, tree*, $T = (V, E)$ is a directed, acyclic, connected graph, with $V = \{0, 1, \cdots, n\}$ as the set of vertices (or nodes), $E = \{(x, y)|x, y \in V\}$ as the set of edges. One distinguished vertex $r \in V$ is designated the *root*, and for all $x \in V$, there is a *unique* path from r to x. Further, $l : V \to L$ is a labeling function mapping vertices to a set of *labels* $L = \{\ell_1, \ell_2, \cdots\}$. In an *ordered tree* the children of each vertex are ordered (i.e., if a vertex has k children, then we can designate them as the first child, second child, and so on up to the kth child), otherwise, the tree is *unordered*.

If $x, y \in V$ and there is a path from x to y, then x is called an *ancestor* of y (and y a *descendant* of x), denoted as $x \leq_p y$, where p is the length of the path from x to y. If $x \leq_1 y$ (i.e., x is an immediate ancestor), then x is called the *parent* of y, and y the *child* of x. If x and y have the same parent, x and y are called *siblings*, and if they have a common ancestor, they are called *cousins*.

We also assume that vertex $x \in V$ is synonymous with (or numbered according to) its position in the depth-first (pre-order) traversal of the tree T (for example, the root r is vertex 0). Let $T(x)$ denote the subtree rooted at x, and let y be the rightmost leaf (or highest numbered descendant) under x. Then the *scope* of x is given as $s(x) = [x, y]$. Intuitively, $s(x)$ demarcates the range of vertices under x.

SubTrees Given a tree $S = (V_s, E_s)$ and tree $T = (V_t, E_t)$, we say that S is an *isomorphic subtree* of T *iff* there exists a one-to-one mapping $\varphi : V_s \to V_t$, such that $(x, y) \in E_s$ iff $(\varphi(x), \varphi(y)) \in E_t$. If φ is onto, then S and T are called *isomorphic*. S is called an *induced subtree* of $T = (V_t, E_t)$, denoted $S \preceq_i T$, iff S is an isomorphic subtree of T, and φ preserves labels, i.e., $l(x) = l(\varphi(x)), \forall x \in V_s$. That is, for induced subtrees φ preserves the parent-child relationships, as well as vertex labels. The induced subtree obtained by deleting the rightmost leaf in T is called an *immediate prefix* of T. The induced tree obtained from T by a series of rightmost node deletions is called a *prefix* of T. In the sequel we use prefix to mean an immediate prefix, unless we indicate otherwise.

$S = (V_s, E_s)$ is called an *embedded subtree* of $T = (V_t, E_t)$, denoted as $S \preceq_e T$ iff there exists a 1-to-1 mapping $\varphi : V_s \to V_t$ that satisfies: i) $(x, y) \in E_s$ iff $\varphi(x) \leq_p \varphi(y)$, and ii) $l(x) = l(\varphi(x))$. That is, for embedded subtrees φ preserves ancestor-descendant relationships and labels. A (sub)tree of size k is also called a k-(sub)tree. If $S \preceq_e T$, we also say that T *contains* S or S *occurs* in T. Note that each occurrence of S in T can be identified by its unique *match label*, given by the sequence $\varphi(x_0)\varphi(x_1)\cdots\varphi(x_{|S|})$, where $x_i \in V_s$. That is a match label of S is given as the set of matching positions in T.

Support Let $\delta_T(S)$ denote the number of occurrences (induced or embedded, depending on context) of the subtree S in a tree T. Let d_T be an indicator variable, with $d_T(S) = 1$ if $\delta_T(S) > 0$ and $d_T(S) = 0$ if $\delta_T(S) = 0$. Let D denote a database (a *forest*) of trees. 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. In some domains one might be interested in using weighted support, instead of support. Both of them are allowed our mining approach, but we focus mainly on support.

Tree Mining Tasks Given a collection of trees D and a user specified *minsup* value, several tree mining tasks can be defined, depending on the choices among rooted/free, ordered/unordered or induced/embedded trees. *In this paper we focus on efficiently enumerating all frequent, embedded, unordered subtrees in D.*

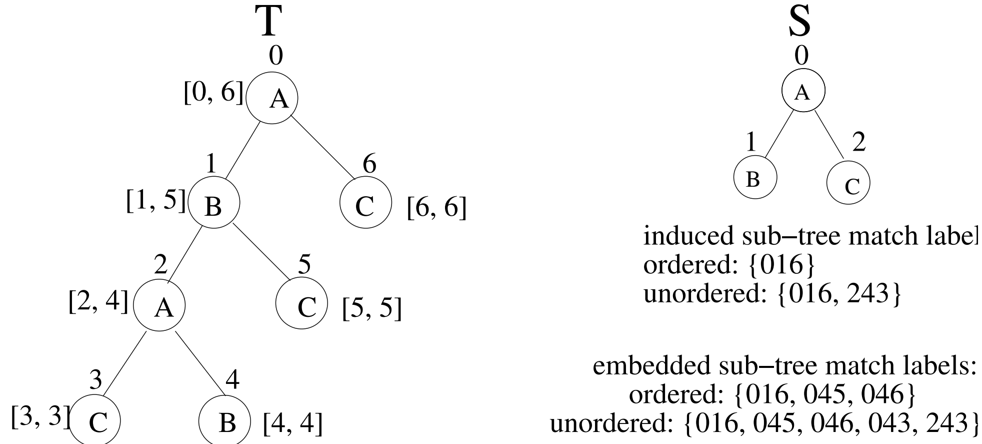

Figure 2. An Example: Tree and Subtree

Example 2.1. Consider Figure 2, which shows an example tree T with vertex labels drawn from the set $L = \{A, B, C\}$, and vertices identified by their depth-first number. The figure shows for each vertex, its label, depth-first number, and scope. For example, the root is vertex 0, its label $l(0) = A$, and since the right-most leaf under the root is vertex 6, the scope of the root is $s(0) = [0, 6]$.

Consider S; it is clearly an induced subtree of T. If we look only at ordered subtrees, then the match label of S in T is given as: $012 \rightarrow \varphi(0)\varphi(1)\varphi(2) = 016$ (we omit set notation for convenience). If unordered subtrees are considered, then 243 is also a valid match label. S has additional match labels as an embedded subtree. In the ordered case, we have additional match labels 045 and 046, and in the unordered case, we have on top of these two, the label 043.

Thus the induced weighted support of S is 1 for ordered and 2 for unordered case. The embedded weighted support of S is 3, if ordered, and 5, if unordered. The support of S is 1 in all cases. ■

Tree Representation: String Encoding As described in [25], we represent a tree T by its *string encoding*, denoted $\mathcal{T}$, generated as follows: Add vertex labels to $\mathcal{T}$ in a depth-first preorder traversal of T, and add a unique symbol $\$ \notin L$ whenever we backtrack from a child to its parent. For example, for T shown in Figure 2, its string encoding is $A\ B\ A\ C\ \$\ B\ \$\ \$\ C\ \$\ \$\ C\ \$$. We use the notation $\mathcal{T}[i]$ to denote the element at position i in $\mathcal{T}$, where $i \in [1, |\mathcal{T}|]$, and $|\mathcal{T}|$ is the length of the string $\mathcal{T}$.

Database Representation: Scope-Lists We refer to the tree database in the string encoding format as the horizontal database. In SLEUTH, we represent the database in the vertical format [25], in which for every distinct label we store its scope-list, which is a list of tree ids and vertex scopes where that label occurs. For label ℓ, we denote its scope-list as $\mathcal{L}(\ell)$; each entry in the scope list is a pair (t, s), where t is a tree id (tid) in which ℓ occurs, and s is the scope of a vertex with label ℓ in tid t.

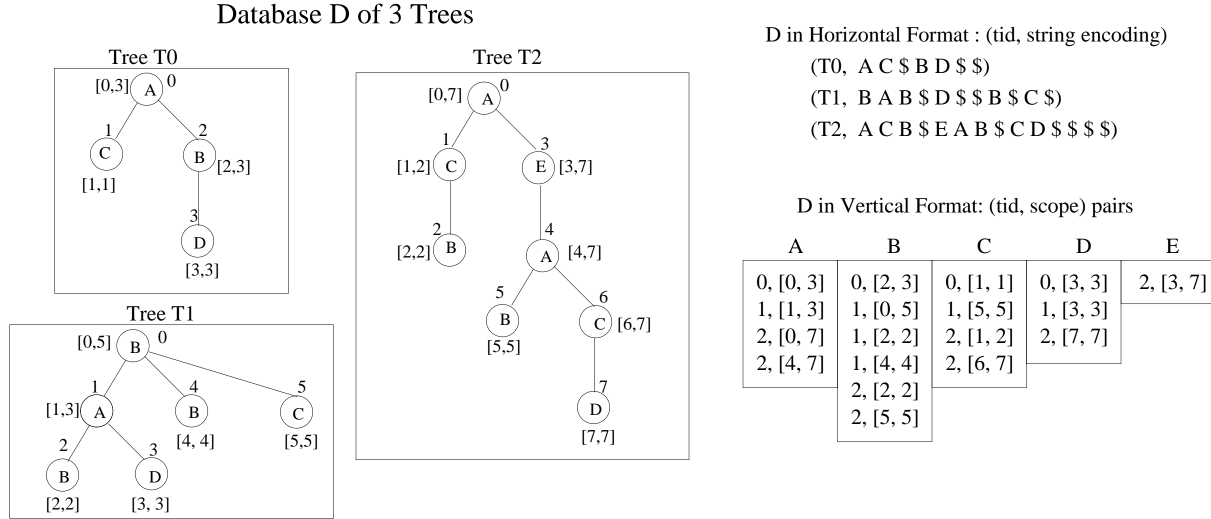

Figure 3. Scope-Lists

Example 2.2. Figure 3 shows a database of 3 trees, along with the horizontal format for each tree, and the vertical scope-lists format for each label. Consider label A; since it occurs at vertex 0 with scope $[0, 3]$ in tree T_0, we add $(0, [0, 3])$ to its scope list. A also occurs in T_1 with scope $[1, 3]$, and in T_2 with scopes $[0, 7]$ and $[4, 7]$, thus we add $(1, [1, 3])$, $(2, [0, 7])$ and $(2, [4, 7])$ to $\mathcal{L}(A)$. In a similar manner, the scope lists for other labels are created. ■

We use the scope-lists to represent the list of occurrences in the database, for any k-subtree S. Let x be the label of the rightmost leaf in S. The scope list of S consists of triples (t, m, s), where t is a tid where S occurs, s is the scope of vertex with label x in tid t, and m is a match label for the prefix subtree of S. Thus our vertical database is in fact the set of scope-lists for all 1-subtrees (and since they have no prefix, there is no match label).

3. Related Work

Tree mining, being an instance of frequent structure mining, has obvious relation to association [1] and sequence [2] mining. Frequent tree mining is also related to tree isomorphism [18] and tree pattern matching [8]. 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. 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.

Recently tree mining has attracted a lot of attention. We developed TreeMiner [25] to mine labeled, embedded, and ordered subtrees. The notions of scope-lists and rightmost extension were introduced in that work. TreeMiner was also used in building a structural classifier for XML data [26]. Asai et al. [3] presented FreqT, an apriori-like algorithm for mining labeled ordered trees; they independently proposed the rightmost candidate generation scheme. 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 [1] approach, and they mine induced subtrees only. There are several other recent algorithms that

mine different types of tree patterns, which include FreeTreeMiner [5] which mines induced, unordered, free trees (i.e., there is no distinct root); and PathJoin [21], uFreqt [17], uNot [4], and HybridTreeMiner [6] which mine induced, unordered trees. CMTreeMiner [7] mines maximal and closed induced, unordered trees. TreeFinder [19] uses an Inductive Logic Programming approach to mine unordered, embedded subtrees, but it is not a complete method, i.e, it can miss many frequent subtrees, especially as support is lowered or when the different trees in the database have common node labels. Our focus here is on an efficient algorithm to mine the complete set of frequent, embedded, unordered trees.

There has also been recent work in mining frequent graph patterns. The AGM algorithm [12] discovers induced (possibly disconnected) subgraphs. The FSG algorithm [15] improves upon AGM, and mines only the connected subgraphs. Both methods follow an Apriori-style level-wise approach. Recent methods to mine graphs using a depth-first tree based extension have been proposed in [22, 23]. Another method uses a candidate generation approach based on Canonical Adjacency Matrices [11]. 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. Work on molecular feature mining has appeared in [14]. The SUBDUE system [9] also discovers graph patterns using the Minimum Description Length principle. An approach termed Graph-Based Induction (GBI) was proposed in [24], which uses beam search for mining subgraphs. However, both SUBDUE and GBI may miss some significant patterns, since they perform a heuristic search. In contrast to these approaches, we are interested in developing efficient, complete algorithms for tree patterns.

4. Generating Unordered, Embedded Trees

There are two main 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 to the extent possible; ideally, each subtree should be generated as most once. Second, we need efficient ways of counting the number of occurrences of each candidate tree 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 candidate generation.

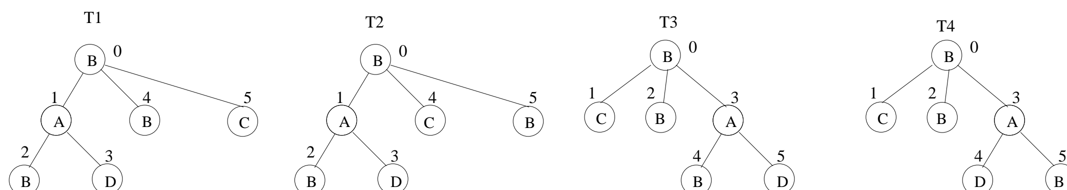

Figure 4. Some Automorphisms of the Same Graph

Automorphism Group An *automorphism* of a tree is a isomorphism with itself. Let $Aut(T)$ denote the *automorphism group*, i.e., the set of all label preserving automorphisms, of T. Henceforth, by automorphism, we mean label preserving automorphisms. The goal of candidate generation is to enumerate only one *canonical* representative from $Aut(T)$. For an unordered tree T, there can be many automorphisms. For example, Figure 4 shows some of the automorphisms of the same tree.

Let there be a linear order $\leq$ defined on the elements of the label set L. Given any two trees X and Y, we can define a linear order $\leq$, called *tree order* between them, recursively as follows: Let r_x and

r_y denote the roots of X and Y, and let $c_1^{r_x}, \cdots, c_m^{r_x}$ and $c_1^{r_y}, \cdots, c_n^{r_y}$ denote the ordered list of children of r_x and r_y, respectively. Also let $T(c_i^{r_x})$ denote the subtree of X rooted at vertex $c_i^{r_x}$. Then $X \leq Y$ (alternatively, $T(r_x) \leq T(r_y)$) iff either:

1.) $l(r_x) < l(r_y)$, or

2.) $l(r_x) = l(r_y)$, and either a) $n \leq m$ and $T(c_i^{r_x}) = T(c_i^{r_y})$ for all $1 \leq i \leq n$, i.e., Y is a prefix (not necessarily immediate prefix) of or equal to X, or b) there exists $j \in [1, \min(m, n)]$, such that $T(c_i^{r_x}) = T(c_i^{r_y})$ for all $i < j$, and $T(c_j^{r_x}) < T(c_j^{r_y})$.

This tree ordering is essentially the same as that in [17], although their tree coding is different.

We can also define a *code order* on the tree encodings directly as follows: Assume that the special backtrack symbol $\$ > \ell$ for all $\ell \in L$. Given two string encodings $\mathcal{X}$ and $\mathcal{Y}$. We say that $\mathcal{X} \leq \mathcal{Y}$ iff either:

i.) $|\mathcal{Y}| \leq |\mathcal{X}|$ and $\mathcal{X}[k] = \mathcal{Y}[k]$ for all $1 \leq k \leq |\mathcal{Y}|$, or

ii.) There exists $k \in [1, \min(|\mathcal{X}|, |\mathcal{Y}|)]$, such that for all $1 \leq i < k$, $\mathcal{X}[i] = \mathcal{Y}[i]$ and $\mathcal{X}[k] < \mathcal{Y}[k]$.

Incidentally, a similar tree code ordering was independently proposed in CMTreeMiner [7].

Lemma 4.1. *$X \leq Y$ iff $\mathcal{X} \leq \mathcal{Y}$.*

Proof Sketch: Condition i) in code order holds if $\mathcal{X}$ and $\mathcal{Y}$ are identical for the entire length of $\mathcal{Y}$, but this is true iff Y is a prefix of (or equal to) X.

Condition ii) holds if and only if $\mathcal{X}$ and $\mathcal{Y}$ are identical up to position $k-1$, i.e., $\mathcal{X}[1, \cdots, k-1] = \mathcal{Y}[1, \cdots, k-1]$. This is true iff both X and Y share a common prefix tree P with encoding $\mathcal{P} = \mathcal{X}[1, \cdots, k-1]$). Let v_X^i (and v_Y^j) refer to the node in tree X (and Y), that corresponds to position $\mathcal{X}[i] \neq \$$ (and $\mathcal{Y}[j] \neq \$$).

If $k = 1$, then P is an empty tree with encoding $\mathcal{P} = \emptyset$. It is clear that $l(r_x) < l(r_y)$ iff $\mathcal{X}[1] < \mathcal{Y}[1]$. If $k > 1$, then $\mathcal{X}[k] < \mathcal{Y}[k]$, iff one of the following cases is true: A) $\mathcal{X}[k] \neq \$$ and $\mathcal{Y}[k] \neq \$$: We immediately have $\mathcal{X}[k] < \mathcal{Y}[k]$ iff $T(v_X^k) < T(v_Y^k)$ iff $X < Y$. B) $\mathcal{X}[k] \neq \$$ and $\mathcal{Y}[k] = \$$: let v_X^j be parent of node v_X^k ($j < k$), and let v_Y^j be the corresponding node in Y (which refers to $\mathcal{Y}[j] \neq \$$). We then immediately have that $T(v_Y^j)$ is a prefix of $T(v_X^j)$, since $\mathcal{X}[j, \cdots, k-1] = \mathcal{Y}[j, \cdots, k-1]$, and v_X^j has an extra child v_X^k, whereas v_Y^j doesn't. □

Given $Aut(T)$ the canonical representative $T_c \in Aut(T)$ is the tree, such that $T_c \leq X$ for all $X \in Aut(T)$. For any $P \in Aut(T)$ we say that P is in *canonical form* if $P = T_c$. For example, $T_c = T_1$ for the automorphism group $Aut(T_1)$, four of whose members are shown in Figure 4. We can see that the string encoding $\mathcal{T}_1 = BAB\$D\$\$B\$C\$$ is smaller than $\mathcal{T}_2 = BAB\$D\$\$C\$B\$$ and also smaller than other members.

Lemma 4.2. *A tree T is in canonical form iff for all vertices $v \in T$, $T(c_i^v) \leq T(c_{i+1}^v)$ for all $i \in [1, k]$, where $c_1^v, c_2^v, \cdots, c_k^v$ is the list of ordered children of v.*

Proof Sketch: T is in canonical form implies that $T \leq X$ for all $X \in Aut(T)$. Assume that there exist some vertex $v \in T$ such that $T(c_i) > T(c_{i+1})$ for some $i \in [1, k]$, where $c_1, c_2, \cdots, c_k$ are the ordered children of v. But then, we can obtain tree T' by simply swapping the subtrees $T(c_i)$ and $T(c_{i+1})$ under node v. However, by doing so, we make $T' < T$, which contradicts the assumption that T is canonical. □

Prefix Extension Let $R(P) = v_1 v_2 \cdots v_m$ denote the rightmost path in tree P, i.e., the path from root P_r to the rightmost leaf in P. Given a seed frequent tree P, we can generate new candidates P_x^i obtained by adding a new leaf with label x to any vertex v_i on the rightmost path $R(P)$. We call this process as *prefix-based* extension, since each such candidate has P as its prefix tree.

It has been shown that prefix-based extension can correctly enumerate all ordered embedded or induced trees [3, 25]. For unordered trees, we only have to do a further check to see if the new extension is the canonical form for its automorphism group, and if so, it is a valid extension. For example, Figure 5 shows the seed tree P, with encoding $\mathcal{P} = CDA\$B$ (omitting trailing \$'s). To preserve the prefix tree, only rightmost branch extensions are allowed. Since the rightmost path is $R(P) = 013$, we can extend P by adding a new vertex with label x any of these vertices, to obtain a new tree P_x^i ($i \in \{0, 1, 3\}$). Note, how adding x to node 2 gives a different prefix tree encoding $CDAx$, and is thus disallowed, as shown in the figure.

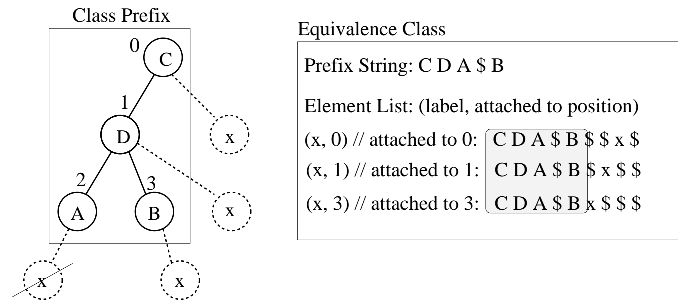

Figure 5. Prefix Extension and Equivalence Class

In [17] it was shown that for any tree in canonical form its prefix is also in canonical form. Thus starting from vertices with distinct labels, using prefix extensions, and retaining only canonical forms for each automorphism group, we can enumerate all unordered trees non-redundantly. For each candidate, we can count the number of embedded occurrences in database D to determine which are frequent. Thus the main challenges in tree extension are to: i) efficiently determine whether an extension yields a canonical tree, and ii) determine extensions which will potentially be frequent. The former step considers only valid candidates, whereas the latter step minimizes the number of frequency computations against the database.

Canonical Extension To check if a tree is in canonical form, we need to make sure that for each vertex $v \in T$, $T(c_i) \leq T(c_{i+1})$ for all $i \in [1, k]$, where $c_1, c_2, \cdots, c_k$ is the list of ordered children of v. However, since we extend only canonical trees, for a new candidate, its prefix is in canonical form, and we can do better.

Lemma 4.3. *Let P be a tree in canonical form, and let $R(P)$ be the rightmost path in P. Let P_x^k be the tree extension of P when adding a vertex with label x to some vertex v_k in $R(P)$. For any $v_i \in R(P_x^k)$, let $c_{l-1}^{v_i}$ and $c_l^{v_i}$ denote the last two children of v_i* [2]*. Then P_x^k is in canonical form iff for all $v_i \in R(P_x^k)$, $T(c_{l-1}^{v_i}) \leq T(c_l^{v_i})$.*

Proof Sketch: Let $R(P) = v_1 v_2 \cdots v_k v_{k+1} \cdots v_m$ be the rightmost path in P. By Lemma 4.2, P is in canonical form implies that for every node $v_i \in R(P)$, we have $T(c_{l-1}^{v_i}) \leq T(c_l^{v_i})$.

[2] If v_i is a leaf, then both children are empty, and if v_i has only one child, then $c_{l-1}^{v_i}$ is empty

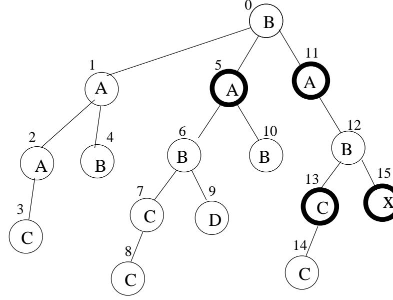

Figure 6. Check for Canonical Form

When we extend P to P_x^k, we obtain a new rightmost path $R(P_x^k) = v_1 v_2 \cdots v_k v_n$, where v_n is the new last child of v_k (with label x). Thus both $R(P)$ and $R(P_x^k)$ share the vertices $v_1 v_2 \cdots v_k$ in common. Note that for any $i > k$, $v_i \in R(P)$ is unaffected by the addition of vertex v_n. On the other hand, for all $i < k$, the last child $c_l^{v_i}$ of $v_i \in R(P)$ (i.e., $v_i \in R(P_x^k)$) is affected by v_n, whereas $c_{l-1}^{v_i}$ remains unchanged. Also for $i = k$, the last two children of v_k change in tree P_x^k; we have $c_{l-1}^{v_k} = v_{k+1}$ and $c_l^{v_k} = v_n$.

Since P is in canonical form, we immediately have that for all $v_i \in \{v_1, v_2, \cdots, v_k\}$, $T(c_j^{v_i}) \leq T(c_{l-1}^{v_i})$ for all $j < l-1$. Thus we only have to compare the new subtree $T(c_l^{v_i})$ with $T(c_{l-1}^{v_i})$. If $T(c_{l-1}^{v_i}) \leq T(c_l^{v_i})$ for all $v_i \in R(P_x^k)$, then by Lemma 4.2, we immediately have that P_x^k is in canonical form. On the other hand if $T(c_{l-1}^{v_i}) > T(c_l^{v_i})$ for some $v_i \in R(P_x^k)$, then P_x^k cannot be in canonical form. □

According to lemma 4.3 we can check if a tree P_x^k is in canonical form by starting from the rightmost leaf in $R(P_x^k)$ and checking if the subtrees under the last two children for each node on the rightmost path are ordered according to $\leq$. By lemma 4.1 it is sufficient to check if their string encodings are ordered by $\leq$. For example, given the candidate tree P_x^{12} shown in Figure 6 which has a new vertex 15 with label x attached to node 12 on the rightmost path, we first compare 15 with its previous sibling 13. For $T(13) \leq T(15)$, we require that $x \geq C$. After skipping node 11 (with empty previous sibling), we reach node 0, where we compare $T(5)$ and $T(11)$. For $T(5) \leq T(11)$ we require that $x \geq D$, otherwise P_x^{12} is not canonical. Thus for any $x < D$ the tree is not canonical. It is possible to speed-up the canonicality checking by adopting a different tree coding [17], but here we will continue to use the string encoding of a tree. The corresponding checks for canonicality based on lemma 4.1 among the subtree encodings are shown below:

Compare 13 and 15: $BAAC\$\$B\$\$ABCC\$\$D\$\$B\$\$AB\underline{CC}\$\$\underline{x}$

Compare 5 and 11: $BAAC\$\$B\$\$\underline{ABCC\$\$D\$\$B}\$\$\underline{ABCC}\$\$x$

Based on the check for canonical form, we can determine which labels are possible for each rightmost path extension. Given a tree P and the set of frequent labels F_1, we can then try to extend P with each label from F_1 that leads to a canonical extension. Even though all of these candidates are non-redundant (i.e., there are no isomorphic duplicates), this extension process may still produce too many candidate trees, whose frequencies have to be counted in the database D. To reduce the number of such trees, we try to extend P with a vertex that is more likely to result in a frequent tree, using the idea of a prefix equivalence class.

Equivalence Class-based Extension We say that two k-subtrees X, Y are in the same *prefix equivalence class* iff they share the same prefix tree. Thus any two members of a prefix class differ only in the last vertex. For example, Figure 5 shows the class template for subtrees with the same prefix subtree P with string encoding $\mathcal{P} = C\ D\ A\ \$\ B$. The figure 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, i) pair, where x is the *label* of the last vertex, and i specifies the vertex in P to which x is attached. For example $(x, 1)$ refers to the case where x is attached to vertex 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 vertex. These subtrees are shown only for illustration purposes; we only store the element list in a class.

Let P be a prefix subtree of size $k-1$; we use the notation $[P]$ to refer to its class (we will use P and its string encoding $\mathcal{P}$ interchangeably). 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 vertex labeled x, attached to vertex i in P. We use the notation P_x^i to refer to the new prefix subtree formed by adding (x, i) to P.

Let P be a $(k-1)$-subtree, and let $[P] = \{(x,i) | P_x^i \text{ is frequent}\}$ be the set of all possible frequent extensions of prefix tree P. Then the set of potentially frequent candidate trees for the class $[P_x^i]$ (obtained by adding an element (x, i) to P), can be obtained by prefix extensions of P_x^i with each element $(y, j) \in [P]$, given as follows: i) *cousin extension*: If $j \leq i$ and $|P| = k - 1 \geq 1$, then $(y, j) \in [P_x^i]$, and in addition ii) *child extension*: If $j = i$ then $(y, k-1) \in [P_x^i]$.

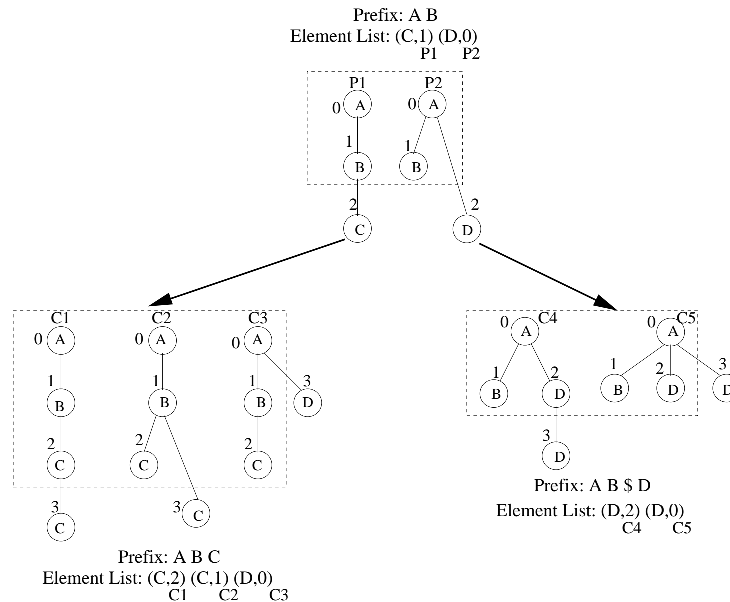

Figure 7. Equivalence Class-based Extension

Example 4.1. Consider Figure 7, showing the prefix class $\mathcal{P} = AB$, which contains 2 elements, $(C, 1)$ and $(D, 0)$. Let's consider the extensions of first element, i.e., of $[P_C^1] = [ABC]$. First we must consider element $(C, 1)$ itself. As child extension, we add $(C, 2)$ (tree C_1), and as cousin extension, we add $(C, 1)$ (tree C_2). Extending with $(D, 0)$, since $0 < 1$, we only add cousin extension $(D, 0)$ (tree C_3) to $[ABC]$.

When considering extensions of $[P_D^0] = [AB\$D]$, we consider $(C, 1)$ first. But since C is attached to vertex 1, it cannot preserve the prefix tree P_D. Considering $(D, 0)$, we add $(D, 0)$ as a cousin extension and $(D, 2)$ as a child extension, corresponding to trees C_4 and C_5. ■

The main observation behind equivalence class extension is that only known frequent elements from the same class are used for extending P_x^i, which itself is known to be frequent from the previous extension step. Furthermore, we only extend P_x^i, if it is in canonical form. However, to guarantee that all possible extensions are members of $[P]$, we have to relax the non-redundant tree generation idea. That is, whereas we extend only canonical P_x^i, all possible extensions are added to $[P_x^i]$ (which are not necessarily canonical). This contrasts with the purely canonical extension approach, where only canonical extensions are considered. In essence canonical and equivalence class extensions represent a trade-off between the number of redundant (isomorphic) candidates generated and the number of potentially frequent candidates to count. Canonical extensions generate non-redundant candidates, but many of which may turn out not to be frequent. On the other hand, equivalence class extension generates redundant candidates, but considers a smaller number of (potentially frequent) extensions. In our experiments we found equivalence class extensions to be more efficient. One consequence of using equivalence class extensions is that SLEUTH doesn't depend on any particular canonical form; it can work with any systematic way of choosing a representative from an automorphism group. Provided only one representative is extended, its class contains all information about the extensions that can be potentially frequent. This can provide a lot of flexibility on how tree enumeration is performed.

5. Frequency Computation

SLEUTH uses scope-list joins for fast frequency computation for a new extension. We assume that each element (x, i) in a prefix class $[P]$ has a scope-list which stores all occurrences of the tree P_x^i (obtained by extending P with (x, i)). The vertical database contains the initial scope lists $\mathcal{L}(\ell)$ for each distinct label ℓ. To compute the scope-lists for members of $[P_x^i]$ we need to join the scope-lists of (x, i) with every other element $(y, j) \in [P]$. If the resulting tree is frequent, we insert the element in $[P_x^i]$.

Let $s_x = [l_x, u_x]$ be a scope for vertex 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 $l_x < l_y$ and $u_x \geq u_y$, i.e., the interval s_y is a proper subset of s_x.

Recall from the equivalence class extension that when we extend element $[P_x^i]$ there can be at most two possible outcomes, i.e., child extension or cousin extension. The use of scopes allows us to compute in constant time whether y is a descendant of x or y is a cousin of x. We describe below how to compute the embedded support for child and cousin (unordered) extensions, using the descendant and cousin tests.

Descendant Test Given $[P]$ and any two of its elements (x, i) and (y, j). In a child extension of P_x^i the element (y, j) is added as a child of (x, i). For embedded frequency computation, we have to find all occurrences where label y occurs as a descendant of x, sharing the same prefix tree P_x^i in some $T \in D$, with tid t. This is called the *descendant test*. To check if this subtree occurs in an input tree T with tid t, we search if there exists triples $(t_y, m_y, s_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x) \in \mathcal{L}(x)$, 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 then extend the match label m_y of the old prefix P, to get the match label for the new prefix P_x^i (given as $m_y \cup l_x$), and add the triple $(t_y, \{m_y \cup l_x\}, s_y)$ to the scope-list of $(y, |P|)$ in $[P_x^i]$.

Cousin Test Given $[P]$ and any two of its elements (x, i) and (y, j). In a cousin extension of P_x^i the element (y, j) is added as a cousin of (x, i). For embedded frequency computation, we have to find all occurrences where label y occurs as a cousin of x, sharing the same prefix tree P_x^i in some input tree $T \in D$, with tid t. This is called the *cousin test*. To check if y occurs as a cousin in some tree T with tid t, we need to check if there exists triples $(t_y, m_y, s_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x) \in \mathcal{L}(x)$, 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) $s_x < s_y$ or $s_x > s_y$, i.e., either x comes before y or y comes before x in depth-first ordering, and their scopes do not overlap. This allows us to find the unordered frequency and is one of the crucial differences compared to ordered tree mining, as in TreeMiner [25], which only checks if $s_x < s_y$.
If these conditions are satisfied, we add the triple $(t_y, \{m_y \cup l_x\}, s_y)$ to the scope-list of (y, j) in $[P_x^i]$.

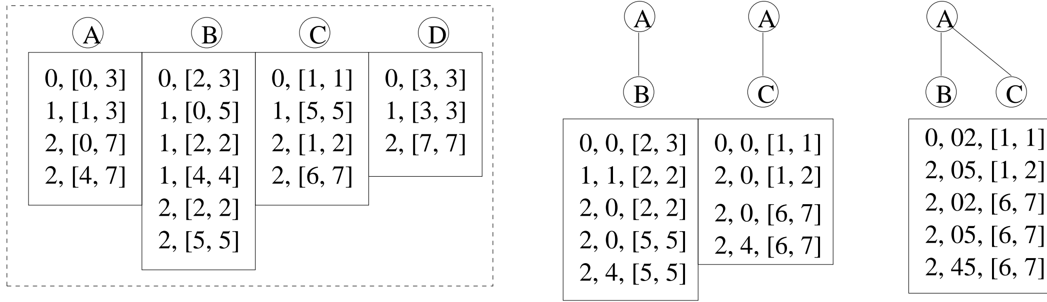

Figure 8. Scope-list Joins

Example 5.1. Figure 8 shows an example of how scope-list joins work, using the database D from Figure 3. The initial class with empty prefix consists of four frequent labels (A, B, C, and D), with their scope-lists. All pairs (not necessarily distinct) of elements are considered for extension.

Two of the frequent trees in class $[A]$ are shown, namely $AB\$$ and $AC\$$. $AB\$$ is obtained by joining the scope lists of A and B and performing descendant tests, since we want to find those occurrences of B that are within some scope of A (i.e., under a subtree rooted at A). Let s_x denote a scope for label x. For tree T_0 we find that $s_B = [2, 3] \subset s_A = [0, 3]$. Thus we add the triple $(0, 0, [2, 3])$ to the new scope list. Similarly, we test the other occurrences of B under A in trees T_1 and T_2. If a new scope-list occurs in at least *minsup* tids, the pattern is considered frequent.

The next candidate shows an example of testing frequency of a cousin extension, namely, how to compute the scope list of $AB\$C$ by joining $\mathcal{L}(AB)$ and $\mathcal{L}(AC)$. For finding all unordered embedded occurrences, we need to test for disjoint scopes, with $s_B < s_C$ or $s_C < s_B$, which have the same match label. For example, in T_0, we find that $s_B = [2, 3]$ and $s_C = [1, 1]$ satisfy these condition. Thus we add the triple $(0, 02, [1, 1])$ to $\mathcal{L}(AB\$C)$. Notice that the new prefix match label (02) is obtained by adding to the old prefix match label (0), the position where B occurs (i.e.,2). The other occurrences are noted in the final scope-list. ■

6. The SLEUTH Algorithm

Figure 9 shows the high level structure of SLEUTH. The main steps include the computation of the frequent labels (1-subtrees) and 2-subtrees, and the enumeration of all other frequent subtrees via recursive (depth-first) equivalence class extensions of each class $[P]_1 \in F_2$. We will now describe each step in some more detail.

```
SLEUTH (D, minsup):
1.  F_1 = { frequent 1-subtrees };
2.  F_2 = { classes [P]_1 of frequent 2-subtrees }; //create scope-lists
3.  for all [P]_1 ∈ F_2 do Enumerate-Frequent-Subtrees([P]_1);

ENUMERATE-FREQUENT-SUBTREES([P]):
4.  for each element (x, i) ∈ [P] do
5.    if check-canonical(P_x^i) then
6.      [P_x^i] = ∅;
7.      for each element (y, j) ∈ [P] do
8.        if do-child-extension then L_d = descendant-scope-list-join((x, i), (y, j));
9.        if do-cousin-extension then L_c = cousin-scope-list-join((x, i), (y, j));
10.       if child or cousin extension is frequent then
11.           Add (y, j) and/or (y, k − 1) to equivalence class [P_x^i]; //add scope-list also
12.     Enumerate-Frequent-Subtrees([P_x^i]);
```

Figure 9. SLEUTH Algorithm

Computing F_1 and F_2: SLEUTH assumes that the initial database is in the horizontal string encoded format. To compute F_1 (line 1), for each label $i \in \mathcal{T}$ (the string encoding of tree T), we increment i's count in a count 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 $[P]_0 = [\emptyset] = \{(i, _),\ i \in F_1\}$, and the position _ indicates that i is not attached to any vertex. Total time for this step is $O(n)$ per tree, where $n = |T|$.

For efficient F_2 counting (line 2) we compute the supports of all candidate by using a 2D integer array of size $F_1 \times F_1$, where $cnt[i][j]$ gives the count of the candidate (embedded) subtree with encoding $(i\ \ j\ \ \$)$. 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 $i \in F_1$, and before each call of `Enumerate-Frequent-Subtrees` $([P]_1 \in E)$ (line 3) we also compute the scope lists of all frequent elements (2-subtrees) in the class.

Computing $F_k (k \geq 3)$: Figure 9 shows the pseudo-code for the recursive (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.

Before extending the class $[P_x^i]$ we first make sure that P_x^i is the canonical representative of its automorphism group (line 5). If not, the pattern will not be extended. If yes, we try to extend P_x^i with

every element $(y, j) \in [P]$. We try both child and cousin extensions, and perform descendant or cousin tests during scope-list join (lines 8,9). If any candidate is frequent, it is added to the new class $[P_x^i]$. This way, 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 $O(qn^2)$, where q is the cost of a scope-list join. The cost of scope-list join is $O(me^2)$, where m is the average number of distinct tids in the scope list of the two elements, and e is the average number of embeddings of the pattern per tid. The total cost of generating a new class is therefore $O(m(en)^2)$.

In terms of memory management, we need memory to store classes along a path in DFS search. In fact we need to store intermediate scope-lists for two classes at a time, i.e., the current class $[P]$, and a new candidate class $[P_x^i]$. Thus the memory footprint of SLEUTH is not much, unless the scope-lists become too big, which can happen if the number of embeddings of a pattern is large. If the lists are too large to fit in memory, we can do joins in stages. That is, we can bring in portions of the scope-lists for the two elements to be joined, perform descendant or cousin tests, and write out portions of the new scope-list.

Lemma 6.1. *The equivalence class-based extension in* SLEUTH *correctly generates all possible embedded, unordered, frequent subtrees.*

Proof Sketch: We prove the correctness of SLEUTH by induction on the length k of the mined k-subtrees. Let's consider the base cases. For $k = 1$, SLEUTH considers each label and counts its frequency, thus all 1-subtrees (F_1) are found correctly. Let x, y, z be any three labels (not necessarily distinct). For $k = 2$, SLEUTH considers all possible 2-subtrees of the form $xy\$$, and counts their frequency; 2-subtrees are by definition canonical and thus all frequent 2-subtrees are recorded in F_2. For $k = 3$, SLEUTH considers all 3-subtrees of the form $xyz\$\$$ or $xy\$z\$$, computes their frequency, and creates prefix equivalence classes $[xy\$]$, where each such class contains *all* frequent extensions of $xy\$$. The class can contain non-canonical elements, but only canonical elements will be output and considered for extension. Thus all possible 3-subtrees are correctly mined.

For the inductive step, let's assume that SLEUTH mines the set of frequent k-subtrees, organized as a set of prefix-based equivalence classes $\mathbf{P}$, where each class $P \in \mathbf{P}$ has a shared $(k-1)$ length prefix tree, and consists of all frequent extensions of P (not necessarily canonical extensions). Also note that SLEUTH correctly outputs only the canonical frequent k-subtrees from each class.

We will now show that SLEUTH will enumerate all canonical frequent $(k+1)$-subtrees. Consider any class $[P]$; let (x, i) and (y, j) be any two elements of the class (not necessarily distinct). From the previous step we already know that P_x^i and P_y^j (i.e, extensions of P with (x, i) and (y, j)) are frequent. The equivalence class extension step enumerates all frequent subtrees by using the rightmost path extension, whose correctness has been proved in [3, 25]. Since SLEUTH extends only canonical subtrees, each new class $[P_x^i]$ has a canonical prefix k-subtree P_x^i, and all of its frequent extensions are elements of the class. Out of these only the canonical $(k+1)$-subtrees will be output. This proves that SLEUTH enumerates all possible, embedded, unordered, frequent subtrees. □

Equivalence Class vs. Canonical Extensions As described above, SLEUTH uses equivalence class extensions to enumerate the frequent trees. The prefix P is known to be frequent from the previous step, and we extend it only if it is in canonical form. To ensure that all possible extensions are members of

$[P]$, we had to compromise on non-redundant tree generation, by adding all possible extensions of P to the class $[P]$, whether they are canonical or not.

```
SLEUTH-FKF2 (D, minsup):
1.  F_1 = { frequent 1-subtrees };
2.  F_2 = { classes [P]_1 of frequent 2-subtrees }; //create scope-lists
3.  for all [P]_1 ∈ F_2 do Enumerate-Frequent-Subtrees([P]_1, F_2);

ENUMERATE-FREQUENT-SUBTREES([P], F_2):
4.  for each element (x, i) ∈ [P] do
5.     [P_x^i] = ∅;
6.     for each element (y, j) ∈ [P] ∪ [x], where [x] ∈ F_2 do
7.        if check-canonical(P_x^i extended with (y, j)) then
8.           if do-child-extension then L_d = descendant-scope-list-join((x, i), (y, j));
9.           if do-cousin-extension then L_c = cousin-scope-list-join((x, i), (y, j));
10.          if child or cousin extension is frequent then
11.               Add (y, j) and/or (y, k − 1) to equivalence class [P_x^i]; //add scope-list also
12.    Enumerate-Frequent-Subtrees([P_x^i], F_2);
```

Figure 10. SLEUTH-FKF2 Algorithm

For comparison we implemented another approach, called SLEUTH-FKF2, which performs only canonical extensions. The main idea is to extend a canonical and frequent subtree, with a known frequent subtree from F_2. The pseudo-code is shown in Figure 10. The computation of F_1 and F_2 is the same as for SLEUTH(lines 1-2). We then call `Enumerate-Frequent-Subtrees` for each class in F_2 (line 3). This function takes as input a class $[P]$, all of whose elements are known to be both *frequent and canonical*. Each member (x, i) of $[P]$ (line 4) is either extended with another element of $[P]$ or with elements in $[x]$ (line 6), where $[x] \in F_2$ denotes all possible frequent 2-subtrees of the form $xy\$$; to guarantee correctness we have to extend $[P_x^i]$ with all $y \in [x]$. Note that elements of both $[P]$ and $[x]$ represent canonical subtrees, and if the child or cousin extension is canonical (line 7), we perform descendant and cousin joins, and add the new subtree to $[P_x^i]$ if is is frequent. This way, each class only contains elements that are both canonical and frequent.

Lemma 6.2. SLEUTH-FKF2 *correctly generates all possible embedded, unordered, frequent subtrees.*

Proof Sketch: The proof is similar to that for SLEUTH. The main difference is that instead of storing all possible frequent extensions in a prefix class, SLEUTH-FKF2 stores only the canonical, frequent extensions. To generate new $(k + 1)$ length candidates, all possible rightmost path extensions with elements of F_2 are considered. If any extension is both canonical and frequent the process continues to the next level, until all possible embedded, unordered, frequent subtrees have been mined. □

As we mentioned earlier pure canonical and equivalence class extensions denote a trade-off between the number of redundant candidates generated and the number of potentially frequent candidates to count. Canonical extensions generate non-redundant candidates, but many of which may turn out not to be frequent (since, in essence, we join F_k with F_2 to obtain F_{k+1}). On the other hand, equivalence class extension generates redundant candidates, but considers a smaller number of (potentially frequent)

extensions (since, in essence, we join F_k with F_k to obtain F_{k+1}). In the next section we compare these two methods experimentally; we found SLEUTH, which uses equivalence class extensions to be more efficient, than SLEUTH-FKF2, which uses only canonical extensions.

7. Experimental Results

All experiments were performed on a 2.8GHz Pentium 4 processor with 1GB main memory, and with a 250GB, 7200rpms disk, running RedHat Linux 9. Timings are based on total wall-clock time, and include all preprocessing costs (such as creating scope-lists).

Synthetic Datasets We constructed a synthetic data generation program to create a database of artificial website browsing behavior [25]. We first construct a master website browsing tree 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. For each node in master tree W, we assign probabilities of following its children nodes, including the option of backtracking to its parent, such that sum of all the probabilities is 1. Using the master tree, one can generate a subtree $T_i \preceq W$ by randomly picking a subtree of W as the root of T_i and then recursively picking children of the current node according to the probability of following that link.

We used the following default values for the parameters: the number of labels $N = 100$, the number of vertices 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.

7.1. Performance Evaluation

Figure 11 shows the performance of SLEUTH on different datasets for different values of minimum support, and compares the run time against TreeMiner and SLEUTH-FKF2. Note that, whereas SLEUTH and SLEUTH-FKF2 mine unordered embedded patterns, TreeMiner mines ordered embedded patterns. The second column in the figure shows the distribution of frequent embedded, unordered patterns for various supports. Finally the third column shows the difference between the number of frequent embedded unordered and ordered patterns; a positive value means that there are more frequent unordered patterns than ordered ones.

Let's consider the $F5$ dataset. We find that unordered and ordered pattern mining (SLEUTH and TreeMiner, respectively) are comparable, but TreeMiner takes less time. There are two main reasons for this behavior. First, the number of unordered patterns is more than ordered patterns for this dataset. Second, SLEUTH needs to perform canonical form tests, while TreeMiner doesn't, since for ordered tree mining, the automorphism group for a tree only contains one member, the tree itself ($Aut(T) = \{T\}$).

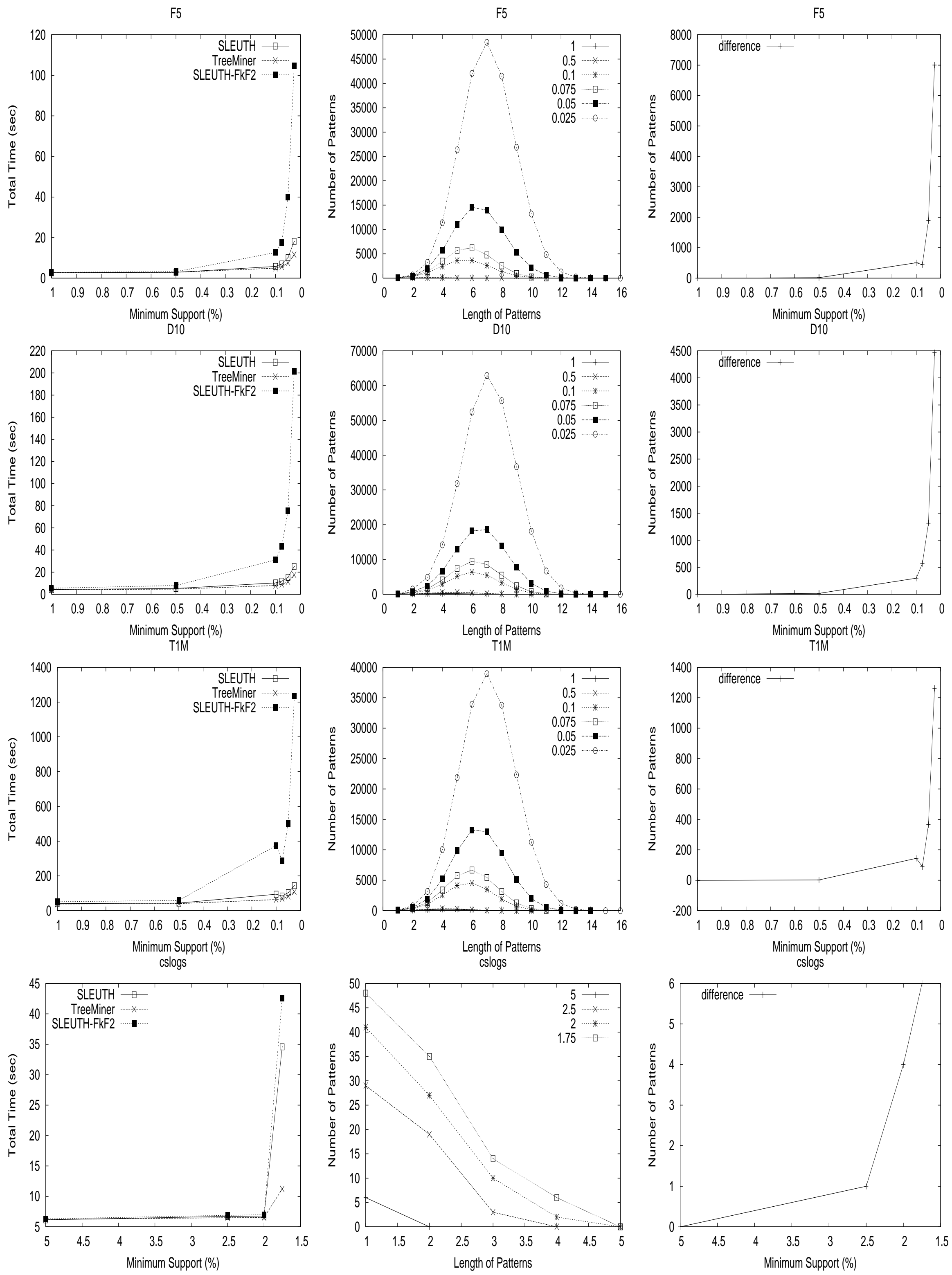

Figure 11. Performance Evaluation

Looking at the length distribution, we find it to be mainly symmetric across the support values, and also, generally speaking more unordered tree are found as compared to ordered ones, especially as minimum support is lowered. Comparing SLEUTHwith SLEUTH-FKF2, we find that there is a big performance loss for the pure canonical extensions due to the joins of F_k with F_2, which result in many infrequent candidates; SLEUTH-FKF2 can be 5 times slower than SLEUTH. Similar trends are obtained for $D10$ and $T1M$ datasets; TreeMiner is slightly faster than SLEUTH, whereas SLEUTH can be 5-10 times faster than SLEUTH-FKF2. This shows clearly that the strategy of generating some redundant candidates, but extending only canonical prefix classes is superior to generating many infrequent but purely canonical candidates.

The web-log dataset $CSLOGS$ has different characteristics for the supports at which we mined. Looking at the pattern length distribution, we find that the number of patterns keep decreasing as length increases. Also there is not much difference between the number of unordered and ordered embedded trees. Considering run times, SLEUTH remains faster than SLEUTH-FKF2 and both of them take the same time as ordered tree mining for higher values of support, but for a low value (1.75%) it takes much longer to mine unordered patterns. The reason for this gap is that SLEUTH keeps track of all possible "unordered" mappings from a candidate to a dataset tree. For the $CSLOGS$ dataset, this results in longer scope-lists than for TreeMiner, which keeps only the ordered mappings. Longer scope-lists lead to higher execution time for the joins.

Summarizing from the results over synthetic and reals datasets, we can conclude that SLEUTH is an efficient, complete, algorithm for mining unordered, embedded trees. Even though it mines more patterns than TreeMiner, and has to perform canonical form tests, its performance is comparable to ordered tree mining.

8. Conclusions

In this paper we presented, SLEUTH, the first algorithm to mine all unordered, embedded subtrees in a database of labeled trees. Among our contributions is the procedure for systematic candidate subtree generation using self-contained equivalence prefix classes. All frequent patterns are enumerated by unordered scope-list joins via the descendant and cousin tests. We also compared SLEUTH with SLEUTH-FKF2, which also mines unordered, embedded trees, but uses pure canonical extensions. Our experiments show that SLEUTH is more efficient than SLEUTH-FKF2, and is generally comparable to TreeMiner, which mines only ordered subtrees, even though SLEUTH has to check if a subtree is in canonical form.

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 given 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 compelling applications, such as finding common tree patterns in phylogenetic data within bioinformatics, as well as the extraction of structure from XML documents and their use in classification, clustering, and so on.

References

[1] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A. I.: Fast Discovery of Association Rules, *Advances in Knowledge Discovery and Data Mining* (U. Fayyad, et al, Eds.), AAAI Press, Menlo Park, CA, 1996.

[2] Agrawal, R., Srikant, R.: Mining Sequential Patterns, *11th Intl. Conf. on Data Engg.*, 1995.

[3] Asai, T., Abe, K., Kawasoe, S., Arimura, H., Satamoto, H., Arikawa, S.: Efficient Substructure Discovery from Large Semi-structured Data, *2nd SIAM Int'l Conference on Data Mining*, April 2002.

[4] Asai, T., Arimura, H., Uno, T., Nakano, S.: Discovering Frequent Substructures in Large Unordered Trees, *6th Int'l Conf. on Discovery Science*, October 2003.

[5] Chi, Y., Yang, Y., Muntz, R. R.: Indexing and Mining Free Trees, *3rd IEEE International Conference on Data Mining*, 2003.

[6] Chi, Y., Yang, Y., Muntz, R. R.: HybridTreeMiner: An Efficient Algorihtm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms, *16th International Conference on Scientific and Statistical Database Management*, 2004.

[7] Chi, Y., Yang, Y., Xia, Y., Muntz, R. R.: CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees, *8th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2004.

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

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

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

[11] Huan, J., Wang, W., Prins, J.: Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism, *IEEE Int'l Conf. on Data Mining*, 2003.

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

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

[14] Kramer, S., Raedt, L. D., Helma, C.: Molecular Feature Mining in HIV data, *Int'l Conf. on Knowledge Discovery and Data Mining*, 2001.

[15] Kuramochi, M., Karypis, G.: Frequent Subgraph Discovery, *1st IEEE Int'l Conf. on Data Mining*, November 2001.

[16] Morell, V.: Web-Crawling up the Tree of Life, *Science*, **273**(5275), aug 1996, 568–570.

[17] Nijssen, S., Kok, J. N.: Efficient Discovery of Frequent Unordered Trees, *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.

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

[19] Termier, A., Rousset, M.-C., Sebag, M.: TreeFinder: a First Step towards XML Data Mining, *IEEE Int'l Conf. on Data Mining*, 2002.

[20] Wang, K., Liu, H.: Discovering Typical Structures of Documents: A Road Map Approach, *ACM SIGIR Conference on Information Retrieval*, 1998.

[21] Xiao, Y., Yao, J.-F., Li, Z., Dunham, M. H.: Efficient Data Mining for Maximal Frequent Subtrees, *International Conference on Data Mining*, 2003.

[22] Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining, *IEEE Int'l Conf. on Data Mining*, 2002.

[23] Yan, X., Han, J.: CloseGraph: Mining Closed Frequent Graph Patterns, *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, August 2003.

[24] Yoshida, K., Motoda, H.: CLIP: Concept Learning from Inference Patterns, *Artificial Intelligence*, **75**(1), 1995, 63–92.

[25] Zaki, M. J.: Efficiently Mining Frequent Trees in a Forest, *8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, July 2002.

[26] Zaki, M. J., Aggarwal, C.: Xrules: An effective structural classifier for XML data, *9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, August 2003.