MINING GRAPH DATA

A JOHN WILEY & SONS, INC., PUBLICATION

CHAPTER 1

A UNIFIED APPROACH TO ROOTED TREE MINING: ALGORITHMS AND APPLICATIONS

1.1 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 [15], indicating evolutionary history of several organisms, one might be interested in discovering if there are common subtree patterns.

Recently there has been tremendous interest in mining increasingly complex pattern types such as trees [30, 1, 4, 6, 26, 18, 2, 5] and graphs [12, 14, 27]. For example, several algorithms for tree mining have been proposed recently, which include TreeMiner [30], which mines embedded, ordered trees; SLEUTH [31], which mines embedded, unordered trees; FreqT [1], which mines induced ordered trees, FreeTreeMiner [4] which mines induced, unordered, free trees (i.e., there is no distinct root); TreeFinder [22], which mines embedded, unordered trees (but it may miss some patterns; it is not complete); and PathJoin [26],

uFreqt [18], uNot [2], CMTreeMiner [6] and HybridTreeMiner [5] which mine induced, unordered trees.

In this paper we extend SLEUTH [1] to obtain an efficient, unified algorithm for the problem of mining frequent subtrees. The key contributions of our work are as follows: 1) We present a unified approach to tree mining that can handle both ordered or unordered, and both induced or embedded trees. 3) 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. We study the trade-off between non-redundant versus potentially frequent candidate generation. 4) We extend the notion of scope-list joins (first proposed in [30]) for fast frequency computation for unordered/induced trees. 5) We also propose a method to count only distinct tree occurrences, instead of all mappings. 6) Finally, we conduct performance evaluation on several synthetic datasets and a real weblog dataset to show that SLEUTH is an efficient algorithm for different types of tree patterns. We present applications of tree mining in bioinformatics, such as mining frequent RNA structures, and common phylogenetic tree patterns.

1.2 PRELIMINARIES

A *rooted tree* $T = (V, E)$ is a directed, acyclic, graph with vertex set $V = \{0, 1, \cdots, n\}$, edge set $E = \{(x, y) | x, y \in V\}$, and with one distinguished vertex $r \in V$ called the *root* such that for all $x \in V$, there is a *unique* path from r to x. In a *labeled tree*, $l : V \rightarrow L$ 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., 1st child, 2nd child, etc.), 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

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

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.

As suggested in [30], 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 1.1., 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 $\mathcal{T}$.

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 $\varphi : V_s \rightarrow V_t$, 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 $\varphi : V_s \rightarrow V_t$ 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.

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 using our mining approach, but we focus mainly on support.

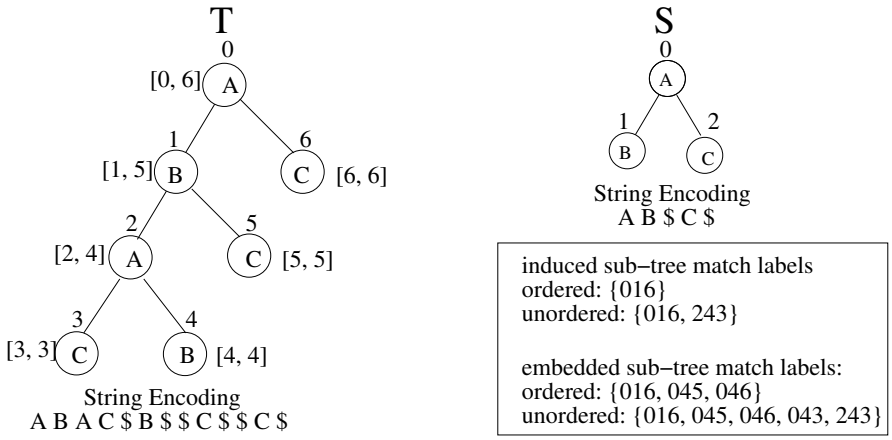


Figure 1.1. An Example: Tree and Subtree

Given a collection of trees D and a user specified *minsup* value, several rooted tree mining tasks can be defined, depending on the choices among ordered/unordered or induced/embedded trees. Consider Figure 1.1., 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 the 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.

1.3 RELATED WORK

Recently tree mining has attracted a lot of attention. Zaki proposed TreeMiner [30] 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 [32]. XSpanner [24] is a pattern-growth based method for mining embedded ordered subtrees. Asai et al. [1] presented FreqT, an apriori-like algorithm for mining labeled ordered trees; they independently proposed the rightmost candidate generation scheme. Wang and Liu [25] developed an algorithm to mine frequently occurring subtrees in XML documents. Their algorithm is also reminiscent of the level-wise Apriori approach, and they mine induced subtrees only. There are several other recent algorithms that mine different types of tree patterns, which include FreeTreeMiner [4] which mines induced, unordered, free trees (i.e., there is no distinct root); SingleTreeMining [21], which mines rooted, unordered trees, with application to phylogenetic tree pattern mining; and PathJoin [26], uFreqt [18], uNot [2], and HybridTreeMiner [5] which mine induced, unordered trees. CMTreeMiner [6] mines maximal and closed induced, unordered trees. TreeFinder [22] 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. Dryade [23] is a recent method to mine embedded unordered subtrees, with the restriction that no two siblings have the same labels. In recent work, Zaki [31] proposed SLEUTH, the first complete algorithm to mine embedded unordered trees. Our focus here is on an efficient, unified approach to mine the complete set of frequent, induced/embedded, ordered/unordered trees.

Frequent tree mining is also related to tree isomorphism [20] and tree pattern matching [7]. 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. Here we are interested in enumerating all common subtrees in a collection of trees.

There has also been recent work in mining frequent graph patterns. The AGM algorithm (see [12] and Chapter 10 of this book) discovers induced (possibly disconnected) subgraphs. The FSG algorithm (see [14] and Chapter 7 of this book) 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 [27, 28] (see also Chapter 6 of this book). Another method uses a candidate generation approach based on Canonical Adjacency Matrices [11]. GASTON [17] adopts an interesting step-wise approach using a combination of path, free tree and finally graph mining to discover all frequent subgraphs. The SUBDUE system (see [8] and Chapter 8 of this book) also discovers graph patterns using the Minimum Description Length principle. An

approach termed Graph-Based Induction (GBI) was proposed in [29], which uses beam search for mining subgraphs. In contrast to these approaches, we are interested in developing efficient, complete algorithms for tree patterns.

1.4 GENERATING CANDIDATE SUBTREES

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 at 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 in Section 1.5. We begin with the problem of candidate generation in this section.

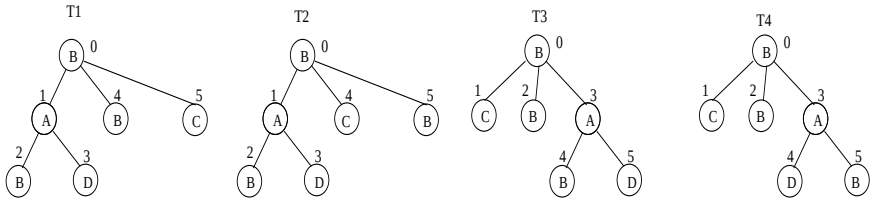


Figure 1.2. Some Automorphisms of the Same Tree

An *automorphism* of a tree is an 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 1.2. shows some of the automorphisms of the same tree. On the other hand, ordered trees have either only one automorphism (the trivial one that maps T to itself), or if there are several of them, they are indistinguishable (for example, when some node has at least two identical subtrees). Whether a tree is induced or embedded does not impact its automorphism group.

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 [18], 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 [6].

Lemma 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 1.2.. 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 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. ■

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 trees [30, 1]. SLEUTH follows the same strategy for ordered, embedded trees. 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 1.3. 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.

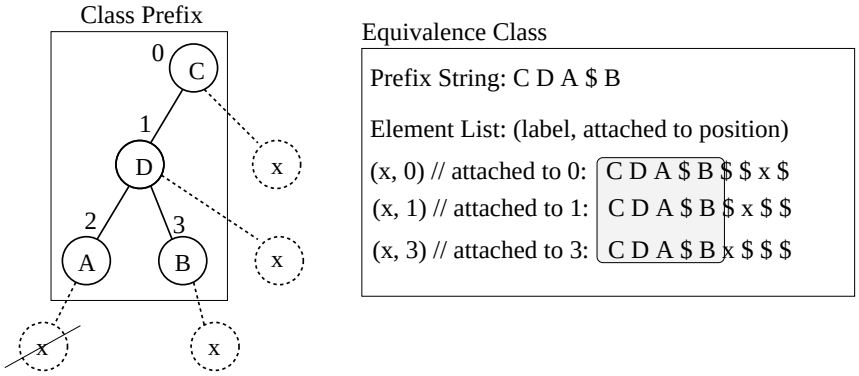


Figure 1.3. Prefix Extension and Equivalence Class

In [18] 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 retain-

ing only canonical forms for each automorphism group, we can enumerate all unordered trees non-redundantly. For each candidate, we can count the number of 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.

1.4.1 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.

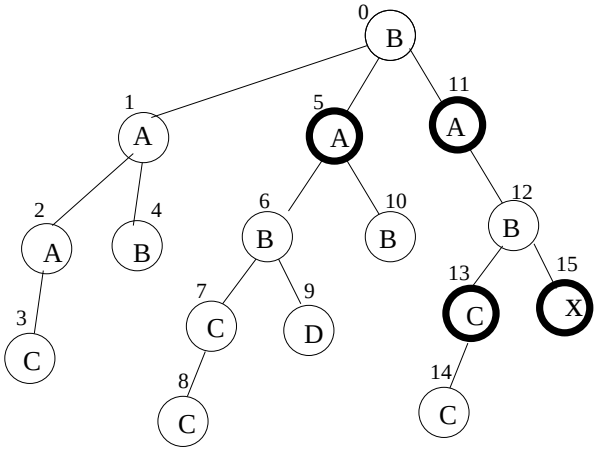


Figure 1.4. Check for Canonical Form

Lemma 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 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

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 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 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 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 1.4. 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 [18], but here we will continue to use the string encoding of a tree. The corresponding checks for canonicality based on lemma 1 among the subtree encodings are shown below:

T(13) vs. T(15): BAAC$$B$$ABCC $$D $$B $$ABCC $$x

T(5) vs. T(11): BAAC$$B$$ABCC $$D $$B $$ABCC $$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 edges F_2 (or the frequent labels F_1), we can then try to extend P with each edge from F_2 (or each item in 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, and many of the candidates may not be frequent. 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.

1.4.2 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 1.3. 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) *descendant extension*: If $j = i$ then $(y, k-1) \in [P_x^i]$.

Consider Figure 1.5., 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 descendant 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 descendant extension, corresponding to trees C_4 and C_5.

1.4.3 Discussion

Note that for ordered and embedded tree mining, we can guarantee that each equivalence class is complete, i.e., all potential embedded, ordered $(k+1)$-subtrees can be obtained by

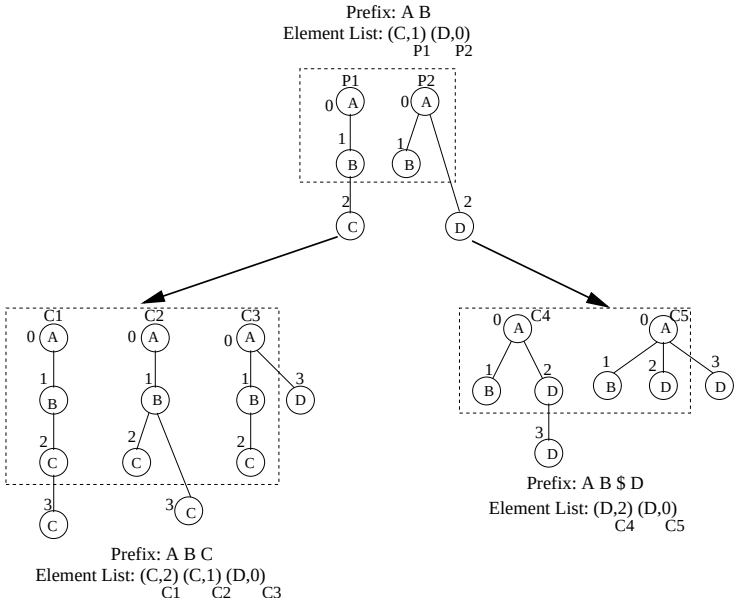


Figure 1.5. Equivalence Class-based Extension

joining two embedded, ordered k-subtrees in the class. For unordered subtrees, we have to allow non-canonical subtrees, to preserve completeness. Thus any embedded, unordered $(k+1)$-subtree can be obtained by joining two embedded, unordered k-subtrees in the class, where the tree being extended is canonical. For induced subtrees also the equivalence class is not complete, if we only keep induced subtrees as class members. For example, in Figure 1.1., there are two induced 2-subtrees with B as a root, namely, $BA\$$ and $BC\$$. Thus $[B] = \{(A,0),(C,0)\}$. It is clear that we cannot obtain the pattern $BAB\$\$$ by joining only two elements within the class $[B]$. To guarantee completeness, we must allow embedded subtrees to be class members. For example, if we add the embedded pattern $BB\$$ to $[B]$, then we will be able to obtain $BAB\$\$$ by joining only two elements within the class $[B]$. In general we can obtain any induced, (un)ordered $(k+1)$-subtree, from two k-subtrees within a class, provided the tree being extended is both canonical and induced. Thus the main observation behind equivalence class extension is the $F_k \times F_k$ candidate generation process, where only known frequent k-elements from the same class are used for extending P_x^i. Furthermore, we only extend P_x^i, if it is in canonical form, and satisfies the given tree properties. However, to guarantee that all possible extensions are members of $[P]$, we have to relax the canonicality or induced requirements.

As opposed to equivalence class extensions, for pure canonical extensions, an equivalence class contains only canonical members, and only those members that satisfy the tree properties (embedded or induced). To guarantee that all possible extensions will be tried, we extend $[P_x^i]$ by considering all members of the form $(x, y) \in F_2$, which itself stores only canonical or embedded/induced elements as the case may require. Canonical extension thus corresponds to an $F_k \times F_2$ (or $F_k \times F_1$) candidate generation process. 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 (see Section 1.8). 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.

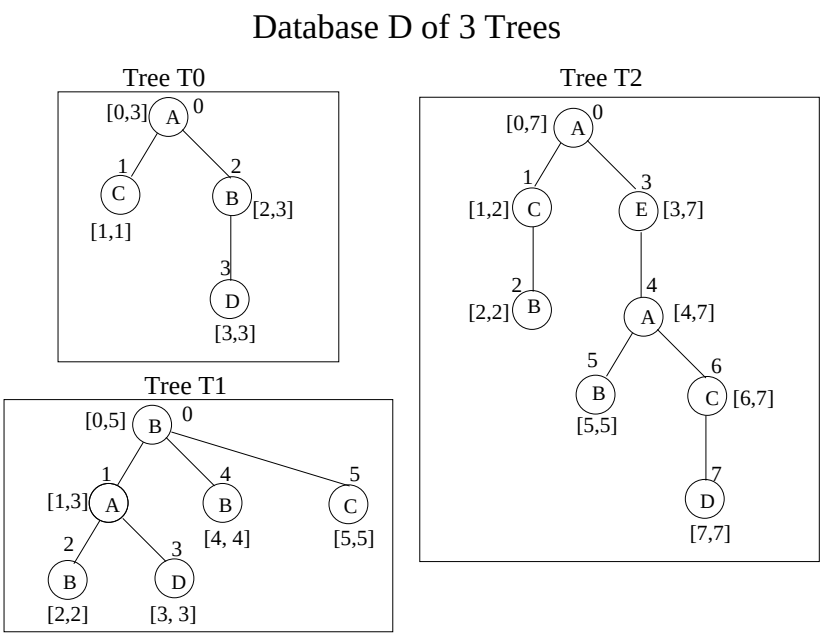
1.5 FREQUENCY COMPUTATION

The candidate generation step allow us to enumerate potentially frequent ordered/unordered subtrees in a systematic manner. The goal of the frequency counting step is to quickly find the support of a candidate. We first look at the task of finding the frequency of embedded subtrees, and then extend the method to compute the support of induced subtrees.

1.5.1 Embedded Subtrees

In SLEUTH, we represent the database in the vertical format [30], 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 1.6. shows a database of 3 trees, and the scope-lists 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.



D in Vertical Format: (tid, scope) pairs

A	B	C	D	E
0, [0, 3]	0, [2, 3]	0, [1, 1]	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 1.6. Scope-Lists

We also 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 the 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).

SLEUTH uses scope-list joins for fast frequency computation for a new embedded 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., descendant 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 (un)ordered extensions, using the descendant and cousin tests.

Descendant Test. Given $[P]$ and any two of its elements (x, i) and (y, j). In a descendant 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 [30], 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 $[P_x^i]$.

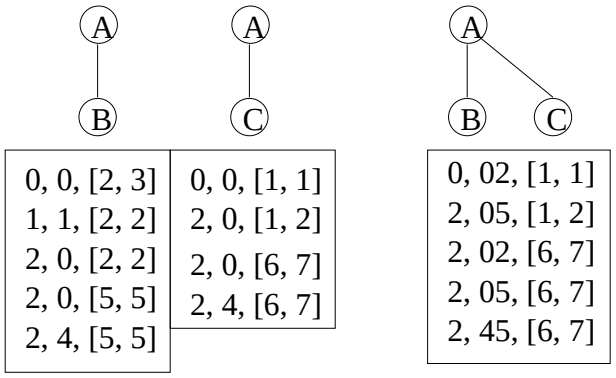


Figure 1.7. Scope-list Joins: Embedded

Figure 1.7. shows an example of how scope-list joins work, using the database D from Figure 1.6.. 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.

1.5.2 Induced Subtrees

For counting the support of only induced trees, SLEUTH extends the scope-list to be a five-tuple of the form (t, m, s, d, i), where in addition to the tid t, prefix match-label m, and last node scope s, we keep the last node's depth d (i.e., the number of edges on the path from the root to the given node), and a boolean flag i indicating whether this tuple contributes to the induced-support of the candidate. Initially, for single items, d is the actual depth of the item in tree t, but for k-subtrees ($k \geq 2$), d denotes the depth of the node in the candidate subtree. Figure 1.8. shows the single item scope-lists for induced mining; only the triples (t, s, d) are shown for single items, since the match-label $m = \emptyset$, and the induced flag $i = 1$ for all elements. For example, in tree T_o, A occurs at vertex 0 with scope $[0, 3]$ and at depth 0, in tree T_0, we add $(0, [0, 3], 0)$ to its scope list. In tree T_1, A occurs at node 1 with scope $[1, 3]$ and at depth 1, so we add $(1, [1, 3], 1)$ to its scope list, and so on.

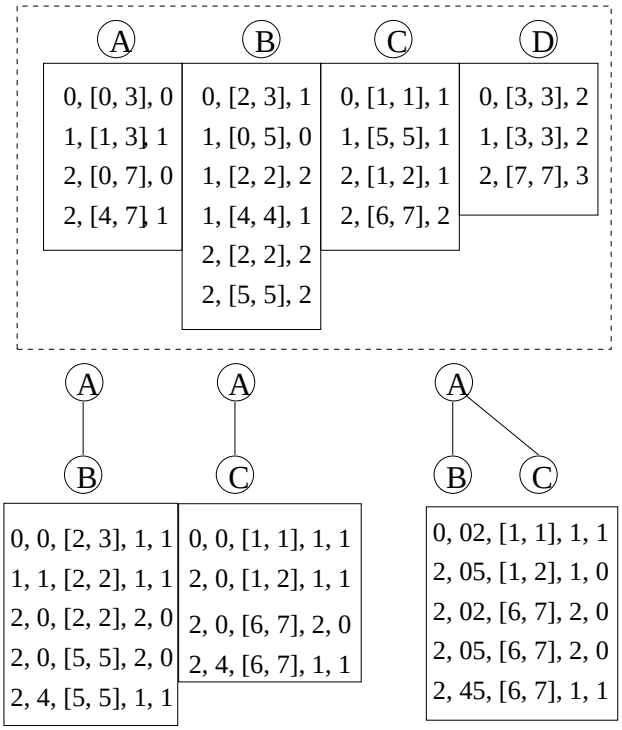


Figure 1.8. Scope-list Joins: Induced

Instead of cousin and descendant tests, for induced mining, we have to consider only sibling and child tests. Let $[P]$ be an equivalence class, let $(x, i) \in [P]$. For canonical extensions, let $(y, j) \in F_2$, and for equivalence class extensions, let $(y, j) \in [P]$.

Child Test. In a child extension of P_x^i the element (y, j) is added as a child of (x, i). For induced frequency computation, we first find all occurrences where label y occurs as a descendant of x, but we increment the support only for those tuples, where y is a direct child of x. Note that we keep all embedded occurrences to preserve the equivalence-class completeness property. Thus for induced support counting, like in the embedded case, we begin by searching if there exists tuples $(t_y, m_y, s_y, d_y, i_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x, d_x, i_x) \in \mathcal{L}(x)$, such that the tids ($t_y = t_x = t$) and match-labels ($m_y = m_x = m$) are equal, and y lies within the scope of x ($s_y \subset s_x$). In addition, we compute the difference in the depth of nodes y and x, $\delta = d_y - d_x$. If $i_x = 1$ then x represents an induced subtree, and therefore, if $\delta = 1$, then y must also be an induced extension of x. In this case we add the new tuple $(t_y, \{m_y \cup l_x\}, s_y, d, 1)$ to the scope-list of $(y, |P|)$ in $[P_x^i]$, where $d = \delta$ when transitioning from absolute depth of y in t to relative depth of y in the candidate, i.e., for 2-subtrees, otherwise $d = d_y$ (for $k > 2$). If $i_x \neq 1$, then y cannot be an induced extension of x, but rather is an embedded extension. In this case we add the new tuple $(t_y, \{m_y \cup l_x\}, s_y, d, 0)$ to the scope-list, but only if we are using equivalence-class extensions.

Cousin Test. In a cousin extension of P_x^i the element (y, j) is added as a cousin of (x, i). For induced support counting, we require that both y and x are induced extensions of the same parent node. That is we look for tuples $(t_y, m_y, s_y, d_y, i_y) \in \mathcal{L}(y)$ and $(t_x, m_x, s_x, d_x, i_x) \in \mathcal{L}(x)$, such that: $t_y = t_x = t$, $m_y = m_x = m$, and $s_x < s_y$ for ordered trees, and in addition if $s_x > s_y$ for unordered trees. Further, if $i_x = i_y = 1$, then it is an induced extension, and we add the tuple $(t_y, \{m_y \cup l_x\}, s_y, d_y, 1)$ to the scope-list of (y, j) in $[P_x^i]$. Otherwise, if we are using equivalence class extensions, we add $(t_y, \{m_y \cup l_x\}, s_y, d_y, 0)$ to the scope-list. Note that since a cousin test can only be applied to k-subtrees, with $k \geq 3$, the depth d_y is already relative to the root of the candidate.

Figure 1.8. shows an example of how induced scope-list joins work, using the database D from Figure 1.6.. It shows the initial scope-lists for the four frequent items. Consider the candidate $AB\$$ obtained by joining the scope lists of A and B and performing child tests. Consider $(0, [2, 3], 1) \in \mathcal{L}(B)$ and $(0, [0, 3], 0) \in \mathcal{L}(A)$. We find that $s_B = [2, 3] \subset s_A = [0, 3]$, and $\delta = d_y - d_x = 1 - 0 = 1$, which means it is an induced occurrence of the pattern, so we add the tuple $(0, 0, [2, 3], 1, 1)$ to the new scope list $\mathcal{L}(AB)$. As another example, for

T_2 we have $(2, [2, 2], 2) \in \mathcal{L}(B)$ and $(2, [0, 7], 0) \in \mathcal{L}(A)$, with $s_B = [2, 2] \subset s_A = [0, 7]$, and $\delta = d_y - d_x = 2 - 0 = 2$, which means it is only an embedded extension, so we add $(2, 0, [2, 2], 2, 0)$ to the new scope-list. In a similar manner we compute other elements of $\mathcal{L}(AB)$ and of $\mathcal{L}(AC)$. For induced support we only count those tuples with the induced flag $i = 1$. Thus the induced support of $AB\$$ is 3 (since there is at least one element with $i = 1$ for each t), and the induced support of $AC\$$ is 2. As an example of cousin testing, consider the scope-list of $AB\$C$, obtained by joining $\mathcal{L}(AB)$ and $\mathcal{L}(AC)$. For finding all (un)ordered induced occurrences, we need to test for disjoint scopes, with $s_B < s_C$ or $s_C < s_B$, which have the same match label, and where both tuples are induced. For example, for T_0, we find that the tuple $(0, 0, [2, 3], 1, 1)$ and $(0, 0, [1, 1], 1, 1)$, in $\mathcal{L}(AB)$ and $\mathcal{L}(AC)$, respectively, satisfy these conditions. Thus we add the tuple $(0, 02, [1, 1], 1, 1)$ to $\mathcal{L}(AB\$C)$. If we were interested only in ordered subtrees, this tuple would not be valid, since $[1, 1] < [2, 3]$. Note also that tuples $(2, 0, [5, 5], 2, 0)$ and $(2, 0, [1, 2], 1, 1)$, in $\mathcal{L}(AB)$ and $\mathcal{L}(AC)$, respectively, represent a sibling, rather than a cousin extension. So we add the tuple $(2, 05, [1, 2], 1, 0)$ to the new list, The other tuples are obtained similarly, and the induced support of $AB\$C$ is 2. In this example, we showed the scope-lists assuming equivalence class extensions; for pure canonical extension, all those tuples with $i = 0$ will not be added to the scope-lists.

1.6 COUNTING DISTINCT OCCURRENCES

SLEUTH is inherently a very efficient method for weighted support computation since it counts all embeddings of a frequent pattern within each database tree, using the scope-list joins. Many applications, however, may require only the support, i.e., instead of finding all embeddings of a subtree in the entire database, we may simply want to know the number of database trees that contain at least one embedding of a subtree. If there are relatively few embeddings per tree, SLEUTH continues to be very effective for support counting. On the other hand, if there are many duplicate labels, and if the tree is highly branched, the number of embeddings can get large, resulting in long scope-lists and increased running time. If the application calls for the use of weighted support, the increased cost is acceptable, but if we want only support, it is possible to optimize SLEUTH to count only distinct occurrences of each pattern.

To count only distinct occurrences SLEUTH uses a different scope-list representation for computing pattern frequency. It does not maintain the match-labels, which keep track

of all embeddings. Instead, it stores the scopes for all nodes on the right-most path within a tree; we call the new scope-lists as *scope-vector-lists* or *SV-lists* for short. Thus each element of the new list is a pair of the form $(t, \mathbf{s})$, where t is a tree id and $\mathbf{s} = \{s_1, s_2, \cdots, s_m\}$ is the scope-vector of matching node scopes s_i on the rightmost path. Furthermore, $\mathbf{s}$ represents a *minimal* occurrence of the pattern within a database tree, i.e., there does not exist another scope-vector $\mathbf{s}'$ strictly contained in $\mathbf{s}$ [3], such that the pattern also occurs at nodes with scopes given by $\mathbf{s}'$. Note that for induced mining, we can extend the tuple to be of the form $(t, \mathbf{s}, d, i)$, where d is the depth information, and i is an induced flag, as previously described in Section 1.5.2. For simplicity, we only illustrate the embedded case below; it is straight-forward to extend it to the induced case.

SV-List Joins

Given two trees (x, i) and (y, j) within the same equivalence class $[P]$, we perform SV-list as follows: Let $(t_x, \mathbf{s}_x)$ and $(t_y, \mathbf{s}_y)$ be any SV-list elements for nodes x and y, respectively. Let $\mathbf{s}_x = \{s_x^1, s_x^2, \cdots, s_x^m\}$ and $\mathbf{s}_y = \{s_y^1, s_y^2, \cdots, s_y^n\}$.

Descendant Test. For the descendant test we first make sure that $t_x = t_y$, i.e., both nodes x and y occur in the same database tree. Next, we look at the last node-scope of scope-vectors $\mathbf{s}_x$ and $\mathbf{s}_y$, namely s_x^m and s_y^n. If $s_y^n \subset s_x^m$, and there does not exist another last node-scope, say $s_x^{m'}$, in another element of x's SV-list, such that $s_y^n \subset s_x^{m'} \subset s_x^m$ (i.e., this is a minimal occurrence of the pattern), then we add the pair $(t_x, \mathbf{s}' = \{s_x^1, \cdots, s_x^k, \cdots, s_y^n\})$ new SV-list (where, $\mathbf{s}'$ represents the scope-vector for only those nodes on the rightmost path of the extended pattern).

Cousin Test. After checking $t_x = t_y$, we make sure that $s_x^m < s_y^n$ or $s_x^m > s_y^n$, i.e., the last nodes of each element are disjoint. Note that when extending P_x^i with (y, j) we obtain a new tree with prefix P_x^i, and which has y as the label of the rightmost node, attached to node j in the prefix. The next step in the cousin test is to compare the scopes at position j in both x and y, (i.e., s_x^j and s_y^j), and s_y^n. There are two cases to consider: a) $s_y^n \subset s_x^j$ and either $s_y^n > s_x^{j+1}$, or $s_y^n < s_x^{j+1}$, (i.e., the last node of y is contained within the j-th node of x (say, with label z), but it is not contained within the $(j+1)$-th node's scope), or b) either $s_y^n > s_x^j$ or $s_y^n < s_x^j$, and $s_x^j \subset s_y^j$, i.e., the last node of y is before or after the j-th

[3] We say that a scope-vector $\mathbf{s}' = \{s'_1, s'_2, \cdots, s'_n\}$ is contained within another scope-vector $\mathbf{s} = \{s_1, s_2, \cdots, s_m\}$ if $(s_1 < s'_1 \wedge s'_n \leq s_m)$ or $(s_1 \leq s'_1 \wedge s'_n < s_m)$

node of x and the j-th node of x is contained in the j-th node of y. If a) is true, then we add the pair $(t_x, \{s_x^1, \cdots, s_x^j, s_y^n\})$ to the SV-list of the new candidate, or if b) is true we add $(t_x, \{s_y^1, \cdots, s_y^j, s_y^n\})$. To maintain minimality, we store the pair only for the nearest j-th node to y in a database tree.

Figure 1.9. shows an example of how SV-list joins work, using the database D from Figure 1.6.. The initial SV-lists are the same as the item scope-lists in Figure 1.6.. While

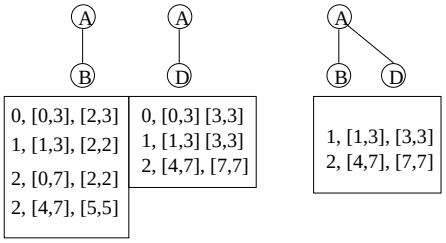


Figure 1.9. SV-List Joins

computing the new SV-lists for the subtrees $AB\$$ and $AB\$$, we have to perform only descendant tests. The key is to keep only minimal occurrences. For example, in tree T_2, the node scopes $[0, 7]$ and $[4, 7]$ for label A both contain the scope $[5, 5]$ for label B. In this case, the SV-list for $AB\$$ contains only the pair $(2, [4, 7], [5, 5])$. In a similar manner the complete SV-lists for both patterns are obtained, as shown in the figure. These two lists are joined to compute the frequency of $AB\$D\$$, using the cousin test. In our example, all tree ids belong to case a) of the cousin test. For example, for T_2, node label D has scope $[7, 7]$, whereas node label B has occurrences at scopes $[2, 2]$ and $[5, 5]$. Here $j = 0$ and thus D's scope $[7, 7]$ is contained in B's j-th node's scope $[0, 7]$, and also it is after B's $(j + 1)$-th node's scope $[2, 2]$. The cousin test is true, but it is not minimal, since the test is also satisfied for B's scope $[4, 7]$, and thus we add $(2, [4, 7], [7, 7])$ to the new candidate's SV-list.

1.7 THE SLEUTH ALGORITHM

Figure 1.10. 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 $\}$;
3. **for** all $[P]_1 \in F_2$ **do**
4. *Enumerate-Frequent-Subtrees*($[P]_1$);

ENUMERATE-FREQUENT-SUBTREES($[P]$):

5. **for each** element $(x, i) \in [P]$ **do**
6. **if** check-canonical(P_x^i) **then**
7. $[P_x^i] = \emptyset$;
8. **for each** element $(y, j) \in [P]$ **do**
9. **if** do-descendant-extension **then**
   $\mathcal{L}_d$ = descendant-scope-list-join($(x, i), (y, j)$);
10. **if** do-cousin-extension **then**
    $\mathcal{L}_c$ = cousin-scope-list-join($(x, i), (y, j)$);
11. **if** descendant or cousin extension is frequent **then**
12. Add (y, j) and/or $(y, k-1)$ & their scope-lists to equivalence class $[P_x^i]$;
13. Enumerate-Frequent-Subtrees($[P_x^i]$);

Figure 1.10. 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-FrequentSubtrees` $([P]_1 \in E)$ (line 4) we also compute the scope lists of all frequent elements (2-subtrees) in the class.

Computing $F_k (k \geq 3)$:. Figure 1.10. 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 (or SV-lists). Frequent subtrees are generated by joining the scope-lists (SV-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 6). 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 descendant and cousin extensions, and perform descendant or cousin tests during scope-list join or SV-list join (lines 9-10). 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 4 SLEUTH *correctly generates all possible induced/embedded, ordered/unordered, frequent subtrees.*

Equivalence Class vs. Canonical Extensions

As described above, SLEUTH uses equivalence class extensions to enumerate the frequent trees. For comparison we also implemented the pure-canonical extension in a method called SLEUTH-FK F2. The main idea is to extend a canonical and frequent (induced/embedded) subtree, with a known frequent (induced/embedded) subtree from F_2. The main difference is that `Enumerate-Frequent-Subtrees` takes as input a class $[P]$, all of whose elements are known to be both *frequent and canonical*. Each member (x, i) of $[P]$ is either extended with another element of $[P]$ or with elements in $[x]$, 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 descendant or cousin extension is canonical, 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, and is induced/embedded as the case requires.

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 Section 1.8 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.

1.8 EXPERIMENTAL RESULTS

All experiments were performed on a 3.2GHz Pentium 4 processor with 1GB main memory, and with a 200GB, 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 used the synthetic data generation program to create a database of artificial website browsing behavior [30]. The program constructs 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, the generator assigns 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 [30]. The logs touched 13361 unique web pages within the department's web site. After processing the raw logs 59691 user browsing subtrees of the CS department website were obtained. The average string encoding length for a user subtree was 23.3.

1.8.1 Performance Evaluation

We first compare four options for SLEUTH for different types of tree mining tasks, namely SLEUTH-EU (embedded, unordered), SLEUTH-EO (embedded, ordered), SLEUTH-IU (induced, unordered), and SLEUTH-IO (induced, ordered). Note that SLEUTH-EO is essentially the same as the TreeMiner algorithm [30] (which also uses vertical scope-lists to mine embedded, ordered trees). Figure 1.12. shows their performance on different datasets for different values of minimum support. Figure 1.11. also shows the length distribution of the different types of frequent trees for the highest minsup value.

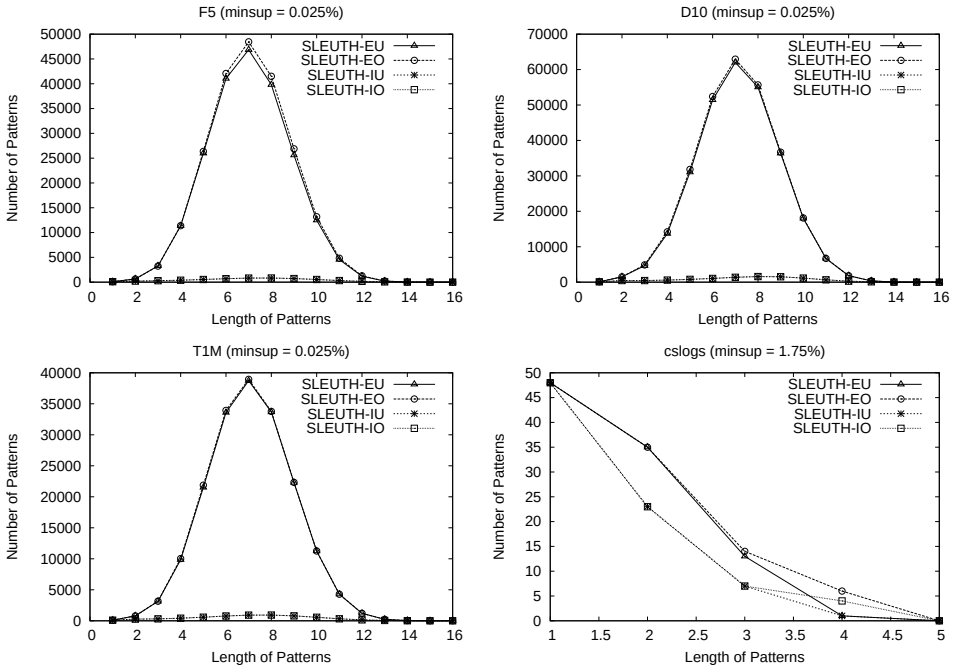


Figure 1.11. Distribution of Patterns

Let's consider the $F5$ dataset. We find that for embedded trees, there is a gap between unordered and ordered pattern mining. Ordered pattern mining (SLEUTH-EO) is slower, even though unordered pattern mining (SLEUTH-EU) needs to check for canonical forms, whereas SLEUTH-EO does not. The reason this happens is because there are more ordered rather than unordered frequent patterns for this dataset (as shown in Figure 1.11.). Looking at the length distribution, we find it to be mainly symmetric across the support values, and also, generally speaking more ordered tree are found as compared to ordered ones, especially as minimum support is lowered. Similar trends are obtained for the $D10$ and $T1M$ datasets.

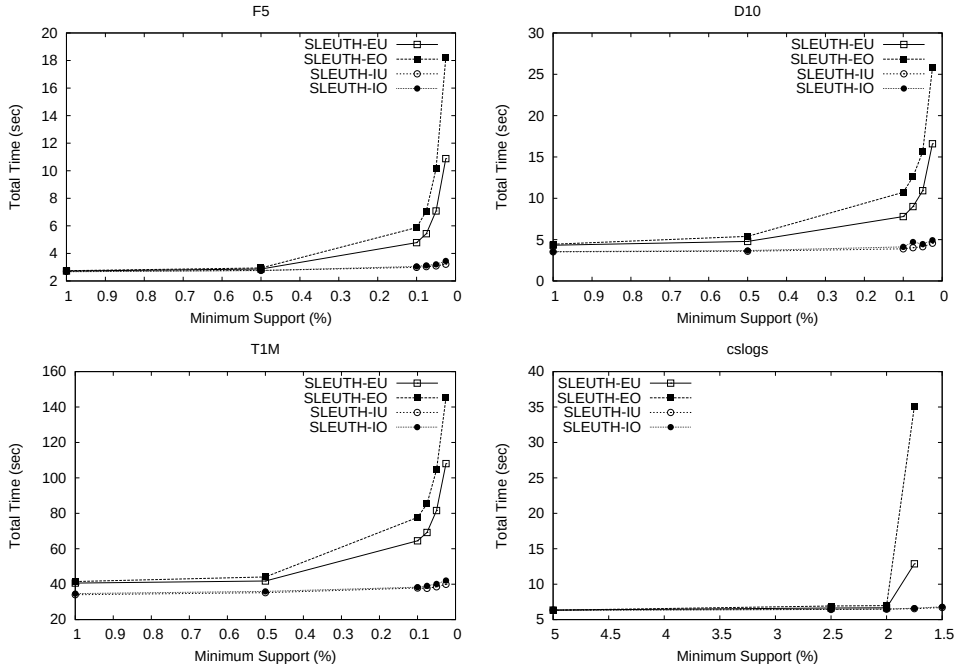


Figure 1.12. SLEUTH-EU vs. SLEUTH-EO vs. SLEUTH-IU vs. SLEUTH-IO

Comparing the induced, unordered (SLEUTH-IU) and ordered (SLEUTH-IO) mining methods, we see once again that SLEUTH-IU is slightly faster then SLEUTH-IO; the difference is not large, since there is little difference in the length distributions of these pattern types. Similar trends are obtained for the $D10$ and $T1M$ datasets.

Comparing embedded versus induced trees, we find a very big difference in the running times (embedded mining can be 4-5 times slower than induced mining). Once look at the length distributions explains why this is the case. We see that the number of induced

patterns is orders of magnitude smaller than the number of embedded patterns. The shape of the distribution is also symmetric.

The web-log dataset $CSLOGS$ has different characteristics than the synthetic ones. Looking at the pattern length distribution, we find that the number of patterns keep decreasing as length increases. Like before there are more ordered, than unordered patterns, and the timing trends remain the same as in the synthetic datasets. That is, SLEUTH-EO is slower than SLEUTH-EU, there is not much difference between SLEUTH-IO and SLEUTH-IU, and induced mining is faster than embedded mining.

Figure 1.13. compares SLEUTH-EO (which uses equivalence class extensions), with SLEUTH-D (which mines only distinct occurrences), and with SLEUTH-FKF2 (which uses pure canonical extensions). We evaluate the case only for embedded, ordered trees, since the results are similar for other pattern types.

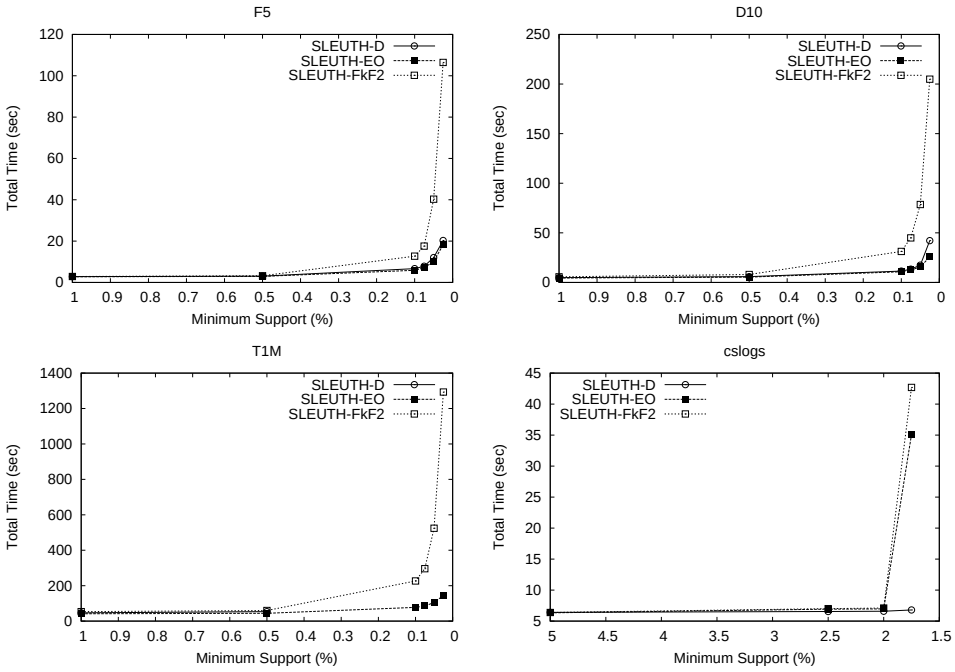


Figure 1.13. SLEUTH-EO vs. SLEUTH-D vs. SLEUTH-FKF2

Comparing SLEUTH-EO with SLEUTH-FKF2, for all the datasets, 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-EO. This shows clearly that the equivalence-class based strategy of gener-

ating some redundant candidates, but extending only canonical prefix classes is superior to generating many infrequent but purely canonical candidates. Comparing SLEUTH-EO with SLEUTH-D, we find that for the synthetic datasets, SLEUTH-D is slightly slower. This happens because, for these datasets, the number of possible embeddings is small, and SLEUTH-EO requires potentially smaller memory (since an SV-list element can be twice the size of a scope-list element). On the other hand, $CSLOGS$ has highly branched trees, and consequently, there are many embeddings, as support is lowered. In this case, we find that SLEUTH-D can be about 7 times faster than SLEUTH-EO, confirming that counting distinct occurrences is clearly beneficial when the number of mappings increases rapidly.

Summarizing from the results over synthetic and reals datasets, we can conclude that SLEUTH is an efficient, complete, unified algorithm for mining ordered/unordered, induced/embedded trees. Furthermore, it has optimizations to mine only distinct occurrences, and its equivalence class extension scheme is more effective than a pure canonical extension process.

1.9 TREE MINING APPLICATIONS IN BIOINFORMATICS

In this section we look at two applications of tree mining within the domain of bioinformatics: RNA structure and phylogenetic tree analysis.

1.9.1 RNA Structure

RNA molecules perform a variety of important biochemical functions, including translation; RNA splicing and editing; and cellular localization. Predicting RNA structure is thus an important problem; if a significant match to an RNA molecule of known structure and function is found, then a query molecule may have a similar role. Here we are interested in finding common motifs in a database of RNA structures [10].

Whereas RNA has a three-dimensional (3D) shape, it can be viewed in terms of its secondary structure, which is composed mainly of double-stranded regions formed by folding the single-stranded RNA molecule back on itself. To produce these double-stranded regions a subsequence of bases (made up of four letters: A,C,G,U) must be complementary to another subsequence so that base-pairing can occur (G-C and A-U). It is these pairings that contribute to the energetic stability of the RNA molecule. Moreover, bulges may also form, for example, when the middle portion of a complementary subsequence doesn't participate

in the base-pairing. Thus there are different RNA secondary structures that are possible, such as: single-stranded RNA, double-stranded RNA helix, stem and loop or hairpin loop, bulge loop, interior loop, junction and multi-loops, and so on [16]. In addition there may be tertiary interactions between RNA secondary structures, e.g., pseudo-knots, kissing hairpins, hairpin-bulge contacts, etc. Figure 1.14. shows a two-dimensional (2D) representation of a (transfer) RNA secondary structure. There are 5 loops (as numbered in the center); loop 1 is a bulge loop, 3, 4, and 5 are hairpin loops, and 2 is a multi-junction loop.

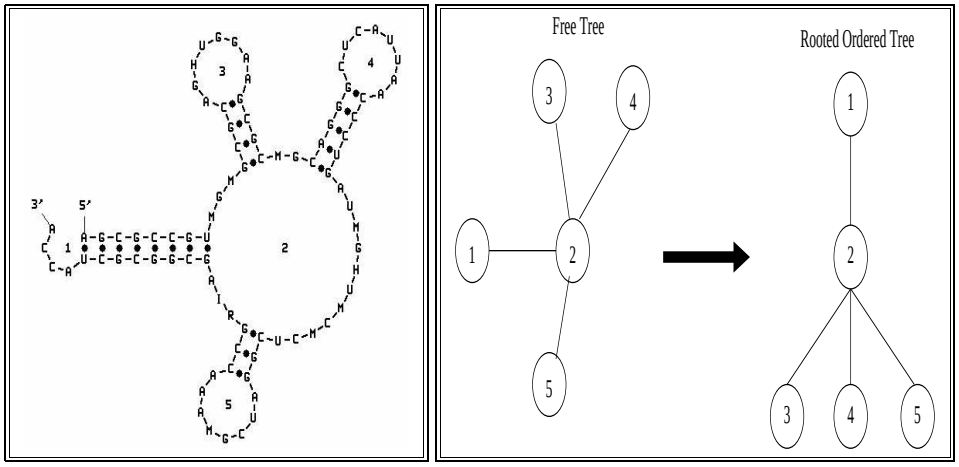


Figure 1.14. An Example RNA Structure and its Tree Representation

To mine common RNA motifs or patterns, we use a tree representation of RNA secondary structure obtained from the RNA Matrix method used in the RAG (RNA-as-graph) database [9]. In the RNA tree, a nucleotide bulge, hairpin loop, or internal loop is considered a vertex if there is more than one unmatched nucleotide or non-complementary base pair. The 3' and 5' ends of a helical stem are considered vertices, and so is a junction. An RNA stem with complementary base pairs (more than 1) is considered an edge. The resulting free tree captures the topological aspects of RNA structure. To turn the free tree into a rooted labeled tree, we label each vertex from 1 to n, numbered sequentially from the 5' to the 3' end of the RNA strand. We choose the root to be vertex 1, and children of a node are ordered by their label number. For example, Figure 1.14. shows the RNA free tree representing the RNA secondary structure, and its rooted version.

We took 34 Eukarya RNA structures from the Ribonuclease P (Rnase P) database [3]. Rnase P is the ribonucleoprotein endonuclease that cleaves transfer (and other) RNA precursors. Rnase P is generally made up of two sub-units, an RNA and a protein, and it is

the RNA subunit that acts as the catalytic unit of the enzyme. The RNase P database is a compilation of currently available RNase P sequences and structures. For a given RNase P RNA subunit, we obtained a free tree using the RNA Matrix program [4], and then converted it into a rooted ordered tree. The resulting RNA tree dataset has 34 trees, with the smallest having 2 vertices and the largest having 12 vertices. We then ran SLEUTH on this RNA tree dataset. Figure 1.15. shows the total time taken to mine the dataset and the number of patterns found at different values of minimum support. We observe that mining at minimum support of one occurrence took less than 0.1 seconds, and found 5593 total patterns. An example of a common topological RNA pattern is also shown (rightmost figure); this pattern appears in at least 10 of the 34 Eukarya RNA. By applying tree mining, it is thus possible to analyze RNA structures to enumerate all the frequent topologies. Such information can be a useful step in characterizing existing RNA structures, and may help in structure prediction tasks [10]. Enumerating frequent RNA trees also helps in cataloging the kinds of RNA structures seen in nature [9].

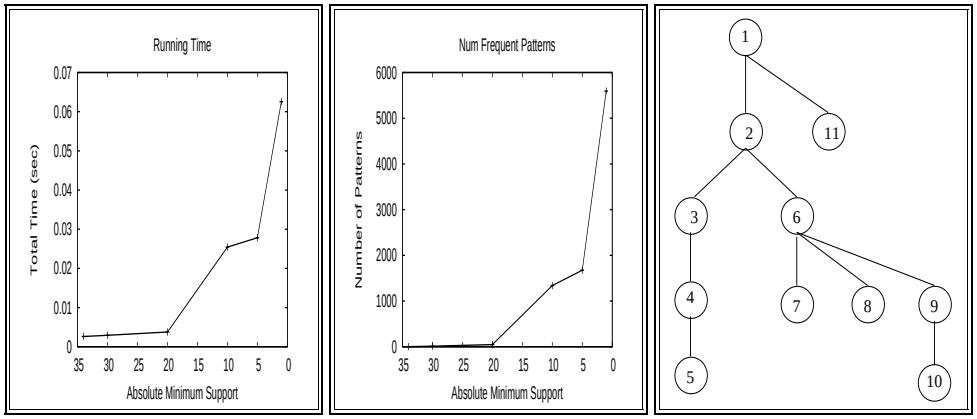


Figure 1.15. RNase P Database: A) Time, B) Num. Patterns, C) Example Pattern

1.9.2 Phylogenetic Trees

Given several phylogenies (i.e., evolutionary trees) from the Tree of Life [15], indicating evolutionary history of several organisms, one might be interested in discovering if there are common subtree patterns. This is an important task, since there are many algorithms for inferring phylogenies, and biologists are often interested in finding consensus subtrees (those shared by many trees) [19]. Tree mining can also be used to mine cousin pairs in

[4] http://monod.biomath.nyu.edu/rna/analysis/rna_matrix.php

phylogenetic trees [21]. A cousin pair is essentially a pair of siblings, and mining pairs that share common ancestors gives important clues about the evolutionary divergence between two organisms or species.

TreeBASE is a relational database designed to manage and explore information on phylogenetic relationships [5]. It stores phylogenetic trees and data matrices used to generate them from published research papers. It includes bibliographic information on phylogenetic studies, as well as details on taxa, methods, and analyses performed; it contains all types of phylogenetic data (e.g., trees of species, trees of populations, trees of genes) representing all biotic taxa. The database is ideally suited to allow retrieval and recombination of trees and data from different studies; it thus provides a means of assessing and synthesizing phylogenetic knowledge.

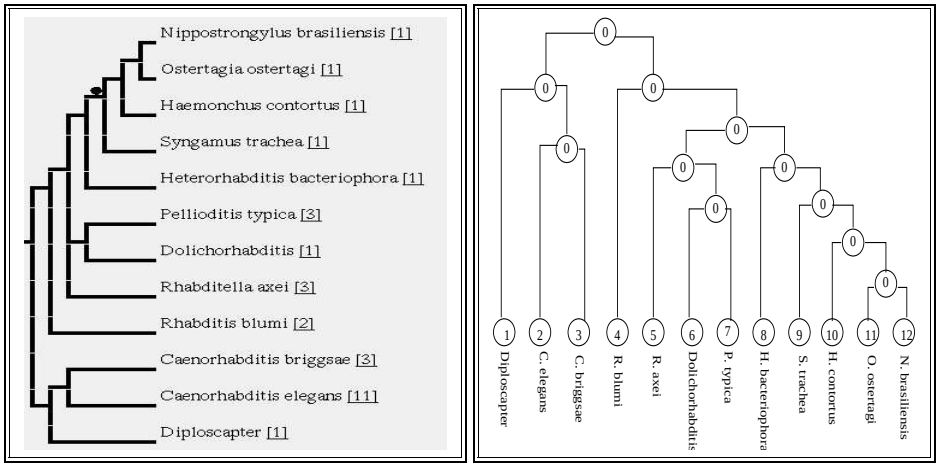


Figure 1.16. A) Part of Phylogenetic Tree of Phylum Nematoda, B) Tree for Mining

Figure 1.16. shows part of the evolutionary relationship between organisms of the plylum Nematoda taken from the TreeBase site. This tree was produced using a parsimony based phylogenetic tree construction method [16]; using different algorithms may produce several variants of the evolutionary relationships. Tree mining can help infer the parts of the phylogeny that are common among many alternate evolutionary trees.

We took 1974 trees from the TreeBase dataset, and converted them into a format suitable for mining. We give each organism a unique label (e.g., C. elegans has label 2), and we give each internal node the same label (e.g., 0). Given the resulting database of 1974 trees, we mine for frequent patterns. To prevent combinatorial effects, we also impose a constrain on

[5]http://www.treebase.org/

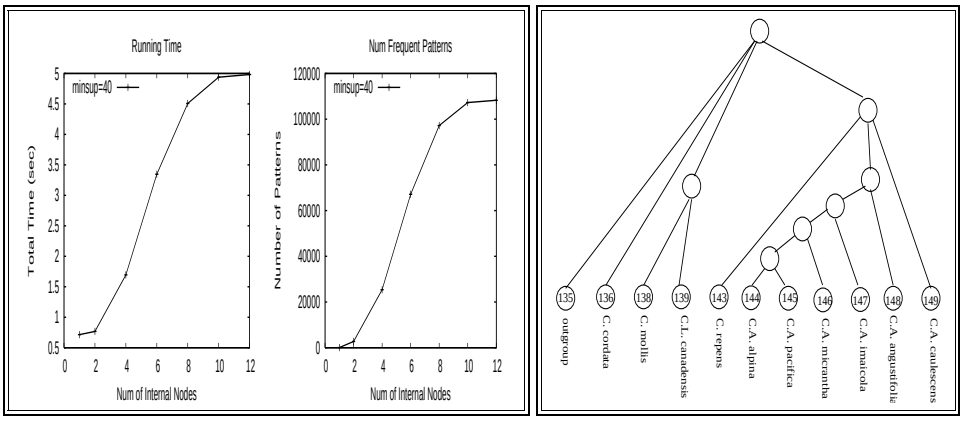


Figure 1.17. Phylogenetic Data: A) Runtime and Num. Patterns, B) Example Pattern

the number of internal nodes allowed in the mined patterns; this constrain is incorporated during mining for efficiency reasons (as opposed to post-processing). Figure 1.17. shows the running time and number of patterns found for an absolute support value of 40, as the number of internal nodes increase from 1 to 12. As we allow more internal nodes, more patterns are found. An example of a mined frequent pattern (with frequency 42) is also shown; this pattern shows the evolutionary relationship between members of the Circaea plant family. Notice how the most closely related organisms, e.g., Circaea Alpina (C.A.), group together (right branch under the root).

1.10 CONCLUSIONS

In this paper we presented SLEUTH, a unified algorithm to mine induced/embedded, ordered/unordered subtrees, and the procedure for systematic candidate subtree generation using self-contained equivalence prefix classes. All frequent patterns are enumerated by scope-list joins via the descendant and cousin tests. Our experiments show that SLEUTH is highly effective in mining various kinds of tree patterns. We studied two applications of tree mining: finding common RNA structures and mining common phylogenetic subtrees.

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 the extraction of structure from XML documents and their use in classification, clustering, and so on.

Acknowledgments

This work was supported in part by NSF Career Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, and NSF grants EIA-0103708 and EMT-0432098.

REFERENCES

1. 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.

2. T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *6th Int'l Conf. on Discovery Science*, October 2003.

3. J.W. Brown. The ribonuclease p database. *Nucleic Acids Research*, 27(1):314–315, 1999.

4. Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *3rd IEEE International Conference on Data Mining*, 2003.

5. Y. Chi, Y. Yang, and R. R. Muntz. Hybridtreeminer: An efficient algorihtm for mining frequent rooted trees and free trees using canonical forms. In *16th International Conference on Scientifi c and Statistical Database Management*, 2004.

6. Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *8th Pacifi c-Asia Conference on Knowledge Discovery and Data Mining*, 2004.

7. 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.

8. D. Cook and L. Holder. Substructure discovery using minimal description length and background knowledge. *Journal of Artifi cial Intelligence Research*, 1:231–255, 1994.

9. H.H. Gan, D. Fera, J. Zorn, N. Shiffeldrim, M. Tang, U. Laserson, N. Kim, and T. Schlick. RAG: RNA-As-Graphs database–concepts, analysis, and features. *Bioinformatics*, 20(8):1285–91, 2004.

10. H.H. Gan, S. Pasquali, and T. Schlick. Exploring the repertoire of rna secondary motifs using graph theory with implications for rna design. *Nucleic Acids Res.*, 31:2926–2943, 2003.

11. Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *IEEE Int'l Conf. on Data Mining*, 2003.

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. V. Morell. Web-crawling up the tree of life. *Science*, 273(5275):568–570, aug 1996.

16. D.W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Press, 2001.

17. S. Nijssen and J.N. Kok. A quickstart in frequent structure mining can make a difference. In *ACM SIGKDD Int'l Conf. on KDD*, 2004.

18. Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.

19. R.D. Page and E.C. Holmes. *Molecular Evolution: A Phylogenetic Approach*. Blackwell Science, 1998.

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

21. D. Shasha, J. Wang, and S. Zhang. Unordered tree mining with applications to phylogeny. In *International Conference on Data Engineering*, 2004.

22. A. Termier, M-C. Rousset, and M. Sebag. Treefinder: a first step towards xml data mining. In *IEEE Int'l Conf. on Data Mining*, 2002.

23. A. Termier, M-C. Rousset, and M. Sebag. Dryade: a new approach for discovering closed frequent trees in heterogeneous tree databases. In *IEEE Int'l Conf. on Data Mining*, 2004.

24. C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient pattern-growth methods for frequent tree pattern mining. In *Pacifi c-Asia Conference on KDD*, 2004.

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

26. Y. Xiao, J.-F. Yao, Z. Li, and M. H. Dunham. Efficient data mining for maximal frequent subtrees. In *International Conference on Data Mining*, 2003.

27. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE Int'l Conf. on Data Mining*, 2002.

28. X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, August 2003.

29. K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artifi cial Intelligence*, 75(1):63–92, 1995.

30. M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, July 2002.

31. M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52, 2005.

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