Mining Different Kinds of Trees: A Tree Mining Overview

Aída Jiménez   Fernando Berzal   Juan-Carlos Cubero
Department of Computer Science and Artificial Intelligence
ETSIIT, University of Granada, 18071 Granada, Spain
aidajm@decsai.ugr.es   fberzal@decsai.ugr.es   jc.cubero@decsai.ugr.es

Abstract

This paper surveys recent work on tree pattern mining and provides an overview of the state of the art in this emerging data mining field. A handful of noteworthy tree pattern mining algorithms are analyzed and they are compared from different points of view in order to highlight their similarities and recognize their dissimilarities.

1 Introduction

Non-linear data structures are becoming more and more common in data mining problems nowadays. Graphs, for instance, are commonly used to represent data and their relationships in different problem domains, ranging from web mining and XML databases to bioinformatics and computer networks. Trees, in particular, are amenable to efficient mining techniques and they have recently attracted the attention of the research community.

The aim of this paper is to describe the state of the art in tree pattern mining. We will try to give the reader a global view of the algorithms that have been developed in this area. We will analyze them from different perspectives in order to spot their commonalities and peculiarities.

Our paper is organized as follows. We introduce some standard terms in Section 2. Section 3 describes the frequent tree pattern mining process. Alternative tree representation strategies are summarized in Section 4. Section 5 tackles the candidate generation problem, i.e. how to identify all potentially frequent tree patterns. Section 6 surveys some of the better known tree mining algorithms and Section 7 concludes our survey with some pointers to future work in this area.

2 Tree Mining Terminology

In this section we introduce some basic concepts from graph theory. In particular, we will focus on labeled trees and use the notation from [4].

A **labeled graph** $G = (V, E, \Sigma, L)$ consists of a vertex set V, an edge set $E \subseteq V \times V$, an alphabet Σ for vertex and edge labels, and a labeling function $L : V \cup E \rightarrow \Sigma \cup \varepsilon$, where ε stands for the empty label.

A **path** is a sequence of consecutive edges between two nodes in the graph. A **cycle** is a path such that the first and the last vertices of the path are the same.

A graph is **directed** when each edge consists of an ordered pair of vertices. In this context, edges are referred to as **arcs**.

A graph is **acyclic** if it contains no cycles. It is **connected** if there exists at least one path between every pair of vertices.

Two graphs G_1 and G_2 are **isomorphic** if there exists a mapping function $\phi : V(G_1) \rightarrow V(G_2)$ so that, when u_1 and u_2 are adjacent in G_1, $\phi(u_1)$ and $\phi(u_2)$ are adjacent in G_2.

A subgraph S of a graph G is a graph such that V(S) $\subseteq$ V(G) and E(S)$\subseteq$ E(G).

A **tree** is a connected and acyclic graph. The size of a tree is defined as the number of vertices it has. In a tree, there is only one path between every pair of vertices. It is also easy to see that, in a tree, $\sharp E = \sharp V - 1$, where

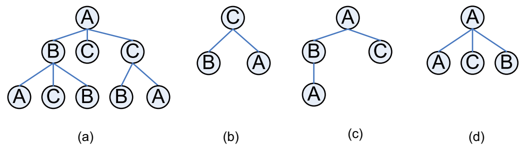

Figure 1: Different kinds of subtrees: (a) original tree, (b) bottom-up subtree, (c) induced subtree, (d) embedded subtree.

$\sharp E$ and $\sharp V$ represent the number of edges and vertices in the tree, respectively.

Different kinds of trees can be defined:

- **Rooted trees**: A tree is rooted if its edges are directed and a special node, called root, can be identified. The root is the node from which it is possible to reach all the other vertices in the tree.

  Rooted trees can be classified as:

  - **Ordered trees**, when there is a predefined order within each set of siblings.
  - **Unordered trees**, when there is not such a predefined order among siblings.

  In rooted trees, if a vertex v is on the path from the root to a vertex w, then v is an ancestor of w and w is a descendant of v. If $(v, w) \in E$, then v is the parent of w and w is a child of v. Sibling nodes share the same parent, while leaf nodes have no descendants. The depth or level of a node is the length of the path form the root to that node.

- **Free trees**: A tree is free if its edges have no direction, i.e. it is an undirected graph. Therefore, the tree has no predefined root.

A subtree could be defined as a subgraph of a tree. However, different kinds of subtrees can also be defined depending on the way we define the matching function between a pattern and a tree:

- **Bottom-up subtrees**: Intuitively, a bottom-up subtree T' of T (with root v) can be obtained by taking one vertex v from T with all of its descents and corresponding edges.
  In a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is a bottom-up subtree of T if and only if:

  1. $V' \subseteq V$
  2. $E' \subseteq E$
  3. The labeling of V' and E' in T is preserved in T'.
  4. If T is ordered, then the sibling ordering in T is preserved in T'.
  5. For every vertex $v \in V$, if $v \in V'$ then all descendants of v are also in V'.

- **Induced subtrees**: We can obtain an induced subtree T' from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T.
  Formally, we can define induced subtrees as bottom-up subtrees without the last constraint above, i.e. for any vertex $v \in V$, if $v \in V'$ there is no need for all its descendants to also be in V'.

- **Embedded subtrees**: An embedded subtree must not break the ancestor relationship among the vertices of T.
  In a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an embedded subtree of T if and only if:

  1. $V' \subseteq V$
  2. The labeling of V' and E' in T is preserved in T'.
  3. $(v_1, v_2) \in E'$ if and only if v_1 is an ancestor of v_2 in T.
  4. If T and T' are rooted ordered trees, then for $v_1, v_2 \in V'$, preorder $(v_1) <$ preorder (v_2) in T' if and only if preorder (v_1)<preorder (v_2) in T, where the preorder of a node is its index in the tree according to its preorder traversal.

3 Frequent Pattern Mining in Trees

The goal of frequent tree mining is the discovery of all the frequent subtrees in a large database of trees D, also referred to as *forest*, or in a unique large tree.

Let δ_T(S) be the occurrence count of a subtree S in a tree T and d_T a variable such that d_T(S)=0 if $\delta_T(S) = 0$ and d_T(S)=1 if $\delta_T(S) > 0$. We define the **support** of a subtree as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e, the number of trees in D that include at least one occurrence of the subtree S. Analogously, the **weighted support** of a subtree is defined as $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., the total number of occurrences of S within all the trees in D.

We say that a subtree S is **frequent** if its support is greater or equal than a minimum threshold threshold defined by the user. We define F_k as the set of all frequent subtrees with size k.

A frequent tree t is **maximal** if it is not a subtree of another frequent tree in T and it is **closed** if is not a subtree of another tree with exactly the same support in D.

Several frequent pattern mining algorithms for trees have been proposed in the literature. All of them, to our knowledge, follow the well-known Apriori [2] iterative pattern mining strategy, where each iteration is broken up into two distinct phases:

- *Candidate Generation*: Potentially frequent candidates are generated from the frequent patterns discovered in the previous iteration. Most algorithms generate candidates of size $k + 1$ by merging two patterns of size k having $k - 1$ elements in common. Different candidate generation strategies for trees are discussed in Section 5.

- *Support Counting*: Given the set of potentially frequent candidates, determine their support and keep only those candidates that are actually frequent.

Before we delve into the details of particular tree mining algorithms, we will survey how these algorithms internally represent trees.

4 Tree Representation

A canonical tree representation is a unique way of representing a labeled tree. This representation makes the problems of tree comparison and subtree enumeration easier.

In the following sections, we describe how different kinds of canonical representations allow for the efficient handling of different kinds of trees:

4.1 Canonical Representation of Rooted Ordered Trees

Three alternatives have been proposed to represent rooted ordered trees as strings:

- **Depth-first codification**: The string representing the tree is built by adding the label of the tree nodes in a depth first order, and a special symbol $\uparrow$, which is not in the label alphabet, when the sequence comes back from a child to his parent. The depth-first codification for the example tree in Figure 1(a) would be ABA$\uparrow$C$\uparrow$B$\uparrow\uparrow$C$\uparrow$CB$\uparrow$A$\uparrow\uparrow$.

- **Breadth-first codification**: Using this codification scheme, the string is obtained by traversing the tree in a breadth-first order, level by level. Again, we need an additional symbol $ that is not in the label alphabet in order to separate sibling families. The breath-first codification for the previous example is A$BCC$ACB$$BA.

- **Depth-sequence-based codification**: This codification scheme is also based on a depth-first traversal of the tree, but it explicitly stores the depth of each node within the tree. The resulting string is built with pairs (d, l) where the first element, d, is the depth of the node and the second one, l, is the node label. The depth sequence for the tree used in the previous examples is (0,A) (1,B) (2,A) (2,C) (2,D) (1,C) (1,C) (2,B) (2,A).

4.2 Canonical Representation of Rooted Unordered Trees

Given a rooted unordered tree, it is possible to derive several ordered trees from it, each one with its corresponding string representation. The canonical representation for an unordered tree is therefore defined as the minimum codification, in lexicographical order, of all the ordered trees that can be derived from it. You can use a depth-first, breath-first or depth-sequence-based codification scheme.

4.3 Canonical Representation of Free Trees

Free trees have no predefined root, but it is possible to select one node as the root in order to get a unique canonical representation of the tree.

The procedure typically used consists of repeatedly removing leaf nodes (with their incident edges) from the free tree until a single vertex or two adjacent vertices remain. In the first case, the tree is called centered. In the second case, the tree is called bicentered.

If a free tree is centered, the center node is uniquely identified and it is designated as the root node in order to obtain a rooted unordered tree, which can be represented as explained above. If the free tree is bicentered, two rooted unordered trees are obtained, each one with a different center as root node. The rooted unordered tree with the smaller string representation in lexicographical order is then selected as the free tree canonical representation.

5 Candidate Subtree Generation

The first step in Apriori-based pattern mining algorithms consists of generating a set of potentially frequent patterns. This set of candidate patterns must be a superset of the actual frequent pattern set. Several candidate generation alternatives have been proposed for trees. The following paragraphs describe some of the most important.

5.1 Rightmost Expansion

One of the most common candidate generation strategies, known as *rightmost expansion* consists of generating subtrees of size $k+1$ from frequent subtrees of size k by adding nodes only to the rightmost branch of the tree.

Let T be a tree pattern of size k and $rml(T)$ the rightmost leaf of T. The rightmost branch of T is the unique path from the root to $rml(T)$. A rightmost expansion of T is any pattern of size $k+1$ that can be obtained by adding a child node to the right of any node in the rightmost branch.

This technique is used, for instance, by FreqT [1].

Rightmost expansion using depth sequences

When using depth sequences, connecting a new node to the rightmost branch of the tree results in a new pair (d, l) added to the end of the depth sequence string.

The depth sequence of a node is defined as the segment of the canonical sequence that matches with the bottom-up ordered tree corresponding to the node. If a node n on the rightmost path has a left sibling and its depth sequence is a prefix of the depth sequence of his left sibling, we say that n is a prefix node. The prefix node in the least level of the tree is called the lowest prefix node, but not all trees have such a lowest prefix node.

As shown in [4], based on the demonstration by [3], a depth pair (d, l) can be concatenated at the end of a canonical depth sequence S to generate a valid candidate if and only if:

- If the tree L has a lowest prefix node, then $d \leq d'$, or $d = d' \wedge l \geq l'$. Given the length i of the depth sequence of the lowest prefix node, (d', l') is the $(i{+}1)$ pair in the depth sequence of the left sibling of the lowest prefix node.

- If the rightmost path has a node at depth d with label l', then $l \geq l'$.

This technique is used by uFreqT [8] and Unot [3].

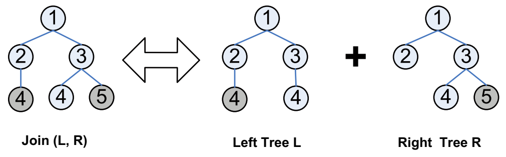

Figure 2: Right-and-left tree join.

5.2 Equivalence Class-based Extension

This technique is based on the depth-first canonical representation of trees. It generates a candidate $(k+1)$-subtree through joining two frequent k-subtrees.

Two k-subtrees are in the same equivalence class $[P]$ if they share the same codification string until the node $k-1$. Each element of the class can be represented by a single pair (x, p) where x is the k-th node label and p specifies the depth-first position of its parent.

Zaki [15] proves that all possible candidates can be generated by joining elements of the same equivalence class. Let (x, i) and (y, j) denote two elements in the same class $[P]$, and $[P_x^i]$ be the set of candidate trees derived from the tree that is obtained by adding the element (x, i) to P. The join procedure is defined as follows [14]:

1. *Cousin extension*: If $j \leq i$ and $|P| = k - 1 \geq 1$, then $(y, j) \in [P_x^i]$.
2. *Child extension*:
   If $j = i$ then $(y, k-1) \in [P_x^i]$.

This candidate generation technique is used by Zaki in TreeMiner [15] and SLEUTH [14].

5.3 Right-and-Left Tree Join

This technique uses both the rightmost and the leftmost leaves of a tree to generate candidates. The rightmost leaf was already defined in Section 5.1. The leftmost leaf of the tree T, $lmt(T)$, is the first leaf node in the preorder traversal of T. Let $Right(T)$ represent the right tree obtained by removing $lml(T)$ from T, and $Left(T)$ the left tree obtained by removing $rml(T)$ from T. Then, we define $Center(T)$ as the center tree where both $rml(T)$ and $lml(T)$ have been removed.

Let L and R be two trees where $Right(L) = Left(R)$. Their right-and-left tree join is defined as: $Join(L, R) = L \cup rml(R) = lml(L) \cup R$.

This join operation can be used to generate candidate patterns, even though serial trees cannot be obtained with this procedure. An additional serial extension is then used for completeness. This serial extension simply adds a frequent node v to the last node of a serial tree.

This technique is used by AMIOT [7].

5.4 Extension and Join

This candidate generation method was proposed by HybridTreeMiner [6]. It is based on the breadth-first codification of trees and defines two operations to generate candidates:

- *Extension*: Adding a new node v' to a leaf node of the subtree of height h yields a subtree of height $h+1$. The resulting tree must be in breadth-first canonical form to be considered.
- *Join*: Two subtrees that are siblings in the enumeration tree are joined to obtain a new candidate subtree. Some constraints are imposed to avoid duplicate candidate generation.

5.5 Apriori-based Generation

Trees can be described in a relational way by representing each tree T as a set of parent-child relationships $Rel(T)$ and their corresponding descendent-ancestor relationships $Rel+(T)$.

This technique uses the Apriori algorithm [2] extended to tree in order to extract frequent itemset in $Rel+(T)$

This technique is used by TreeFinder [10] to extract maximal subtrees but it is only complete if the trees that support the maximal frequent subtrees have no label pairs in common with those trees that do not support them.

5.6 Frequent Path Join

Another alternative consists of finding all maximal frequent paths and, then, mining the frequent subtrees by joining the frequent paths. Joining k maximal frequent paths results in subtrees with k leaf nodes.

This strategy, which is also similar to Apriori-based candidate generation, has been used by PathJoin [13].

6 Frequent Subtree Discovery Algorithms

Once we have described the basic building blocks of tree mining algorithms, we will analyze the implementation details of some relevant tree pattern mining algorithms.

Table 1 summarizes the algorithms we discuss in this paper by indicating the kinds of trees each algorithm is applicable to, as well as the kinds of patterns each algorithm is able to identify.

In our discussion, we group the algorithms according to the kinds of patterns they discover (induced, embedded, maximal) and, within each group, according to the kinds of trees they can work on.

6.1 Mining Induced Subtrees

6.1.1 Induced subtrees in ordered trees

FreqT

FreqT [1] uses the rightmost expansion strategy to generate candidate trees. In the support counting phase, it resorts to rightmost occurrence lists (RMO-lists). RMO-lists are stored with each candidate and contain (t, n) pairs, where n is the node position (or position) of each tree t in the database that matches with the $rml(T)$ node of the candidate pattern.

Update operations can be defined on these lists, as in AprioriTID [2], so that the RMO-list of a new k-pattern can be obtained from the RMO-lists of the $k-1$ patterns it is generated from using the rightmost expansion technique discussed in Section 5.1.

AmioT

This algorithm [7] is based on the right-and-left tree join operation described in Section 5.3. As in FreqT, the support counting phase is performed with the help of RMO-lists.

AMIOT, which stands for *Apriori-based Mining of Induced Ordered Trees*, cleverly uses ancillary lists for each candidate so that it avoids the generation of duplicate candidates. Its strategy is more efficient than the rightmost expansion used by FreqT.

6.1.2 Induced subtrees in unordered trees

uFreqT

This algorithm uses depth sequences to represent unordered trees in a canonical form and it performs rightmost expansions to generate candidates.

In the support counting phase for unordered trees, the main problem is to determine all the possible combinations of the children of each node v in the pattern that can match with the children of a node w in a database tree. The potential mappings can be represented by a bipartite graph G(v,w).

uFreqT uses a data structure that stores all the possible mappings of the graph for each node in the rightmost path along with pointers to database tree nodes with the aim of facilitating the support counting phase for new patterns generated after a rightmost expansion.

HybridTreeMiner

HybridTreeMiner [6] uses the breadth-first canonical codification of trees. The extension and join method is used to generate candidates (Section 5.4). Occurrence lists are also used for support counting.

HybridTreeMiner occurrence lists have elements with the form $(tid, i_1 \dots i_k)$ where tid is the tree identifier and $i_1 \dots i_k$ represents the mapping between the vertex indices in the pattern and vertex indices in the database tree.

6.1.3 Induced subtrees in free trees

HybridTreeMiner

HybridTreeMiner can also be adapted to free trees by using the canonical representation for

	Input trees			Discovered patterns		
Algorithm	Ordered	Unordered	Free	Induced	Embedded	Maximal
FreqT[1]	✓			✓		
AMIOT [7]	✓			✓		
uFreqT [8]		✓		✓		
TreeMiner [15]	✓				✓	
SLEUTH [14]		✓			✓	
Unot [3]		✓			✓	
TreeFinder [10]		✓			✓	✓
PathJoin [13]		✓		✓		✓
CMTreeMiner [12]	✓	✓		✓		✓
FreeTreeMiner [5]		✓	✓	✓		
HybridTreeMiner [6]		✓	✓	✓		

Table 1: Frequent tree mining algorithms.

free trees described in Section 4.3 and modifying the standard extension-and-join candidate generation method (Section 5.4). The join procedure is not modified, but extensions are only applied to centered trees when dealing with free trees (bicentered trees do not have to be extended).

FreeTreeMiner

FreeTreeMiner [5] is a previous Apriori-based frequent pattern mining algorithm for free trees. Candidates of size $k+1$ are generated by combining pairs of frequent trees of size k sharing $k-1$ vertices, as in the standard Apriori algorithm. Indexing techniques based on B+ trees and hash tables are used to speed up the support counting phase.

6.2 Mining Embedded Subtrees

6.2.1 Embedded subtrees in ordered trees

TreeMiner

TreeMiner [15] uses the depth-first codification of trees together with the class-based extension method to generate candidates (Section 5.2). Zaki also proposes how to prune the candidate generation phase in order to avoid duplicate candidates.

Scope-lists are used during the support counting phase. Each scope list stores the different occurrences of a pattern X within each database tree. The scope-lists are composed of triplets (t, m, s) where t is the tree identifier, m is the node label that matches with the $k-1$ prefix of X, and s is the scope of the last node of X. The scope of a node n is a pair $[a, b]$ where a is the position of the node in the depth-first enumeration and b is the depth-first position of its rightmost descendant.

Using the scope-lists, it is easy to determine the support of each pattern. Moreover, the scope-list for a new candidate is built by joining the scope-lists of the subtrees involved in the generation of the candidate.

6.2.2 Embedded subtrees in unordered trees

SLEUTH

As in TreeMiner, Zaki uses the depth-first canonical codification for representing unordered trees and scope-lists to support the support counting phase in SLEUTH [14]. In the case of unordered trees, however, not all the candidates generated by the class-based extension of a canonical frequent subtree are also in canonical form. Zaki proposes two alternatives to generate candidates:

- Using the *class-based extension mecha-*

nism, but checking if each subtree is canonical before extending it with the elements of its equivalence class.

- *Canonical extension*, which extends trees with the elements that belong to the same class and also with those belonging to the 2-subtree classes that share the node that is going to be extended, but only if the result is in canonical form.

As shown in [14], canonical extension generates non-redundant candidates, but many of them may not be frequent. On the other hand, class-based extension generates redundant candidates, but considers a smaller number of potential frequent candidate extensions. Experiments demonstrate that class-based extension is more efficient that canonical extension.

Unot

The Unot [3] algorithm uses depth sequences to canonically represent unordered trees. Rightmost expansion with depth sequences is used to generate frequent candidates. In order to speed up the support counting phase, Unot uses occurrence lists which are similar to those of Zaki's TreeMiner [1].

6.3 Mining Maximal Subtrees

6.3.1 Maximal Induced Subtrees

CMTreeMiner

The enumeration DAG (directed acyclic graph) is a lattice-like data structure that represents all frequent subtrees. Enumeration trees, which are commonly used for frequent subtree mining, are actually spanning trees of the enumeration DAG.

The enumeration DAG joins each frequent k-subtree t with the $(k-1)$-subtrees that can be extended to obtain t, as well as with $(k+1)$-subtrees that can be obtained by extending t.

[1] In the paper that proposed Gaston [9], an efficient graph mining algorithm, there is also a proposal describing how to extract embedded patterns from free trees using maximal length paths called "backbones".

This DAG can be used to prune the set of frequent patterns and generate only those patterns that are maximal. Pruning techniques have been devised for improving the efficiency of maximal subtree mining in rooted unordered trees [12].

PathJoin

This algorithm uses the path joining technique for maximal candidate generation (Section 5.6). PathJoin employs a specialized data structure, FST-Forest, that represents frequent trees in a compact form and supports the support counting phase of candidate patterns.

6.3.2 Maximal Embedded Subtrees

TreeFinder

The TreeFinder algorithm [10] uses the Apriori-based candidate generation technique based on ancestor-descendent relationships (Section 5.5).

Once frequent relationships are clustered by their support count, it is very easy to determine whether a candidate is frequent. However, it is important to note that this method is not complete: TreeFinder is an approximate miner. In the general case, it is only guaranteed to find a subset of the actual frequent trees.

DRYADE

DRYADE [11] searches for closed patterns in tree databases (remember that closed patterns are not necessarily maximal). The basic principle in DRYADE is to discover the closed frequent patterns of depth 1, and then to hook them together in order to build higher-depth closed frequent patterns in a level-wise fashion. DRYADE reformulates search operations in a propositional language. This way, DRYADE can benefit from any progress made in the field of closed itemset mining algorithms.

Algorithm	Tree Representation	Candidate Generation	Implementation Details
FreqT[1]	-	Rightmost expansion	RMO occurrence lists
AMIOT [7]	-	Left-right union	J_L/J_R lists
uFreqT [8]	Depth sequences	Rightmost expansion	Bipartite graphs
TreeMiner [15]	Depth-first codification	Equivalence class extension	Scope lists
SLEUTH [14]	Depth-first codification	Equivalence class extension	Scope lists
Unot [3]	Depth sequences	Rightmost expansion	Occurrence lists
TreeFinder [10]	Relational representation	Apriori itemset generation	Clustering techniques
PathJoin [13]	FST-Forest	Path join	FST-Forest
CMTreeMiner [12]	Depth-first codification		Enumeration DAG
FreeTreeMiner [5]	Breadth-first codification	Apriori	Indexing techniques
HybridTreeMiner [6]	Breadth-first codification	Union/extension method	Ocurrence lists

Table 2: Main features of different tree mining algorithms.

7 Conclusions and Future Work

Table 2 summarizes the main features of some relevant tree mining algorithms: their selected tree representation scheme, the way they generate candidates, and references to optimizations used in their implementation.

All the algorithms we have analyzed follow the candidate generation approach proposed by the Apriori algorithm [2]. The application of FP-Growth-like algorithms to tree mining is an open research problem (i.e. suppressing the explicit candidate generation phase from tree mining algorithms).

XML-related technologies are an important application domain for tree mining techniques. However, the algorithms proposed in the literature work either on ordered or unordered trees. They are unable to deal with trees whose nodes are partially ordered, a common situation when handling XML documents.

Finally, existing algorithms are based on a simplistic tree labeling scheme that is not valid (and should be extended) for many interesting problems, such as entity resolution, data integration, duplication detection, or reference reconciliation.

References

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *PKDD'2002*, pages 1–14, 2002.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'1994: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[3] T. Asai, H. Arimura, T. Uno, and S. ichi Nakano. Discovering frequent substructures in large unordered trees. In *DS'2003*, pages 47–61, 2003.

[4] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok. Frequent subtree mining - an overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.

[5] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, pages 509–512, 2003.

[6] Y. Chi, Y. Yang, and R. R. Muntz. Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, pages 11–20, 2004.

[7] S. Hido and H. Kawano. Amiot: Induced ordered tree mining in tree-structured databases. In *ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining*, pages 170–177, 2005.

[8] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences (MGTS2003), in conjunction with ECML/PKDD'2003*, 2003.

[9] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, 2004.

[10] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards xml data mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 450–457, 2002.

[11] A. Termier, M.-C. Rousset, and M. Sebag. Dryade: A new approach for discovering closed frequent trees in heterogeneous tree databases. In *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 543–546, 2004.

[12] Y. Xia and Y. Yang. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):190–202, 2005. Student Member-Yun Chi and Fellow-Richard R. Muntz.

[13] Y. Xiao, J.-F. Yao, Z. Li, and M. H. Dunham. Efficient data mining for maximal frequent subtrees. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, pages 379–386, 2003.

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

[15] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.