To See the Wood for the Trees: Mining Frequent Tree Patterns

Björn Bringmann

Lab for Machine Learning, Institute of Computer Science,
Albert-Ludwigs-University Freiburg,
Georges-Köhler-Allee 079, 79100 Freiburg, Germany
bbringma@informatik.uni-freiburg.de

Abstract. Various definitions and frameworks for discovering frequent trees in forests have been developed recently. At the heart of these frameworks lies the notion of matching, which determines if a pattern tree matches a tree in a data set. We compare four notions of tree matching for use in frequent tree mining and show how they are related to each other. Furthermore, we show how Zaki's TreeMinerV algorithm can be adapted to employ three of the four notions of tree matching. Experiments on synthetic and real world data highlight the differences between the matchings.

1 Introduction

In recent years, interest has grown in extending the frequent itemset paradigm to more expressive pattern types such as graphs, trees and sequences. Special attention has been devoted to semi-structured [1,2,3,4] and more specifically to tree-structured data [5,6,7]. These approaches aim at finding all frequent trees in a forest of rooted trees. They differ not only in the algorithms and implementation details, but more importantly also in the underlying notion of tree matching. When does one tree match another one? Asai *et al.* [5], Zaki [6] and Termier *et al.* [7] provide different answers to this question. Asai's notion is more restrictive than Zaki's, which is in turn more restrictive than Termier's. Termier *et al.* also have shown that it can be beneficial to work with more permissive notions of matching. However, this typically comes at a computational costs. Indeed, due to the expressiveness of their framework, Termier *et al.* cannot guarantee completeness, whereas the approaches of Zaki and Asai *et al.* are complete.

There are several important real-world applications for tree mining. First of all, consider the web usage mining problem [8]. Thousands of visitors maneuver through the well known web-sites like Amazon, Yahoo! and CNN each and every day. Most of these sites basically follow a hierarchical structure, i.e. a tree structure. Data Mining techniques created to handle trees can be used to gather information from the behavior of the visitors. The toy-example of an online shop shown in Figure (1) compares *tree embedding* and *tree incorporation* which we will discuss in paragraph 2.2. While the latter, more expressive definition yields only one maximally specific pattern, the notion of tree embedding yields two. According to tree embedding some visitors looked at the blouse and some at the Fulgoni purse. The single maximally specific pattern according to the notion of tree incorporation offers some more information: *The visitors looking at the blouse and the visitors looking at the purse are the same persons.* This knowledge might be helpful when restructuring an online-shop to improve accessibility or placement of advertisements.

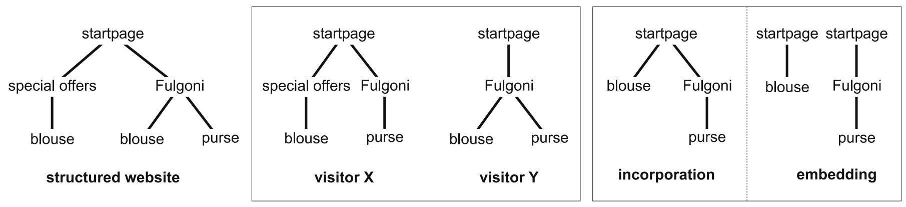


Fig. 1. Two *visitor* subtrees from a hierarchically structured web-site and the maximally specific patterns with regard to tree embedding and tree incorporation

XML has become a popular way of storing semi-structured data. As Goldfarb and Prescod write in their book [9]:

> *XML is a standardized notation for representing structured information. It is well-formed theoretically and is based on extensive industry experience. Although XML documents are simple, readily-transmitted character strings, the notation easily depicts a tree structure. A tree is a natural structure that is richer than a simple flat list, yet also respectful of cognitive and data processing requirements for economy and simplicity.*

XML data thus forms in general a 'source' for several important data mining domains. In bioinformatics tree structures arise as well. RNA structures essentially fold as trees. Newly sequenced RNA is compared with known RNA structures to draw conclusions about the functions of the RNA [10].

In this chapter, we compare the four notions *tree inclusion*, *tree embedding*, *tree incorporation*, and *tree subsumption* used in frequent tree mining and show how they are related to each other. Furthermore, we extend Zaki's TreeMinerV algorithm such that any of the first three named notions (i.e. all but *tree subsumption*) can be employed.

The chapter is organized as follows: Section 2 starts with definitions of trees and related objects. Using these, we formally define the four different notions of tree matching. In Theorem 1 we discuss the order among these notions. Section 2 ends with the definition of the tree-mining problem. In section 3, all concepts necessary for the tree-mining algorithm are described and the RETRO algorithm is explained. The following section discusses a novel pruning technique which reduces the memory consumption and time needed by the algorithm without sacrificing any patterns. In section 5, we show two different ways to extract the maximally specific patterns from the set of frequent patterns found. The experiments in section 6 give insight into the performance of the algorithm employing different notions and pruning techniques as well as into the amount of patterns on real and artificial data. Finally, in sections 7 and 8 we touch upon related work and conclude.

2 Matching Trees

There exist several different matching notions for trees. All notions use a mapping function to match the nodes of one tree onto another tree adhering to several constraints. We will first define *trees* and several concepts regarding trees. Based on those concepts, we then define the four notions of matching we discuss in this chapter.

2.1 Trees

A *graph* $G = (V, E)$ is a set of *vertices* V (i.e. *nodes*), connected by *edges* $E \subseteq V \times V$ (i.e. links, arcs). The *order* of a graph is the number of its vertices $|V|$. If each edge is an ordered pair of vertices, the graph is a *directed graph.* A graph is *undirected* if each edge is an unordered pair of vertices. A graph can have labeled vertices, as well as labeled edges. We will denote a label on a vertex $v \in V$ or edge $e \in E$ with $\lambda(v)$ and $\lambda(e)$ respectively. A sequence of vertices such that each of its vertices has an edge to its successor vertex is called a *path.*

A *free tree* is a graph in which every pair of vertices is connected by *exactly* one path. In a *rooted tree*, the edges are directed (i.e. a rooted tree is a directed graph) and every node has exactly one incoming edge, except one designated node v_0 called *root*, which has no incoming edge. Nodes that have no outgoing edges are called *leaves.* Every node that is not a leaf is an *inner node.* In a rooted tree, a node c is called a *child* node of p if $(p, c) \in E$. Dually, p is called *parent* of c, denoted as $p = \pi(c)$. If there is a path from a node a to a node d, a is called an *ancestor* of d, and d is called a *descendant* of a. Hence, the root node of a tree is an ancestor of all other nodes in the tree. We use $\pi^*(d) =_{\text{def}} \{\pi(d)\} \cup \pi^*(\pi(d))$ to denote the set of all ancestors of a node d. For a tree with order k we write k-tree. In this work we concentrate onto *rooted trees*, *free trees* are not investigated. Hence, we simply use *tree* to denote a *rooted tree.* The child nodes of a node can be ordered. To denote the order from left to right, we use an operator $\prec$. If the child nodes of a node are ordered, the tree is called an *ordered rooted tree.* We can now define a formal language $\mathcal{L}$ composed of all possible labeled, ordered, rooted trees.

Furthermore, we need a notion of the *scope* of a node. This is a very useful notion for tree mining since the following proposition holds:

Proposition 1 (Scope). *Given the scope of a node a in a tree it can be checked in constant time if a second node b of the same tree is an ancestor, descendant, left or right sibling of a.*

The scope of a node n is an interval in $\mathbb{N}$. The nodes in a tree are indexed with their *preorder index*, i.e. they are enumerated in a depth first manner as depicted in Figure (2). Thus, the root node has the index 0 and its leftmost child is vertex 1. The rightmost descendant of the k-tree (i.e. the rightmost leaf) has the index $k-1$, where k is the number of nodes in the tree. Using a function $\gamma(x)$ that returns the index of node x, the scope $[x_l, x_r]$ of a node can be defined as:

$$x_l =_{\text{def}} \gamma(x) \quad \text{and} \quad x_r =_{\text{def}} \begin{cases} c_r & \text{if } c \text{ is the rightmost child of } x \\ \gamma(x) & \text{if } x \text{ is a leaf.} \end{cases} \tag{1}$$

An example for the depth-first enumeration and the scope-definition is shown in Figure (2).

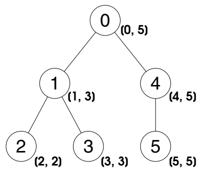


Fig. 2. A tree with its nodes labeled with their preorder index and each node annotated with its scope

2.2 Notions of Tree Matching

Previously [11], we presented three notions of tree matching and introduced also a novel one. In this section, we explain the differences and similarities between the four different notions in more detail.

First, all notions of *matching* map a tree $p = (V_p, E_p)$ onto another tree $t = (V_t, E_t)$, using a function $\varphi : V_p \to V_t$. In all four cases, the labels of the vertices are preserved: A vertex $v_p \in V_p$ can only be mapped to a vertex $v_t \in V_t$ if $\lambda(v_p) = \lambda(v_t)$, i.e. they both have the same labels.

The four notions have further similarities, but none is shared by all four of them. Figure (3) gives an example where the maximally specific patterns are

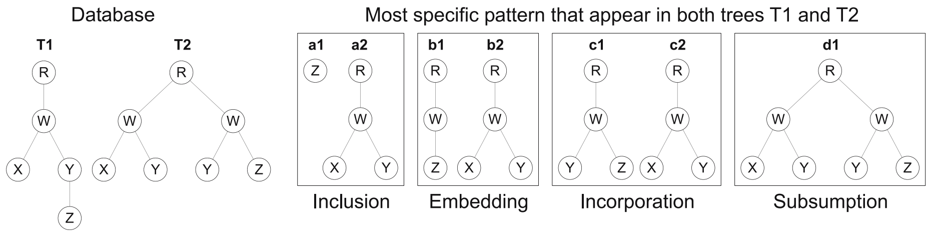


Fig. 3. Given a database consisting of the two trees $T1$ and $T2$ the four different notions yield four different sets of (maximally specific) patterns contained in both trees

different for all four notions. The most restrictive notion, called *tree inclusion*, states that a tree p is included in another tree t if there exists a subtree of t which is identical to p. It is defined as follows:

Definition 1. *Tree Inclusion*
A tree $p = (V_p, E_p)$ is included in a tree $t = (V_t, E_t)$, denoted as $match_{incl}(p, t)$, if there exists an injective mapping $\varphi : V_p \to V_t$ from the nodes of p to the nodes of t such that $\forall u, v \in V_p$

$$\begin{aligned}&\lambda(u) = \lambda(\varphi(u)) \wedge\\ &u \prec v \Leftrightarrow \varphi(u) \prec \varphi(v) \wedge\\ &\pi(u) = v \Leftrightarrow \pi(\varphi(u)) = \varphi(v).\end{aligned}$$

Tree inclusion has been extensively studied (in [12]) and can be decided in linear time. Asai *et al.* use this definition of matching in their algorithm FreqT. This notion might be to limited for several cases, but for other cases exactly this restrictiveness is required. Consider for example the representation of mathematical formulae as trees shown in Figure 4. In such a tree, each node corresponds to an operator or function and the leaves represent variables or numbers. Since the existence or absence of a single operation or function changes the whole meaning of the subtree, one would want patterns that preserve the parent-child relationship. Thus, more relaxed notions like *tree embedding* or *tree incorporation*, which allow to '*skip*' nodes in a pattern (and thus would extract A as a pattern for all three trees shown), are not useful here. Tree patterns in a set of formulae could be used to precalculate or optimize calculations that appear frequently.

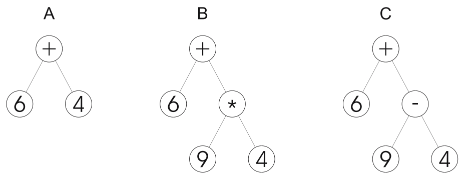


Fig. 4. In settings where a single node might influence the meaning of its whole subtree, tree inclusion will be the notion of choice

A more relaxed notion, called *tree embedding*, was first proposed in [13] and is based on an injective mapping preserving labels and ancestor-descendant relationships in the trees. In other words, we require that a parent-child relationship appears in the pattern p if and only if the two vertices are on the same path from the root to a leaf in the the tree t.

Definition 2. *Tree Embedding*
A tree $p = (V_p, E_p)$ is embedded in a tree $t = (V_t, E_t)$, denoted as $match_{emb}(p, t)$, if there exists an injective mapping $\varphi : V_p \to V_t$ from the nodes of p to the nodes of t such that $\forall u, v \in V_p$

$$\begin{aligned}&\lambda(u) = \lambda(\varphi(u)) \wedge \\ &u \prec v \Leftrightarrow \varphi(u) \prec \varphi(v) \wedge \\ &v \in \pi^*(u) \Leftrightarrow \varphi(v) \in \pi^*(\varphi(u)).\end{aligned}$$

Whereas the notion of tree inclusion is really useful when a node can change the meaning of its whole subtree, it will often be too restrictive if one deals with trees that contain some kind of additional or hierarchical information in the nodes which does not affect the meaning of nodes below. Consider for example a database of vehicles as in Figure 5, each represented as a hierarchical tree describing how the vehicle is composed of all its components. In such a case, trees may contain additional, more detailed information which is not available for all types of vehicles. Still, the make up of the components might be similar. Thus, a notion that allows to '*skip*' nodes can be very useful as shown in the example in Figure 5.

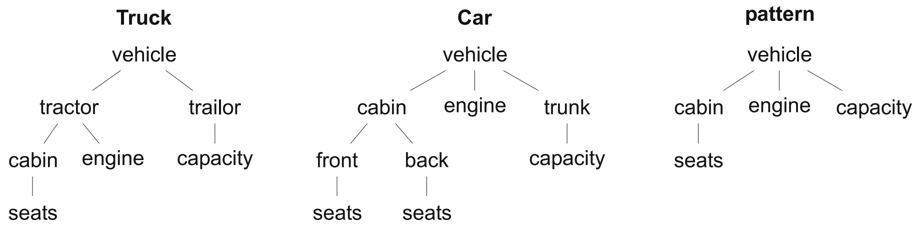


Fig. 5. Not enforcing the preservation of parent-child relationships allows to extract patterns that show more hidden similarities in the input data

Our definition, termed *tree incorporation*, is more relaxed than *tree embedding* since an ancestor-descendant relationship in the data does not have to hold in the pattern. Furthermore, it attempts to preserve the order among the children, but does not enforce it.

Definition 3. *Tree Incorporation*
A tree $p = (V_p, E_p)$ is incorporated in a tree $t = (V_t, E_t)$, denoted as match$_{icpr}$ (p, t), if there exists an injective mapping $\varphi : V_p \to V_t$ from the nodes of p to the the nodes of t such that $\forall u, v \in V_p$

$$\begin{aligned}&\lambda(u) = \lambda(\varphi(u)) \wedge \\ &u \prec v \Leftarrow \varphi(u) \prec \varphi(v) \wedge \\ &v \in \pi^*(u) \Rightarrow \varphi(v) \in \pi^*(\varphi(u)).\end{aligned}$$

The difference between *tree incorporation* and *tree embedding* can be described as follows. Let us consider two disjunct subsets X and Y of a set of trees $\mathcal{D}$ as shown in Figure (6), where all trees $x_i \in X$ and $y_j \in Y$ have two nodes labeled ψ and ϕ being descendants of a node labeled γ. Furthermore, in a tree $x_i \in X$, a node labelled ψ is an ancestor of the node labelled ϕ. For any tree of the other

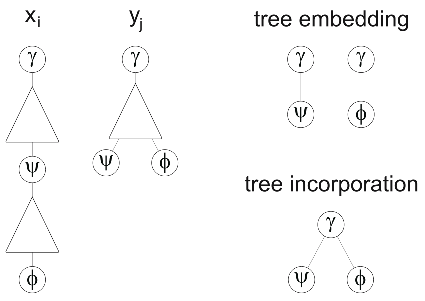


Fig. 6. The notion of *tree embedding* is more restrictive then *tree incorporation*, hence the latter one incorporates more information but less detail in the patterns found

subset Y, the nodes labelled ψ and ϕ are siblings, i.e. they have no ancestor-descendant relationship. Given an α such that $\max(|X|, |Y|) < \alpha \leq |X| + |Y|$, neither the pattern-tree where ψ is ancestor of ϕ nor the pattern-tree where ψ is a sibling of ϕ will be in at least α trees of the set $\mathcal{D}$. With regard to *tree embedding*, there are two tree-patterns with ψ being descendant of γ and ϕ being descendant of γ that match at least α trees of the set $\mathcal{D}$. In contrast, when using the notion of *tree incorporation*, there will be one tree-pattern where ψ and ϕ are siblings in at least α trees in $\mathcal{D}$. That way this result will show the information that the nodes labelled ψ and ϕ always appear in the same tree, while from the result using *tree embedding* this information cannot be obtained.

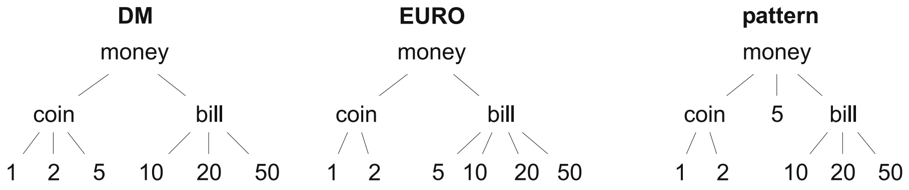


Fig. 7. The node labelled *5* denoting a bill and a coin respectively, appears as direct child of the root-node in the pattern

Furthermore, consider the example in Figure 7 where each tree represents the bills and coins of a currency. Both currencies have coins and bills with similar values. However, there is a 5 *Euro* bill but a 5 *DM* coin. Hence, the pattern that appears in both trees w.r.t. tree incorporation contains the node labelled *5* as child of the root node, since it can not be assigned to bill or coin in both cases. Using tree embedding, the pattern would or contain a *coin* and/or *bill* node or the node labelled *5* but never all three.

Finally, *tree subsumption* was introduced in [7]. It corresponds to representing the trees as relational formulae (cf. [14]), i.e. as a conjunction of all π^*, *edge* and *label* relations that hold in the tree. A tree then matches another tree if it θ-subsumes it. Theta-subsumption defined by Plotkin [15] is commonly employed in the field of inductive logic programming (ILP, cf. [16]) and relational learning. Termier *et al.* [7] use this notion of tree subsumption in their TreeFinder algorithm.

Definition 4. *Tree Subsumption*
A tree $p = (V_p, E_p)$ is subsumed by a tree $t = (V_t, E_t)$, denoted as $match_{sub}(p, t)$, if there exists a mapping $\varphi : V_p \to V_t$ from the nodes of p to the nodes of t such that $\forall u, v \in V_p$

$$\begin{aligned}&\lambda(u) = \lambda(\varphi(u)) \wedge \\ &v \in \pi^*(u) \Rightarrow \varphi(v) \in \pi^*(\varphi(u)).\end{aligned}$$

2.3 Order Among the Notions

The four notions given above are closely related. Indeed, the following theorem holds:

Theorem 1.

$$\begin{aligned}&\forall t, p \in \mathcal{L} \\ &match_{incl}(p,t) \to match_{emb}(p,t) \to match_{icpr}(p,t) \to match_{sub}(p,t)\end{aligned} \tag{2}$$

Proof($match_{incl} \to match_{emb}$):
Since a parent node is also an ancestor node, i.e. $\pi(x) \in \pi^*(x)$, it follows that:

$$\begin{aligned}&\text{if } (v = \pi(u) \Leftrightarrow \varphi(v) = \pi(\varphi(u))) \\ &\text{then } (v = \pi(u) \in \pi^*(u) \Leftrightarrow \varphi(v) = \pi(\varphi(u)) \in \pi^*(\varphi(u))).\end{aligned}$$

Hence, if a tree p is included in a tree t, it is also embedded in t. □

Proof ($match_{emb} \to match_{icpr}$):
Given $u \prec v \Leftrightarrow \varphi(u) \prec \varphi(v)$ and $v \in \pi^*(u) \Leftrightarrow \varphi(v) \in \pi^*(\varphi(u))$, it is obvious that also $u \prec v \Leftarrow \varphi(u) \prec \varphi(v)$ and $v \in \pi^*(u) \Rightarrow \varphi(v) \in \pi^*(\varphi(u))$ hold. Hence, a tree p embedded in a tree t is also incorporated in t. □

Proof ($match_{icpr} \to match_{sub}$):
Given an injective mapping φ from V_p to the nodes of V_t, such that $\forall u, v \in V_p$ $(\lambda(u) = \lambda(\varphi(u))) \wedge (u \prec v \Leftarrow \varphi(u) \prec \varphi(v)) \wedge (v \in \pi^*(u) \Rightarrow \varphi(v) \in \pi^*(\varphi(u)))$, it is obvious that only the part of the conditions required by *tree subsumption* will hold as well. Thus, if there exists a mapping φ such that a tree p is incorporated in a tree t it is also subsumed by t. □

2.4 Generality

On each of the matching notions, a more-general than (more-specific than) relation $\sqsubseteq$ ($\sqsupseteq$) can be defined. Given two trees $t, t' \in \mathcal{L}$ the relation can be defined as

$$t \sqsubseteq t' \Leftrightarrow \text{match}(t, t')$$

Hence, a tree t is called *more general* than a tree t', if and only if t is a pattern of t', according to the tree matching notion used. Corresponding to the four notions of matching there are four generality relations, i.e. for each $match_\chi$ there is a $\sqsubseteq_\chi$ with $\chi \in \{incl, emb, icpr, sub\}$. All four notions of generality induce a partial order on the pattern language $\mathcal{L}$. In contrast to the generality notion commonly used for frequent itemset mining[1], none of the four generality notions induces a lattice over $\mathcal{L}$. Figure (8) gives an example for this. In this example *two* elements S, S' are in the greatest lower bound of the trees T_1 and T_2 for any of the four notions. Since in a lattice the infimum (and supremum) are *unique*, this shows that no generality notion discussed here induces a lattice over $\mathcal{L}$.

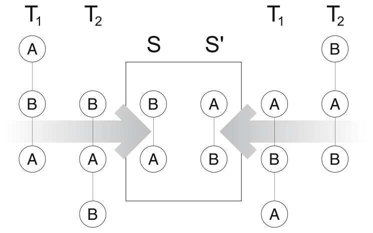


Fig. 8. An example that *none* of the four tree matching notions induces a lattice over $\mathcal{L}$. S and S' are maximally specific generalisations of the set $\{T_1, T_2\}$, i.e. there is no unique element that is the upper bound for this set.

2.5 Pattern Mining

After defining tree matching notions and therewith generality relations over $\mathcal{L}$, we can formalize the frequent tree mining problem. As stated before, let $\mathcal{L}$ be a formal language composed of all possible labeled, ordered, rooted trees and $\mathcal{D} \subseteq \mathcal{L}$ a database. To count the trees $t \in \mathcal{D}$ containing a pattern p with regard to a matching notion χ we define a function $d_{t,\chi} : \mathcal{L} \rightarrow \{0, 1\}$ as

$$d_{t,\chi}(p) =_{\text{def}} \begin{cases} 1 & \text{if } p \sqsubseteq_\chi t \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

that is 1 if the pattern p occurs at least once in the tree t, and 0 otherwise. The frequency of a pattern p in $\mathcal{D}$ can then be defined as

$$\sigma_{\mathcal{D},\chi}(p) =_{\text{def}} \Sigma_{t \in \mathcal{D}} d_{t,\chi}(p) \tag{4}$$

Using this definition, we now can define the task of *frequent tree mining*:

> Given a set of trees $\mathcal{D}$ and a minimum frequency α, the task of tree-mining is to find all patterns $p \in \mathcal{L}$ with regard to a matching notion χ such that $\sigma_{\mathcal{D},\chi}(p) \geq \alpha$ holds.

[1] In frequent itemset mining as introduced in [17], a pattern t is a set of items, and t is more general than t' if and only if $t \subseteq t'$.

All four matching notions can be used for searching patterns that occur with a minimum frequency.

Definition 5. *Antimonotonicity*
A constraint *or* selection predicate $q(p)$ *(such as minimum frequency* $\sigma_{\mathcal{D},\chi}(p) \geq \alpha$*) is* antimonotone *with regard to the specialization relation* $\sqsubseteq$ *if and only if*

$$\forall g \sqsubseteq s \Rightarrow (q(s) \Rightarrow q(g)). \tag{5}$$

The minimum-frequency predicate as defined above is antimonotone with regard to the specialization-relation $\sqsubseteq$, since for any pattern $g \sqsubseteq_{\chi} s$ it holds that $\sigma_{\mathcal{D},\chi}(g) \geq \sigma_{\mathcal{D},\chi}(s)$. Considering the framework introduced by Mannila and Toivonen [18], the theory of a database $\mathcal{D}$ with regard to a matching notion χ and a minimum frequency α can be defined as:

$$\mathcal{Th}(\mathcal{L}, \mathcal{D}, (\alpha, \chi)) = \{p \in \mathcal{L} \mid \sigma_{\mathcal{D},\chi}(p) \geq \alpha\}$$

As a consequence of Theorem 1, the set of frequent trees with regard to a data set $\mathcal{D}$ for *tree inclusion* is smaller than that for *tree embedding*, which is in turn smaller than that for *tree incorporation.* The set of frequent trees with regard to *tree incorporation* finally is smaller than that induced by *tree subsumption.* This motivates the use of the notion of matching as a parameter of frequent tree discovery tasks.

As pointed out by Mannila and Toivonen [18], a set $\mathcal{Th} \subseteq \mathcal{L}$ can be described by giving just the positive or the negative border. This helps to reduce the size for the representation of the solution. While the complete theory $\mathcal{Th}$ may contain thousands of patterns, the borders often contain only a small fraction of the patterns when compared to the whole version space. The *maximally specific set* $\mathbb{S}$, which is the same as the positive border ($\mathcal{B}d^{+}$), is defined as follows:

Definition 6. *Maximally Specific Set* - $\mathbb{S}$

$$\mathbb{S} =_{\text{def}} \{p \in \mathcal{Th} | \forall p' \in \mathcal{L} : p \sqsubset p' \Rightarrow p' \notin \mathcal{Th}\} \tag{6}$$

Minimum support is only one of several constraints that can be used to search for patterns occurring in a database. As long as the constraint is *antimonotone* with regard to the *more-general* relation, the resulting $\mathcal{Th}$ can be represented by its maximally specific set only. Hence, one could define constraints on tree-patterns like a maximum number of vertices, a maximum depth, or a maximum branching factor in a pattern. In addition to this extension, one can also use the generality relation $\sqsubseteq$ to define additional constraints. That will result in constraints like *maximally specific patterns.* I.e., given a set of maximally specific patterns $\mathbb{M}\text{ax} \subset \mathcal{L}$ there will be only patterns p in the result set such that $\forall m \in \mathbb{M}\text{ax} : p \sqsubseteq m$. Please note that the set $\mathbb{M}\text{ax}$ cannot always be reduced to just one tree, since the generality relation does *not* induce a lattice over $\mathcal{L}$.

Even further, *monotone constraints* could be used. I.e., a maximum support could be defined similar to the minimum support or a set of *maximally general patterns.* Similar to the *antimonotonicity* of constraints the *monotonicity* is defined.

Definition 7. *Monotonicity*
A constraint $q(p)$ is monotone *with regard to the specialization relation $\sqsubseteq$ if and only if*

$$\forall g \sqsubseteq s \Rightarrow (q(g) \Rightarrow q(s)). \tag{7}$$

If monotone constraints are used the corresponding border must be given to represent the resulting set $\mathcal{T}h$ in a compact way, i.e. only by its borders. This set $\mathbb{G}$ of *maximally general patterns* is defined analogue to the set $\mathbb{S}$ of *maximally specific patterns.*

Definition 8. *Maximally General Set* - $\mathbb{G}$

$$\mathbb{G} =_{\text{def}} \{p \in \mathcal{T}h | \forall p' \in \mathcal{L} : p' \sqsubset p \Rightarrow p' \notin \mathcal{T}h\} \tag{8}$$

Apart from the *maximum* and *minimum frequency* constraints, none of the constraints mentioned above requires to query the database. Following the notion of Ng *et al.* [19], constraints that do not require to query the database are called *domain constraints.* In the context of constraint-based mining one can see the notion of matching employed as providing a *constraint.* This type of constraint is comparable to the so-called *class constraints* mentioned by Ng *et al.*

Below we focus on the well known minimum frequency. Most of the other constraints can be treated in a similar way.

3 Mining Trees

Algorithms for mining frequent trees with regard to all four notions of tree matching exists. The algorithms for tree inclusion FreqT [5] and tree embedding TreeMinerV [6] work in a levelwise manner. Possible pattern trees are generated by extending known frequent trees with an additional node. After this extension, the supports for the new pattern trees are counted. The algorithm for mining incorporated trees RETRO [11] is an extension to the TreeMinerV algorithm for embedded trees. Thus, it works similarly, searching for frequent patterns in a levelwise manner.

However, the algorithm TreeFinder by Termier *et al.* [7] for mining frequent tree patterns is quite different. First, the trees are represented as relational formulae similar to representing trees within the Inductive Logic Programming (ILP) framework [16]. For every edge there is a binary predicate named after the labels of the nodes that are connected. The two arguments are unique identifiers of the connected nodes. After constructing all edge-predicates, their transitive closure is calculated. The algorithm then searches for all frequent ancestor-descendant relationships and clusters them. The resulting ancestor-descendant sets are re-transformed into edges from which a tree is constructed. For an in-depth explanation we refer the reader to [7].

The algorithm FreqT for searching included trees is not further described here, since the TreeMinerV algorithm will be extended in a way such that it can be used for mining frequent trees with regard to tree inclusion, tree embedding, and tree incorporation.

3.1 Systematically Enumerating Candidate Tree Patterns

As stated before, the algorithms for tree inclusion (FREQT), tree embedding (TREEMINER), and tree incorporation (RETRO) work in a levelwise manner. All three use a method called *rightmost expansion* to canonically enumerate all labeled, ordered, rooted trees. This technique was independently proposed by Zaki [6] and Asai *et al.* [5]. It works in a levelwise manner, adding a single node to a known frequent pattern in such a way that every possible candidate pattern will be generated *exactly* once. Thus the *rightmost expansion* is an optimal refinement operator, since every tree is enumerated but no tree is enumerated several times. Basically a k-tree is expanded to several $k+1$-trees P_i by adding new nodes only to its rightmost path as shown in Figure 9. The new node v_{k+1} in a pattern P_i is called *rightmost leaf* (`RML`) and the subtree without its `RML` is called *prefix* of the tree, denoted as $[P_i]$. The *rightmost path* of a tree is the path from the root node to the rightmost leaf. For efficient candidate generation the antimonotonicity of frequent patterns is used (i.e. a specialization s of a pattern p is not more frequent than p). Thus, we consider only frequent patterns for extension.

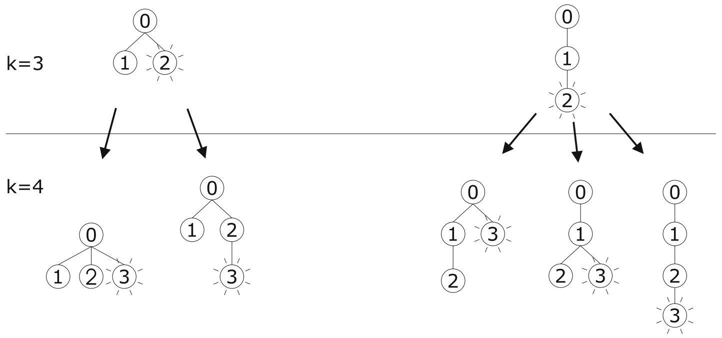


Fig. 9. All 3 and 4-trees of the enumerationtree. The nodes are labeled with their preorder index. New nodes attached at each level are marked.

3.2 Equivalence Classes and Instance Lists

With focus on the rightmost extension, patterns are organized in so called *equivalence classes* (`EQ`). An equivalence class contains all patterns that have the same prefix, i.e. differ in their rightmost leaves only. Each equivalence class contains the prefix $[P_i]$ only once and for each pattern P_i a tuple $\langle\lambda(\mathtt{RML}_i), \gamma(\pi(\mathtt{RML}_i))\rangle$ that contains the label $\lambda(\mathtt{RML}_i)$ of the new node and the index of the node it is attached to. If the `RML` is the root node of the pattern (i.e. the prefix is empty), this is denoted by a tuple $\langle\lambda(\mathtt{RML}_i), -1\rangle$. Since *prefix* and *equivalence class* denote essentially the same concept, $[P]$ is used to denote both, the prefix of a pattern P and the equivalence class that contains all patterns with the same prefix $[P]$. If there are no ambiguities, we use $\langle\lambda, \gamma\rangle$ to refer to a pattern P_i in $[P_i]$. With regard to the definition of *tree incorporation*, a pattern $\langle\lambda, j\rangle$ is a specialization of a pattern $\langle\lambda, i\rangle$ if $j > i$ and both patterns belong to the same equivalence

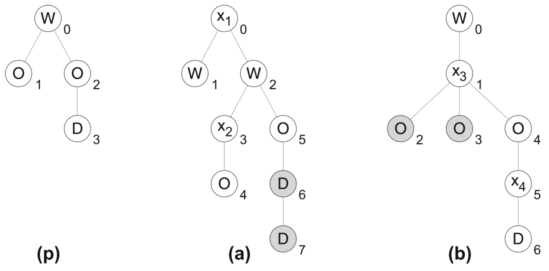


Fig. 10. A sample for four different instances of pattern p incorporated in a database consisting of two trees a and b. Both trees contain the pattern twice. In a the rightmost leaves of both instances are different, whereas in tree b their prefix is different. The grey marked nodes are the reason that there are two instances in each of the trees.

class. Hence, a function φ exists that maps the nodes of $\langle\lambda, i\rangle$ to the nodes of $\langle\lambda, j\rangle$ with respect to Definition 3. For example, for the two left-hand 4-trees in Figure (9), the right one is a specialization of the left one, but not vice versa.

To efficiently count the support of a pattern, the algorithm needs information about the instances in the data that support this pattern. Let pattern X be a k-subtree occurring in a tree T, φ the mapping from the pattern-nodes to the nodes of T, and x_k refer to the rightmost leaf of X. Following Zaki, we use $\mathcal{I}(X)$ to refer to the *instance-list* (also known as *scope-list*) of X. Each element of $\mathcal{I}(X)$ is a triple $\langle t, s, m\rangle$ identifying an instance of X where t is the identifier of the tree T the pattern X occurs in, $m =_{\text{def}} \{\gamma(\varphi(x_0)), \gamma(\varphi(x_1)), \ldots, \gamma(\varphi(x_{k-1}))\}$ is a list called *match label* of the prefix of X, and s is the scope of the node $\varphi(x_k)$, which the rightmost leaf of the pattern is mapped to. These instance lists contain all instances of a pattern with regard to *tree embedding.* An example is shown in Figure (10). Here the instance list for the pattern p is

$$\begin{aligned}\mathcal{I}(p) = \{\ &\langle a,\ [6,7], (2,4,5)\rangle,\\ &\langle a,\ [7,7], (2,4,5)\rangle,\\ &\langle b,\ [6,6], (0,2,4)\rangle,\\ &\langle b,\ [6,6], (0,3,4)\rangle\ \}\end{aligned}$$

For *tree incorporation*, we need the notion of *extended instance lists*. As stated before, a pattern $\langle\lambda, j\rangle$ is a specialization of a pattern $\langle\lambda, i\rangle$ if $j > i$ and both patterns belong to the same equivalence class. Hence, every instance that supports a pattern $\langle\lambda, j\rangle$ is also an instance that supports the pattern $\langle\lambda, i\rangle$. Using this information, the definition of an *extended instance list*, containing all instances that support a pattern X, is as follows:

$$\mathcal{I}^*(X) = \mathcal{I}^*(\langle\lambda, i\rangle) =_{\text{def}} \cup_{j\geq i}\mathcal{I}(\langle\lambda, j\rangle) \tag{9}$$

Given an equivalence class $[P]$, we use Zaki's *class extension* to obtain equivalence classes containing the successors of the patterns in $[P]$ with regard to the canonical enumeration scheme. The main idea is to consider each pair of patterns

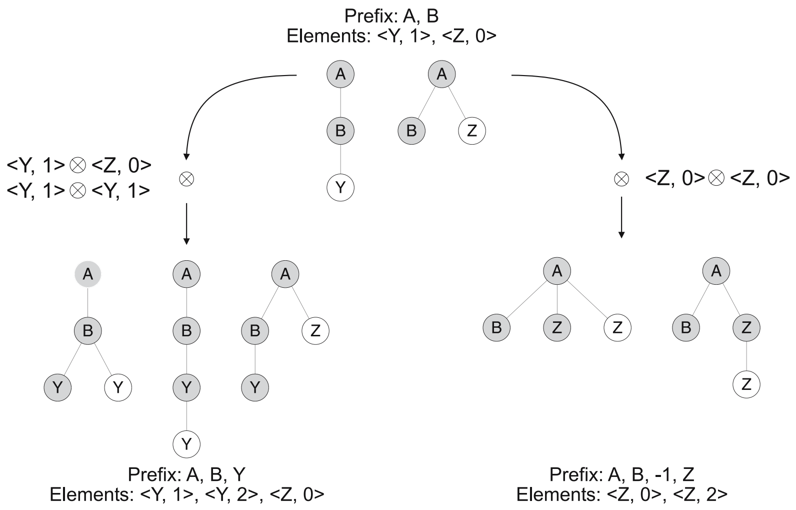


Fig. 11. An example for class extension. A class with two patterns is extended to obtain all its canonical successors. The grey nodes of each pattern represent the prefix common to all patterns in the equivalence class.

in the class for extension, including self extension. There can be up to two new candidates for each pair of patterns to be joined. Zaki's Theorem [6] formalizes this notion:

Theorem 2. *(Class Extension)*
Let $[P]$ be an equivalence class and let $\langle x, i\rangle$ and $\langle y, j\rangle$ denote any two elements in the class. Let $[P_x]$ denote the class representing extensions of element $\langle x, i\rangle$. We define a join operator $\otimes$ on the two elements, denoted $\langle x, i\rangle \otimes \langle y, j\rangle$, as follows:

$$\langle x, i\rangle \otimes \langle y, j\rangle =_{\text{def}} \begin{cases} \{\langle y, 0\rangle\} & \textit{if} \quad i = j = -1 \\ \{\langle y, j\rangle, \langle y, \gamma(x)\rangle\} & \textit{if} \quad i = j > -1 \\ \{\langle y, j\rangle\} & \textit{if} \quad i > j \\ \{\} & \textit{otherwise}\ (i < j) \end{cases} \tag{10}$$

Then all possible $(k+1)$-subtrees in $[P_x]$ with the prefix $[P]$ will be enumerated by applying the join operator to each unordered pair of elements $\langle x, i\rangle$ and $\langle y, j\rangle$.

As formalized in the join operation, there can be up to two outcomes of a join of two patterns. In Figure (11), showing the application of the join operator to an equivalence class consisting of two patterns, the self join of $\langle Z, 0\rangle$ results in two new patterns.

When two patterns A and B of the same equivalence class are joined, their instance lists are joined to obtain the instances that support the resulting patterns of the join. The corresponding operation is denoted $\cap_{\otimes}$. Let $R = \langle x, i\rangle \otimes \langle y, j\rangle$ denote the resulting set of patterns of the join. As stated before, there can be at most two elements (i.e. patterns) in R. In one of the elements, $\ddot{r}$, the node with label y is a sibling of x and in the other element, $\overset{\vdots}{r}$, y is a child of x. Either one of the two elements can be in R. Similarly, the join of the instance lists results in at most two new instance lists, one for each element in R. When joining the instance lists $\mathcal{I}(X)$ and $\mathcal{I}(Y)$ of two patterns X and Y, all pairs $x = \langle t_x, s_x, m_x \rangle \in X$ and $y = \langle t_y, s_y, m_y \rangle \in Y$ of instances are considered. For two instances to be joined, i.e. recombined to a new instance, several conditions have to hold. First, both instances have to appear in the same tree. Second, both instances have to be extensions of the same prefix occurrence. Finally, the scopes $s_x = [l_x, u_x]$ and $s_y = [l_y, u_y]$ of the rightmost leaves of the instances have to be considered. If s_y is contained in s_x, the rightmost leaf of y is a descendant of the rightmost leaf of x. In this case we have a new instance for the pattern $\overset{\vdots}{r}$. If the rightmost leaf of y is a sibling (to the right) of the rightmost leaf of x we have a new instance for $\ddot{r}$. The conditions for a new instance for a pattern $\ddot{r}$ are called *out-scope* test, whereas the conditions for a pattern $\overset{\vdots}{r}$ are called *in-scope* test.

Formally speaking, the *out-scope* test and the *in-scope* test for two instances $x = \langle t_x, s_x, m_x \rangle$ and $y = \langle t_y, s_y, m_y \rangle$ (with $s_x = [l_x, u_x]$ and $s_y = [l_y, u_y]$) are defined as follows:

Definition 9. *In-Scope Test*
Given two instances $x = \langle t_x, s_x, m_x \rangle$ and $y = \langle t_y, s_y, m_y \rangle$ we say that y is in the scope of x if the following conditions hold:

1. $t_x = t_y$
2. $m_x = m_y$
3. $l_x < l_y \wedge u_x \geq u_y$

Definition 10. *Out-Scope Test*
Given two instances $x = \langle t_x, s_x, m_x \rangle$ and $y = \langle t_y, s_y, m_y \rangle$ we say that y is outscope of x if the following conditions hold:

1. $t_x = t_y$
2. $m_x = m_y$
3. $u_x < l_y$

If a new instance z is added to either $\ddot{r}$ or $\overset{\vdots}{r}$, it is composed by combining the instances x and y such that $z =_{\text{def}} \langle t_y, s_y, m_y \oplus \gamma(\texttt{RML}(x)) \rangle$, where the operator $\oplus$ adds a new element to the end of the list. Thus, the node the rightmost leaf of x refers to is now part of the match of the new instance.

This notion works for *embedded* trees. For *incorporated* as well as for *included* trees, some minor changes and extensions have to be made.

For *tree incorporation* we have already introduced *extended instance lists*. These lists contain every instance supporting a pattern with regard to *tree incorporation*. Furthermore the conditions of an *out-scope* test have to be modified such that condition 3 reads:

$$u_x < l_y \vee (l_y < l_x \wedge u_y \geq u_x).$$

That way, an *out-scope* test holds if the node the `RML` of y is mapped to is a right-sibling *or* an ancestor of the node the `RML` of x is mapped to.

For *tree inclusion* there are no ancestor-descendant relationships allowed, but only parent-child relationships. Fortunately, this additional constraint can be incorporated into the algorithm in the following way. The prefix of a tree, i.e. the part of the pattern that is common to all patterns in the same equivalence class is not changed when a pattern is extended. So we have to make sure that every pattern prefix is consistent with the parent-child constraint, such that only the rightmost leaves are allowed to be in an ancestor-descendant relationship[2]. To achieve the parent-child consistency in the prefix, an extension $\langle x, i\rangle \otimes \langle y, j\rangle$ of a pattern $\langle x, i\rangle$ is permitted only if the rightmost leaf of $\langle x, i\rangle$ is a proper child of its parent in the instance both nodes are mapped to. To do that, we keep track of the preorder indices of the nodes in the match-part of a pattern instance. To refer to a single node in the match $m = \{\gamma(\varphi(x_0)), \gamma(\varphi(x_1)), \ldots, \gamma(\varphi(x_n))\}$, we use $m_{[i]} = \gamma(\varphi(x_i))$. Please note that the elements $m_{[i]}$ in the match m of an instance are ordered with regard to the prefix order of the nodes x_i in the pattern tree X , i.e. $m_{[i]} = \gamma(\varphi(x_i))$. In addition, the notion of an instance is changed to a quadruple $\langle t, s, m, p\rangle$ where p is the preorder index of the parent node of the rightmost leaf in the instance. The other three parts remain as before, i.e. t is the identifier of the tree, s is the scope of the rightmost leaf of the instance, and m is the match of the instance. The *in-scope* and *out-scope* tests are modified such that there is an additional condition 4 for both cases which requires:

$$m_{[k]} = p$$

where k is the number of nodes in the prefix, i.e. the number of elements in the match m.

3.3 The RETRO Algorithm

As stated before, the RETRO (F**re**quent **Tr**ee Disc**o**very) algorithm is a modification of Zaki's TreeMinerV algorithm. The main differences are the usage of the extended instance lists and the new condition for the *out-scope* test. The algorithm for computing frequent patterns with regard to tree incorporation is shown in Figure (12).

The first part of the algorithm computes the sets containing all frequent 1-trees (i.e. nodes) and 2-trees. Then the main loop starts by calling the function *Enumerate-Frequent-Subtrees* for every frequent 2-tree. The function *Enumerate-Frequent-Subtrees* generates all possible refinements of patterns in an `EQ` $[P]$. This is done by joining every pair $\langle x, i\rangle \otimes \langle y, j\rangle$ of patterns in $[P]$ including self-joins. Due to the rightmost expansion it is not allowed to join $\langle x, i\rangle \otimes \langle y, j\rangle$ with $i < j$ which would result in non-canonical expansions. A join results in one or two new patterns (R). Afterwards the respective instance lists are created by joining the instance lists of the patterns $\langle x, i\rangle$ and $\langle y, j\rangle$. Any new pattern that turns out to be frequent is added to the new equivalence class $[P_x]$. If all frequent

[2] This is imposed by the design of the algorithm.

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

Enumerate-Frequent-Subtrees($[P]$):
  for each element $(x, i) \in [P]$ do
    $[P_x] = \emptyset$
    for each element $(y, j) \in [P]$ with $i \geq j$ do
      $R = \{(x, i) \otimes (y, j)\}$;
      $\mathcal{I}(R) = \{\mathcal{I}^*(x, i) \cap_\otimes \mathcal{I}^*(y, j)\}$;
      if for any $p \in R$, $p$ is frequent then $[P_x] = [P_x] \cup \{p\}$;
    Enumerate-Frequent-Subtrees($[P_x]$);
```

Fig. 12. TreeMining Algorithm

patterns of the new equivalence class $[P_x]$ are computed further refinements of these patterns are generated. Thus, the algorithm proceeds depth-first.

The Figure (13) shows one path of the enumeration tree when the algorithm is applied to the database consisting of the two trees $(x), (y)$. The extended instance lists are required explicitly in refinement step 3. Without the extended instance lists the algorithm would not refine further, hence it would not reach the incorporated pattern p.

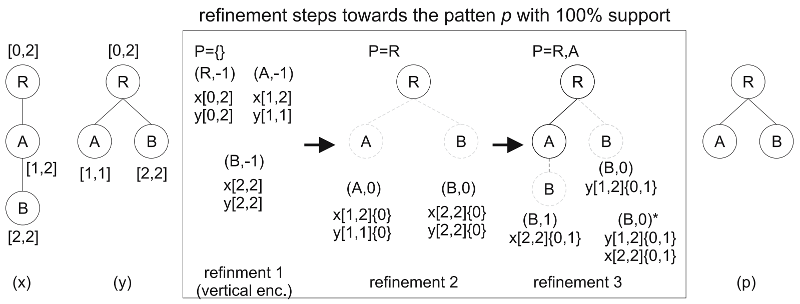
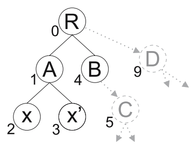


Fig. 13. An example for a tree pattern search with the notion of tree incorporation. Only the search path leading to the most specific pattern (p) in both trees $(x), (y)$ is shown. The shown pattern is *not* valid with regard to tree embedding where two different most specific patterns would be discovered.

4 Instance Pruning

Next to well-known pruning techniques like *node pruning* and *edge pruning* [5], we introduce a novel technique called *instance pruning* (*IP*) that reduces the average computation time by 50%. It is not only applicable to the algorithm working on the novel pattern definition, but also to the TreeMinerV algorithm.

As stated before, $d_{t,\chi}(X)$ returns a 1 if there is at least one occurrence of pattern X in tree t, otherwise it returns 0. Hence, the frequency of a pattern depends only partly on the number of instances. The idea for *IP* is to keep only a subset of the instances necessary to discover all frequent patterns. If there are different instances of a pattern $\langle x, i\rangle$ in tree t, they are represented in the instance list $\mathcal{I}\langle x, i\rangle$ as $I_{1,0} = \langle t, a_0, s\rangle$ and $I_{2,0} = \langle t, b_0, r\rangle$. If the pattern



prefix: R,A,x,-,-,B

match	pattern (C, 3)	pattern (D, 0)
s	[5,8]{0,1,2,4}	[9,42]{0,1,2,4}
r	[5,8]{0,1,3,4}	[9,42]{0,1,3,4}

Fig. 14. One of the groups with the match label $s = \{0, 1, 2, 4\}$ or $r = \{0, 1, 3, 4\}$ can be removed, since both would yield the same result in future joins

$\langle x, i\rangle$ is joined with another pattern $\langle y, j\rangle$, all instances in I_1 and I_2 will be joined with the respective instances of (y, j). Consider two groups of instances $\langle t, a_0, s\rangle, \ldots, \langle t, a_m, s\rangle$ and $\langle t, b_0, r\rangle, \ldots, \langle t, b_n, r\rangle$ of the tree t with match labels s and r as shown in figure 14. If for every triple $\langle t, b_k, r\rangle$ there exists a triple $\langle t, a_l, s\rangle$ with $a_l = b_k$, all triples with the match label r can be removed from the `EQ`. This is possible, as for instances in the same tree with the same match label only the nodes a_l (or b_k) are of relevance for the extension of the instances. If a match label s includes all nodes b_k of a match label r, no instance can be created out of instances with match label r that cannot be created out of instances with match label s. This decrease in the number of instances can effectively reduce the memory consumption of the process. More importantly, it lowers computation time. Not only the removed instances themselves are not joined anymore, but also the ones that would have been created by joining them. For databases, with a low number of labels when compared to the number of nodes, *IP* can reduce the computation time by up to 80%.

5 Towards the $\mathbb{S}$-Set

As stated before, the pattern mining algorithm uses a canonical enumeration scheme for labeled, ordered, rooted trees. Due to this scheme, not every specialization s is created as a refinement of a more general pattern g. During the mining process this restriction is very useful, since it assures that no pattern is generated twice, i.e. it avoids redundancy. However, if we are only interested in the maximally specific patterns, the enumeration strategy gives rise to a problem:

> If a pattern cannot be refined further using the enumeration strategy, that does not imply that there is no further valid specialization.

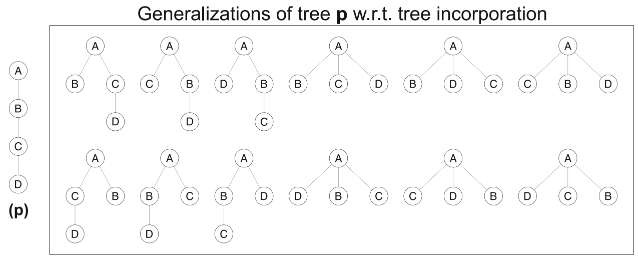


Fig. 15. Altogether there are 35 generalizations (including the empty tree) for the tree (p) on the left. The Figure shows only the 12 4-trees.

We focus on the most specific patterns only, i.e. the set of patterns $\mathbb{S}$ where every $s \in \mathbb{S}$ is frequent, but there is no specialization $s' \sqsupset s$ such that s' is frequent. A run of the algorithm, cf. Figure (12), yields all patterns that are found to be frequent during the search process. For the notions match_{incl} and match_{emb} these are *all* frequent patterns. Since the amount of frequent patterns with regard to the notion of tree incorporation is very high, (cf. Figure (15)) the algorithm focuses on the set $\mathbb{S}$ only. Hence, not all frequent patterns are generated during the search. Obviously, every pattern that can be refined further cannot be part of the set $\mathbb{S}$. But due to the canonical enumeration scheme, not every pattern that cannot be refined further is a maximally specific pattern. An example is shown in Figure (16). Hence, it is necessary to check if there exists a possible extension, whether it is canonical or not. Below we describe two ways to solve this problem.

First, the set $\mathbb{S}^+$ containing all patterns that could not be further refined by the algorithm, which is a superset of the desired set $\mathbb{S}$, can be filtered in a

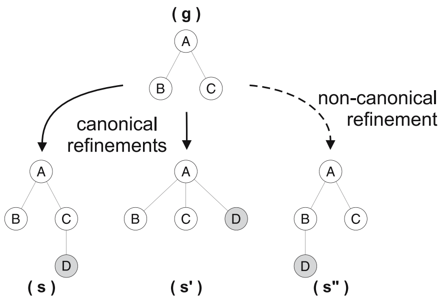


Fig. 16. The trees s, s', and s'' are specializations of the tree g. Whereas s and s' are canonical refinements which are generated by the rightmost expansion, s'' is not generated from g by rightmost expansion. Assume that g and s'' are frequent but neither s nor s' are frequent, there exists no frequent canonical refinement of g but there is a frequent specialization for g: s''. I.e., g would not be a maximally specific pattern.

post processing step. This can be done by applying the algorithm on each pair $s_1, s_2 \in \mathbb{S}^+, s_1 \neq s_2$ to search for patterns that appear in both trees, i.e. that have a frequency of 100%. If there is a pattern p that is equal to s_1 and appears in s_2 we know that $s_1 \sqsubseteq s_2$. Hence, s_2 is a specialization of s_1 and thus s_1 does not belong to the set $\mathbb{S}$. After checking every pair of pattern-trees[3] the set $\mathbb{S}$ is obtained. This method can be used for all three matchings tree inclusion, embedding, and incorporation. Unfortunately, the time-complexity is at $O(|\mathbb{S}|^2)$.

Fortunately, there is another way to discover if further specializations s' of a pattern s exist. Before inserting a frequent pattern s that is not further refinable into the solution set $\mathbb{S}$ it has to be checked if there exists no frequent specialization s' of s. For this, *non-canonical* expansions of the pattern s have to be considered as well.

During the search process, the algorithm traces instances of the patterns found. This trace-data can now be used to search for possible non-canonical expansions of patterns that cannot be generated with regard to the canonical enumeration scheme.

There are three possible specializations to consider: (1) **A new root node**, i.e. adding a parent node of the current root node to the pattern. (2) **A new child node**, i.e. adding a child node of the current root node to the pattern. (3) **Descent of a node**, i.e. specializing a pattern by moving a certain node downwards.

The first specialization, which can basically be described as adding a new node *above* the current pattern-root node is straight forward. For each tree t, we select the node r the pattern's root node is mapped to. We traverse the path starting from r to the root of t upwards and count the unique labels of the nodes seen. If any label's count reaches the minimum frequency, the search can be aborted: there exists a possible specialization of the considered pattern adding a new root node.

In the second case, where a new node is added *below* the pattern-root to generate a frequent non-canonical specialization, the process is more complicated. Since for every pattern p with a node n as some descendant of the root node there exists a more general pattern p with n as direct child of the root node, only expansions with new nodes as direct children of the root node have to be considered. Hence, there are as many possible positions (called *bins*) for a new node as the current pattern has direct children of the root node. expansion with a new node as the rightmost child of the root node, since this is already covered by the canonical expansion. Every node n that is not already part of the pattern could appear in a *bin* to the left as well as to the right of a direct child c of the root node, if n and c are on the same path to the root node. If a node n does not lie on a path with any direct child c of the root node, it could appear only in one *bin* between two existing children c_n, c_{n+1} of the root node (cf. node B in Figure 17). In general in a tree a node n lies on the same path as every ancestor of n and on the same path as every node in the subtree of n. Figure (17) shows a pattern and an instance of the pattern in a data tree. For non-canonical expansions there

[3] Of course, patterns that turn out to have a specialization are not reconsidered.

are two positions x_1, x_2 for a new node. The node D that is a descendant of A could appear at both, whereas the nodes B and E could only appear at position x_2. The second possible position for node E would be a canonical expansion and thus is not considered here. To find possible non-canonical expansions, the algorithm traverses the subtree of each instance root and counts the possible labels for the *bins*. If any label reaches the minimum support, we know that there is a frequent specialization of the pattern.

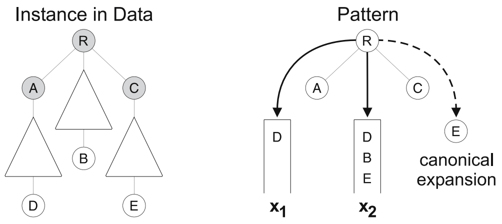


Fig. 17. Searching for non-canonical expansions the algorithm puts every node that is not part of the instance in up to two bins. Each bin is located on the left of each child of the root node. for both bins next to the corresponding between the 'root-childs' where they are located.

In the third and last case, no new node is added to the pattern. Instead, specializations of the pattern are considered where the nodes are moved further down in the tree. For this, every pair of nodes n_1, n_2 in the pattern is considered where n_1 is a sibling of n_2. The algorithm checks if there is an ancestor-descendant relationship in the data between the nodes n_1 and n_2 are mapped to. If the number of trees in which n_1 is an ancestor of n_2 is equal to or larger then the minimum frequency, the pattern considered can be specialized further. I.e., it is not maximally specific.

If none of the three specializations is possible, the pattern is maximally specific and can be added to the set $\mathbb{S}$.

5.1 Cardinality of Maximally Specific Sets

Due to Theorem 1, the resulting pattern space with regard to tree incorporation contains more patterns than of tree embedding, which in turn contains more patterns than that of tree inclusion. In contrast to the whole pattern space, there is no such relation among the $\mathbb{S}$ sets, i.e. the sets containing the most specific patterns only. Two examples are given in Figure (18) and Figure (19). The first shows that with the given database and a minimum frequency of two there are five maximally specific patterns for *tree inclusion* and six for *tree embedding*. With a minimum frequency of three, there are still five maximally specific patterns for *tree inclusion*, but only four for *tree embedding*. The example for *tree embedding* and *tree incorporation* in Figure (19) is similar. In the first case, there are four maximally specific patterns with regard to *tree embedding* and six for

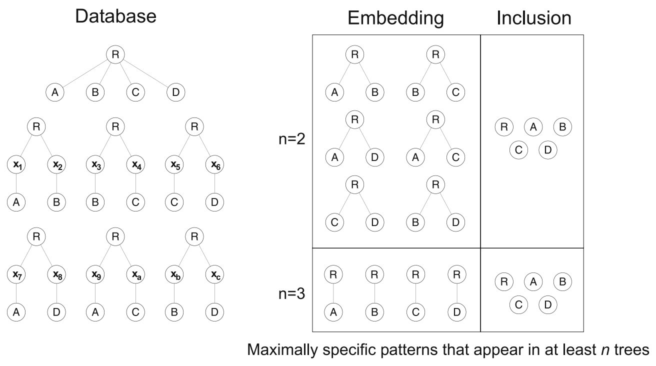

Database
Embedding
Inclusion
n=2
n=3
Maximally specific patterns that appear in at least *n* trees

Fig. 18. A database and maximally specific patterns w.r.t. *tree inclusion* and *tree embedding* for a minimum support of n=2 and n=3 each. For n=2 there are more maximally specific patterns for embedding while for $n = 3$ there are more w.r.t. inclusion.

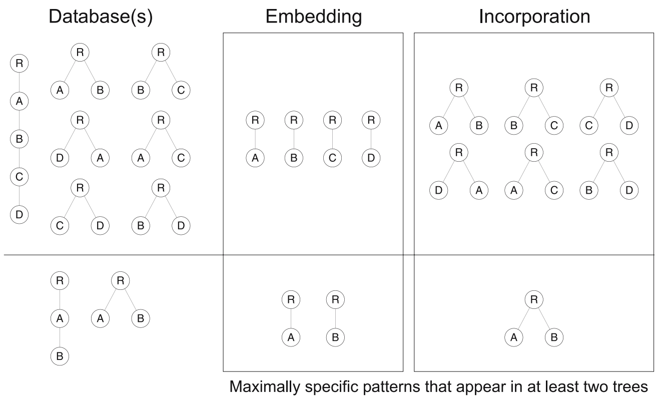

Database(s)
Embedding
Incorporation
Maximally specific patterns that appear in at least two trees

Fig. 19. Two databases and maximally specific patterns w.r.t. *tree embedding* and *tree incorporation* for a minimum support of $n = 2$ in both cases. In the upper case there are more maximally specific patterns for incorporation while for the database on the bottom there are more w.r.t. embedding.

tree incorporation. In the second case, there are more for *tree embedding* than for *tree incorporation.*

6 Experimental Results

A number of experiments were conducted on real-world and synthetic datasets. The real-world dataset (legcare [20]) consists of an online shop's web-log, containing 234942 visits. Each visit is regarded as a subtree of the hierarchically structured web-site. There where 694 unique labels for the database. For the synthetic dataset we implemented a data generator as described by Zaki [6]. All the experiments were performed on a 3.2GHz Intel Pentium 4 with 2GB main memory, running SUSE 9.0. The algorithms were implemented in C++. For the tree embedding and tree incorporation, *instance pruning* is available. We compared the number of frequent patterns found by the algorithms and the size of the $\mathbb{S}$ set on both datasets with different minimum support. To calculate the $\mathbb{S}$ set an additional post-processing step was performed.

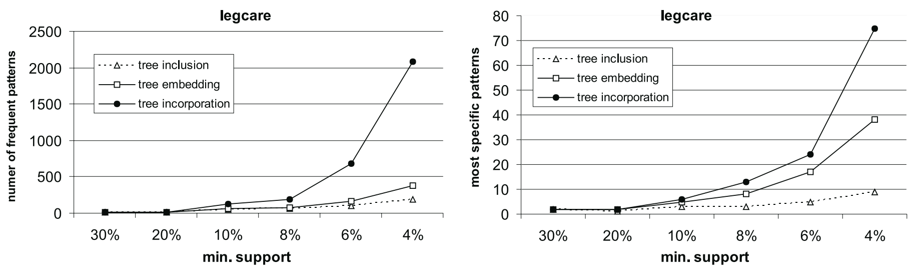


Fig. 20. Comparing the amount of the patterns generated during the search (*left*) and the size of the maximally specific border (*right*) of three tree matching notions for various minimum support levels

Figure (20) shows the number of patterns generated during the search and the number of patterns contributing to the $\mathbb{S}$ set for tree embedding and tree incorporation on the legcare dataset. As mentioned earlier, the notion of tree incorporation is more relaxed than tree embedding such that more patterns are generated and discovered during the search. In contrast to the figures shown here (Figure 20), there is no order between the notions with regard to the size of the according $\mathbb{S}$ set as explained in the Figures (18, 19). All three notions have in common that they exhibit exponential growth in the number of most specific patterns as well as in patterns considered during the search when reducing the minimum support. For the legcare-dataset there was no effect on computation time with and without *IP*. However, using *IP*, the memory consumption dropped dramatically for computing frequent pattern sets with minimum support below 10%. For the experiments with the synthetic data, a master-tree with 100 unique labels and 10000 nodes was generated with a maximum depth and fanout of 10. From this hypothetical web-site we generated 10000 visits, each a smaller subtree of the master-tree, as database. The graph in Figure (21) shows the results of experiments on this dataset regarding computation time and the effect of instance pruning. The plot clearly indicates an exponential growth in computation time when lowering the minimum support. The solid lines depict the required time with *IP*. Both, tree embedding and tree incorporation[4], show a significant speedup for low minimum support levels using *IP*. More experiments on synthetic data sets showed similar behaviour, i.e. the runtimes for the algorithms using *IP* are significantly lower, especially at low minimum support levels.

[4] Since *IP* was not implemented for tree inclusion there are no results here.

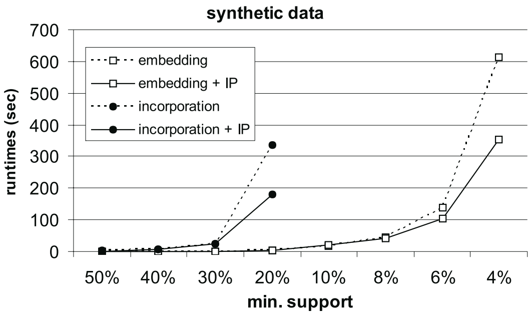


Fig. 21. Comparing the effect of instance pruning for embedding and incorporation on synthetic data

7 Related Work

The most directly related work to this paper is Zaki's TreeMinerV [6] as well as Termier's TreeFinder [7] and FreqT by Asai [5]. Zaki uses a smart, so-called vertical representation to facilitate the candidate count enabling a fast mining process that scales well even with large datasets. We adopted this idea for the presented treemining algorithm RETRO. Since the definition of tree incorporation is more general than tree embedding or tree inclusion, the algorithm yields more patterns even when focusing on the $\mathbb{S}$ set only. Therefore it is not surprising that the algorithm is slower than TreeMinerV. Using the RETRO algorithm to mine embedded trees there is a significant speedup compared to TreeMinerV due to the presented pruning technique. Compared to TreeFinder, it uses a less general pattern definitions, but all of them are complete with regard to the maximally specific patterns. For tree inclusion and tree embedding it is even complete with regard to the whole pattern space.

Other algorithms like FreeTreeMiner by Yun Chi *et al.* [21] and HybridTreeMiner [22] work on free trees. Furthermore there exist several algorithms like FreeTreeMiner by Rückert and Kramer [23] which searches for free-tree patterns in graphs or AGM [1], FSM [2], gSpan [3], and Gaston [24] that work completely on graphs rather than trees. They are restricted to subgraphs consisting of edges and if applied to trees would only discover frequent trees in the sense of subtree inclusion. The Gaston algorithm realizes the graph search stepwise. First frequent sequences are mined that are expanded to trees and later on to graphs.

8 Conclusions

The algorithm presented in this paper improves and expands the TreeMinerV algorithm. We added support for tree incorporation and and tree inclusion. The introduced *instance pruning* technique reduces the computation time as well as the memory usage of the algorithm in many cases. Since the amount of frequent patterns in large databases grows fast when lowering the minimum support it seams to be useful to calculate the set $\mathbb{S}$ of all maximally specific patterns. An approach how to immediately calculate the set of all maximally specific patterns by considering non-canonical expansions was presented.

Furthermore, we have shown that there is an order among the four different matching notions. Each tree that matches another one with regard to tree inclusion also matches the same tree with regard to tree embedding. In the same way, each tree that is embedded in another tree is incorporated in the tree as well. Finally each incorporated tree is subsumed. Hence, the amount of patterns grows from tree inclusion, over embedding, incorporation to subsumption. In contrast this is not true for the set of maximally specific patterns.

With regard to the future, especially in a real-word environment, it would be nice to have more constraints, like maximally-general or maximally-specific pattern, to enable the user to focus the search as in MolFea [4]. Furthermore it would be interesting to extend the tree-mining process to first order logic which would give a much more expressive language for data and patterns. On the other hand the frequent patterns discovered could also be used as features for some classifier as in [25]. Considering the notion of tree incorporation, we still have to evaluate if the additional cost in time and memory is justified by more informative patterns. Finally, it depends on the data and on the requirements of the user which tree matching notion is the best.

Acknowledgments

We sincerely thank Luc De Raedt for motivating this work. We also thank Mohammed Zaki for providing the TreeMiner source code. Further many thanks to Ulrich Rückert, Albrecht Zimmermann, Kristian Kersting, and Andreas Karwath for constructive comments and helpful suggestions. The real world dataset was kindly provided by Blue Martini Software.

This work was partly supported by the EU IST project cInQ (Consortium on discovering knowledge with Inductive Queries). Contract no. IST-2000-26469.

References

1. Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data. In: Proc. of PKDD. (2000) 13–23
2. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: Proc. of ICDM. (2001) 439–442
3. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: Proc. of ICDM. (2002) 721–724
4. De Raedt, L., Kramer, S.: The levelwise version space algorithm and its application to molecular fragement finding. In: Proc. of IJCAI-01. (2001) 853–862
5. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: Proc. of SIAM SDM. (2002) 158–174
6. Zaki, M.: Efficiently mining frequent trees in a forest. In: Proc. of KDD. (2002) 71–80

7. Termier, A., Rousset, M.C., Sebag, M.: Treefinder: a first step towards XML data mining. In: Proc. of ICDM. (2002) 450–457
8. Cooley, R., Mobasher, B., Srivastava, J.: Web mining: Information and pattern discovery on the world wide web. In: Proc. of ICTAI. (1997) 558–567
9. Goldfarb, C.F., Prescod, P.: The XML handbook. Prentice Hall (1998) ISBN 0-13-081152-1.
10. Shapiro, B.A., Zhang, K.: Comparing multiple RNA secondary structures using tree comparisons. Computer Applications in the Biosciences **6** (1990) 309–318
11. Bringmann, B.: Matching in frequent tree discovery. In: Proc. of ICDM. (2004) 335–338
12. Ramesh, R., Ramakrishnan, L.: Nonlinear pattern matching in trees. Journal of the ACM **39(2)** (1992) 295–316
13. Kilpeläinen, P.: Tree Matching Problems with Applications to Structured Text Databases. PhD thesis, University of Helsinki (1992)
14. Lloyd, J.W.: Foundations of logic programming; (2nd extended ed.). Springer-Verlag New York, Inc. (1987)
15. Plotkin, G.D.: A note on inductive generalization. In: Machine Intelligence. Volume 5. Edinburgh University Press (1970) 153–163
16. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming **19/20** (1994) 629–679
17. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Proc. of ICMD, Washington, D.C. (1993) 207–216
18. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery **1** (1997) 241–258
19. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained associations rules. In: SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, ACM Press (1998) 13–24
20. Kohavi, R., Brodley, C., Frasca, B., Mason, L., Zheng, Z.: KDD-Cup 2000 organizers' report: Peeling the onion. SIGKDD Explorations **2** (2000) 85–98
21. Chi, Y., Yang, Y., Muntz, R.R.: Indexing and mining free trees. In: Proc. of ICDM. (2003) 509–512
22. Chi, Y., Yang, Y., Muntz, R.R.: Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: Proc. of SSDBM. (2004) 11–20
23. Rückert, U., Kramer, S.: Frequent free tree discovery in graph data. In: Proc. of ACM symposium on Applied computing, ACM Press (2004) 564–570
24. Nijssen, S., Kok, J.N.: A quickstart in frequent structure mining can make a difference. In: Proc. of KDD, ACM Press (2004) 647–652
25. Bringmann, B., Karwath, A.: Frequent SMILES. In: Proc. of LWA (FGML). (2004) 132–137