FAT-miner: Mining Frequent Attribute Trees

Jeroen De Knijf

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2006-053

www.cs.uu.nl

ISSN: 0924-3275

FAT-miner: Mining Frequent Attribute Trees

Jeroen De Knijf

October 24, 2006

Abstract

Data that can conceptually be viewed as tree structures abounds in domains such as bio-informatics, web logs, XML databases and multi-relational databases. Besides structural information such as nodes and edges, tree structured data also often contains attributes, that represent properties of nodes. Current algorithms for finding frequent patterns in structured data, do not take these attributes into account, and hence potentially useful information is neglected. We present FAT-miner, an algorithm for frequent pattern discovery in tree structured data with attributes. To illustrate the applicability of FAT-miner, we use it to explore the properties of good and bad loans in a well-known multi-relational financial database.

1 Introduction

Frequent tree mining has become an important and popular problem in the field of knowledge discovery and data mining. The main reasons for the increase in interest are the growing amount of semi-structured data (e.g. XML databases) and the urge to analyze and mine these databases. Furthermore, the availability of tree mining algorithms such as [2, 13, 16] to exploit these databases, without losing information on the structure of the data, has increased the interest of the research community. Briefly, given a set of tree data, the problem is to find all subtrees that satisfy the minimum support constraint, that is, all subtrees that occur in at least $n\%$ of the data records.

Applications of frequent tree mining include the following:

- Web log mining: Frequent access trees from a database of web logs, where each record corresponds to the entire forward access of a user, are explored in [16]. These frequent access trees can be used to improve the design of the web site.
- Classification and clustering: The work presented in [17] uses a frequent tree mining algorithm to extract frequent substructures of XML data, the data is then classified according to its structure.
- Database indexing: In [15] a frequent tree mining algorithm is used to extract frequent tree query patterns out of a large collection of XML queries. The answers to the frequent tree query patterns are then stored and indexed for faster retrieval.

- Exploration of the data source: When confronted with a large unknown data source, frequent tree mining can be used to help a user understand the data, because frequently occurring structures give insight in the dataset. This idea is used in the work of Wang and Liu [13], on a subset of the Internet Movie Database (IMDB).

Without doubt the most commonly used tree structured datasets are XML data, but also multi-relational datasets often have a star-shaped or snowflake structure: OLAP-databases are well known examples. These structures are essentially trees and hence tree mining algorithms should be suited to analyze this type of data. However, current tree mining algorithms neglect the existence of attribute values in XML data and are not applicable to multi-relational datasets. In this work, we consider the frequent tree mining problem, where the attributes associated with each node in the tree play a crucial role in the mining algorithm.

This paper is organized as follows: In the next section we motivate the assumptions we made. In section 3 we discusses the basic concepts necessary for frequent attribute tree mining. The following section discuss the frequent tree mining algorithms which will later be used in FAT-miner. In section 5 two different variants of FAT-miner are proposed: first for the induced subtree relation and secondly for the embedded subtree relation. Furthermore, these algorithms are theoretically compared with a straightforward approach to attribute tree mining. In section 6 we apply our algorithms to a well known multi-relational financial dataset and discuss the results. Furthermore, an experimentally comparison is made between FAT-miner and the straightforward approach to attribute tree mining. In section 7 we relate frequent attribute tree mining with multi-relational mining and mention some advantages of the first approach. In the final section we draw conclusions and give directions for further research.

2 Motivation

We define an attribute tree to be a labeled rooted ordered tree, with a non-empty attribute set assigned to each node. The motivation behind this constraint is twofold:

1) Informativeness: When considering tree structured data containing attributes, a pattern where each node has some attributes attached to it is more informative than its counterpart without attributes, that is, the attributes attached to the nodes reveal additional information about this node. In fact, the "at least one attribute" constraint can be viewed as insisting on more specialized patterns. Consequently, from a frequent pattern attribute tree we can also derive all combinations of this frequent pattern without attributes. As an example of non-informative patterns without attributes consider the multi-relational dataset described in subsection 6.1. From the database scheme we apriori know that every loan has an account, and hence it is uninteresting to 'discover' this type of patterns. However, allowing nodes without attributes results in a large number of this type of uninteresting patterns. In semi-structured data we usually do not apriori know that a

node occurs in the tree. Although in such cases nodes without attributes are not entirely uninformative, they are still less informative than their counterparts with attributes. For example, consider the Wikipedia XML dataset as described in subsection 6.2, almost every XML document has a couple of nodes of type Wikipedia link; for finding discriminating patterns this information is not very helpful. However, attributes assigned to this node describe to which document the link points, and hence finding patterns with the attributes associated to the collection link nodes, results in very discriminative patterns.

2) Complexity: As theoretically argued in subsection 5.1 and experimentally shown in subsection 6.2 mining frequent tree patterns without the constraint is unfeasible from a computational point of view, even for small data sets.

Although it make no sense to use FAT-miner for tree structured data without attributes, the previously mentioned advantages also apply when some of the nodes contain attributes and others not; a typical example is XML data. When mining semi-structured data, a pre-processing step is needed to transform these data trees into attribute data trees. Practically, when the data is loaded a special (dummy) attribute is assigned to the nodes in the data trees without any attributes.

The reader should note that our definition differs from the definition used by Termier et. al [12], who use the term attribute tree to denote a rooted (ordered or unordered) tree where no two children of a node have the same label.

3 Preliminaries

In this section we provide the basic concepts and notation used in this paper. A labeled rooted ordered attribute tree $T = \{V, E, \leq, L, v_0, M\}$ is an acyclic directed connected graph which contains a set of nodes V, and an edge set E. The labeling function L is defined as $L : V \rightarrow \Sigma$, i.e., L assigns labels from alphabet Σ to nodes in V. The special node v_0 is called the root of the tree. If $(u, v) \in E$ then u is the parent of v and v is a child of u. For a node v, any node u on the path from the root node to v is called an ancestor of v. If u is an ancestor of v then v is called a descendant of u. Furthermore there is a binary relation '$\leq$' $\subset V^2$ that represents an ordering among siblings. The size of a tree is defined as the number of nodes it contains; we refer to a tree of size k as a k-tree. The set of attributes is denoted by : $\mathcal{A} = \{a_1, \ldots, a_n\}$, where each attribute takes its value from a finite domain. We further assume that there is an ordering among the attributes; i.e., $a_j \prec a_k$. To each node v in V, a non-empty subset of $\mathcal{A}$ is assigned; we call this set the attributes of v. More formally: $M : V \rightarrow \mathcal{P}(\mathcal{A}) \setminus \{\varnothing\}$.

Rooted ordered attribute trees are an extension of rooted ordered trees. Since mining rooted ordered trees is well explored [2, 16], we recall here some definitions which we will later extend for attribute trees.

Definition 1 *Given two labeled rooted trees T_1 and T_2 we call T_2 an* induced *subtree of T_1 and T_1 an* induced *supertree of T_2, denoted by $T_2 \preceq_i T_1$, if there exists an injective matching*

function Φ *of* V_{T_2} *into* V_{T_1} *satisfying the following conditions for any* $v, v_1, v_2 \in V_{T_2}$*:*

1. Φ *preserves the labels:* $L_{T_2}(v) = L_{T_1}(\Phi(v))$.

2. Φ *preserves the order among the siblings: if* $v_1 \leq_{T_2} v_2$ *then* $\Phi(v_1) \leq_{T_1} \Phi(v_2)$.

3. Φ *preserves the parent-child relation:* $(v_1, v_2) \in E_{T_2}$ *iff* $(\Phi(v_1), \Phi(v_2)) \in E_{T_1}$.

Besides the induced subtree relation, another subtree relation is also often used in frequent tree mining:

Definition 2 *Given two labeled rooted trees* T_1 *and* T_2 *we call* T_2 *an* embedded *subtree of* T_1 *and* T_1 *an* embedded *supertree of* T_2*, denoted by* $T_2 \preceq_e T_1$*, if there exists an injective matching function* Φ *of* V_{T_2} *into* V_{T_1}*, satisfying the conditions* 1 *and* 2 *of definition* 1*. Additionally,* Φ *has the following property for any* $v_1, v_2 \in V_{T_2}$*:*

3'. Φ *preserves the ancestor-descendant relation: if* $(v_1, v_2) \in E_{T_2}$ *then* $\Phi(v_1)$ *is an ancestor of* $\Phi(v_2)$ *in* T_1.

In the remainder of this paper we use $T_1 \preceq T_2$ to denote that T_1 is either an induced or an embedded subtree of T_2.

Let $D = \{d_1, \ldots, d_m\}$ denote a database where each record $d_i \in D$, is a labeled rooted ordered tree. For a given labeled rooted ordered tree T we say T occurs in a transaction d_i if T is a subtree of d_i. Let $\sigma_{d_i}(T) = 1$ if $T \preceq d_i$ and 0 otherwise. The support of a tree T in the database D is then defined as $\psi(T) = \sum_{d \in D} \sigma_d(T)$, that is the number of records in which T occurs one or more times. T is called frequent if $\psi(T)/|D|$ is greater than or equal to a user defined minimum support (*minsup*) value. The goal of frequent tree mining is to find all frequently occurring subtrees in a given database. Notice that ψ is an anti-monotone function: $T_i \preceq T_j \Rightarrow \psi(T_i) \geq \psi(T_j)$. The anti-monotonicity property of ψ is used to efficiently compute all the frequent subtrees of a database.

4 Mining Frequent Trees

In this section we briefly discuss frequent tree mining without attributes. Enumeration of all frequent induced subtrees is accomplished by using the rightmost extension techniques described in [2]: a $(k-1)$-tree is expanded to a k-tree by adding a new node *only* to a node on the rightmost branch of the $(k-1)$-tree. The rightmost branch of a tree is the unique path from the root to the rightmost leaf. Note that for each k-tree its parent is uniquely defined by removing the rightmost node. This procedure of extending pattern trees ensures that each pattern tree is counted exactly once. The algorithm starts by first computing the frequent 1-patterns. Then each previously found pattern is extended until no more frequent extensions are possible. For support counting and extension of the frequent pattern tree, occurrence lists are used. For each subtree T, there is an occurrence list of T where all mappings from the nodes of T to the database are recorded. More formally,

$occ(T,D) = \cup_{d \in D}\{\{\Phi^1_d(v_1),\ldots,\Phi^1_d(v_k)\},\ldots,\{\Phi^l_d(v_1),\ldots,\Phi^l_d(v_k)\}\}$ where $(v_1,\ldots,v_k) \in V_T$ and $|T| = k$; with $\Phi^1_d,\ldots,\Phi^l_d$ the distinct matching functions from T into d.

Zaki [16] introduces methods to enumerate all embedded subtrees. Trees are encoded as strings: the labels of the tree are added in depth first order to the string. Whenever the depth first search backtracks, a special symbol (-1 with $-1 \notin \Sigma$) is added to the string, to preserve the topological structure of the tree. The string encoding of a tree T has an interesting property: if either the last or second last label from the string is removed, the resulting string is an embedded subtree of T. This property is used to enumerate all frequent subtrees: a candidate $(k+1)$-tree is generated by joining two frequent k-subtrees that share the same $k-1$ prefix. Starting from the frequent 1-patterns, all frequent trees are enumerated by performing a depth first search. For support counting and joining of trees a scope list representation of the database is used. For each frequent subtree T of size k, the corresponding scopelist of T records all occurrences of T. Per occurrence a triple (t,m,s) is used , where t is a tree id of d_i; i.e., the identifier of the database tree in which T occurs, m the (pre-order) positions of the nodes in d_i of the first $k-1$ nodes from T; finally s the scope of the rightmost node of T. The scopes of the rightmost nodes are used when two k-trees are joined: one can compute from it whether one node is an ancestor/descendant of the other. Formally the scopelist of a tree is defined as $\mathcal{L}(T,D) = \cup_{d \in D}$ $\{(id(d),\{\Phi^1_d(v_1),\ldots,\Phi^1_d(v_{k-1})\},Scope(\Phi^1_d(v_k))),\ldots,(id(d),\{\Phi^l_d(v_1),\ldots,\Phi^l_d(v_{k-1})\},Scope(\Phi^l_d(v_k)))\}$ where $(v_1,\ldots,v_k) \in V_T$ and $|T| = k$; with $\Phi^1_d,\ldots,\Phi^l_d$ the distinct matching functions from T into d; $id(d)$ the unique identifier of a database record and $Scope(v)$ the scope of the node.

5 Mining Frequent Attribute Trees

To mine attribute trees we first need to define the subtree relations for attribute trees. An intuitive and simple additional criterion to the definitions 1 and 2 is that the subtree relation should preserve the attributes; i.e., we add to definitions 1 and 2:

4. $\forall v \in V_{T_2} : M(v) \subseteq M(\Phi(v))$.

5.1 Naive Approach

A straightforward approach to mine attribute trees is to create a new node for every attribute-value pair of a node v and add these newly created nodes as child nodes of v. Since there is an ordering among the attributes, these newly created nodes can be added according to that order and by convention all get an ordering before (or after) the original children of v.

The advantage of this encoding is that one needs to only slightly modify the existing rooted ordered tree mining algorithms discussed in the previous section to be able to mine attribute trees. The disadvantage is however that the algorithms may return trees containing nodes without any attributes; as we have argued in section 2 such trees are

unlikely to be interesting. A possible solution is to post-prune the output such that all non-attribute trees (that is, trees having empty attribute sets) are thrown away. A drawback of the naive approach with post-pruning is that the algorithm spends effort in computing trees that will later be removed. The question arises, how many trees the naive algorithm would produce that turn out not to be attribute trees. To answer this question we compute both a lower and an upper-bound for the number of non-attribute trees the naive approach would generate.

To compute the upper-bound, we give an analysis of the worst case scenario of the naive mining algorithm. Suppose we have a tree T with n nodes, and m attribute-value pairs assigned to each node. For our analysis we require T to have the maximal number of induced subtrees possible. This is the case when we have a root with $n-1$ children attached to it. Then the number of trees reported with the naive approach can be derived as follows:

1. The number of trees of size one $= n$ and since each tree has m attribute-value pairs there are $n \times 2^m$ different trees of size one.

2. The number of trees of size two equals $\binom{n-1}{1}$, that is, if we pick the root node we still have to choose one node out of $n-1$. Each of these trees has m attribute-value pairs associated with each node; hence there are $\binom{n-1}{1} \times (2^m)^2$ different trees of size two.

Generalizing this we get:

$$\mathcal{N}(naive) = \quad n2^m + \sum_{k=1}^{n-1} \binom{n-1}{k} (2^m)^{(k+1)} . \tag{1}$$

While the number of attribute trees $\mathcal{N}(ideal)$ equals:

$$\begin{aligned} \mathcal{N}(ideal) &= \overbrace{n2^m - n}^{c} + \sum_{k=1}^{n-1} \binom{n-1}{k} (2^m - 1)^{(k+1)} \\ {\scriptstyle [x:=2^m-1]} &= c - x + x \sum_{k=0}^{n-1} \binom{n-1}{k} (x)^k \, 1^{(n-1)-k} \\ &= c - x + x\,(x+1)^{(n-1)} \\ {\scriptstyle [2^m-1:=x]} &= \underbrace{n2^m - n}_{c} + 2^{mn} - 2^{m(n-1)} - 2^m + 1. \end{aligned} \tag{2}$$

If we subtract equation 2 from equation 1, we get the number of non-attribute trees the naive approach produces in the worst case. We observe that: $\mathcal{N}(naive) - \mathcal{N}(ideal) \gg 2^{m(n-1)}$.

To determine the lower-bound, we give an analysis of the (non-trivial) best case scenario of the naive mining algorithm. Suppose T is a tree with n nodes and to each node one attribute-value pair is assigned. For the induced subtree relation, the best possible case is achieved when all nodes of T are on a single path in T. Suppose T has a frequent pattern of size $k > 0$, then the number of trees reported with the naive approach is at least:

$$\mathcal{M}(naive) = \sum_{i=0}^{k-1}(k-i)2^{(i+1)}. \tag{3}$$

While the number of attribute trees equals:

$$\mathcal{M}(ideal) = \sum_{i=0}^{k-1}(k-i). \tag{4}$$

Hence, if we subtract equation 4 from equation 3 we get the total number of non-attribute trees the naive algorithm produces. As a result, $\mathcal{M}(naive) - \mathcal{M}(ideal) \geq 2^k - 1$.

We omit the worst and best case scenarios for the naive embedded tree mining algorithm, but it can be done similarly to the analysis for induced subtrees. Since every embedded subtree is also an induced subtree, the upper-bound is also a least upper-bound, of the naive embedded tree mining algorithm. Likewise, the stated lower-bound is also the least lower-bound of the naive embedded tree mining algorithm. In fact they both are of the same order.

Combining the lower and upper-bound previously determined, the number of attribute trees $\#(ideal)$ compared with the numbers of trees generated by the naive algorithm $\#(naive)$ is bounded by the following formula:

$$(2^k - 1) + \#(ideal) \leq \#(naive) \leq 2^{m(n-1)} + \#(ideal).$$

Where k denotes the size of the largest attribute tree, m denotes the number of attribute-value pairs of each node and n the number of nodes in the tree. Our conclusion from this analysis is as follows. Although, as the analysis shows, the complexity of the naive approach and the ideal approach only differs by an additive term, this term increases at least exponentially with the size of the largest attribute tree. Furthermore, as the experiments in subsection 6.2 show this difference is of considerable practical importance. Hence it is worthwhile to develop a mining algorithm that directly computes the desired trees, rather than generates trees with empty attribute sets that have to be post-pruned, as the naive approach does.

5.2 Global and Local Mining

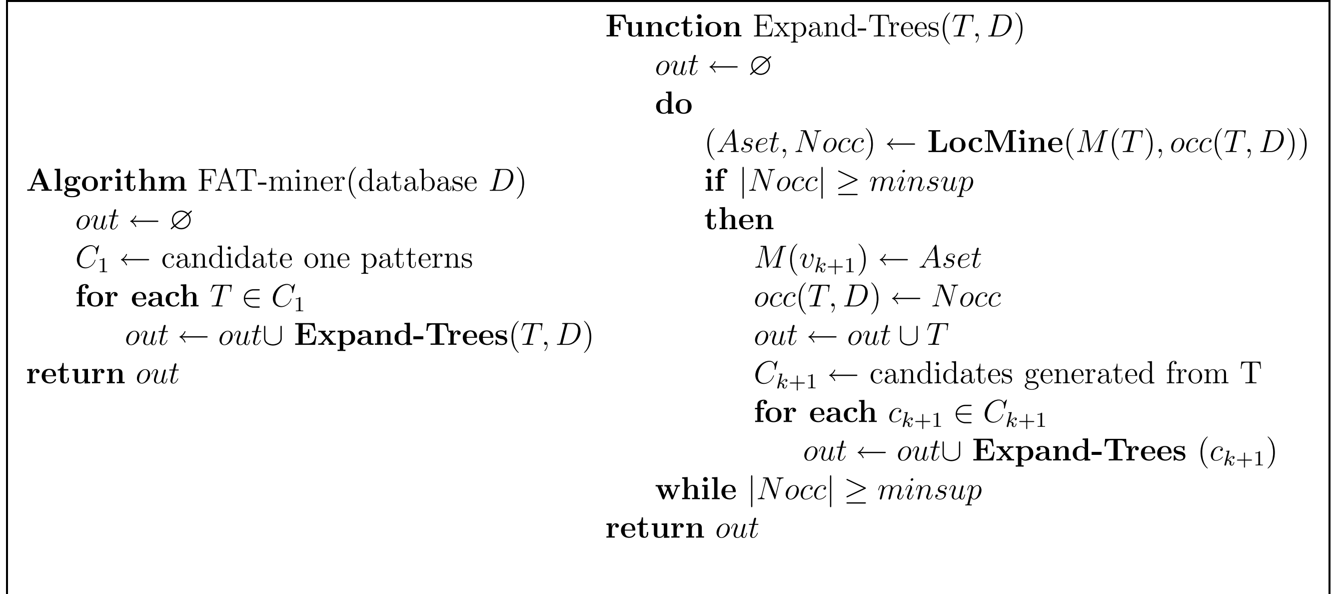
In the previous section we determined the advantage of an "ideal" algorithm, in this section we present such an algorithm. The main idea for the ideal attribute tree mining algorithm is to split the mining process in a global mining stage and a local one. Loosely speaking the

```
Function LocMine ( X, OCL)
    if X = ∅
    then
        l ← 1
    else
        l ← k + 1
    do
        while l ≤ n
            if support(X ∪ a_l) ≥ minsup
            then
                (X ← X ∪ {a_l})
                return(X, ComputeOcc(X, OCL))
            else
                l ← l + 1
        Y ← X
        if X ≠ ∅
        then
            X ← X \ {a_k}
            l ← k + 1
    while Y ≠ ∅
return (∅, ∅)
```

```
Function ComputeOcc( X, OCL)
    out ← ∅
    for each d_i ∈ OCL
        for each Φ^j_{d_i} ∈ d_i
        if X ⊆ M(Φ^j_{d_i}(v_{k+1}))
        then
            out ← out ∪ Φ^j_{d_i}
return out
```

Figure 1: The local mining algorithm

global mining part consist of a slightly modified rooted tree mining algorithm as described in section 4. The local mining must be done for every node of the subtrees. It boils down to the computation of frequent attribute sets from all the attributes to which the node of the subtree is mapped in the database. In this setting, the local mining algorithm is slightly different from the work done on frequent itemset mining [1, 3]. This difference is mainly due the fact that the node labels can occur multiple times in a tree. These two mining methods have to be combined: the idea is that whenever a candidate tree T (of size k) is generated, the local mining algorithm determines the first frequent attribute set of v_k, say A_1. If there is none, then T must be pruned. Otherwise all supertrees of T are computed, where the attributes of v_k equals A_1. When there are no more frequent extensions of T left, the next frequent itemset of v_k is computed and this frequent pattern is then further extended. For both the global and local mining a depth-first search through the enumeration lattice is used.



Algorithm FAT-miner(database D)
    $out \leftarrow \varnothing$
    $C_1 \leftarrow$ candidate one patterns
    for each $T \in C_1$
        $out \leftarrow out \cup$ **Expand-Trees**(T, D)
return out

Function Expand-Trees(T, D)
    $out \leftarrow \varnothing$
    do
        $(Aset, Nocc) \leftarrow$ **LocMine**$(M(T), occ(T, D))$
        if $|Nocc| \geq minsup$
        then
            $M(v_{k+1}) \leftarrow Aset$
            $occ(T, D) \leftarrow Nocc$
            $out \leftarrow out \cup T$
            $C_{k+1} \leftarrow$ candidates generated from T
            for each $c_{k+1} \in C_{k+1}$
                $out \leftarrow out \cup$ **Expand-Trees** (c_{k+1})
    while $|Nocc| \geq minsup$
return out

Figure 2: The global mining algorithm.

Consider a candidate $(k+1)$-tree, that is generated from the frequent k-tree T. Then the local mining for node v_{k+1} consists of determining all frequent itemsets of the following transaction database TD_s:

$$\begin{array}{llll} tid = id(d_i) & M(\Phi^k_{d_i}(v_{k+1})) & \dots & M(\Phi^l_{d_i}(v_{k+1})) \\ \quad\vdots & & & \\ tid = id(d_j) & M(\Phi^k_{d_j}(v_{k+1})) & \dots & M(\Phi^l_{d_j}(v_{k+1})) \end{array}$$

So each $d \in D$ for which both $T \preceq d$ and T can be extended in d with a node v_{k+1} according to definition 1 or 2—hence ignoring the attributes of node v_{k+1} for the moment—is considered as a transaction in the local mining setting. Since a node label can occur multiple times in a tree, each transaction can contain multiple attribute sets. In this setting the support of an attribute set X is the number of transactions in which X occurs, where X is said to occur in a transaction $TD_s\{A_1, \dots, A_n\}$ if X is a subset of *at least one* of the attribute sets of TD_s; i.e. $\exists A_i \in TD_s : X \subseteq A_i$. Since the local mining is interleaved with the global mining, we don't compute all frequent attribute sets at once when we start the local mining for the rightmost node of the candidate $(k+1)$-tree. The reason is that even though computing all the frequent attribute sets at once leads to the generation of fewer candidates, it is expensive in memory usage. So the local mining algorithm gets as input a frequent attribute set (which can be possible empty), computes the next one, and returns all nodes in the transaction database that cover the newly computed attribute set. In figure 1 the pseudo code for the local mining algorithm is given. Notice that in the

pseudo code the computation of the occurrences that match the frequent attribute set is done naively; this is mainly done for clarity and simplicity. In the implementation of the algorithm, whenever a frequent attribute set is computed also the nodes that cover the frequent attribute set are known. In the function LocMine in case of a non-empty attribute set X, we represent with a_k the attribute which comes last in the ordering of the attributes in X. Recall that we earlier defined that the set of attributes has as element with the highest order a_n. First, for every possible extension of X with a higher ordered attribute than a_k, the frequency is determined. If one of these extensions is frequent, the function ComputeOcc is called. This function determines all mappings in the occurrence list (or scopelist), for which the rightmost node of the mapping covers the frequent extension. If none of the previous extensions is frequent, a_k is removed from X and X is again extended with attributes.

In the global mining algorithm, as described in figure 2, candidate trees are generated by the induced or embedded rooted tree mining algorithm used. For the candidate generation the attributes are ignored. Notice that in the pseudo code $occ(T, D)$ should be replaced with $\mathcal{L}(T, D)$ in case of embedded subtrees. For each candidate one-pattern the function Expand-Trees is called. This function first calls the local mining function, which determines the next frequent attribute set. If there is one, this attribute set is assigned to the rightmost node of the current tree, and the result is added to the solution. Then the occurrence list (or scopelist) is updated and this tree, with the updated occurrence list, is further extended. Otherwise the current tree is pruned.

6 Experimental Results

The goal of the experiments is twofold: first to show the applicability of FAT-miner, and second to compare the runtime behavior of FAT-miner and the naive approach. To demonstrate the applicability of FAT-miner, we analyzed a multi-relational financial dataset and found interesting discriminating patterns. For the behavior of FAT-miner in comparison with the naive algorithm we used two real datasets. These experiments support our theoretical observations made in subsection 5.1.

6.1 Interesting rules in a financial database

We applied FAT-miner to a financial multi-relational database, provided for the PKDD1999 and PKDD2000 discovery challenge [4]. The database contains data from a Czech bank, and describes the operations of 5369 clients holding 4500 accounts. The data is distributed over eight tables, which are shown together with the database scheme in the left part of figure 3.

The clients with a granted loan are divided into two subgroups: the first with a good loan status; the second with a bad loan status. A good loan is defined as one where the contract is finished and the loan has been paid back or the contract is still running and there are no payment problems so far. A loan is defined as bad if the contract is finished

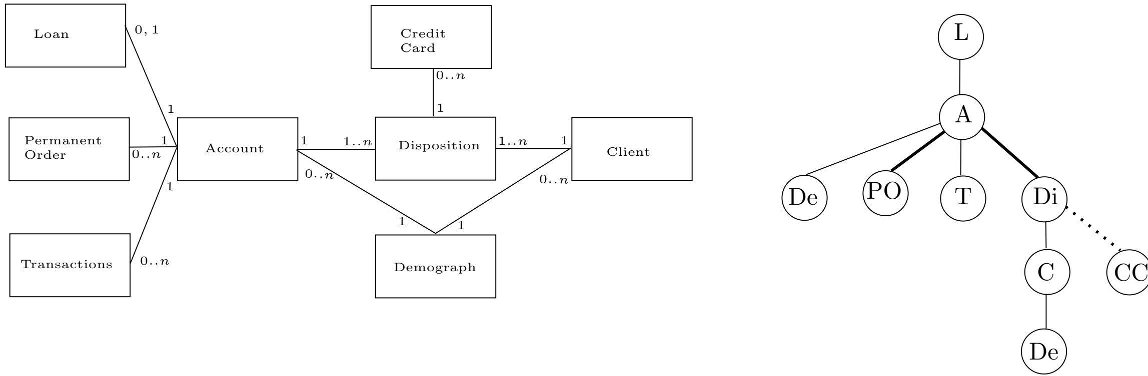


Figure 3: Left the database scheme is given. Right the corresponding tree is shown, where bold lines denotes that there can be multiple occurrences of that node; the dotted line denotes that the node labeled 'CC' can occur zero or more times. Note that the node labels are abbreviations of the table names.

and the loan has not been paid back or the contract is still running and the client is in debt. The objective of the experiments is to find discriminating patterns that describe the clients with good and bad loan respectively.

The database was restricted to the part relevant for the analysis (this was done by Krogel [4]), all accounts, clients and transactions that have no associated loans were removed. Furthermore, all transactions that occurred after a loan was granted were removed. After the data cleaning, we aggregated the transaction relation. This was done because of the huge number of transactions and because it is questionable whether many individual transaction-patterns leads to interesting results. The aggregation was done as follows: for each client the number of transactions was counted and divided by the time period (in days) in which these transactions took place. Furthermore, for every nominal attribute (like mode of transaction, type of transaction and characterization of the transaction) the values were counted per client and divided by the time period. These extra attributes were added to the dataset. Finally for the numeric attributes, the average values over the time period were computed per client.

The resulting dataset consists of 682 records in the relations loan, transaction and account, 827 records in the relation clients and disposition, 36 records in the relation credit card and 1513 records in the relation permanent orders. The demograph table was not modified. The next step was to discretize numeric attributes, this was done with the ProSaffarii [9] preprocessing toolkit. For each relation the numeric attributes of this relation are divided into five consecutively numbered intervals containing approximately the same number of records. Each attribute is then changed into a nominal attribute where the value equals the number of the interval.

Finally the database was transformed to a rooted ordered attribute tree. Strictly speaking the financial database structure is not a tree: there is a cycle connecting the tables account,demograph, disposition and client. However, because demograph is a static back-

ground table, this cycle can be removed without loss of information. Further we need to choose a root node and define an order among the siblings. Because the primary interest of the analysis is to describe properties of good and bad loans, the loan table is chosen as the root of the tree. The order among the siblings is chosen arbitrarily, because for the analysis the particular order used is irrelevant. The resulting tree is shown in figure 3.

The primary measure of interest is the support of a pattern within a class, i.e., $P(T|\text{class})$. Given our classes of bad and good loans (c_0 and c_1 respectively), let D_{c_0} denote the selection of all $d \in D$ for which it holds that $d \in c_0$, equivalently D_{c_1} is the selection of all $d \in D$ that satisfy $d \in c_1$. Note that for each $d_i \in D$ either $d_i \in c_0$ or $d_i \in c_1$ holds. Then the support of T within class c_i is defined as: $\Psi_{c_i}(T) = \sum_{d \in D_{c_i}} \sigma_d(T)$ and its relative support within class c_i as: $\Psi_{c_i}(T)/|D_{c_i}|$. Another measure of interest is by what factor observing a pattern changes the class probability, compared to the class prior probability, i.e., $P(\text{class}|T)/P(\text{class})$ also known as the lift of a rule $T \rightarrow \text{class}$.

We conducted two experiments on the dataset. In the first experiment the subtree inclusion relation in FAT-miner was set to induced subtrees, in the second run we used the embedded subtree relation. In both experiments the minimum relative-support for both good and bad loans was set to 5%. The database proportion of good versus bad loans in the database is 606 vs. 76, hence 11.1% of the records in the database describe bad loans.

Some examples of discovered patterns for the induced subtree relation are shown in figure 4, in this figure **FG** stands for the relative support of the pattern within the class of good loans, **FB** for the relative support within the class of bad loans and finally the rule and the lift of the rule are given. The patterns with a high relative support within the class of good loans (numbers 1–3), have a relatively low lift. This is mainly because the prior probability of the class of good loans equals 0.89. As a consequence of the prior probability, the patterns with a high relative support within the class of bad loan (numbers 4–6) have a relative high lift value.

For the embedded subtree relation some example-results are shown in figure 5. Note that we show here examples of patterns that are embedded trees only. As in the case of the induced patterns, here also the lift of the rules varies a lot between the two classes. A typical type of pattern found is best illustrated with an example. Consider patterns two, three and five; these patterns start with the node labeled 'loan' followed by children of the node 'account'. In these cases the pattern describes subgroups that have the same values for the attributes of loan, and for the children of the account node but, that may not have a common value for one of the attributes of the intermediate relation account.

1. **FG** 21.45 %, **FB** 3.95%, **LIFT**($T \rightarrow$ **G**)=1.10
   Loan where monthly payments is in range (304–2051) while the overall range = (304–9910)
   Account where frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='loan payment'.
2. **FG** 19.97%, **FB** 0% , **LIFT**($T \rightarrow$ **G**)=1.13
   Account where frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='loan payment'
   Disposition where type of disposition='owner'
   Disposition where type of disposition='user'.
3. **FG** 28.22%, **FB** 3.95% , **LIFT**($T \rightarrow$ **G**)=1.11
   Account where frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='household payment'
   Permanent Order where characterization of the payment ='loan payment'
   Transaction where the average number of transactions a day of type = 'credit' is in range (0.081–0.098), while the overall range = (0.0224–0.157)
   Disposition where type ='owner'.
4. **FB** 39.47%, **FG** 16%, **LIFT**($T \rightarrow$ **B**)=2.12
   Accountwhere frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='loan payment'
   Transaction where the average number of transactions a day with characterization =' interest credited' is in range (0.033–0.1), while the overall range = (0–0.1)
   Disposition where type of disposition='owner'.
5. **FB** 18.42%, **FG** 4.78%, **LIFT**($T \rightarrow$ **B**)=2.92
   Account where frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='loan payment' & debited amount is in range (6630–14882), while the overall range = (2–14882)
   Transaction where the average number of transactions a day of type = 'credit' is in range (0.081–0.098), while the overall range = (0.022–0.157)
   Disposition where type of disposition='owner'
   Client where sex ='female' .
6. **FB** 13.16% ,**FG** 1.48%, **LIFT**($T \rightarrow$ **B**)=4.72
   Account where frequency of issuance statements='monthly'
   Permanent Order where characterization of the payment ='loan payment'
   Transaction where the average number of transactions a day with characterization =' interest credited' is in range (0.033–0.1), while the overall range = (0–0.1)
   Disposition where type of disposition='owner'
   Client where sex ='Male'
   District where number of municipalities with inhabitants 2000–9999 is in range (0–3), while the overall range = (0–20).

Figure 4: Example patterns returned by FAT-miner for the class of good loans ($1 - -3$) and the class of bad loans ($4 - -6$). The minimum relative-support for both classes was set to 5% and the induced subtree relation was used.

1. **FG** 31.19%, **FB** 5.26% , **LIFT**$(T \rightarrow \mathbf{G})$=1.11
Account where frequency of issuance statements='monthly'
Permanent Order where characterization of the payment ='household payment'
Transaction where the average number of transactions a day of type = 'credit' is in range (0.081–0.098), while the overall range = (0.022–0.157)
Client where sex ='Male'.
2. **FG** 22%, **FB** 3.95% , **LIFT**$(T \rightarrow \mathbf{G})$=1.10
Loan where the amount that is monthly payed is in range (304–2051), while the overall range = (304–9910)
Permanent Order where characterization of the payment ='loan payment'
Transaction where the average number of transactions a day of type ='credit' is in range (0.033–0.1), while the overall range = (0–0.1)
Disposition where type of disposition='owner'.
3. **FG** 19.14%, **FB** 2.63% , **LIFT**$(T \rightarrow \mathbf{G})$=1.11
Loan where the amount of money borrowed is in range (4980–52788), while the overall range = (4980–590820)
Permanent Order where characterization of the payment ='loan payment'
Transaction where the average number of transactions a day of type ='credit' is in range (0.033–0.1), while the overall range = (0–0.1)
Disposition where type of disposition='owner'.
4. **FB** 25%,**FG** 6.76%, **LIFT**$(T \rightarrow \mathbf{B})$=3.41
Account where frequency of issuance statements='monthly'
Permanent Order where characterization of the payment ='loan payment'
Transaction where the average number of transactions a day with characterization =' interest credited' is in range (0.033–0.1), while the overall range = (0–0.1)
Disposition where type of disposition='owner'
District where the number of municipalities with inhabitants $> 1000 = 2$, with the overall range = (0–5).
5. **FB** 19.74%,**FG** 8.74%, **LIFT**$(T \rightarrow \mathbf{B})$=1.98
Loan where duration of the loan ='24 months'
Permanent Order where characterization of the payment ='loan payment'
Transaction where the average number of transactions a day of type ='credit' is in range (0.033–0.1), while the overall range = (0–0.1) & where the average number of transactions a day with type = 'withdrawal' is in range (0.075–0.101), while the overall range = (0–0.26)
Disposition where type of disposition='owner'.
6. **FB** 21.05%, **FG** 4.95%,**LIFT**$(T \rightarrow \mathbf{B})$=3.12
Account where frequency of issuance statements='monthly'
Permanent Order where characterization of the payment ='loan payment'
Transaction where the average number of transactions a day with characterization =' interest credited' is in range (0.033–0.1), while the overall range = (0–0.1)
Disposition where type of disposition='owner'
District where the number of committed crimes in 1995 are in range (5179–85677), while the overall range = (818–85677) & the number of committed crimes in 1996 are in range (4987–99107), while the overall range = (888–99107).

Figure 5: Example patterns returned by FAT-miner for the class of good loans $(1 - -3)$ and the class of bad loans $(4 - -6)$. The minimum relative-support for both classes was set to 5% and the embedded subtree relation was used.

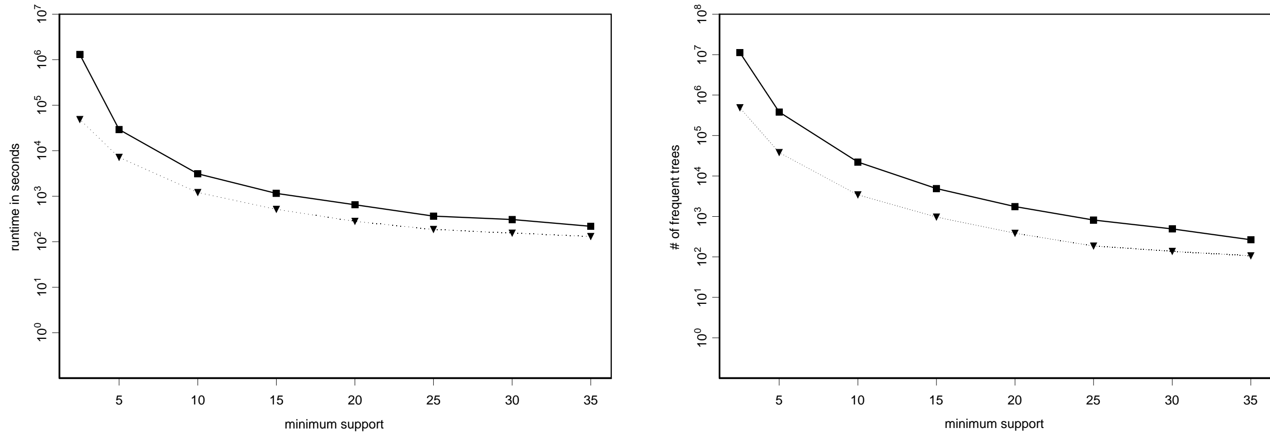


Figure 6: The running time (left) and the number of frequent trees reported (right), compared for FAT-miner (dotted line) and the naive approach (solid line). These experiments were done on the Wikipedia XML dataset with the induced subtree relation. Note the log scale on the y-axis.

6.2 Performance Experiments

In this subsection, both the run time and the number of frequent trees generated by FAT-miner and the naive approach are experimentally compared. All algorithms were implemented in C++ and were evaluated on a 2.8GHz PC with 500 MB of RAM. The comparison was done on two real datasets: the first is the financial multi-relational dataset, as described in the previous subsection, but without the class labels. The second is the Wikipedia XML corpus [7], which is an XML collection based upon Wikipedia. The collection consist of 75,047 documents and 60 classes. For our experiments we discarded the class labels and performed the experiments on a random sample of approximately one third of the original collection.

A sample was taken such that both algorithms that use the induced subtree relation were able to run this dataset on a desktop computer with 500MB of main memory. Besides the memory constraint, the run time for the naive algorithm already took quite some time, which should have been worse for a larger collection. Unfortunately, the algorithms that use the embedded subtree relation could not run on a server having 2GB of main memory available. The reason for this is described in [5]: in the worst case, for the embedded subtree mining algorithm [16] the scopelist size is exponential in the size of the data tree. Since FAT-miner with the embedded subtree relation is built upon treeminer [16] it has the same memory requirements.

The XML sample consist of 25,127 trees and 8,310 distinct node labels; of these nodes 389 contained attributes. When the data was loaded into FAT-miner, each node in the database that had no attribute was assigned a dummy attribute. The average number of nodes in the trees equals 85, with each attribute counted as a single node, the average size

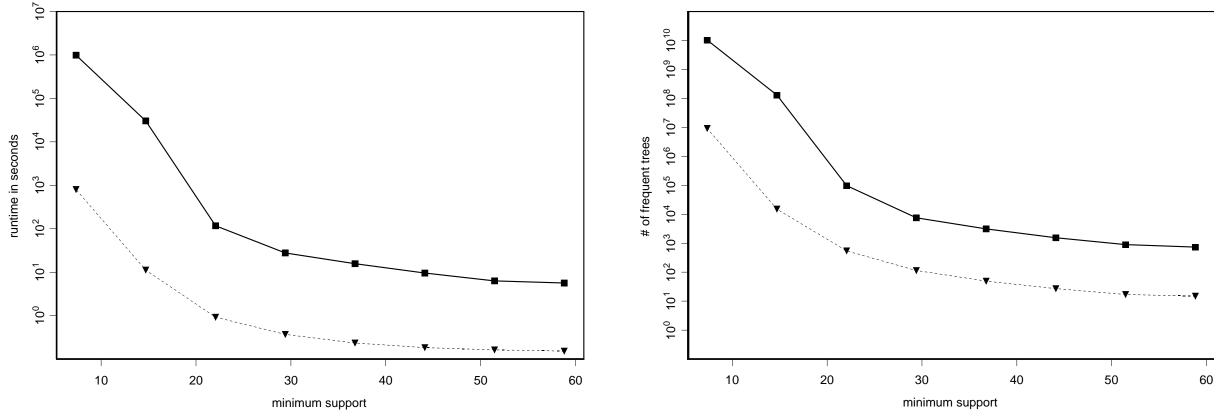


Figure 7: The running time (left) and the number of frequent trees reported (right), compared for FAT-miner(dotted line) and the naive approach (solid line). These experiments were done on the financial multi-relational dataset with the induced subtree relation. Note the log scale on the y-axis.

of the trees was 128. Both FAT-miner and the naive approach where run with different minimum support values; the run time and the number of frequent trees for different support values are shown in figure 6. Due to the long execution time of the naive algorithm at the lowest minimum support level, this run was terminated after two weeks. Remarkable is that the difference between FAT-miner and the naive approach for both the execution time and the number of frequent trees initially is relatively small, but is increasing drastically at the lowest levels of support. This is likely caused by the fact that only a relatively small part of the data contained attributes, and hence the speedup of FAT-miner is only based on this small part. However, with lower support the attributes that become frequent are dominating both the run time and the number of frequent trees, and hence FAT-miner achieves a better reduction in the number of frequent trees and consequently a better speedup.

The financial multi-relational dataset consist of 682 trees and 9 distinct node labels, each of these nodes contained attributes. The average number of nodes in the trees equals 9, with each attribute counted as a single node, the average size of the trees was 65. Comparing the statistics of the financial and the XML dataset, reveals that the financial dataset is small, but has a large number of attributes per tree in the database. Also in these experiments, both FAT-miner and the naive approach where run with different minimum support values and with the two different subtree relations. The amount of main memory used for both FAT-miner and the naive approach was 8MB for the runs with the induced subtree relation and 31MB for the runs with the embedded subtree relation. The results are shown in figure 7 for the induced subtree relation and in figure 8 for the embedded one. Note that for the latter, the run of the naive algorithm with the lowest minimum support was terminated after more than two weeks of execution time. Both The number of additional trees and the additional run time of the naive approach over FAT-

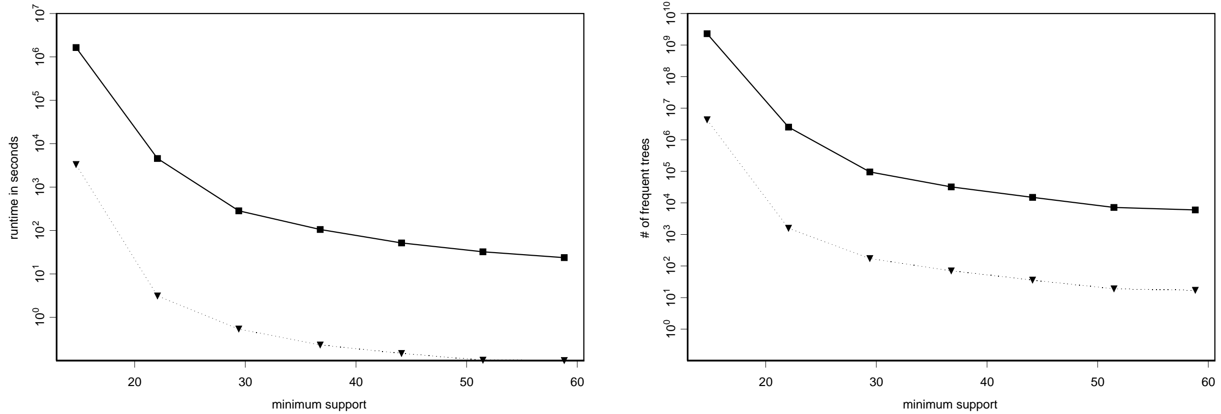


Figure 8: The running time (left) and the number of frequent trees reported (right), compared for FAT-miner(dotted line) and the naive approach (solid line). These experiments were done on the financial multi-relational dataset with the embedded subtree relation. Note the log scale on the y-axis.

miner is initially large and increase drastically as the minimum support drops. However, the difference is a slowly increasing factor for the higher minimum support values, and increased drastically for the lower minimum support values.

To conclude, the experimental results show that when mining frequent attribute trees FAT-miner reduces the run time drastically in comparison with a straightforward approach. Even for the relative small financial dataset the execution time is reduced from more then eleven days in less then fifteen minutes. This conclusion is in accordance with the theoretical comparison in subsection 5.1.

7 Related Work

Besides the tree mining algorithms of which FAT-miner is an extension [2, 16], there are some more algorithms available to mine frequent trees. They mostly differ in what type of tree is handled (ordered,unordered or free trees) or use subtly different enumeration techniques; for a detailed overview see [5]. Further generalization are algorithms that mine frequent graph patterns such as [14, 8]. A common property of these algorithms to mine structured data is that there is no proper way to handle attributes. This in contrast with the multi-relational data mining field, where attribute values are included into the mining algorithms. Related work in the multi-relational data mining field, includes the following. The work done by Ng et. al [11] describes an algorithm to mine association rules from star-shaped databases. While their algorithm is limited to database schemes that are star shaped, Safarii [10] and Warmr [6] can both mine graph-structured data. However, Safarii uses a heuristic search to mine multi-relational databases, in contrast

with the complete search done by FAT-miner. Like FAT-miner Warmr also performs a complete search, but Warmr faces a high computational complexity due to equivalence checking under θ-subsumption. In addition, FAT-miner has the following advantages:

- The ability to find hidden patterns: our algorithm is able to mine for both induced and embedded subtrees. With the embedded subtree relation it is not necessary that there is a direct link between different parts of the pattern; an ancestor/descendant relation is sufficient. This in contrast to multi-relational data mining algorithms.

- Exclusive pattern matching: whenever two parts of a pattern are present, it is assumed that these two parts are distinct. In multi-relational mining, on the contrary, multiple parts of the pattern can map on the same part of the data. That is, as opposed to FAT-miner, multi-relational data mining algorithms can not differentiate between the following two example patterns: a node labeled 'account' that has two children labeled 'disposition' of type 'user' and a node labeled 'account' that has one child 'disposition' of type 'user'.

8 Conclusion

In this work we developed FAT-miner, a frequent tree mining algorithm that takes attributes of tree structured data into account. As a result FAT-miner is able to mine multi-relational databases with tree structured database schemes, as well as XML data. We compared the complexity of FAT-miner with a naive approach both theoretically as well as experimentally. Both analyses revealed that the execution time is reduced drastically by FAT-miner. Furthermore, the usability of FAT-miner was shown on a well known financial database. Further research includes the development of condensed representations for attribute trees and algorithms to mine these directly. In addition we intend to investigate the incorporation of user defined constraints to assist mining process. Finally, building classifiers from frequent attribute trees is a topic we plan to investigate in the near future.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[3] R. Bayardo. Efficiently mining long patterns from databases. In A. T. Laura and M. Haas, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 85–93, 1998.

[4] P. Berka. Guide to the financial data set. http://lisp.vse.cz/challenge/. Workshop notes on Discovery Challenge PKDD2000.

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

[6] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 125–132. Springer-Verlag, 1997.

[7] L. Denoyer and P. Gallinari. The Wikipedia XML Corpus. *SIGIR Forum*, 2006.

[8] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In D. A. Zighed, H. J. Komorowski, and J. M. Zytkow, editors, *Principles of Data Mining and Knowledge Discovery (PKDD 2000)*, pages 13–23, 2000.

[9] A. Knobbe. Prosafarii. http://www.kiminkii.com/safarii.html.

[10] A. Knobbe. *Multi-Relational Data Mining*. PhD thesis, Universiteit Utrecht, 2004.

[11] E. Ng, A. Fu, and K. Wang. Mining association rules from stars. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 322–329, 2002.

[12] A. Termier, M. Rousset, M. Sebag, K. Ohara, T. Washio, and H. Motoda. Efficient mining of high branching factor attribute trees. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*, pages 785–788, 2005.

[13] K. Wang and H. Liu. Discovering structural association of semistructured data. *Knowledge and Data Engineering*, 12(2):353–371, 2000.

[14] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724, 2002.

[15] L. H. Yang, M. Lee, W. Hsu, and S. Acharya. Mining frequent query patterns from XML queries. In *Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, pages 355–362, 2003.

[16] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002.

[17] M. J. Zaki and C. C. Aggarwal. Xrules: an effective structural classifier for XML data. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 316–325, 2003.