Universidad de Granada

Departamento de Ciencias de la Computación
e Inteligencia Artificial

Knowledge discovery in non-linear structures

Tesis Doctoral

Aída Jiménez Moscoso del Prado

Granada, Febrero de 2011

Editor: Editorial de la Universidad de Granada
Autor: Aída Jiménez Moscoso del Prado
D.L.: GR 2418-2011
ISBN: 978-84-694-2919-8

Universidad de Granada

Knowledge discovery in non-linear structures

MEMORIA QUE PRESENTA

Aída Jiménez Moscoso del Prado

PARA OPTAR AL GRADO DE DOCTOR EN INFORMÁTICA

Febrero de 2011

DIRECTORES

Fernando Berzal Galiano

Juan Carlos Cubero Talavera

Departamento de Ciencias de la Computación
e Inteligencia Artificial

La memoria titulada "*Knowledge discovery in non-linear structures*", que presenta Dña. Aída Jiménez Moscoso del Prado para optar al grado de doctor, ha sido realizada dentro del Programa Oficial de Posgrado en *"Ciencias de la Computación y Tecnología Informática"* del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada bajo la dirección de los doctores D. Fernando Berzal Galiano y D. Juan Carlos Cubero Talavera.

Granada, Febrero de 2011

La Doctoranda

Los directores

Fdo: Aída Jiménez Moscoso del Prado

Fdo: Fernando Berzal Galiano

Fdo: Juan Carlos Cubero

Table of Contents

Part I. PhD dissertation

"If you torture the data long enough,
it will confess"
[Ronald Coase]

1 Introduction

Knowledge discovery from data, or KDD, has been defined as the non-trivial extraction of implicit, previously unknown, and potentially useful information from large amounts of data [58]. Knowledge discovery is a process that consists of an iterative sequence of the following steps [30]:

- Data preprocessing: Data is cleaned, integrated, selected, and transformed into appropriate form for data mining.
- Data mining, where algorithmic techniques are applied to the data in order to extract patterns.
- Pattern evaluation: The extracted patterns are evaluated using some interestingness measures to identify the truly interesting ones.
- Knowledge presentation, where visualization and knowledge representation techniques are used to present the mined knowledge to the user.

Data mining is an essential step in this process because it uncovers hidden patterns from data. Most of the data mining techniques that have been developed in the last two decades deal with linear structures (e.g., identifying frequent patterns and sequences in lists of transactions or data tables).

However, many scientific and commercial applications require the analysis of more complex patterns [30]. The identification of patterns in non-linear

```
<customer>
   <order>
      <item>milk</item>
      <item>bread</item>
   </order>
   <order>
      <item>beer</item>
      <item>diapers</item>
   </order>
</customer>
```

```
public int TestNestedFor (int n) {
  int sum = 0; 1
  for (int i=0; 2  i<n; 8  i++) 7
     for (int j=0; 3  j<n; 6  j++) 5
        sum += i*j; 4
  return sum; 9
}
```

Figure 1: A simple XML document (left) and a simple code snippet (right).

structures (like trees, networks, or graphs, in general) is useful for modeling these structures and for studying the interactions among their components.

In Bioinformatics, for example, graphs provide a natural representation for chemical compounds and, with the analysis of these graphs, it is possible to identify characteristics that compounds have in common and also to highlight their differences. This is useful in databases like Mutagenesis [20], which contains a set of chemical compounds labeled as muted or non-muted, and in other similar datasets [22][40].

Another field of application is the analysis of social networks, which constitutes an approximation of the study of the social interactions within different groups of people. These social relationships are usually represented as graphs and can be used, for example, to study community structure [18].

In the Internet, the structure of HTML pages (and the links between them), allows the identification of patterns in tree and graph format. These patterns can be used for classifying web pages, which is useful for improving the results to Web queries [43][65]. Furthermore, it is possible to extract information about the structure of the Web in order to identify new web page communities or study the relevance index of a web page within the results of a query (as done by Google's PageRank) [13].

Figure 1 shows two application areas where trees provide a good representation model: XML documents and software representation.

XML has become a popular format for the storage and exchange of data due to its flexible semi-structured nature, which allows the representation of a wide variety of databases as XML documents. XML data thus forms an important data mining domain [68]. In the past few years, some novel techniques have been proposed for dealing with XML documents and their structure. Frequent discriminatory substructures within XML documents have been used to perform rule-based classification of XML data in XRules [68] and in effective clustering algorithms for XML data [2]. Apart from being at the heart of many interesting and challenging data mining problems, the discovery of frequent XML patterns is also useful in many other situations. For instance, the discovery of frequent XML query patterns in the history log of XML queries can be used to expedite XML query processing, as the answers to these queries can be cached and reused when the future queries "hit" such frequent patterns [7].

In Nature, complexity frequently takes the form of hierarchic systems - the complex system being composed of subsystems that in turn have their own subsystems -, maybe because hierarchic systems can evolve far more quickly than non-hierarchic systems of comparable size [55]. Hierarchies also appear in human problem solving and in Software Engineering (a domain with no obvious connections with natural evolution). In the form of trees, hierarchies provide relatively simple descriptions for complex systems when they are decomposable (or near decomposable). Dependence higraphs [10], for instance, can be used for representing a hierarchical model of software systems.

Apart from the aforementioned applications, there are many other applications of trees and graphs in the resolution of current problems [18]: VLSI circuit design, the Internet Movie Database(http://www.imdb.com), information retrieval, multirelational databases, etc..

The use of non-linear data structures in many interesting problems has spurred the interest of data mining researchers in the development of efficient and scalable data mining techniques for these special data structures. Trees, in particular, have recently attracted the attention of the research community, in part because they are particularly amenable to efficient pattern mining techniques. Identifying frequent patterns in a database of trees is an important task in solving many tree mining problems. These frequent patterns, also known as frequent subtrees, represent common substructures in a tree database.

Formally, a **tree** is a connected and acyclic graph. In a tree, these two properties hold:

- There is an unique path between each two vertices.
- $\#E = \#V - 1$, where $\#E$ and $\#V$ are the number of edges and vertices, respectively.

Trees can be classified according to their structural properties as free trees or rooted trees:

- A tree is said to be **free** if its edges have no direction, that is, when it is an undirected graph. A free tree, therefore, has no predefined root.

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

  Rooted trees can be classified as:

  - **Ordered trees**, when there is a predefined order within each set of sibling nodes in the tree.
  - **Unordered trees**, when there is not such a predefined order among sibling nodes in the tree.
  - **Partially-ordered trees** contain both ordered and unordered sets of sibling nodes within the same tree. This distinction is useful when there is an order between some sibling node sets but it is not necessary to establish an order relationship within all the sibling node sets of the tree.

Partially-ordered trees appear in different application domains, for example, Figure 2 (left) shows the partially-ordered tree representing the purchase history of a particular customer shown in the XML document in Figure 1 (left). In this case, taken from a typical scenario in market basket analysis, it is important to preserve the order among the different orders placed by the same customer, while the order among the items in the same order is irrelevant.

Another example application of partially-ordered trees can be found in software mining. Figure 2 (right) shows the higraph [10] representation of the code snippet in Figure 1(right). When we represent the structure of a software program as a hierarchy, program segments can be represented as nodes within trees. Existing dependences among program segments can be made explicit as order relationships among tree nodes. The resulting trees are, therefore, partially-ordered trees.

As the goal in itemset mining is to find the frequent itemsets in a transactional database, the **frequent tree pattern mining task** consists of discovering of all the frequent subtrees in a large database of trees, also referred to as a forest, or in a unique large tree.

A **subtree** can be formally defined just as a subgraph of a tree. However, different kinds of subtrees can result depending on the way the relationships between the nodes within a tree are preserved:

- **Bottom-up subtrees**: A bottom-up subtree T' of T (with root v) can be obtained by taking one vertex v from T with all of its descendants and corresponding edges, and preserving the order between siblings if it exists.

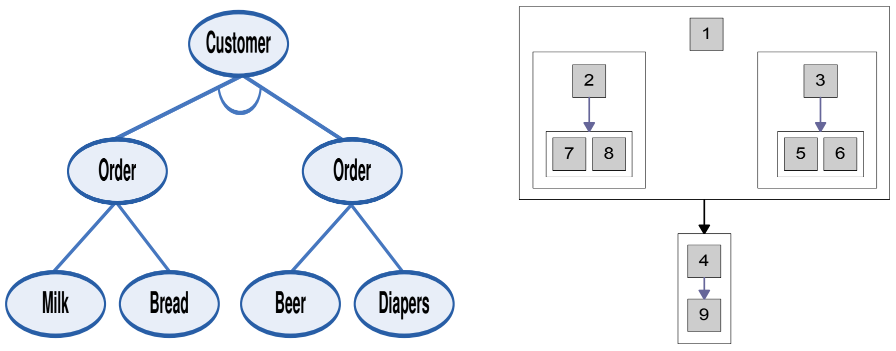


Figure 2: Partially-ordered tree representing XML document in Figure 1 (left) and the dependence higraph corresponding to the code snippet in Figure 1 (right), which can also be interpreted as a partially-ordered tree.

- **Induced subtrees**: We can obtain an induced subtree T' from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T.
- **Embedded subtrees**: An embedded subtree T' cannot break the ancestor relationships among the vertices of T.

The approach we present in this Ph.D. dissertation is embraced in the tree mining field. We propose a new algorithm, called POTMiner, which deals with situations that previous algorithms could not address. In order to improve the time efficiency and reduce the memory consumption of POTMiner, improved versions of our algorithm have also been developed.

These algorithms correspond to in the data mining step of the KDD process. In this dissertation, however, we also address three different applications where other stages of the whole KDD process, such as data preprocessing or pattern evaluation, are involved:

- Mining multirelational databases, where we apply our algorithms to extract frequent patterns and association rules from this kind of databases.
- Group association rules, a new kind of association rules that lets us discover and study groups of individuals within a given database and rank them in order to highlight the most interesting ones according to different criteria.
- The identification of musical motifs, where we are able to find frequent melodic and rhythmical patterns in music.

This dissertation is divided into two parts. The first one is devoted to the description of the problems that we have addressed and the discussion of the results we have obtained. The second one collects the publications where our research has been featured.

In this introduction to Part I, we have already motivated the issues we address in this Ph.D dissertation. In Section 2, we present the goals we pursued during the elaboration of this dissertation. The summary of the results we have achieved in each one of the specific problems we have addressed is presented in Section 3. Section 4 presents our conclusions while, in Section 5, we provide some pointers to future work.

Part II of this Ph.D. dissertation collects the journal papers that have resulted from the research we have conducted. Those papers have been classified into two groups: those that are directly related to frequent tree pattern mining and those that correspond to applications.

2 Objectives

The present dissertation is organized around six objectives that involve the analysis of existing tree mining techniques, the development of efficient algorithms that address the partially-ordered tree mining problem, and the application of data mining techniques to different problem domains.

In particular, the objectives we aimed at during the elaboration of this Ph.D dissertation were the following ones:

- **Analysis of existing tree mining techniques.** One of the goals of this dissertation is to provide a full survey of the state of the art in the tree mining area, studying the different tree mining algorithms proposed in the literature. An overview that analyzes a few tree pattern mining algorithms in terms of their efficiency was proposed in 2005 [15]. However, many tree mining algorithms have been developed since then. We will also propose a different approach in our survey, where the algorithms are compared in terms of their similarities and their peculiarities are highlighted. Our review will provide a classification of existing algorithms according to the kind of input trees they can be applied to and the kind of subtrees they are able to identify within their input trees.

- **Partially-ordered tree mining.** Although many tree mining algorithms had been developed prior to this work, none of them was able to deal with partially-ordered trees. Partially-ordered trees are useful in several domains like XML document mining or program representation, as explained in Section 1. One of the main goals of this dissertation is to deal with this kind of patterns. Hence, we will develop an algorithm, called POTMiner (Partially-Ordered Tree Miner), which will be able to mine partially-ordered trees as well as completely-ordered and completely-unordered trees. Our algorithm will also be able to identify induced and embedded patterns in all those trees.

- **The design of efficient algorithms** is another objective of this dissertation. Since tree mining algorithms must work with huge amounts of data, efficiency is a requirement for those algorithms, not only in CPU time but also in memory consumption. Therefore, refinements of the POTMiner algorithm will be devised in order to be more efficient both in CPU time and in memory consumption. The results of these refinements will be a parallel version of POTMiner and two new variants of the POTMiner algorithm that make different memory/CPU-time trade-offs.

- With respect to the applications of our fundamental research, we will address several problems in three different areas:

  – **Multirelational data mining**: Several algorithms have been proposed in order to mine multirelational databases. Those that resort to join-based techniques do not preserve the proper support counts, and others are especially devised for solving classification or clustering problems. Our goal in this area is to extract frequent patterns from multirelational databases. Our proposal will be based on transforming the multirelational database into a set of trees. This way, efficient tree pattern mining algorithms can be applied to multirelational databases. We will also extract association rules from multirelational databases using the identified tree patterns.

  – **Group association rules:** We will show how a new kind of association rule which we have called group association rule, is useful to identify groups in databases. Using classical interestingness measures defined for association rules as our basis, we will also define interestingness measures for group association rules, which will let us rank the group association rules and the groups within the database. This ranking will highlight the most interesting groups and will help expert users sift through the vast number of rules that are often obtained when mining a real-word database.

  – **Identifying transposed motifs in music.** The problem of motif extraction in a piece of music can be seen as a particular case of mining frequent patterns from a sequence when using the symbolic representation of the song. Another goal of this dissertation is to adapt our algorithms to deal with sequences, as well as with the particular kind of similarity metric between motifs that is suitable for this particular application domain (i.e., transpositions).

3 Discussion of results

This section summarizes the different proposals contained within this dissertation and the main results we have obtained fom each one. Two different subsections group our results in the frequent tree pattern mining area and in the applications we have addressed.

3.1 Frequent tree pattern mining

In this section, we present the results obtained during the elaboration of this dissertation that are directly related to the frequent tree pattern mining problem: a full review of the state of the art, the design and implementation of a new tree mining algorithm devised to deal with partially-ordered trees, and the improvements on this algorithm that make it more efficient in terms of CPU time as well as in terms of memory consumption.

3.1.1 State of the art

Two well-known algorithms are commonly used to identify frequent patterns in transactional databases: Apriori [4] and FP-Growth [31].

Both Apriori and FP-growth algorithms have been extended to mine frequent substructures in tree databases, and many frequent tree pattern mining algorithms proposed in the literature are derived from them. Most of the existing proposals, however, follow the Apriori iterative pattern mining strategy. Using the Apriori iterative pattern mining strategy [4], each iteration is broken up into two distinct phases:

- *Candidate Generation*: Potentially frequent subtree candidates are generated from the frequent patterns discovered in the previous iteration. Most Apriori-like algorithms generate candidates of size $k+1$ by merging two trees of size k having $k-1$ elements in common.
- *Support Counting*: Given the set of potentially frequent candidates, this phase consists of determining their actual support and keeping only those candidates that are actually frequent (a subset of the candidates generated in the previous phase).

Different algorithms have been devised in order to deal with the different kinds of trees and subtrees we mentioned in Section 1. FreqT [1], AMIOT [32], TreeMiner [67], X3Miner [56], and IMB3Miner [57] are algorithms that work with ordered trees. Unordered trees have also been mined using algorithms such as Unot [5], SLEUTH [66], or Uni3 [29]. Most of the algorithms devised to mine free trees are also able to deal with unordered trees, such as HybridTreeMiner [17], Gaston [47] and two different algorithms that were given the same name, FreeTreeMiner [16] [52].

There are also algorithms that follow the FP-Growth pattern growth approach, which does not explicitly generate candidates. For instance, the PathJoin algorithm [62] uses compacted structures called FP-Trees to encode input data, while Chopper and XSpanner [59] use a sequential codification for trees, which lets them extract frequent subtrees by discovering frequent subsequences.

In order to obtain a complete survey of the state of the art in this area, a handful of noteworthy tree pattern mining algorithms have been analyzed [37]. They have been compared from different points of view in order to highlight their similarities and recognize their dissimilarities.

On the one hand, these algorithms have been classified according to the kinds of input trees they can be applied to (ordered, partially-ordered, unordered, or free) and the kinds of subtrees they are able to identify within their input trees (induced, embedded, incorporated, or subsumed). We have also pointed out which of those algorithms have been especially devised for discovering closed and/or maximal tree patterns.

On the other hand, the following structural properties of tree pattern mining algorithms have been fully studied:

- *Tree representation*: When mining a tree database, it is useful to have a unique representation for each tree, commonly a string. This canonical representation makes the problems of tree comparison and subtree enumeration easier. Several tree representation schemes have been proposed in the literature.

- *Candidate generation approach*: As explained above, most of the algorithms proposed in the literature follow the Apriori iterative pattern mining strategy. Therefore, they decompose each iteration into two main steps: candidate generation and support counting. We have studied the different candidate generation approaches and, when applicable, each algorithm has been classified according to its candidate generation strategy. Within the Apriori-based algorithms, most of them resort to the rightmost expansion method or employ Zaki's class-based extension technique for candidate generation [67] [66].

- *Implementation details*: The specific implementation details of each algorithm have also been studied. In some cases, these details correspond to the support counting strategy chosen for each algorithm. In other cases, those details involve the introduction of ancillary data structures, which are used by the algorithms to be more efficient. For example, vertical [29] [57], RMO [1] [32], or plain occurrence lists [17] [5] are typically used with the right-most expansion method, while scope lists [67] [66] are used for those that resort to Zaki's class-based extension, in order to speed up the support counting phase.

The journal article associated to this part of the dissertation is:

A. Jiménez, F. Berzal, and J.C. Cubero.
Frequent tree pattern mining: A survey
Intelligent Data Analysis 14(6):603-622,2010.
DOI 10.3233/IDA-2010-0443.

3.1.2 POTMiner: Mining partially-ordered trees

We have extended Zaki's TreeMiner [67] and SLEUTH [66] algorithms to mine partially-ordered trees. The algorithm we have devised is able to identify frequent subtrees, both induced and embedded, in ordered, unordered, and partially-ordered trees. Hence its name, POTMiner [38], which stands for Partially-Ordered Tree Miner.

Our algorithm is based on Apriori [4], just like TreeMiner and SLEUTH. Therefore, it follows an iterative pattern mining strategy. The $k-th$ iteration of the algorithm mines the frequent subtrees of size k starting from the frequent subtrees of size $k-1$ discovered in the previous iteration. Each iteration, as in Apriori, is subdivided in two phases, candidate generation and support counting.

- *Candidate generation*: POTMiner uses Zaki's class-based extension strategy. This method generates $(k+1)$-subtree candidates by joining two frequent k-subtrees with $k-1$ elements in common. Candidates that share $k-1$ elements in common are grouped into equivalence classes. Then, elements in the same equivalence class are joined to create a new candidate. It is important to note that this strategy does not generate duplicate candidates.

- *Support counting*: Once POTMiner has generated the potentially frequent candidates, it is necessary to determine which ones are actually

```
algorithm POTMiner

Obtain frequent nodes (frequent patterns of size 1)
Build candidate classes $C_1$ from the frequent nodes
for k=2 to MaxSize
    for each class $P \in C_{k-1}$
        for each element $p \in P$.
            Compute the frequency of $p$
            if $p$ is frequent
            then
                Create a new class $P'$ from $p$.
                Add $P'$ to $C_k$
```

Figure 3: POTMiner algorithm pseudocode.

frequent by counting their support. Instead of checking if each candidate is present in each database tree, which is a costly operation when dealing with trees, occurrence lists are used to preserve the occurrences of each pattern in the database, thus facilitating the support counting phase. Checking whether a pattern is frequent, therefore, consists of counting the elements in its occurrence list. POTMiner employs scope lists to preserve the occurrences of a pattern in the tree database. The scope list for a new candidate is built by joining the scope lists of the subtrees involved in the generation of the candidate. In order to work with partially-ordered trees, the elements in the scope lists used in POTMiner include an additional parameter that is not needed when dealing with completely-ordered or completely-unordered trees.

The pseudocode of the resulting algorithm is shown in Figure 3. POTMiner is $O(t*(Ln^2)^{MaxSize}/(MaxSize-1)!)$, where t is the number of trees in the database, L the number of frequent patterns of size 1, and n the size of the trees. Therefore, our algorithm is linear with respect to the number of trees in the tree database and its execution time is also proportional to the number of considered patterns. It should also be noted, however, that POTMiner has to store the scope lists of all the candidates in memory, which can be up to $L^{MaxSize-1} * (MaxSize-2)!$ different lists in the worst case.

The experiments we have performed show that our algorithm is as efficient and scalable as existing algorithms that exclusively work on either ordered or unordered trees.

The journal article associated to this part of the dissertation is:

A. Jiménez, F. Berzal, and J.C. Cubero.
POTMiner: Mining ordered, unordered, and partially-ordered trees
Knowledge and Information System 23(2): 199-224, 2010
DOI 110.1007/s10115-009-0213-3 199-224.

3.1.3 POTMiner improvements

In order to improve the performance of the POTMiner algorithm, several issues have been taken into account. On the one hand, the design of a parallel version of our algorithm has been tackled in order to reduce its response time. On the other hand, the candidate generation process is very memory consuming due to the huge number of scope lists that have to be maintained. Moreover, the size of each scope list is proportional to the number of embedded occurrences of each pattern, which can also be huge in large databases. We have devised two variants of the POTMiner algorithm (POTMiner Light and POTMiner DP) that make different memory/CPU-time trade-offs to compute scope lists on demand, instead of keeping all of them in memory, which is impractical in many real-world situations [35].

Parallel POTMiner

Parallelism can be used to improve the performance of data mining algorithms. The parallel implementation of these algorithms lets us exploit the architecture of modern multi-core processors and multiprocessors.

In order to parallelize POTMiner, we have followed a candidate distribution approach [3]. In POTMiner, generating candidates in parallel simply involves partitioning the set of all candidate classes to be extended among the available processors.

The main idea behind the parallel version of POTMiner is that, in each step of the algorithm, the extension of each candidate class and the corresponding scope list join operations can be performed in parallel. Since these operations are independent from each other, each one can be assigned to a different processor without incurring into significant coordination costs.

The parallel version of POTMiner independently processes each class of size k to obtain candidates of size $k + 1$ and, at the end of each iteration, the results of the independently-processed classes are combined to return all the candidate classes of size $k + 1$. The pseudocode of the parallel version of POTMiner is shown in Figure 4.

```
algorithm ParallelPOTMiner
    Obtain frequent nodes (frequent patterns of size 1)
    Build candidate classes $C_1$ from the frequent nodes
    for k=2 to MaxSize
        for each class $P \in C_{k-1}$
            Extend $P$ in parallel to obtain $P_{extended}$
        for each class $P \in C_{k-1}$
            $C_k = C_k \bigcup P_{extended}$
```

Figure 4: Parallelization of the POTMiner algorithm.

POTMiner Light

POTMiner builds the scope list of a new candidate pattern by joining the scope lists of the patterns involved in its generation. POTMiner Light recursively computes such scope lists directly from the scope lists of the frequent nodes, i.e., the tree patterns of size 1. The key idea of this recursive process is that it is always possible to infer which subtrees were used to generate the tree pattern and obtain the corresponding scope lists. With respect to the efficiency of this approach, POTMiner Light is $O(t * (2Ln^2)^{MaxSize}/(MaxSize-1)!)$, i.e., we introduce a $2^{MaxSize}$ factor in the execution time of POTMiner Light in order to reduce its memory consumption. On the other hand, LightPOTMiner only has to store L lists, where L is the number of frequent patterns of size 1.

Figure 5 shows how the candidate subtree (top) is decomposed in order to obtain the trees that were used to generate it and its corresponding scope lists.

POTMiner DP

The recursive on-demand computation of scope lists in POTMiner Light repeats the generation of some of these scope lists. We can use dynamic programming in order to avoid the repeated computation of the same scope lists, which leads us to a new variant of our tree pattern mining algorithm: POTMiner DP. Our dynamic programming algorithm uses a bottom-up strategy to build scope lists and, by memorizing the scope lists that are recomputed by POTMiner Light, it trades CPU time for memory space (the space needed to store the scope lists that will be reused). The trees depicted in a darker color in Figure 5 are the ones whose scope lists are recomputed by POT-

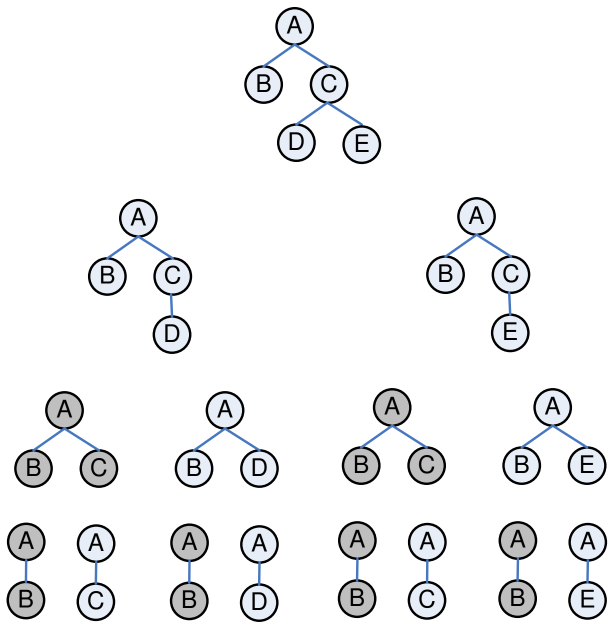


Figure 5: Recursive scope list generation in POTMiner Light.

Miner Light and they are the trees whose scope lists will be kept in memory by POTMiner DP.

POTMiner DP is $O(t * MaxSize^2 * (Ln^2)^{Maxsize}/(Maxsize-1)!)$. In POTMiner DP, we introduce a $(MaxSize^2)$ factor in its execution time with respect to our original POTMiner algorithm. It should be noted that this factor is much lower than the factor introduced by POTMiner Light, $2^{MaxSize}$, especially for large values of $MaxSize$. On the other hand, POTMiner DP needs to store L scope lists corresponding to the frequent patterns of size 1, as POTMiner Light, plus $k-1$ ancillary lists to speed up the scope list computation using a dynamic programming algorithm.

Figure 6 shows the order in which operations are performed by POTMiner DP in order to build the scope list of a pattern of size 5 (the same pattern that was used in Figure 5 to illustrate the redundant computations in POTMiner Light). During a first step, the subtrees involved in the scope list computation are identified from right to left and top-down following the hollow arrows. In a second step, scope lists are generated from left to right and bottom-up following the solid arrows.

Table 3.1.3 summarizes the efficiency of the three POTMiner variants we have devised. POTMiner is the fastest version, although it is also the one that requires more memory. On the other hand, POTMiner Light is the one with less memory consumption, although it is also the slowest. POTMiner DP provides the most attractive trade-off in terms of time and space.

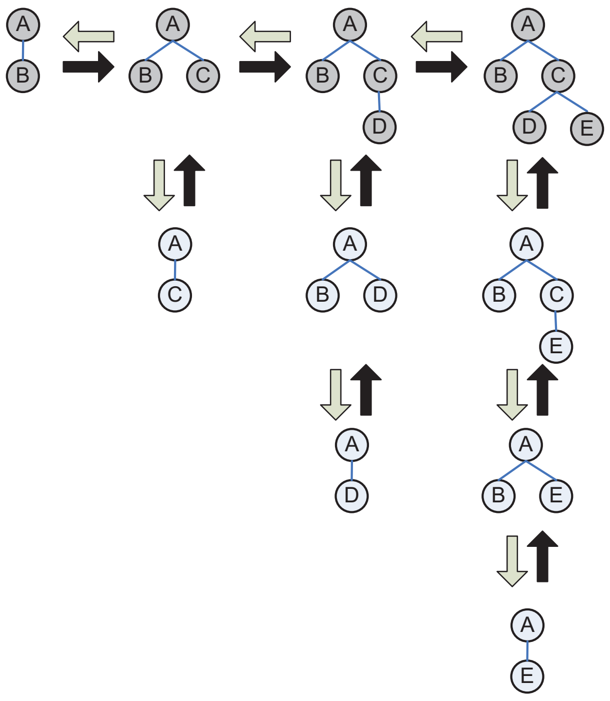


Figure 6: POTMiner DP computation of scope lists.

Algorithm	CPU Time	Memory consumption (number of scope lists)
POTMiner	$O\left(\frac{t * (Ln^2)^{MaxSize}}{(MaxSize-1)!}\right)$	$O(L^{MaxSize-1} * (MaxSize-2)!)$
POTMiner Light	$O\left(\frac{t * (2Ln^2)^{MaxSize}}{(MaxSize-1)!}\right)$	O(L)
POTMiner DP	$O\left(\frac{t * MaxSize^2 * (Ln^2)^{Maxsize}}{(Maxsize-1)!}\right)$	O($L+k-1$)

The journal article associated to this part of the dissertation is:

A. Jiménez, F. Berzal, and J.C. Cubero.
Mining frequent patterns from XML data: Efficient algorithms and design trade-offs
Submitted to Expert Systems with Applications

3.2 Applications

In this section, we present three applications that have been addressed during the preparation of this dissertation: mining multirelational databases, the identification of group association rules, and the identification of transposed motifs in music.

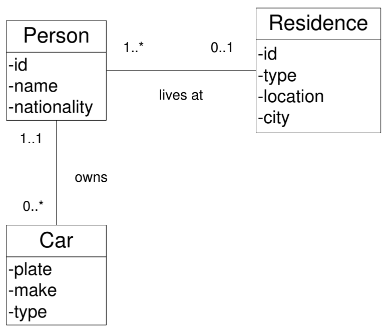
3.2.1 Multirelational data mining

Many data mining techniques, ranging from clustering and classification to association rule mining, have been developed to extract useful information from databases. However, these techniques usually require all the information to be in a single relation.

In multirelational database mining problems, there are several relations involved. A typical solution to these problems consists of joining all the relations of our interest into a single relation, usually called *universal relation* [24] [41][42]. Then, classical data mining techniques can be applied to this universal relation. Other techniques have been specially devised to deal with classification [48] [63] or clustering [64] problems.

Our proposal consist of identifying frequent patterns in multirelational databases. In this dissertation, we introduce two different schemes for representing multirelational databases as sets of trees. Once we have transformed a multirelational database into a set of trees, we can apply existing tree mining techniques to identify frequent patterns in the trees representing the multirelational database [36].

The analysis of multirelational databases typically starts from a particular relation. This relation, which we will call target relation (or target table), is selected by the end user according to her specific goals. The main idea behind our two representation schemes is building a tree from each tuple in the target relation and following the links between relations (i.e. the foreign keys) to collect all the information related to each tuple in the target relation. Our tree-based representation schemes are the following:



Person

id	name	nationality	houseID
1	Peter	US	5
...	...	...	

Residence

id	type	location	city
5	apartment	suburb	London
...	...	...	

Car

plate	make	type	ownerID
1234BCD	Toyota	SUV	1
5678JKL	Chevrolet	sedan	1
...	...	...	

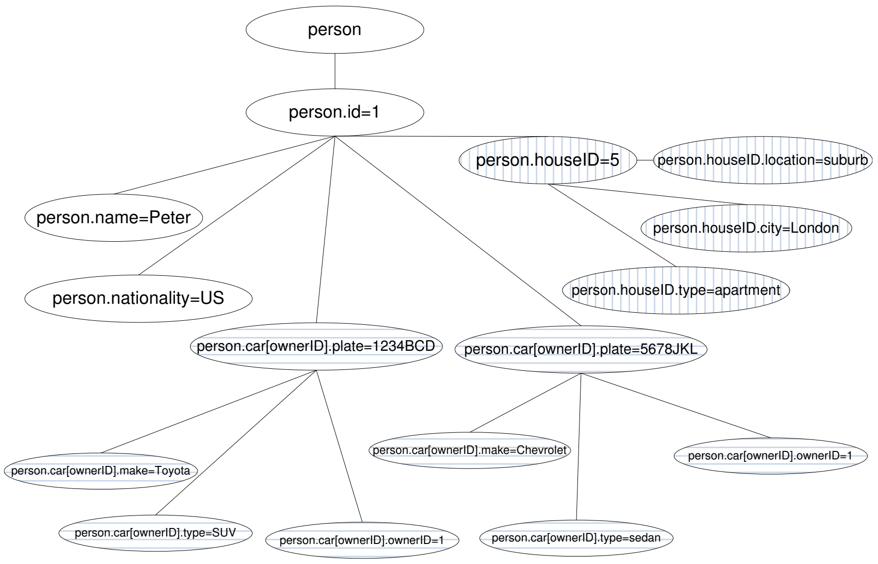
Figure 7: Multirelational database schema in UML notation (left) and its corresponding relations with some tuple examples (right).

- The key-based tree representation scheme is inspired by the concept of identity in the relational model. Relational databases rely on primary keys to ensure that each table row can be univocally referenced. In these trees, each tuple is identified by its primary key, which is placed at the root of the subtree representing the tuple. The values of each attribute in the tuple will be children nodes of the one representing its primary key.

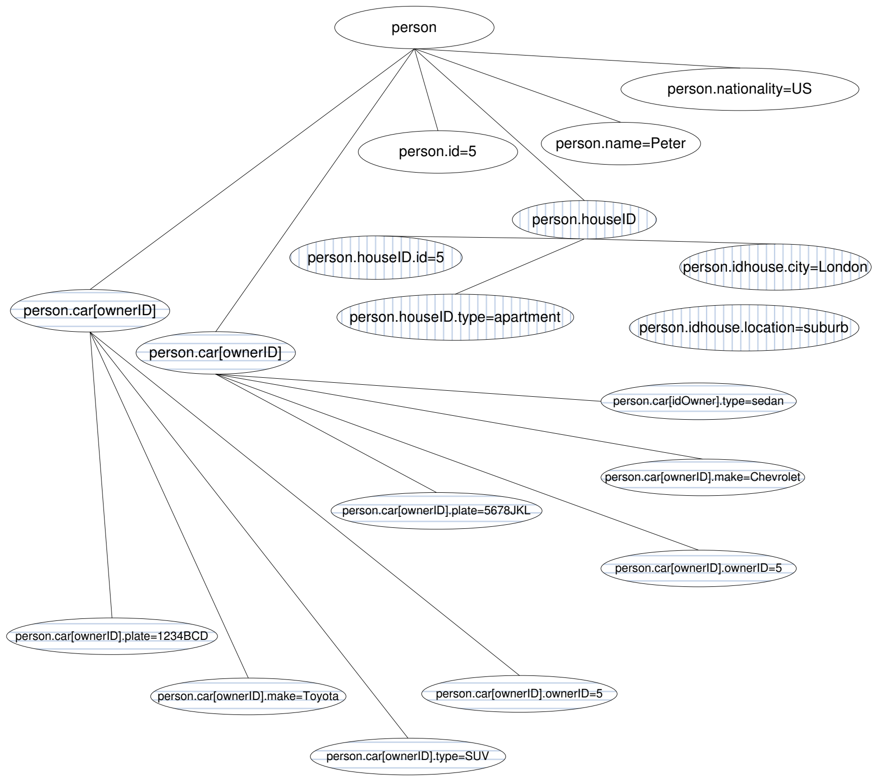
- The object-based tree representation scheme is based on the concept of identity in object-oriented models. In the object model, each object is already unique, and no specific key is needed. That means that you can create objects that have identical field values but are still different objects. In this representation scheme, we will use intermediate nodes as roots of the subtrees representing each tuple in the multirelational database. All the attribute values within the tuple, including the primary key attribute values, will be children of the root node representing the tuple in the tree.

Figure 7 (left) depicts the conceptual schema of a multirelational database using the UML notation [12]. Given such a conceptual schema, we can derive the suitable logical data model for a relational database [27] [54]. The relations obtained from the conceptual model are shown in Figure 7 (right) with some example tuples. Figure 8 shows the key-based (top) and the object-based (bottom) tree representation of the multirelational database instance shown in Figure 7.

We have identified induced and embedded subtrees using both representation schemes. Therefore, we have obtained four different kinds of patterns: induced key-based (IK), induced object-based (IO), embedded key-based (EK) and embedded object-based (EO) patterns. We have studied the formal relationships among these four set of patterns and we have proved that:


person
person.id=1
person.houseID=5
person.houseID.location=suburb
person.name=Peter
person.houseID.city=London
person.nationality=US
person.houseID.type=apartment
person.car[ownerID].plate=1234BCD
person.car[ownerID].plate=5678JKL
person.car[ownerID].make=Chevrolet
person.car[ownerID].ownerID=1
person.car[ownerID].make=Toyota
person.car[ownerID].type=SUV
person.car[ownerID].ownerID=1
person.car[ownerID].type=sedan

a) Key-based tree representation.


person
person.nationality=US
person.id=5
person.name=Peter
person.houseID
person.houseID.id=5
person.idhouse.city=London
person.car[ownerID]
person.houseID.type=apartment
person.idhouse.location=suburb
person.car[ownerID]
person.car[idOwner].type=sedan
person.car[ownerID].make=Chevrolet
person.car[ownerID].plate=5678JKL
person.car[ownerID].ownerID=5
person.car[ownerID].plate=1234BCD
person.car[ownerID].make=Toyota
person.car[ownerID].ownerID=5
person.car[ownerID].type=SUV

b) Object-based tree representation.

Figure 8: Tree representation schemes for the multirelational database shown in Figure 7.

$IK \subseteq EK \subset_{eq} IO \subseteq EO$

Induced and embedded patterns provide us different information about the multirelational database. Induced patterns preserve the structure of the original trees in the database. However, in order to obtain useful information from the multirelational database, the identification of large induced patterns is often necessary. Embedded patterns are typically smaller than the induced patterns required to represent the same information. However, if we use embedded patterns, some of the relationships among the nodes in the original trees will not be preserved.

The key-based and the object-based tree representation schemes also provide us different information about the multirelational database. When we use the key-based representation scheme, no induced patterns with information about the target table can be identified. Therefore, the object-based representation scheme is our only choice if we are interested in induced patterns, which preserve the original structure of the trees in the database.

Using the object-based representation scheme we can represent patterns that show that a particular object in a given relation is related to at least n objects in another relation. An example of this kind of pattern is shown in Figure 9. That pattern indicates that people living in suburbs with at least two cars are frequent in our database, without any references to particular car features. This kind of pattern cannot be identified with traditional data mining techniques and it cannot be identified using the key-based representation scheme.

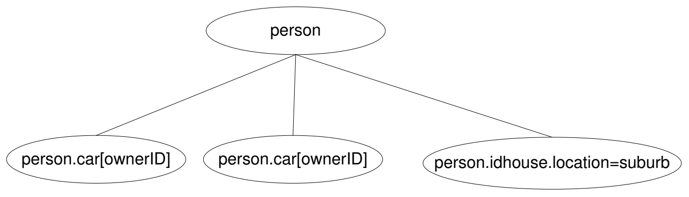


Figure 9: Embedded pattern from the object-based tree representation of the multirelational database in Figure 7, as shown in Figure 8 b).

In this dissertation, we also describe how tree patterns can be employed to discover association rules from multirelational databases. We also study the use of constraints, which are useful for reducing the number of rules that are returned to the user, and help us improve the performance of the rule mining process.

The journal article associated to this part of the dissertation is:

A. Jiménez, F. Berzal, and J.C. Cubero.
Using trees to mine multirelational
Submitted to Data Mining and Knowledge Discovery

3.2.2 Group association rules

The approach we propose in this section intends to solve the second-order data mining problem that often arises in practice. Researchers in the data mining field have traditionally focused their efforts on obtaining fast and scalable algorithms in order to deal with huge amounts of data. When dealing with association rules, for instance, the overwhelming number of discovered rules, usually in the order of thousands or even millions, makes them of limited use in practice. The mere volume of these sets of rules causes the aforementioned second-order data mining problem [9].

Databases can naturally contain groups of individuals that share some characteristics [50]. For example, within a census database, we can find many different groups of individuals (e.g. according to their sex, their marital status, whether they have children, or just by combining several of such features). Our proposal consists of automatically identifying potentially useful groups of related association rules and ranking the resulting group association rules so that expert users can easily sift through vast amounts of association rules [34].

Let $I = \{I_1, I_2, ..., I_m\}$ be a set of items. Let D be a set of database transactions where each transaction T is a set of items such that $T \subseteq I$. Let S be a set of items. A transaction T is said to contain S if and only if $S \subseteq T$ [30]. An **association rule** is an implication of the form $A \Rightarrow C$, where $A \subseteq I$, $C \subseteq \mathrm{I}$, and $A \cap C = \emptyset$.

We define a **group** as a set of items $G = \{G_1, G_2, ..., G_n\}$ such that $G \subseteq I$. A **group association rule** $G : A \Rightarrow C$ is an association rule $A \Rightarrow C$ defined over the group G. In other words, a group association rule $G : A \Rightarrow C$ is equivalent to the classical association rule $GA \Rightarrow C$.

In this dissertation, we explain how to adapt the classical measures defined to evaluate the interestingness of association rules [53] [28] [8] to group association rules and we also propose some novel interestingness measures. We have studied some of the formal properties of these new measures as well as how these new measures can be useful for evaluating the interestingness of group association rules:

- **Group gain** represents the difference between the confidence in the presence of the consequent when we know that the antecedent appears in the group, and the support of the consequent within the group.

- **Group gain factor** highlights the most frequent subgroups within a group.
- **Variation** highlights anomalies because it overweighs those subgroups that have a low support in the group.
- **Impact** represents the number of individuals within a group that are directly affected by a given rule.
- **Impact ratio** represents the ratio of the impact of the rule within the group with respect to the size of the group.

We have used group association rules to describe groups in a database and to provide a ranking of the most interesting rules. Given that groups can be described by many different association rules, if we intend to rank groups, we must somehow compute an aggregate value that represents the overall interestingness of the group. Impact seems to be a good candidate for estimating the potential interestingness of the group, since it takes into account the number of individuals that are affected by each rule within the group. We have then defined a weighted impact measure for the rules within a group using different interestingness measures as weights. We have also defined a weighted impact ratio measure in order to obtain a more balanced ranking for groups of disparate size. Depending on their particular goals, users should choose which measure to employ in order to highlight the potentially most interesting groups.

We have performed experiments on a real-world database that have corroborated our intuitions on the behavior of different ranking criteria and demonstrated that our approach can be useful in practice for dealing with the huge amount of association rules that can be derived from real-world databases.

The journal article associated to this part of the dissertation is:

A. Jiménez, F. Berzal, and J.C. Cubero.
Interestingness measures for association rules within groups
Submitted to Knowledge and Information Systems.

3.2.3 Identifying transposed motifs in music

The third applied line of research we address in this dissertation is the identification of transposed motifs in music.

The discovery of frequent musical patterns (motifs) is a relevant problem in musicology. In music, we can find several entities that can be repeated, such as notes, intervals, rhythms, and harmonic progressions. In other words, music can be seen as a string of musical entities, such as notes or chords, on which pattern recognition techniques can be applied [33] [51] [44] [49].

We can define a music motif as the smallest meaningful melody element. As a rule, motifs are groups of notes no longer than one measure. In human speech, a word is a motif. In the same way that sentences consist of words, motifs form musical phrases. A melody is formed by several main motifs, which are repeated, developed, and opposed one against another within the melody evolution.

When analyzing a music work, musicians carry out a deep analysis of the musical material. This analysis includes motif extraction as a basic task. Musician studies include contextual information (such as the author, the aim, or the period) but also morphological data from the music itself [14]. Looking for the motifs that build the whole work is the first step that a musician takes when faced with a music sheet.

In this dissertation, we address the problem of discovering long motifs that are repeated within a single song. Our hypothesis is that those patterns probably correspond to the chorus or the more significant part of the song [39]. However, it is frequent that, within a melody, motifs are repeated not only in an exact way but also in a similar way.

In this dissertation, hence, we considered transposed motifs: those motifs that preserve the interval distances within notes although the notes are not the same than in the original motif. We also took into account the similarity between notes that have the same pitch but differ in their duration. Therefore, our goal in this application was to find frequent melodic and rhythmical patterns in music starting from the MusicXML[61] representation of a song.

First, we transformed this symbolic representation into a sequence of notes that are defined at their lowest level (pitch and duration). Then, we adapted the POTMiner algorithm to work with sequences, developing a new version, called SSMiner (Similar sequence miner), that mines sequences (i.e., degenerate trees where each node has, at most a single child). SSMiner can be used to find common motifs in several songs and also find repetitions within a song.

Figure 10 shows a piece of an example song. In this example, the motif G4 A4 G4 E4 appears three times within the sequence. The motif D5 E5 D5 B4 is transposed up one fifth with respect to G4 A4 G4 E4. Therefore, if we consider this transposition, the support of the motif G4 A4 G4 E4 in this piece is 4, even though its exact support is 3.

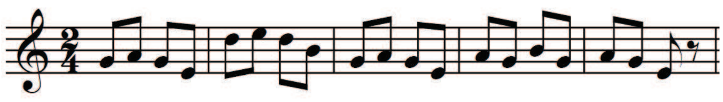


Figure 10: Sample piece of music: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4.

SSMiner, as POTMiner, is an Apriori-based algorithm. Therefore, it has two phases: candidate generation and support counting. The main differences between POTMiner and SSMiner are:

- In the candidate generation phase of SSMiner, only candidates with a sequence form can be generated, i.e., each node of the candidate has an unique child.
- In the support counting phase, it is necessary to add a new parameter to the scope lists in order to preserve the similarity between the candidate pattern and the particular occurrence (i.e. it can be an exact occurrence or a transposed one).

Regarding SSMiner computation time, SSMiner is $O(L^{MaxSize} \cdot t \cdot (S - MaxSize + 1)^2)$. Therefore, our algorithm execution time is proportional to the number of sequences in the sequence database ($t = 1$ in our motifs identification problem), and to the number of patterns than can be identified ($L^{MaxSize}$). Finally, its execution time is quadratic with respect to the size of the sequences (S).

Our experiments suggest that our approach performs well in a set of randomly-selected songs. It is remarkable that considering transposition but not rhythm results in more patterns belonging to the chorus of the songs. This fact indicates that patterns are not always exactly repeated as themselves, but slightly modified. Therefore, our approach for mining transposed motifs is useful in this particular scenario.

Audio-thumbnailing (i.e., summarizing or abstracting) is an interesting application in the musical domain that is related to motif extraction[69][6]. It provides the user with a brief excerpt of a song that (ideally) contains the main features of the work. This technique is also important for indexing large datasets of songs, which can be browsed more quickly and searched more efficiently if indexed by those small patterns instead of being indexed by the whole song.

The journal article associated to this part of the dissertation is:

A. Jiménez, M. Molina-Solana, F. Berzal and W. Fajardo.
Mining transposed motifs in music
Journal of Intelligent Information Systems 36:99-115, 2011.
DOI 110.1007/s10844-010-0122-7.

4 Concluding remarks

The results of this Ph.D. thesis can be grouped into two broad categories, those directly related to tree pattern mining techniques and those that refer to applications.

Frequent tree pattern mining

- We have completed a full survey of the state of the art in tree pattern mining by studying the different tree mining algorithms proposed in the literature. This review provides a classification of the algorithms according to the kinds of input trees they can be applied to and the kinds of subtrees they are able to identify within their input trees. We have also examined these algorithms and compared them in order to highlight their similarities and differences.
- We have devised a new algorithm, called POTMiner, that is able to work with partially-ordered trees, as well as completely-ordered and completely-unordered trees. This algorithm is as efficient and scalable as existing algorithms that exclusively work on either ordered or unordered trees.
- The POTMiner algorithm has been improved in order to reduce its execution time and its memory consumption. In order reduce its execution time, a parallel version of POTMiner has been implemented. Parallel POTMiner lets us exploit the architecture of modern multicore processors and multiprocessors. Furthermore, POTMiner Light and POTMiner DP, other two variants of POTMiner, have been devised to reduce its memory consumption.

Applications

- Multirelational data mining: Several algorithms have been developed in order to mine multirelational databases. Those that resort to join-based techniques do not preserve the proper support counts, while others are especially oriented to classification or clustering problems. Our proposal consists of transforming the multirelational database into a set of trees. This way, efficient pattern mining algorithms that exist for working with trees can be applied. Furthermore, we discover patterns that previous techniques could not identify within the multirelational databases. Several kinds of tree patterns can now be extracted from the multirelational database and we have studied the relationships among these sets of patterns. Moreover, we have extracted association rules from the multirelational database using these tree patterns.

- Group association rules: Databases can naturally contain groups of individuals that share some characteristics. The goal of our approach is to discover such groups in the database and describe them using a new kind of association rule which we have called group association rules. We have described how to adapt some of the interestingness measures that have been defined for standard association rules to group association rules and how these modified measures will help us rank the different groups in a database in order to highlight the most interesting ones.

- Identifying transposed motif in music: The study of frequent patterns, called motifs in this context, that appear in the melody of a song is useful, for example, for determining the chorus of the song. The problem of motif extraction in a piece of music can be seen as a particular case of mining frequent patterns from a sequence when using the symbolic representation of the song. Therefore, we have adapted our POTMiner algorithm to deal with sequences. The resulting algorithm, called SSMiner, is able to identify frequent motifs in songs even when they are transposed.

5 Future work

In this section, we provide some pointers to future lines of research that stem from the proposals described in this dissertation.

On the one hand, there are several extensions and modifications that can be done to POTMiner in order to work on other interesting situations:

- Following the idea of finding similar sequences in SSMiner, we plan to extend POTMiner to deal with partial or approximate tree matching [60]. This is a natural extension to POTMiner that can take advantage of the similarity parameter included in SSMiner to represent this approximate matching. The transpositions in the music problem are, in fact, a crude kind of approximate matching. This feature will be useful, for instance in many real-world problems, from entity resolution in XML documents to program element matching in software mining.

- Our algorithms have been applied to crisp databases. It would also be interesting to apply it to fuzzy databases, where the attributes can have fuzzy values [19]. The use of trees in this kind of databases can be also useful in order to extract association rules from them [21].

Our tree mining approach has already been applied in three different areas, but we also consider some improvements on this applications:

- In the case of the multirelational databases, more constraints can be applied into the tree pattern mining process as well as in the rule mining part to reduce the number of rules to be considered. The use of this constraints can be useful, for instance, when using the resulting rules to build classifications models [11].
- We have extracted group association rules from databases using a completely unsupervised approach. The use of expert knowledge can help focus on some groups that are specially interesting in the database and to reduce the number of groups and group association rules considered. This way, a semi-supervised tool for analyzing groups within a database could be devised.
- With respect to the music application, we intend to employ interval strings to represent notes rather than the absolute pitches we have used in the experiments reported in this paper. We will also consider more abstract representations of melodies, such as the one proposed by Narmour [45], where melodies are represented at a higher level than notes but low enough to capture the essence of the melody.

Finally, we also find interesting the application of our algorithm to mine partially-ordered trees in new application domains:

- Longitudinal studies are research studies where each individual is repeatedly observed through time, so that it is possible to evaluate changes in the individuals over time [25] [23]. Our proposal would consist of representing the longitudinal study as a set of trees where a partially-ordered tree is built for each individual in the study. Then, the different attributes of the study will be unordered nodes in the tree, while the different observations will preserve its relative order. We can extract patterns from these studies and evaluate if different individuals present similar or different patterns. Furthermore, groups of individuals could be identified in longitudinal studies using the group association rules described in this dissertation.

- XML [46] documents, due to their hierarchical structure, are directly amenable to tree mining techniques. Since XML documents can contain both ordered and unordered sets of nodes, partially-ordered trees provide a better representation model for them and POTMiner is, therefore, better suited for mining them than previous tree mining techniques.

- In Software Engineering, it is usually acknowledged that mining the wealth of information stored in software repositories can "support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques" [26]. For instance, there are hierarchical program representations, such as dependence higraphs [10], that can be viewed as partially-ordered trees, hence the potential of tree mining techniques in software mining.

Part II. Publications: Published, accepted, and submitted papers

"The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it." [Elbert Hubbard]

1 Frequent tree pattern mining

1.1 State of the art

The journal paper associated to this part of the dissertation is:

- A. Jiménez, F. Berzal, and JC Cubero, Frequent tree pattern mining: A survey. Intelligent Data Analysis 14 (6):603-622, 2010 DOI 10.3233/IDA-2010-0443
  - Status: **Published**.
  - Impact Factor (JCR 2009): 0.929.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 77 / 103.

Intelligent Data Analysis 14 (2010) 603–622
DOI 10.3233/IDA-2010-0443
IOS Press

Frequent tree pattern mining: A survey

Aída Jiménez*, Fernando Berzal and Juan-Carlos Cubero
Department of Computer Science and AI, University of Granada, Granada, Spain

Abstract. The use of non-linear data structures is becoming more and more common in many data mining scenarios. Trees, in particular, have drawn the attention of researchers as the simplest of non-linear data structures. Many tree mining algorithms have been proposed in the literature and this paper surveys some of the recent work that has been performed in this area. We examine some of the most relevant tree mining algorithms and compare them in order to highlight their similarities and differences.

Keywords: Data mining, frequent patterns, tree patterns

1. Introduction

Many data mining problems are best represented with the help of non-linear data structures. Graphs, for instance, are commonly used to represent data and their relationships in different problem domains, ranging from web mining [18] and XML document mining [2,20,39] to bioinformatics [5] and social networks [17].

The use of non-linear data structures in many interesting problems has spurred the interest of data mining researchers in the development of efficient and scalable data mining techniques for these special data structures. Trees, in particular, have recently attracted the attention of the research community, in part because they are particularly amenable to efficient pattern mining techniques.

Identifying frequent patterns in a database of trees is an important task in solving many tree mining problems. These frequent patterns, also known as frequent subtrees, represent common substructures in a tree database.

The aim of this paper is to survey the state of the art in tree pattern mining. Chi et al.'s survey [8] analyzes a few tree pattern mining algorithms in terms of their efficiency. Our survey provides a more comprehensive overview of the different tree pattern mining algorithms that have been proposed in the literature. We try to highlight the commonalities among different frequent tree pattern mining algorithms and reveal the peculiarities of each one.

Our paper is organized as follows. We introduce some standard terminology and notation in Section 2. Tree mining techniques are analyzed in Section 3. A survey of tree mining algorithms according to the kinds of input trees they can be applied to and the kinds of subtrees they are able to identify within their input trees can be found in Section 4. Finally, Section 5 summarizes and concludes our survey paper.

*Corresponding author. E-mail: aidajm@decsai.ugr.es.

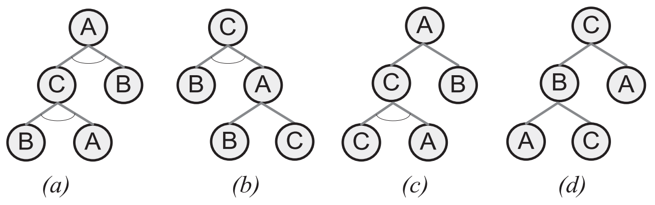


Fig. 1. Example dataset with different kinds of rooted trees (from left to right): (*a*) completely-ordered tree, (*b*) and (*c*) partially-ordered trees, (*d*) completely-unordered tree.

2. Terminology and notation

We will first review some basic concepts and provide some definitions related to labeled trees, their representation, and the kinds of tree patterns we might be interested in.

2.1. Trees

A **labeled tree** is a connected acyclic graph that consists of a vertex set V, an edge set $E \subseteq V \times V$, an alphabet Σ for vertex and edge labels, and a labeling function $L : V \cup E \rightarrow \Sigma \cup \varepsilon$, where ε stands for the empty label. Optionally, a tree can also have a predefined root, v_0, and a binary ordering relationship $\leqslant$ defined over its nodes (i.e. '$\leqslant$' $\subseteq V \times V$). The size of a tree is defined as the number of nodes it contains.

A tree is rooted if its edges are directed and a special node v_0, the root, can be identified. The root is the node from which it is possible to reach all the other vertices in the tree. In contrast, a tree is free if its edges have no direction, that is, when the tree is undirected. Therefore, a free tree has no predefined root. Rooted trees can be further classified into:

– **Ordered trees**, when there is a predefined order $\leqslant$ within every set of siblings in the trees.
– **Unordered trees**, when there is not such a predefined order among siblings.
– **Partially-ordered trees**, which are trees that have both ordered and unordered sets of siblings. This can be useful when the order within some sets of siblings is important but it is not necessary to establish an order relationship for all sets of siblings.

Figure 1 shows some rooted trees, from a completely-ordered tree (left) to a completely-unordered tree (right). In this figure, sibling nodes are joined by an arc when there is an order relationship defined between them. When such an arc is not present, siblings do not present a predefined order.

2.2. Tree representation

A canonical tree representation is an unique way of representing a labeled tree. This canonical representation makes the problems of tree comparison and subtree enumeration easier. We now proceed to describe the most common canonical tree representation schemes.

Three alternatives have been proposed in the literature [8] to represent trees as strings:

– **Depth-first codification**: The string representing the tree is built by adding the label of each tree node in a depth-first order. A special symbol $\uparrow$, which is not in the label alphabet, is used when the sequence comes back from a child to its parent.
– **Breadth-first codification**: Using this codification scheme, the string is obtained by traversing the tree in a breadth-first order, i.e., level by level. Again, we need an additional symbol \$, which is not in the label alphabet, in order to separate sibling families.

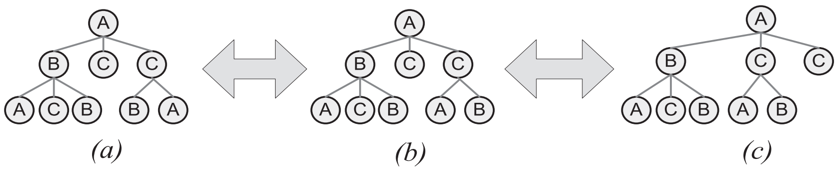


Fig. 2. Three alternative representations of the same unordered tree.

– **Depth-sequence-based codification**: This codification scheme is also based on a depth-first traversal of the tree, but it explicitly stores the depth of each node within the tree. The resulting string, known as depth sequence, is built with (d, l) pairs where the first element, d, is the depth of the node in the tree and the second one, l, is the node label.

2.2.1. *Ordered tree representation*

The three schemes explained above can be directly applied to the representation of rooted ordered trees. Since all sets of siblings in such trees are ordered, there is a single string that represent each ordered tree. This unique string is usually referred to as the canonical representation of the tree.

The depth-first codification of the (a) tree in Fig. 1 is ACB↑A↑↑B↑, the breadth-first codification for that tree is A$CB$BA, and its depth-sequence-based codification is (0,A) (1,C) (2,B) (2,A) (1,B).

2.2.2. *Unordered tree representation*

The representation of an unordered rooted tree might change depending on how we order its sets of sibling nodes, as shown in the example in Fig. 2. If we used a depth-first codification scheme, we would have to choose among the following strings:

– ABA↑C↑B↑↑C↑CB↑A↑↑.
– ABA↑C↑B↑↑C↑CA↑B↑↑.
– ABA↑C↑B↑↑CA↑B↑↑C↑.

The canonical representation for an unordered tree is defined as the minimum codification, in lexicographical order, of all the ordered trees that can be derived from it.

In the previous example, we would choose the third string as the canonical representation because it is the minimum depth-first codification in lexicographical order (the symbol ↑ is considered to be lexicographically greater that any other symbol in the tree labeling alphabet). It should be noted that, even though we have used a depth-first codification in our example, we could have used any of the other codification schemes we have described above.

2.2.3. *Partially-ordered tree representation*

Partially-ordered trees also have different representations depending on the order of their sibling nodes. As in unordered trees, their canonical representation is the minimum codification, in lexicographical order, of all the ordered trees that can be derived from their depth-first representation strings.

The possible codifications for the tree in Fig. 1 (c) are:

– ACC↑A↑↑B↑
– AB↑CC↑A↑↑

The canonical representation for this tree is the second one because it is the minimal codification.

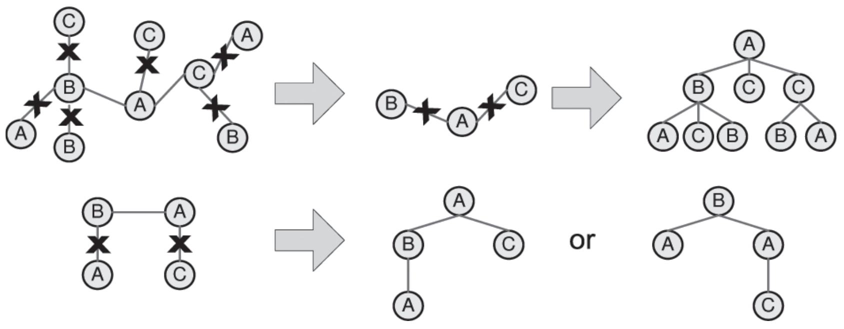


Fig. 3. Two free trees: a centered tree (top) and a bicentered tree (bottom).

2.2.4. *Free tree representation*

Free trees have no predefined root, but it is possible to select one node as the root in order to get a unique canonical representation of the tree, as described in [8,11].

The procedure consists of repeatedly removing leaf nodes (with their incident edges) from the free tree until a single vertex or two adjacent vertices remain. If there is a single remaining node, the tree is centered. If there are two remaining nodes, then the tree is said to be bicentered. Figure 3 shows examples of both kinds of trees.

When the tree is centered, the center node is uniquely identified as the root node to obtain a rooted unordered tree whose canonical representation can be obtained as we explained above.

When the tree is bicentered, two rooted unordered trees can be obtained from the free tree, one from each bicenter node. The rooted unordered tree with a smaller canonical representation is then selected as the canonical representation of the free tree.

2.3. *Tree patterns*

A subtree could be formally defined just as a subgraph of a tree. However, different kinds of subtrees can result depending on the way the relationships between the nodes within a tree are preserved:

– **Bottom-up subtrees**: A bottom-up subtree T' of T (with root v) can be obtained by taking one vertex v from T with all of its descendants and corresponding edges, and preserving the order between siblings if it exists. Formally, in a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is a bottom-up subtree of T if and only if:

  1. $V' \subseteq V$
  2. $E' \subseteq E$
  3. The labeling of V' and E' in T is preserved in T'.
  4. The order among siblings, when it exists in T, is preserved in T'.
  5. For every vertex $v \in V$, if $v \in V'$ then all descendants of $v \in T$ are also in V'.

– **Induced subtrees**: We can obtain an induced subtree T' from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T.
  We can define induced subtrees as bottom-up subtrees without the last fifth constraint above. For any vertex $v \in V$, when $v \in V'$, there is no need for all of its descendants to be also in V'. However, if v is the parent of w in T and both of them are present in T', then v must also be the parent of w in T', since $E' \subseteq E$.
  In a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an induced subtree of T if and only if:

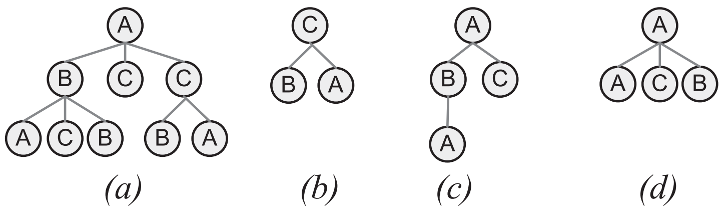


Fig. 4. Different kinds of subtrees (from left to right): (*a*) original tree, (*b*) bottom-up subtree, (*c*) induced subtree, (*d*) embedded subtree.

1. $V' \subseteq V$
2. $E' \subseteq E$
3. The labeling of V' and E' in T is preserved in T'.
4. The order among siblings, when it exists in T, is preserved in T'.

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

1. $V' \subseteq V$
2. The labeling of V' and E' in T is preserved in T'.
3. If $(v_1, v_2) \in E'$ then v_1 is an ancestor of v_2 in T.
4. For all $v_1, v_2 \in V'$: $v_1 < v_2$ in T' if and only if $v_1 < v_2$ in T.

– **Incorporated subtrees**: The ancestor-descendant relationships in the original tree do not have to hold in its incorporated subtrees [6]. Incorporated subtrees still preserve the order relationship within the sets of siblings.
In a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an incorporated subtree of T if and only if:

1. $V' \subseteq V$
2. The labeling of V' and E' in T is preserved in T'.
3. If v_1 is an ancestor of v_2 in T' then v_1 is an ancestor of v_2 in T.
4. For all $v_1, v_2 \in V'$: $v_1 < v_2$ in T' implies that $v_1 < v_2$ in T.

It should be noted that, when v_1 is an ancestor of v_2 in T, v_1 and v_2 might end up as siblings or cousins in the incorporated subtree T'.

– **Subsumed subtrees**: Subsumption, which was introduced in [30], is even laxer than incorporation. The ancestor-descendant relationships in the original tree do not have to hold in its subsumed subtrees, as happened with incorporated subtrees. In addition, subsumed trees do not preserve the order relationship among the nodes in the original tree.
In a rooted tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is a subsumed subtree of T if and only if:

1. $V' \subseteq V$
2. The labeling of V' and E' in T is preserved in T'.
3. If v_1 is an ancestor of v_2 in T' then v_1 is an ancestor of v_2 in T.

As above, when v_1 is an ancestor of v_2 in T, v_1 and v_2 might also end up as siblings or cousins in the subsumed subtree T'. It should also be noted that tree subsumption is based on Prolog subsumption.

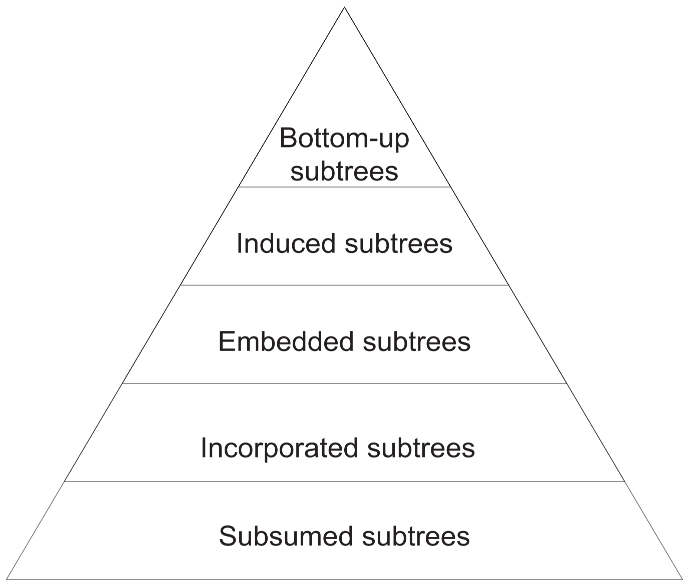


Fig. 5. Inclusion relationships among tree patterns.

This means that the mapping of pattern nodes to data nodes is not required to be injective (i.e., the same data node might appear several times in the subsumed pattern).

As shown in Fig. 5, bottom-up tree inclusion is more restrictive than induced tree inclusion. Induced tree inclusion, in turn, is more restrictive than tree embedding. Therefore, all induced subtrees are also embedded subtrees. Likewise, tree embedding is more restrictive than tree incorporation. Finally, tree incorporation is more restrictive than tree subsumption, which defines the weakest inclusion relationship between trees.

Formally, we can assert that, for all trees T, T': T' is a bottom-up subtree of $T \Rightarrow T'$ is an induced subtree of $T \Rightarrow T'$ is an embedded subtree of $T \Rightarrow T'$ is an incorporated subtree of $T \Rightarrow T'$ is a subsumed subtree of T.

Figure 4 shows some examples of the different kinds of subtrees that could be identified in the tree shown as Fig. 4 (a). If we start from that original tree, we can identify the bottom-up subtree shown in (b) by taking the rightmost node C and all its descendant. The induced subtree in (c) can be obtained by taking the root node A with all its descendants and repeatedly removing some leaves. Finally, the embedded subtree shown in (d) can also be derived from the original tree by removing some leaves and two of the nodes in the intermediate level of the tree, yet still preserving the ancestor-descendant relationship between the nodes in the original tree.

It is important to note that we can consider the (d) subtree to be present in the (a) tree in at least three different ways depending on how we match the subtree nodes to the original tree. For example, the B node in the embedded subtree can be matched to any of the leaves in (a) with the label B.

Figure 6 shows examples of incorporation and subsumption. Please note that all the subtrees shown in the second row of Fig. 6 are present in both trees of the example dataset. Since incorporated and subsumed subtrees generalize embedded subtrees, mining these kinds of subtrees typically leads to a huge number of identified patterns, which can be unmanageable in practice. Hence, they have been used only in marginal applications [6].

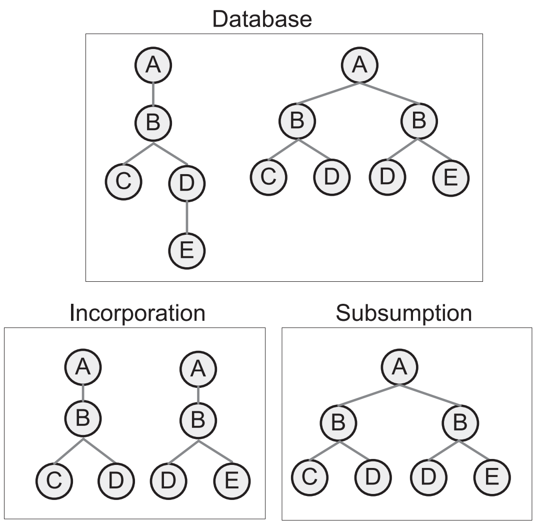


Fig. 6. Examples of incorporated and subsumed subtrees that appear in both trees of the database [7].

3. Tree pattern mining

Once we have introduced some basic terminology and notation, we can proceed to analyze the algorithms that have been proposed for tree pattern mining.

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

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

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

A frequent tree T is **maximal** (also called maximal agreement subtree [41]) if T is not a subtree of another frequent tree in D, while T is **closed** if it is not a subtree of another frequent tree with exactly the same support in D.

3.1. Pattern mining strategies

Many frequent tree pattern mining algorithms have been proposed in the literature. These algorithms are usually derived from one of the following two frequent pattern mining algorithms: Apriori [3] or FP-Growth [14]. These well-known algorithms are commonly used to identify frequent patterns in transactional databases and they have been extended to mine frequent substructures in tree databases.

Most tree pattern mining algorithms follow the Apriori iterative pattern mining strategy [3], where each iteration is broken up into two distinct phases:

– *Candidate Generation*: Potentially frequent subtree candidates are generated from the frequent patterns discovered in the previous iteration. Most Apriori-like algorithms generate candidates of size $k + 1$ by merging two trees of size k having $k - 1$ elements in common.

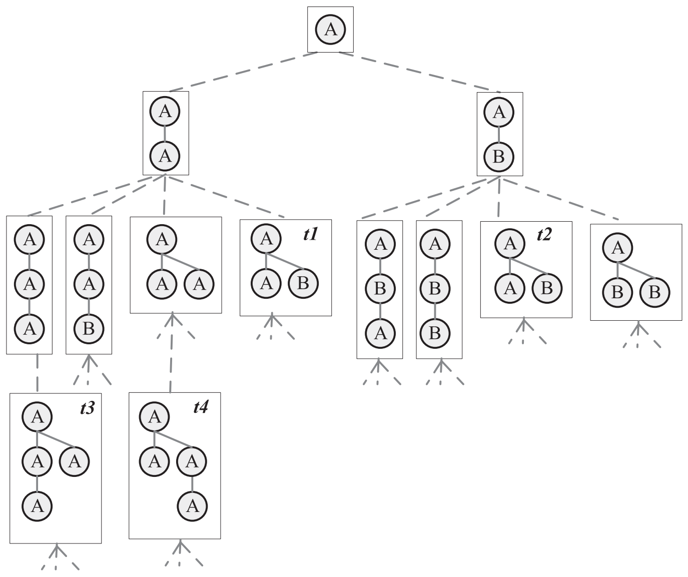


Fig. 7. Candidates generated by the rightmost expansion method when the label alphabet is {A,B}.

– *Support Counting*: Given the set of potentially frequent candidates, this phase consist of determining their actual support in D and keeping only those candidates that are actually frequent (a subset of the candidates generated in the previous phase).

Other algorithms follow the FP-Growth [14] pattern growth approach and they do not explicitly generate candidates. For instance, the PathJoin algorithm [35] uses compacted structures called FP-Trees to encode input data, while Chopper and XSpanner [34] use a sequential codification for trees and they extract frequent subtrees by discovering frequent subsequences.

Since most algorithms follow the Apriori approach, we will first review several strategies that have been proposed for candidate subtree generation and support counting. Later, we delve into the implementation details of particular tree mining algorithms.

3.2. Candidate generation

The first step in all Apriori-based pattern mining algorithms consists of generating a set of potentially frequent patterns. This set of candidate patterns is complete if it is a superset of the actual frequent pattern set. Several candidate generation strategies have been proposed for trees and all of the strategies we discuss in this paper are complete. They are also valid for all kinds of trees since they rely on the canonical tree representation schemes discussed in Section 2.2.

We describe some candidate generation strategies in the following paragraphs:

– **Rightmost expansion**
The rightmost expansion strategy generates subtrees of size $k+1$ from frequent subtrees of size k by adding nodes only to the rightmost branch of the tree.
Figure 7 illustrates candidate generation using rightmost expansion. It should be noted that, even though this candidate generation technique generates all potentially frequent patterns, it can also generate some duplicate patterns. For example, patterns $t3$ and $t4$ in Fig. 7 correspond to the same tree. Likewise, $t1$ and $t2$ also represent the same pattern.

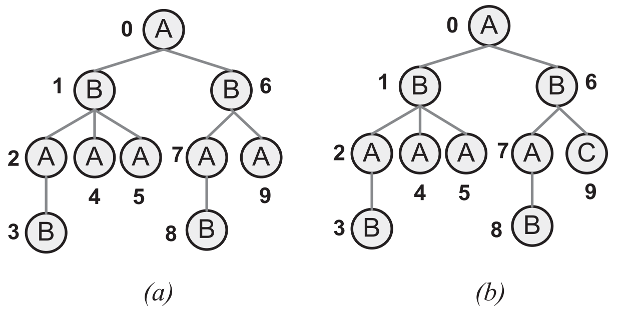


Fig. 8. Two examples of trees with (left) and without (right) a prefix node.

The rightmost expansion candidate generation technique was used by some of the earlier tree mining algorithms, such as FreqT [1]. A specialization of this technique for candidate generation, which avoids the generation of duplicate candidates, is known as Tree-Model-Guided (TMG) candidate enumeration [13,26–28].
The TMG enumeration method uses embedded lists to represent the database trees. These lists contain, for each node of the tree, its descendants in a preorder enumeration. TMG enumeration uses references to the embedded lists of the trees in which a pattern appears, called occurrence coordinates, in order to represent each occurrence of the pattern in a tree. TGM uses the embedded lists and the occurrence coordinates to extend a candidate subtree by adding one node at the time starting from the last node of its rightmost path up to its root [27].
Another variant of the rightmost expansion technique using depth-sequence-based codification scheme was proposed in [4].

– **Rightmost expansion with depth sequences**
This method was designed to identify patterns in unordered trees. It generates candidates that are always in canonical form to avoid the generation of duplicate candidates.
As we explained in the previous section, when we use a depth sequence-based-codification a tree is represented with a sequence of pairs. When a new pair (d, l) is appended to the end of a canonical depth sequence, we are in fact connecting a new node to the rightmost path of the tree. It can be shown that the prefix of a canonical depth sequence is also a canonical depth sequence [4]. Therefore, if we only allow extensions to the canonical depth sequences representing frequent patterns that also lead to canonical depth sequences, we can avoid the generation of duplicate candidates.
The depth sequence of a tree node is defined as the segment of the canonical sequence that matches with the bottom-up subtree rooted at the node. We say that a node is a prefix node when it is in the rightmost path of the tree and its depth sequence is a prefix of his left sibling's depth sequence. The prefix node in the lowest level of the tree is called the lowest prefix node (not all trees have a lowest prefix node). When the tree has a lowest prefix node, the *next prefix node* is the descendant of the lowest prefix node's left sibling that appears in the $(i + 1)$-th position of the left sibling's depth sequence, being i the length of the depth sequence of the lowest prefix node.
As shown in [8] based on the demonstration by [4], a pair (d, l) can be concatenated to the end of a canonical depth sequence S if and only if:

∗ $d \leqslant d'$ or $(d{=}d'$ and $l \geqslant l')$ where (d', l') corresponds to the next prefix node when the tree has a lowest prefix node, and
∗ $l \geqslant l'$ if the rightmost path has a node at depth d with label l'.

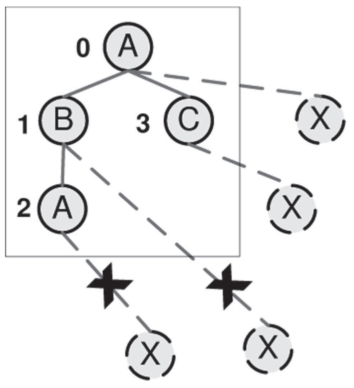


Fig. 9. Valid and invalid equivalence class-based extensions.

Figure 8 (a) shows a tree with two prefix nodes. The node number 9 is a prefix node because the bottom-up subtree rooted at this node has the codification (2,A), which is a prefix of its left sibling's codification (node 7), i.e., (2,A) (3,B). The second prefix node in this tree is node 6, whose bottom-up subtree is (1,B) (2,A) (3,B) (2,A), which is a prefix of the bottom-up subtree rooted at its left sibling (node 1): (1,B) (2,A) (3,B) (2,A) (2,A). Node 6 is the lowest prefix node because it is at a lower level than node 9 in the tree, while node 5 is the next prefix node. Unlike the tree in Fig. 8 (a), the tree shown in Fig. 8 (b) does not have a lowest prefix node.

The rightmost expansion candidate generation technique based on depth sequences has been used by Unot [4], uFreqt [21], Gaston [22], and TRIPS [29].

– **Equivalence class-based extension**

The equivalence class-based extension technique is based on the depth-first canonical representation of trees and it was proposed by Zaki in [40].

Two trees of size k are in the same equivalence class if they share a $(k-1)$-prefix in their depth-first canonical codification. Zaki's candidate generation strategy generates a candidate $(k+1)$-subtree by joining two frequent k-subtrees with $(k-1)$ nodes in common provided that they are in the same equivalence class.

Let us consider the equivalence class whose prefix is ABA ↑↑ C. Figure 9 shows which extensions lead to elements belonging to this class and which ones do not. Adding a child node to node 0 (ABA ↑↑ C ↑ X) or node 3 (ABA ↑↑ C X) will lead to trees that belong to this equivalence class, while adding a child to node 1 (ABA ↑ X ↑↑ C) or node 2 (ABA X ↑↑↑ C) would lead to trees that do not belong to this class because they would not share the class prefix.

This method avoids the generation of duplicates candidates. Zaki used this candidate generation technique in his TreeMiner [38] and SLEUTH [37] algorithms. Its derivatives, such as POTMiner [16], RETRO [7], or Phylominer [41], also employ this class-based extension method.

– **Right-and-left tree join**

The right-and-left tree join method was proposed in conjunction with the AMIOT algorithm [15]. In its candidate generation phase, this algorithm takes the rightmost and leftmost leaves of two trees of size k sharing the rest of their nodes in order to generate a candidate of size $k+1$. Even though the naive implementation of this candidate generation strategy might easily lead to duplicate patterns, AMIOT interleaves its operations so that no duplicates are generated.

Figure 10 shows the right-and-left tree join of a left tree L and a right tree R that only differ in one node: the leftmost leaf in L and the rightmost leaf in R. This join operation leads to a new tree that has the all nodes L and R share plus the two leaves that they do not have in common.

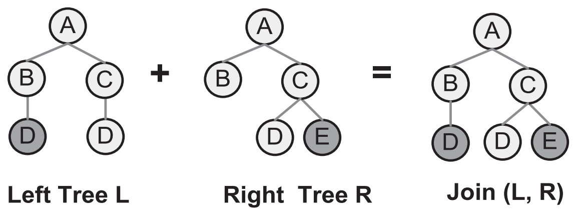


Fig. 10. Right and left tree join.

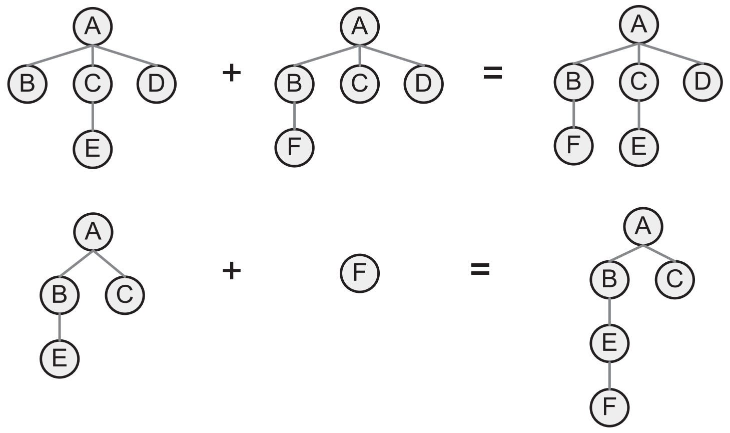


Fig. 11. Extension and join candidate generation [11]: Union of two sibling trees (top) and extension of the ABE ↑↑ C tree with a new node F (bottom).

– **Extension and join**
  The extension and join technique is based on the breadth-first codification of trees and it defines two operations to generate candidates: an extension mechanism to increase the tree depth and a join operation that expands the tree 'width' by joining two trees of size k that share $k-1$ nodes to generate a new candidate of size $k+1$.
  Figure 11 shows an example of the two operations required by the extension and join candidate generation procedure. It should be noted that the extension mechanism is needed because the join operation always produces trees that have the same height as their parents.
  This candidate generation method is used by HybridTreeMiner [11].

3.3. Support counting

Once we have generated the candidates it is necessary to check if they are actually frequent.

Testing whether a given pattern is a subtree of a tree in the database is one of the most time-consuming operations in tree pattern mining algorithms. Therefore, it comes as no surprise that several techniques have been devised to make this test as efficient as possible.

Many of the different proposals that can be found in the literature just use **occurrence lists** to represent the trees, following the approach already proposed by AprioriTID in 1994 [3]. These ancillary lists represent all the occurrences of each pattern X in the database (i.e. the tree database in vertical format) and they are designed so that you it is not necessary to perform tests on trees once these lists have been built. This occurrences are usually represented by the mapping between the pattern and the tree, i.e., the positions of the nodes in the database tree that match those of the pattern.

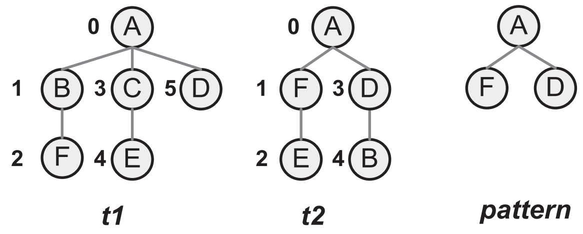


Fig. 12. Two trees ($t1$ and $t2$) and a pattern that is embedded in both trees.

Figure 12 shows two trees and a pattern that is a embedded subtree of both trees. The mappings of the pattern to the trees $t1$ and $t2$ are (0,2,5) and (0,1,3), respectively.

Some of the different kinds of occurrence lists that have been proposed in the literature are the following:

– **Standard occurrence lists** [11] preserve the identifiers of the trees as well as the matching between the pattern nodes and the database tree nodes using a breadth-first codification scheme. Each element of the occurrence list for pattern X has the form $(tid, i_1 \dots i_k)$, where tid is the tree identifier and $i_1 \dots i_k$ represent the mapping between the nodes in the pattern X and those in the database tree.
– **Rightmost occurrence lists** [1] only need to store the rightmost occurrence of X in each database tree, since rightmost expansion techniques do not have to take into account other occurrences of the pattern in the tree database (let us remember that these candidate generation strategies only had to append new nodes to the rightmost branch of a tree). Rightmost occurrence lists, also known as RMO-lists, are stored with each candidate and contain (tid, n) pairs, where n is the position of each tree node in the database tree tid that matches with the rightmost leaf of the candidate pattern. These lists are only useful for mining induced subtrees since they only preserve one node for each occurrence representing the whole subtree. Update operations can be performed on these lists as in AprioriTID [3] so that the RMO-list of a new k-pattern can be obtained from the RMO-lists of the $k - 1$ patterns it is generated from.
– **Scope-lists** [38]: The previous occurrence lists can be used regardless of the candidate generation method used. However, scope-list are typically used with the equivalence class-based candidate generation technique because they are specially designed for it. These lists are composed of triplets (tid, m, s), where tid is the tree identifier, m is the mapping of the $k - 1$ prefix of the pattern X to the database tree, and s is the scope of the last node in X. The scope of a node n delimits the subtree rooted at n. Formally, the scope of a node is a pair $[a, b]$ where a is the position of the node in the depth-first enumeration of the tree and b is the depth-first position of its rightmost descendant.
– **Vertical occurrence lists** [13,28] group the occurrence coordinates of each subtree, as employed by the Tree Model Guide candidate generation method [27], which is a specialization of the rightmost extension technique we described in Section 3.2.

Occurrence lists are usually sufficient for the efficient implementation of the support counting phase of Apriori-based tree pattern mining algorithms. However, other algorithms have resorted to more complex data structures. uFreqt [21] uses bipartite graphs to deal with the isomorphisms that might appear when working with unordered trees. Algorithms that follow the FP-Growth approach and those algorithms that have been devised for mining closed and maximal tree patterns also use special data structures, such as the FST-Forest structures employed by PathJoin [35].

This section has surveyed the main strategies used by the existing tree pattern mining algorithms. Tree pattern mining algorithms usually derive from Apriori [3] or FP-Growth [14]. Apriori-based algorithms

Table 1
Frequent tree mining algorithms

	Input trees				Identified patterns			
Algorithm	Ordered trees	Partially -ordered	Unordered trees	Free trees	Induced subtrees	Embedded subtrees	Incorporated /Subsumed	Maximal /Closed
FreqT [1]	●				●			
AMIOT [15]	●				●			
uFreqT [21]			●		●			
HybridTreeMiner [11]			●	●	●			
Unot [4]			●		●			
FreeTreeMiner [10]			●	●	●			
FreeTreeMiner’ [25]			●	●	●			
GASTON [22]			●	●	●			
X3Miner [26]	●					●		
MB3Miner [27]	●					●		
IMB3Miner [28]	●					●		
TreeMiner [38]	●					●		
TreeMinerD [38]	●					●		
RETRO [7]	●				●	●	●	
Chopper [34]	●					●		
XSpanner [34]	●					●		
Uni3 [13]			●			●		
Phylominer [41]			●			●		
SLEUTH [37]			●			●		
POTMiner [16]	●	●	●		●	●		
TRIPS [29]	●		●		●	●		
TIDES [29]	●		●		●	●		
CMTreeMiner [9]	●		●		●			●
PathJoin [35]			●		●			●
DRYADE [31]			●			●		●
TreeFinder [30]			●				●	●

proceed in two phases: *candidate generation* and *support counting*. Several strategies have been proposed for the candidate generation phase: the *rightmost extension* strategy ensures that all the candidates are generated but it can also generate some duplicates, while the *equivalence class-based extension*, the *right-and-left tree join*, and the *extension and join* methods avoid the generation of duplicate patterns. *Occurrence lists* are often employed during the support counting phase to improve the performance of tree pattern mining algorithms. In particular, *rightmost occurrence lists* can be used for counting the occurrences of induced tree patterns, while *scope-lists* are usually employed in conjunction with the *equivalence class-based candidate generation method* to count the support of both induced and embedded tree patterns.

4. Tree mining algorithms

In the previous section, we have described the main techniques behind most tree pattern mining algorithms. We now proceed to survey some of the particular frequent tree mining algorithms that have been proposed in the literature.

Table 1 provides a bird’s-eye view of frequent tree mining algorithms. This table classifies pattern mining algorithms according to the kinds of input trees they can be applied to (ordered, partially-ordered, unordered, or free) and the kinds of subtrees they are able to identify within their input trees (induced, embedded, incorporated, or subsumed). This table also points out which of those algorithms have been specially devised for discovering closed and/or maximal tree patterns.

In the following paragraphs we will provide a brief overview of the algorithms collected in Table 1. For that, we will group these algorithms with respect to the kind of patterns they identify.

4.1. Identifying bottom-up subtrees

There are not many tree mining algorithms specially designed to identify bottom-up subtrees because they can be represented as strings and sequence indexation techniques can be applied to mine them [24, 33]. Luccio et al. [19], for instance, create a sorted array with the canonical representation of each database tree and use binary search to determine if a pattern is a bottom-up subtree of the tree.

4.2. Identifying induced subtrees

In this section, we focus our attention on the algorithms that identify induced subtrees and classify them according to the kinds of trees they can work with.

4.2.1. Induced subtrees in ordered trees

FreqT [1] uses the rightmost expansion strategy to generate candidate trees. In the support counting phase, it resorts to rightmost occurrence lists (RMO-lists).

AMIOT [15], which stands for *Apriori-based Mining of Induced Ordered Trees*, is based on the right-and-left tree join candidate generation strategy. As in FreqT, the support counting phase is performed with the help of RMO-lists.

AMIOT uses ancillary lists (J_L and J_R lists) for each candidate to avoid the generation of duplicate candidates. This strategy has been shown to be more efficient than the rightmost expansion used by FreqT.

4.2.2. Induced subtrees in unordered trees

The uFreqT [21] algorithm uses depth sequences to represent unordered trees in canonical form and it performs rightmost expansions to generate candidates.

During the support counting phase for unordered trees, the main problem is to determine all the possible combinations of the children of a node v in the pattern that can match with the children of a node w in a database tree. These potential mappings can be efficiently represented by a bipartite graph $G(v, w)$, for which a bipartite matching can be computed that maps each child of v to a different child of w.

uFreqT uses an ancillary data structure that stores exactly those mappings that are included in some solvable bipartite graph $G(v, w)$ for each node v in the rightmost path along with pointers to the database tree nodes in order to facilitate the support counting phase for unordered patterns.

HybridTreeMiner [11] uses the breadth-first canonical codification of trees and the extension and join method for candidate generation. Occurrence lists are used for speeding up the support counting phase and the experiments performed show that HybridTreeMiner is faster than uFreqT.

The Unot [4] algorithm uses depth sequences to canonically represent unordered trees. Rightmost expansion with depth sequences is used to generate frequent candidates. In order to speed up the support counting phase, Unot uses occurrence lists that are similar to HybridTreeMiner's occurrence lists [11].

4.2.3. Induced subtrees in free trees

HybridTreeMiner can also be adapted to work with free trees by using a canonical representation of such trees and modifying the standard extension-and-join candidate generation method.

HybridTreeMiner's join procedure does not need to be modified for free trees. However, as we explained in section 2.2.4, free trees can be centered or bicentered. The extension mechanism must be applied only to centered trees in order to generate all the needed candidate patterns, since bicentered trees do not have to be extended.

FreeTreeMiner [10] is an earlier Apriori-based frequent pattern mining algorithm for free trees. Candidates of size $k+1$ are generated by combining pairs of frequent trees of size k sharing $k-1$ vertices, using the standard Apriori algorithm. Indexing techniques based on B+ trees and hash tables are used to speed up the support counting phase on trees. Experiments in [11] also show that HybridTreeMiner is more efficient than FreeTreeMiner.

Another algorithm, which was also called FreeTreeMiner, was proposed in [25] and is shown as FreeTreeMiner' in Table 1. This algorithm generates potentially frequent candidates by extending the leaves with maximum height in the tree (which are called extension points) and checks the support of each candidate by scanning the database (and saving the potential extensions in an extension table). While HybridTreeMiner is a descendant of Apriori, this algorithm is closer to gSpan [36], a graph mining algorithm.

Finally, an algorithm to extracts induced pattern in free trees using paths of maximal length is proposed in the same paper that introduces GASTON [22], an efficient graph mining algorithm.

4.3. Identifying embedded subtrees

Once we have described the algorithms that identify induced subtrees, we introduce those algorithms that help us to identify embedded subtrees.

4.3.1. Embedded subtrees in ordered trees

Zaki's TreeMiner [38] uses the depth-first codification of trees together with the class-based extension method to generate candidates. Zaki describes how you can prune the candidate generation phase in order to avoid the generation of duplicate candidates using his method, as we discussed in section 3.2.

Scope-lists are used by TreeMiner to determine the support of each pattern. These scope-lists can be built by joining the scope-lists of the subtrees involved in the candidate generation, thus eliminating the need to access to the tree database once the initial scope-list have been built.

TreeMinerD [38] is a variant of TreeMiner that is more efficient than TreeMiner in case we are interested in obtaining the patterns support instead of their weighted support (see Section 2).

Chopper [34] is an FP-Growth-based algorithm to identify embedded subtrees in ordered trees that employs a depth-sequence-based codification. This algorithm has two main phases: first, it applies an algorithm to mine frequent sequences and then, using these sequences, it determines which sequences correspond to frequents subtrees in the original database.

XSpanner [34] is a variant of Chopper that integrates Chopper's two phases in order to improve its efficiency.

Chopper and XSpanner are reported to be more efficient than TreeMiner. XSpanner and Chopper can save time and space cost by avoiding false candidate generation, while the performance of XSpanner is more stable than that of Chopper [34].

Finally, X3Miner [26], MB3-Miner [27] and IMB3-Miner [28] are algorithms that generate candidates by using the Tree Model Guided (TMG) enumeration, a specialization of the rightmost path extension method that avoids the generation of duplicate candidates. Vertical occurrence lists are used to aid in the support counting phase. MB3 variants ensure that only valid candidates are generated, while the join approach used in TreeMiner can generate many false subtrees, which degrades its performance [28].

4.3.2. Embedded subtrees in unordered trees

SLEUTH [37] is Zaki's proposal for mining unordered trees. As in TreeMiner, Zaki resorts to the depth-first canonical codification for representing unordered trees and scope-lists for the support counting phase. In order to generate all the candidates, however, it is sometimes necessary to use non-canonical trees to build a canonical one. Hence, Zaki proposes two alternative techniques to generate candidates:

– Using the *class-based extension mechanism*, but checking if each subtree is canonical before extending it with the elements of its equivalence class.
– Using the *canonical extension* which extends trees with the elements that belong to the same class and also with those belonging to the subtree classes of size 2 that share the node that is going to be extended, but only if the result is in canonical form.

Canonical extension generates non-redundant candidates, but many of them may not be frequent. On the other hand, class-based extension generates redundant candidates, but considers a smaller number of potential frequent candidate extensions. The experiments in [37] demonstrate that the class-based extension method is more efficient that the canonical extension.

Phylominer [41] is a special-purpose algorithm devised to identify embedded subtrees in Phylogenetic trees, which can be defined as rooted leaf-labeled unordered trees where all internal nodes have no labels, being the fan-out of each internal node (i.e. its number of children) at least 2.

The Uni3 [13] algorithm is yet another variant of IMB3Miner [28] to identify patterns in unordered trees using the TMG candidate generation technique.

4.3.3. Embedded subtrees in both ordered and unordered trees

TRIPS and TIDES [29] are two algorithms designed to identify embedded subtrees in ordered or unordered trees. They use embedded lists to generate candidates and an ancillary hash table, dubbed *support structure*, to aid in the support counting phase. The difference between these two algorithms is their canonical representation: TRIPS uses a post-order traversal representation (numbered Prüfer sequences [12]) while TIDES employs depth sequences.

POTMiner [16] is a variant of the TreeMiner/SLEUTH tandem that can be used to mine induced and embedded subtrees in ordered, unordered, and partially-ordered trees. As its predecessors, POTMiner uses Zaki's class-based extension method to generate candidates and scope-lists during the support counting phase.

4.4. Identifying incorporated and subsumed subtrees

RETRO [7] is yet another variant of TreeMiner. In this case, this algorithm can be used to identify incorporated and subsumed subtrees, as well as the more usual induced and embedded tree patterns.

The TreeFinder algorithm [30] uses the Apriori-based candidate generation technique based on ancestor-descendent relationships in order to discover subsumed subtrees. We will return to this algorithm in the following section.

4.5. Closed and maximal subtrees

This final section deals with those algorithms that identify closed and maximal subtrees. Again, we can further classify them depending on the kind of subtrees they identify (induced or embedded).

4.5.1. Closed and maximal induced subtrees

In CMTreeMiner [9], an enumeration DAG (directed acyclic graph) is a lattice-like data structure that represents all frequent subtrees. Enumeration trees, which are sometimes used for frequent itemset mining, are just spanning trees of these enumeration DAGs.

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

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

PathJoin [35] discovers maximal frequent induced subtrees from a database of unordered labeled trees. It employs a specialized data structure, a Frequent-Subtree Forest or FST-Forest, that is used to represent frequent trees in a compact form and improve the efficiency of the support counting phase.

4.5.2. Closed and maximal embedded subtrees

As mentioned above, the TreeFinder algorithm [30] uses the plain Apriori-based candidate generation technique, albeit it applies this technique directly to ancestor-descendent relationships instead of the other candidate generation strategies we have discussed in this paper. This direct use of ancestor-descendent relationships lets TreeFinder discover subsumed subtrees (see Section 2).

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

DRYADE [31], on its hand, searches for closed patterns in tree databases (recall that closed patterns are not necessarily maximal). Discovering the closed frequent patterns of depth 1 is the basic mechanism behind DRYADE. Then, they are hooked together in order to build higher-depth closed frequent patterns in a level-wise fashion. DRYADE reformulates search operations in a propositional language. This way, DRYADE can benefit from any progress made in the field of closed itemset mining algorithms, since it just delegates on a standard closed itemset mining algorithm.

DryadeParent is a variant of the original DRYADE algorithm that outperforms CMTreeMiner by several orders of magnitude on datasets where the frequent patterns have a high branching factor [32].

In this section, we have reviewed the better-known tree mining algorithms that have been proposed in the literature. We have classified them according to the kind of patterns they can identify and also according to the kind of trees they can deal with (see Table 1). AMIOT [15] has been shown to be more efficient than FREQT [1] for induced subtrees in ordered trees, while HybridTreeMiner [11] and Unot [4] obtain better results for unordered trees. HybridTreeMiner [11] can also used for mining free trees, as well as both FreeTreeMiners [10,25] and GASTON [22]. Finally, TreeMiner [38] and SLEUTH [37] are two well-known Apriori-based algorithms for mining embedded subtrees, whereas Chopper and XSpanner [34] are FP-Growth based algorithms that have been reported to be more efficient for ordered trees.

5. Conclusions

In this paper, we have studied existing tree mining algorithms. First, we introduced some basic terminology and definitions. Later, we discussed the usual strategies frequent tree mining algorithms

Table 2
Main features of the different frequent tree mining algorithms analyzed in this survey

Algorithm	Tree representation	Candidate generation approach	Implementation details
FreqT [1]	–	Rightmost expansion	RMO occurrence lists
AMIOT [15]	–	Right and left union	RMO occurrence lists
uFreqT [21]	Depth sequences	Rightmost expansion with depth sequences	Bipartite graphs
HybridTreeMiner [11]	Breadth-first codification	Union-extension method	Occurrence lists
FreeTreeMiner [10]	Depth-first codification	Apriori itemset generation	Indexation techniques
FreeTreeMiner' [25]	Depth-first codification	Maximal-depth extension	
TreeMiner [38]	Depth-first codification	Equivalence classes	Scope lists
TreeMinerD [38]	Depth-first codification	Equivalence classes	Scope lists for non-weighted support
RETRO [7]	Relational representation	Equivalence classes	Scope lists
Chopper [34]	Depth sequences	N/A	Frequent subsequences (PrefixSpan [23])
X3Miner [26]	Depth-first codification	Rightmost expansion – TMG enumeration	Vertical occurrence lists
MB3Miner [27]	Depth-first codification	Rightmost expansion – TMG enumeration	Vertical occurrence lists
IMB3Miner [28]	Depth-first codification	Rightmost expansion – TMG enumeration	Vertical occurrence lists
XSpanner [34]	Depth sequences	N/A	Frequent subsequences (PrefixSpan [23])
SLEUTH [37]	Depth-first codification	Equivalence classes	Scope lists
Unot [4]	Depth sequences	Rightmost expansion with depth sequences	Occurrence lists
Phylominer [41]	Depth-first codification	Equivalence classes	No labels in internal nodes
Uni3 [13]	Depth-first codification	Rightmost expansion – TMG enumeration	Vertical occurrence lists
GASTON [22]	Depth sequences	Rightmost expansion with depth sequences	
TRIPS [29]	Post-order codification	Embedding lists	Support Structure (hash table)
TIDES [29]	Depth sequences	Rightmost expansion with depth sequences	Support Structure (hash table)
POTMiner [16]	Depth-first codification	Equivalence classes	Scope lists
CMTreeMiner [9]	Depth-first codification	N/A	DAG enumeration graph
TreeFinder [30]	Relational representation	Apriori itemset generation	Clustering techniques
PathJoin [35]	FST-Forest structure	N/A	
DRYADE [31]	Propositional language	–	External closed itemset miner

employ in order to create a consistent framework that lets us easily understand the details behind the implementation of particular frequent tree mining algorithms.

We have classified existing algorithms according to the kind of trees each algorithm is designed to work with (ordered, unordered, partially-ordered, and free trees) as well as the kind of patterns they are able to identify (induced, embedded, incorporated, subsumed, closed, and maximal).

Table 2 summarizes the main features of the tree mining algorithms described in this paper: their underlying tree representation mechanism, their candidate generation strategy (where applicable), and some relevant implementation details that are needed for their efficient implementation.

Table 2 shows that most tree mining algorithms follow the well-known Apriori approach [3], while only a few employ the FP-Growth [14] pattern growth strategy. Within the Apriori-based algorithms we have surveyed, most of them resort to the right-most expansion method or employ Zaki's class-based extension technique for candidate generation. Vertical, RMO, or plain occurrence lists are typically used with the former, while scope-lists are used with the latter in order to speed up the support counting phase.

Acknowledgements

Work partially supported by research projects TIN2006-07262 (Spanish Ministry of Science and Innovation), TIN2009-08296 (Spanish Ministry of Science and Innovation) and P07-TIC-03175 (Junta de Andalucía).

References

[1] A. Kenji, K. Shinji, A. Tatsuya, A. Hiroki and A. Setsuo, Efficient substructure discovery from large semi-structured data. In Proceedings of the 2nd SIAM International Conference on Data Mining, 2002.

[2] C. Charu, Aggarwal, Na Ta, J.Y. Wang, J.H. Feng and M.J. Zaki, Xproj: a framework for projected structural clustering of XML documents. In Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 12–15, 2007, pp. 46–55.

[3] R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases. In Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, pp. 487–499.

[4] T. Asai, H. Arimura, T. Uno and S. ichi Nakano, Discovering frequent substructures in large unordered trees. In Discovery Science, volume 2843 of Lecture Notes in Artificial Intelligence, Springer, 2003, pp. 47–61.

[5] C. Borgelt and M.R. Berthold, Mining molecular fragments: Finding relevant substructures of molecules. In Proceedings of the 2nd IEEE International Conference on Data Mining, 2002, pp. 51–59.

[6] B. Bringmann, Matching in frequent tree discovery. In Proceedings of the 4th IEEE International Conference on Data Mining, 1-4 November, Brighton, UK, 2004, pp. 335–338.

[7] B. Bringmann, To see the wood for the trees: Mining frequent tree patterns. In Constraint-Based Mining and Inductive Databases, European Workshop on Inductive Databases and Constraint Based Mining, Hinterzarten, Germany, March 11–13, 2004, Revised Selected Papers, volume 3848 of Lecture Notes in Computer Science, Springer, 2006, pp. 38–63.

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

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

[10] Y. Chi, Y. Yang and R.R. Muntz, Indexing and mining free trees. In Proceedings of the 3rd IEEE International Conference on Data Mining, 2003, pp. 509–512.

[11] Y. Chi, Y. Yang and R.R. Muntz, HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In The 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 11–20.

[12] J. Dieter, In Graphs, Networks and Algorithms, volume 5 of Algorithms and Computation in Mathematics. Springer, 2007.

[13] F. Hadzic, H. Tan and T.S. Dillon, Uni3 – efficient algorithm for mining unordered induced subtrees using tmg candidate generation. In Computational Intelligence and Data Mining, 2007, pp. 568–575.

[14] J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation. In Proceedings of the 6th ACM SIGKDD Interna-tional Conference on Knowledge Discovery and Data Mining, 2000, pp. 1–12.

[15] S. Hido and H. Kawano, AMIOT: induced ordered tree mining in tree-structured databases. In Proceedings of the 5th IEEE International Conference on Data Mining, 2005, pp. 170–177.

[16] A. Jimenez, F. Berzal and J. Carlos Cubero, Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. In the 17th International Symposium on Methodologies for Intelligent Systems, volume 4994 of Lecture Notes in Artificial Intelligence, Springer, 2008, pp. 111–120.

[17] J.M. Kleinberg, Challenges in mining social network data: processes, privacy, and paradoxes. In Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12–15, 2007, 2007, pp. 4–5.

[18] R. Kosala and H. Blockeel, Web mining research: A survey, *SIGKDD Explorations* **2**(1) (2000), 1–15.

[19] F. Luccio, A. Mesa Enriquez, P. Olivares Rieumont and L. Pagli, Exact rooted subtree matching in sublinear time. In ANaCC/ACM/IEEE Inter- national Congress on Computer Science, CENIDET, 2002, pp. 27–35.

[20] R. Nayak and M. Javeed Zaki, eds, Proceedings of the First International Workshop of Knowledge Discovery from XML Docu- ments,Singapore, April 9, volume 3915 of Lecture Notes in Computer Science. Springer, 2006.

[21] S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. In First International Workshop on Mining Graphs, Trees and Sequences (MGTS2003), in conjunction with ECML/PKDD'03, 2003, pp. 55–64.

[22] S. Nijssen and J.N. Kok, A quickstart in frequent structure mining can make a difference. In Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004, pp. 647–652.

[23] Jian Pei, Jiawei Han, Bezhad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proceedings of the 17th International Conference on Data Engineering, 2001, pp. 215–225.

[24] B. Phoophakdee and M.J. Zaki, Genome-scale disk-based suffix tree indexing. In Proceedings of the 27th ACM SIGMOD International Conference on Management of Data, 2007, pp. 833–844.

[25] U. Rückert and S. Kramer, Frequent free tree discovery in graph data. In Proceedings of the 2004 ACM symposium on Applied computing, 2004, pp. 564–570.

[26] H. Tan, T.S. Dillon, L. Feng, E. Chang and F. Hadzic, X3-Miner: Mining Patterns from an XML Database. In The 6th International Conference on Data Mining, Text Mining and their Business Applications, May 2005, Skiathos, Greece, 2005, pp. 287–296.

[27] H. Tan, T.S. Dillon, F. Hadzic, E. Chang and L. Feng, MB3-Miner: mining eMBedded subTREEs using Tree Model Guided candidate generation. In Proceedings of the First International Workshop on Mining Complex Data, 2005, pp. 103–110.

[28] H. Tan, T.S. Dillon, F. Hadzic, E. Chang and L. Feng, IMB3-Miner: Mining induced/embedded subtrees by constraining the level of embedding. In Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2006, pp. 450–461.

[29] Shirish Tatikonda, Srinivasan Parthasarathy, and Tahsin Kurc. TRIPS and TIDES: new algorithms for tree mining. In Proceedings of the 15th ACM International Conference on Information and Knowledge Management, 2006, pp. 455–464.

[30] A. Termier, M.-C. Rousset and M. Sebag, TreeFinder: a first step towards XML data mining. In Proceedings of the 2nd IEEE International Conference on Data Mining, 2002, pp. 450–457.

[31] Alexandre Termier, Marie-Christine Rousset, and Michele Sebag. DRYADE: a new approach for discovering closed frequent trees in heterogeneous tree databases. In Proceedings of the 4th IEEE International Conference on Data Mining, 2004, pp. 543–546.

[32] A. Termier, M.-C. 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, 2005, pp. 785–788.

[33] Y. Tian, S. Tata, R.A. Hankins and J.M. Patel, Practical methods for constructing suffix trees. In Proceedings of the 31st International Conference on Very Large Data Bases, volume 14, 2005, pp. 281–299.

[34] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang and B. Shi, Efficient patterngrowth methods for frequent tree pattern mining. In Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining, volume 3056 of Lecture Notes in Computer Science, Springer, 2004, pp. 441–451.

[35] Y. Xiao, J.-F. Yao, Z. Li and M.H. Dunham, Efficient data mining for maximal frequent subtrees. In Proceedings of the 3rd IEEE International Conference on Data Mining, 2003, pp. 379–386.

[36] X. Yan and J. Han, gSpan: Graph-based substructure pattern mining. In Proceedings of the 2nd IEEE International Conference on Data Mining, pp. 721–724, 2002.

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

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

[39] M.J. Zaki and C.C. Aggarwal, XRules: an effective structural classifier for XML data. In Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, (2003), pp. 316–325.

[40] M.J. Zaki, Efficiently mining frequent trees in a forest. In Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23–26, 2002, pp. 71–80.

[41] S. Zhang and J.T.L. Wang, Discovering frequent agreement subtrees from phylogenetic data, *IEEE Transactions on Knowledge and Data Engineering* **20**(1) (2008), 68–82.

1.2 Mining partially-ordered trees

The journal paper associated to this part of the dissertation is:

- A. Jiménez, F. Berzal, and JC Cubero, POTMiner: Mining Ordered, Unordered, and Partially-Ordered Trees. Knowledge and Information System 23(2):199-224 2010, DOI 10.1007/s10115-009-0213-3.
  - Status: **Published**.
  - Impact Factor (JCR 2009): 2.211.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 27 / 103.
    - Computer Science, Information Systems. Ranking 24 / 116.

Knowl Inf Syst (2010) 23:199–224
DOI 10.1007/s10115-009-0213-3

REGULAR PAPER

POTMiner: mining ordered, unordered, and partially-ordered trees

Aída Jiménez · Fernando Berzal · Juan-Carlos Cubero

Received: 12 September 2008 / Revised: 20 February 2009 / Accepted: 20 April 2009 /
Published online: 3 June 2009

Abstract Non-linear data structures are becoming more and more common in data mining problems. Trees, in particular, are amenable to efficient mining techniques. In this paper, we introduce a scalable and parallelizable algorithm to mine partially-ordered trees. Our algorithm, POTMiner, is able to identify both induced and embedded subtrees in such trees. As special cases, it can also handle both completely ordered and completely unordered trees.

Keywords Data mining · Frequent patterns · Partially-ordered trees · Induced and embedded subtrees

1 Introduction

Some data mining problems are best represented with non-linear data structures like trees. Trees appear in many different problem domains, ranging from the Web and XML documents to bioinformatics and computer networks.

The aim of this paper is to introduce a new algorithm, POTMiner, that is able to identify frequent patterns in partially-ordered trees, a particular kind of tree that appears in several problems domains. However, existing tree mining algorithms cannot be directly applied to this important kind of tree because they work either with completely-ordered or with completely-unordered trees.

A. Jiménez (✉) · J.-C. Cubero
Department of Computer Science and Artificial Intelligence, ETSIIT,
University of Granada, Office 37, 18071 Granada, Spain
e-mail: aidajm@decsai.ugr.es

J.-C. Cubero
e-mail: JC.Cubero@decsai.ugr.es

F. Berzal
Department of Computer Science and Artificial Intelligence, ETSIIT,
University of Granada, Office 17, 18071 Granada, Spain
e-mail: berzal@acm.org

For instance, POTMiner can be applied to XML documents, whose hierarchical structure can be viewed as a tree. Furthermore, XML schemata define the structure of XML documents and they determine which sets of nodes in the trees correspond to ordered data and which ones correspond to unordered data.

As another example, POTMiner can also be used in software mining when we represent the structure of a software program as a hierarchy. In such hierarchies, program segments can be represented as nodes within trees. Existing dependences between program segments can be made explicit as order relationships among tree nodes. The resulting trees are, therefore, partially-ordered trees.

In a more typical data mining scenario, POTMiner can be used to improve existing multi-relational techniques. We can build a tree from each tuple in a particular relation by following foreign keys in order to collect all the data that are related to the original tuple, even when they are stored in different relations.

This paper is organized as follows. We introduce the idea of partially-ordered trees as well as some standard terms in Sect. 2. Our algorithm is presented in Sect. 3. In Sects. 4 and 5 we explain the details of the two main phases of our algorithm, candidate generation and support counting, respectively. Section 6 shows, by means of a particular example, how POTMiner deals with partially-ordered trees. We discuss some implementation issues that are particularly relevant in practice and how we have dealt with them in Sect. 7, while we analyze the experimental results we have obtained in Sect. 8. Finally, we present some conclusions and provide pointers to future work in Sect. 9.

2 Background

We will first review some basic concepts related to labeled trees before we formally define the tree pattern mining problem we address in this paper.

2.1 Trees

A *tree* is a connected and acyclic graph. A tree is rooted if its edges are directed and a special node, called root, can then be identified. The root is the node from which it is possible to reach all the other nodes in the tree. In contrast, a tree is said to be free if its edges have no direction, that is, when it is an undirected graph. A free tree, therefore, has no predefined root.

Rooted trees can be classified as *ordered trees*, when there is a predefined order within each set of sibling nodes in the tree, or *unordered trees*, when there is no such a predefined order among sibling nodes in the tree.

In this paper, we consider *partially-ordered trees*, which contain both ordered and unordered sets of sibling nodes in the same tree. They can be useful when the order within some sets of siblings is important but it is not necessary to establish an order relationship within all the sets of sibling nodes.

Figure 1 shows an example with different kinds of rooted trees. In this figure, ordered sibling nodes are joined by an arc, while unordered sets of sibling nodes do not share such arc.

2.2 Tree representation

A canonical tree representation is an unique way of representing a labeled tree. This canonical representation makes the problems of tree comparison and subtree enumeration easier. We now proceed to describe the most common canonical tree representation schemes.

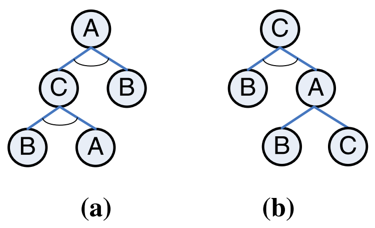
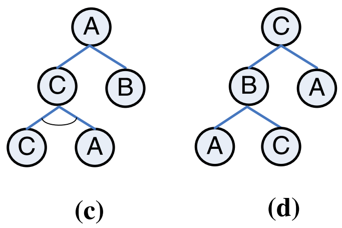


Fig. 1 Different kinds of rooted trees (from *left* to *right*): **a** completely-ordered tree, **b**, **c** partially-ordered trees, **d** completely-unordered tree



Three alternatives have been proposed in the literature [9,12] to represent trees as strings:

- *Depth-first codification*: The string representing the tree is built by adding the label of each tree node in a depth-first order. A special symbol ↑, which is not in the label alphabet, is used when the sequence comes back from a child to its parent.
- *Breadth-first codification*: Using this codification scheme, the string is obtained by traversing the tree in a breadth-first order, i.e., level by level. Again, we need an additional symbol $, which is not in the label alphabet, in order to separate sibling families.
- *Depth-sequence-based codification*: This codification scheme is also based on a depth-first traversal of the tree, but it explicitly stores the depth of each node within the tree. The resulting string, known as depth sequence, is built with (d, l) pairs where the first element, d, is the depth of the node in the tree and the second one, l, is the node label.

In our algorithm, we resort to a depth-first codification to represent the trees. For instance, the depth-first codification of the ordered tree shown in Fig. 1a is $ACB{\uparrow}A{\uparrow}{\uparrow}B{\uparrow}$. The partially-ordered tree in Fig. 1b, however, could be represented as either $CB{\uparrow}AB{\uparrow}$C$\uparrow\uparrow$ or $CB{\uparrow}AC{\uparrow}$B$\uparrow\uparrow$. This example shows that, when a tree contains unordered sets of sibling nodes, different representation strings are possible. The smallest string according to the lexicographical order is chosen as the canonical representation of the tree, where ↑ is considered to be larger than all the symbols in the label alphabet. Therefore, $CB{\uparrow}AB{\uparrow}$C$\uparrow\uparrow$ is chosen as the canonical representation of the tree in Fig. 1b. Likewise, $AB{\uparrow}CC{\uparrow}A{\uparrow}{\uparrow}$ is the canonical representation of partially-ordered tree in Fig. 1c and $CA{\uparrow}BA{\uparrow}$C$\uparrow\uparrow$ is the canonical representation of the unordered tree in Fig. 1d.

2.3 Tree patterns

A subtree is a subgraph of a tree. Different kinds of subtrees can be defined depending on the way we define the matching function between the subgraph and the tree it derives from:

- A *bottom-up subtree* T' of T (with root v) can be obtained by taking one vertex v from T with all its descendants and their corresponding edges.
- An *induced subtree* T' can be obtained from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T.
- An *embedded subtree* T' can be obtained from a tree T by repeatedly removing nodes, provided that ancestor relationships among the vertices of T are not broken.

Figure 2 shows a tree (a) wherein we have identified a bottom-up subtree (b), an induced subtree (c) and an embedded subtree (d).

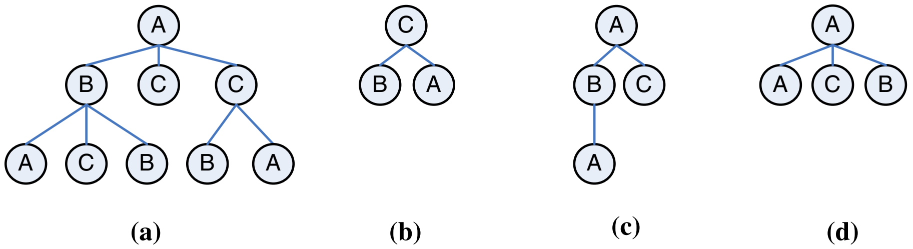


Fig. 2 Different kinds of subtrees (from *left* to *right*): **a** original tree, **b** bottom-up subtree, **c** induced subtree, **d** embedded subtree

2.4 The tree pattern mining problem

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

Let $\delta_T(\mathrm{S})$ be the occurrence count of a subtree S in a tree T and d_T a variable such that $d_T(S) = 0$ if $\delta_T(S) = 0$ and $d_T(S) = 1$ if $\delta_T(S) > 0$. We define the *support* of a subtree as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e, the number of trees in D that include at least one occurrence of the subtree S. Analogously, the *weighted support* of a subtree is defined as $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., the total number of occurrences of S within all the trees in D. We say that a subtree S is *frequent* if its support is greater than or equal to a predefined minimum support threshold. We define L_k as the set of all frequent k-subtrees (i.e., subtrees of size k).

2.5 Tree mining algorithms

Several frequent tree pattern mining algorithms have been proposed in the literature. Tables 1 and 2 summarize some of them. Table 1 indicates the kinds of input trees they can be applied to (ordered, unordered, or free), while Table 2 shows the kinds of subtrees they are able to identify (induced or embedded).

These algorithms are mainly derived from two well-known frequent pattern mining algorithms: Apriori [4] and FP-Growth [16].

Most of the algorithms in Tables 1 and 2 follow the Apriori iterative pattern mining strategy [4], where each iteration is broken up into two distinct phases:

- *Candidate generation*: A candidate is a potentially frequent subtree. In Apriori-like algorithms, candidates are generated from the frequent patterns discovered in the previous iteration. Most algorithms generate candidates of size $k + 1$ by merging two patterns of size k having $k - 1$ elements in common. The most common strategies to generate such candidates are the following:
  - *Rightmost expansion* generates subtrees of size $k + 1$ from frequent subtrees of size k by adding nodes only to the rightmost branch of the tree. This technique is used in algorithms like FREQT [1], uFreqt [19], and UNOT [5].
  - The *equivalence class-based extension* technique generates a candidate $(k + 1)$-subtree by joining two frequent k-subtrees with $(k - 1)$ nodes in common and that share a $(k - 1)$-prefix in their string codification. This extension mechanism is used, for instance, in Zaki's TreeMiner [36] and SLEUTH [35] algorithms.

Table 1 Some frequent tree mining algorithms classified by the kind of input trees they can be applied to

Algorithm	Ordered trees	Unordered trees	Free trees
FreqT [1]	●		
AMIOT [17]	●		
uFreqT [19]		●	
HybridTreeMiner [11]		●	●
FreeTreeMiner [12]		●	●
FreeTreeMiner' [22]		●	●
X3Miner [25]	●		
MB3Miner [26]	●		
IMB3Miner [27]	●		
TreeMiner [36]	●		
TreeMinerD [36]	●		
RETRO [7]	●		
Chopper [31]	●		
XSpanner [31]	●		
Uni3 [15]		●	
Unot [5]		●	
GASTON [20]			●
Phylominer [37]		●	
SLEUTH [35]		●	
TRIPS [28]	●		●
TIDES [28]	●		●
CMTreeMiner [10]	●		●
PathJoin [32]			●
DRYADE [30]			●
TreeFinder [29]			●

  - The *right-and-left tree join* method, which was proposed with the AMIOT algorithm [17], considers both the rightmost and the leftmost leaves of a tree in the generation of candidates.
  - Finally, the *extension and join* technique defines two extension mechanisms and a join operation to generate candidates. This method is used by HybridTreeMiner [11].

- *Support counting*: Given the set of potentially frequent candidates, this phase consists of determining their actual support and keeping only those candidates whose support is above the predefined minimum support threshold (i.e., those candidates that are actually frequent).

Some of the proposed algorithms have instead been derived from the FP-Growth algorithm [16]. Within this category, the PathJoin algorithm [32] uses compact structures called FP-Trees to encode input data, while CHOPPER and XSpanner [31] use a sequence-based codification for trees to identify frequent subtrees using frequent sequences.

Table 2 Some frequent tree mining algorithms and the kind of patterns they can identify

Algorithm	Induced subtrees	Embedded subtrees	Maximal /closed
FreqT [1]	●		
AMIOT [17]	●		
uFreqT [19]	●		
HybridTreeMiner [11]	●		
FreeTreeMiner [12]	●		
FreeTreeMiner' [22]	●		
X3Miner [25]		●	
MB3Miner [26]		●	
IMB3Miner [27]		●	
TreeMiner [36]		●	
TreeMinerD [36]		●	
RETRO [7]	●	●	
Chopper [31]		●	
XSpanner [31]		●	
Uni3 [15]		●	
Unot [5]	●		
GASTON [20]	●		
Phylominer [37]		●	
SLEUTH [35]		●	
TRIPS [28]		●	
TIDES [28]		●	
CMTreeMiner [10]		●	●
PathJoin [32]		●	●
DRYADE [30]			●
TreeFinder [29]			●

3 Mining partially-ordered trees

The algorithms referred to in the previous section extract frequent patterns from trees that are completely ordered or completely unordered, but none of them works with partially-ordered trees.

However, partially-ordered trees appear in different application domains. For example, Fig. 3 shows an XML document and its representation as a partially-ordered tree. This document represents the purchase history of a particular customer. In market basket analysis, it is important to preserve the order between the different orders placed by the customer, but the order between the items in the same order is not relevant.

We have extended Zaki's TreeMiner [36] and SLEUTH [35] algorithms to mine partially-ordered trees. The algorithm we have devised is able to identify frequent subtrees, both induced and embedded, in ordered, unordered, and partially-ordered trees. Hence its name, POTMiner, which stands for *Partially-ordered tree miner*.

Our algorithm is based on Apriori [4], just like TreeMiner and SLEUTH. Therefore, it follows an iterative pattern mining strategy. The kth iteration of the algorithm mines the

```
<customer>
   <order>
      <item>milk</item>
      <item>bread</item>
   </order>
   <order>
      <item>beer</item>
      <item>diapers</item>
   </order>
</customer>
```

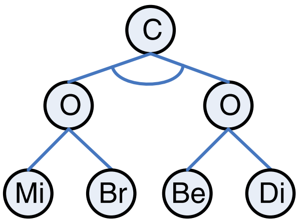


Fig. 3 An XML document and its partially-ordered tree representation

```
algorithm POTMiner
    Obtain frequent nodes (frequent patterns of size 1)
    Build candidate classes $C_1$ from the frequent nodes
    for k=2 to MaxSize
        for each class $P \in C_{k-1}$
            for each element $p \in P$.
                Compute the frequency of $p$
                if $p$ is frequent
                then
                    Create a new class $P'$ from $p$.
                    Add $P'$ to $C_k$
```

Fig. 4 The POTMiner algorithm

frequent subtrees of size k starting from the frequent subtrees of size $k-1$ discovered in the previous iteration. Each iteration is subdivided in two phases, candidate generation and support counting, as we mentioned in Sect. 2.5 when we discussed existing tree mining algorithms derived from Apriori.

Our algorithm employs Zaki's class-based extension technique to generate candidates. Candidates are grouped into equivalence classes, each class containing all the trees that share the same prefix in their codification string.

The following example illustrates the approach we follow for candidate generation. Let us suppose that the AA and AB trees are both frequent. These two trees belong to the same class, denoted by $[A]$, because they share the same prefix, i.e., A. When the AA tree of the class $[A]$ is extended, the resulting trees will have three nodes and all of them will belong to the class $[AA]$. When we generate candidates of size three derived from AA, we must combine AA of with all the frequent patterns of size two belonging to the same class than AA, i.e., $[A]$. If AA and AB are the only elements in $[A]$, the class-based extension technique will consider AAA, AAB, $AA\uparrow A$, and $AA\uparrow B$ as potential members of the $[AA]$ class. Similarly, the extension of the AB tree will produce the trees ABA, ABB, $AB\uparrow A$, and $AB\uparrow B$ as candidate patterns belonging to the class $[AB]$. Next, our algorithm will check which candidates are actually frequent.

The pseudocode of the resulting algorithm is shown in Fig. 4. The details of the candidate generation and support counting phases will be described in the following sections.

4 Candidate generation for partially-ordered trees

We use Zaki's equivalence class-based extension method to generate candidates. This method generates $(k+1)$-subtree candidates by joining two frequent k-subtrees with $k-1$ elements in common.

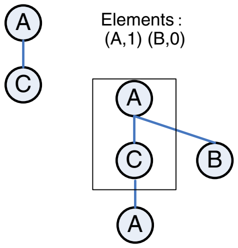


Fig. 5 Equivalence class with two elements, ACA and $AC\uparrow B$, that share the prefix AC

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

In Fig. 5, we represent an equivalence class with two elements, ACA and $AC\uparrow B$. The part of the tree that the class elements share is within the rectangle. Each node outside the rectangle corresponds to a different element in the equivalence class. This way, ACA is represented by $(A, 1)$ in the class $[AC]$, while $AC\uparrow B$ is represented by $(B, 0)$.

TreeMiner [36] and SLEUTH [35] use the class-based extension method to mine ordered and unordered embedded subtrees, respectively. The main difference between TreeMiner and SLEUTH is that only those extensions that produce canonical subtrees are allowed in SLEUTH, which works with unordered trees. This constraint avoids the duplicate generation of candidates corresponding to different representations of the same unordered trees. However, since we must handle both ordered and unordered sets of sibling nodes in partially-ordered trees, all extensions must be allowed in POTMiner, as happened in TreeMiner.

Elements in the same equivalence class are joined to generate new candidates. This join procedure must consider two scenarios [35]: cousin extension and child extension. The following paragraphs describe these two mechanisms as employed by POTMiner.

4.1 Cousin extension

Let (x, i) and (y, j) denote two elements in the same class $[P]$, and $[P_x^i]$ be the set of candidate trees derived from the tree that is obtained by adding the element (x, i) to P.

Cousin extension is performed when the father of (y, j) precedes (or is) the father of the element (x, i) in preorder, where the father of (y, j) is the node in position j and the father of (x, i) is in position i. Formally,

$$\textit{if } j \leq i \textit{ and } |P| = k - 1 \geq 1, \textit{ then } (y, j) \in [P_x^i].$$

The first condition checks that only nodes on the rightmost branch of P are extended. The second one makes cousin extension possible only when patterns have more than one node.

Figure 6 shows the elements generated by the cousin extension procedure from the elements in the class $[AC]$ in Fig. 5.

We have extended the pattern AC with the element $(A, 1)$ to generate the class $[P_A^1]$ whose codification is ACA. This class contains two elements that can be obtained by the cousin extension mechanism: first, the element $(A, 1)$ in the class $[P_A^1]$ is the result of the union of the element $(A, 1)$ from $[P]$ with itself; second, the element $(B, 0)$ in the class $[P_A^1]$ is the result of the union of the elements $(A, 1)$ and $(B, 0)$ from $[P]$.

Cousin extension can also be applied to the pattern P using the element $(B, 0)$, which generates the class $[P_B^0]$ whose codification is $AC\uparrow B$. This class contains the element $(B, 0)$, which can be obtained by the union of $(B, 0)$ from $[P]$ with itself. The union between $(B, 0)$

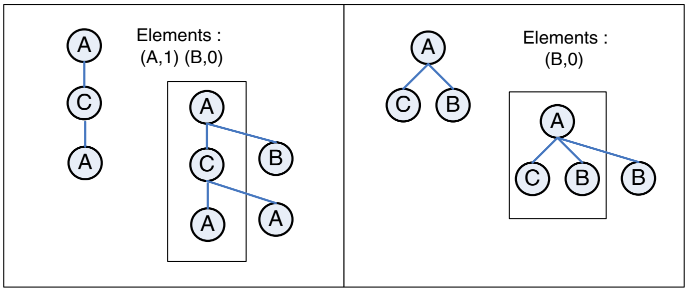


Fig. 6 Cousin extension applied to the elements of the class in Fig. 5

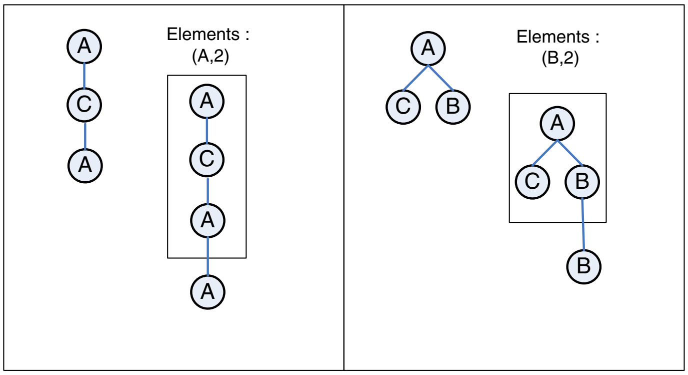


Fig. 7 Child extension applied to the elements of the class in Fig. 5

and $(A, 1)$ in $[P_B^0]$ is not possible since the father or $(A, 1)$ (i.e, the node in position 1), does not precede the father of $(B, 0)$ in $[P]$ (i.e, the node in position 0).

4.2 Child extension

As above, let (x, i) and (y, j) denote two elements in the same class $[P]$, and $[P_x^i]$ be the set of candidate trees derived from the tree that is obtained by adding the element (x, i) to P.

When (y, j) is a sibling node of (x, i), the child extension mechanism lets us add y as a child of the rightmost leaf of the tree. The formal definition of child extension is:

$$\text{if} \quad j = i \quad \text{then} \quad (y, k - 1) \in [P_x^i].$$

Child extension is used to make tree patterns grow in depth, while cousin extension lets them grow in width.

Figure 7 shows the elements generated by the child extension method from the elements in the class AC shown in Fig. 5.

The extension of the pattern AC with the element $(A, 1)$ generates the elements of the class $[P_A^1]$. This class, apart from the two elements generated by the cousin extension mechanism, also contains a new element, $(A, 2)$, which is obtained by the union of the element $(A, 1)$ in $[P]$ with itself. The union of $(A, 1)$ with $(B, 0)$ does not generate a new element because $(B, 0)$ is not a sibling of $(A, 1)$ in $[P]$.

The extension of pattern AC with the element $(B, 0)$ results in the class $[P_B^0]$. This class contains the element $(B, 0)$, generated by the cousin extension mechanism above, and the element $(B, 2)$, resulting from the child extension of $(B, 0)$ in $[P]$ with itself. As above, the element $(B, 0)$ cannot be joined with $(A, 1)$ because $(B, 0)$ is not a sibling of $(A, 1)$ in $[P]$.

5 Support counting for induced and embedded subtrees

Once POTMiner has generated the potentially frequent candidates, it is necessary to determine which ones are actually frequent.

The support counting phase in POTMiner follows the strategy of AprioriTID [4]. Instead of checking if each candidate is present in each database tree, which is a costly operation, special lists are used to preserve the occurrences of each pattern in the database, thus facilitating the support counting phase.

Several kinds of occurrence lists have been proposed in the tree mining literature:

- *Standard occurrence lists* [11] preserve the identifiers of the trees as well as the matching between the pattern nodes and the database tree nodes using a breadth-first codification scheme. Each element of the occurrence list for pattern X has the form $(tid, i_1 \dots i_k)$, where tid is the tree identifier and $i_1 \dots i_k$ represent the mapping between the nodes in the pattern X and those in the database tree.
- *Rightmost occurrence lists* [1] only need to store the occurrences of the rightmost leaf of X in each database tree. Rightmost occurrence lists, also known as RMO-lists, are stored with each candidate and contain (tid, n) pairs, where n is the position of each tree node in the database tree tid that matches with the rightmost leaf of the candidate pattern. These lists are only useful for mining induced subtrees, since they only preserve one node for each occurrence representing the whole subtree. Update operations can be performed on these lists as in AprioriTID [4] so that the RMO-list of a new k-pattern can be obtained from the RMO-lists of the $k - 1$ patterns it was generated from.
- *Vertical occurrence lists* [15,27] group the occurrence coordinates of each subtree, as employed by the Tree Model Guide candidate generation method [26], which is a specialization of the rightmost extension technique we mentioned in Sect. 2.5.
- *Scope lists* [35] preserve each occurrence of a pattern X in the database using a tuple (t, m, s) where t is the tree identifier, m stores which nodes of the tree match those of the $(k - 1)$ prefix of the pattern X in depth-first order, and s is the scope of the last node in the pattern X. The scope of a node is defined as a pair [l,u] where l is the position of the node in depth-first order and u is the position of its rightmost descendant. The scope list of a pattern X is the list of all the tuples (t, m, s) representing the occurrences of X in the tree database.

In POTMiner, we use scope lists to preserve the occurrences of a pattern in the tree database. The main difference between our scope lists and the ones proposed by Zaki in TreeMiner [36] and SLEUTH [35] is that we must add two new elements to the tuples in the scope lists, d and Θ, in order to deal with induced subtrees and partially-ordered trees, respectively.

Our scope lists, then, contain tuples (t, m, s, d, Θ) where t is the tree identifier, m stores which nodes of the tree match those of the $(k - 1)$ prefix of the pattern X in depth-first order, s=[l,u] is the scope of the last node in the pattern X, d is a depth-based parameter used for mining induced subtrees (it is not needed when mining embedded subtrees), and Θ indicates whether the last node of the pattern is ordered (Θ =o) or unordered (Θ =u) in the database tree.

When building the scope lists for patterns of size 1, m is empty and the element d is initialized with the depth of the pattern only node in the original database tree.

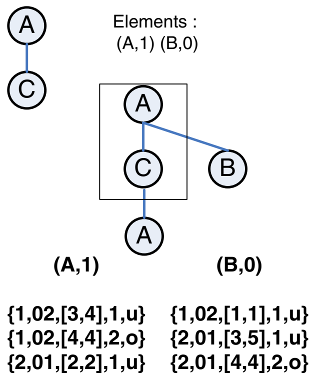


Fig. 8 Class with two elements, $(A, 1)$ and $(B, 0)$, and their scope lists

We obtain the scope list for a new candidate of size k by joining the scope lists of the two subtrees of size $k-1$ that were involved in the generation of the candidate. Let $(t_x, m_x, s_x, d_x, \Theta_x)$ and $(t_y, m_y, s_y, d_y, \Theta_y)$ be the scope lists of the subtrees involved in the generation of the candidate. The scope list for this candidate is built by a join operation that depends on the candidate extension method used to generate the candidate, i.e., whether the candidate was generated by cousin extension or by child extension.

Figure 8 shows the class from Fig. 5 including the scope lists of its elements, which we will use to illustrate how scope lists are joined.

5.1 In-scope join

The in-scope join, which is used in conjunction with the child extension mechanism, proceeds as follows:
if

1. $t_x = t_y = t$ and
2. $m_x = m_y = m$ and
3. $d_x = 1$ when we are looking for induced patterns, and
4. $s_y \subset s_x$ (i.e., $l_x < l_y$ and $u_x \geq u_y$),

then add $[t, m \bigcup \{l_x\}, s_y, d_y - d_x, \Theta_y]$ to the scope list of the generated candidate.

The third constraint is needed when we are interested in obtaining only the induced subtrees that are present in the tree database. This constraint is not used when mining embedded subtrees.

Figure 9 shows the candidates obtained using child extension from the elements in the class shown in Fig. 8. Let us suppose that we are identifying embedded patterns, i.e., the value of d_x is not taken into account.

$ACAA$ was generated from the union of $(A, 1)$ with itself using child extension. Therefore, we perform the in-scope join of the elements in the scope list of $(A, 1)$ with themselves. The only pair that matches is (1,02,[3,4],1,u) and (1,02,[4,4],2,o), hence the scope list of $ACAA$ contains a single tuple, which is (1,023,[4,4],1,o).

$AC\uparrow BB$ results from the child extension of $(B, 0)$ with itself. In this case, it is only possible to join (2,01,[3,5],1,u) and (2,01,[4,4],2,o), generating a scope list containing just (2,013,[4,4],1,o).

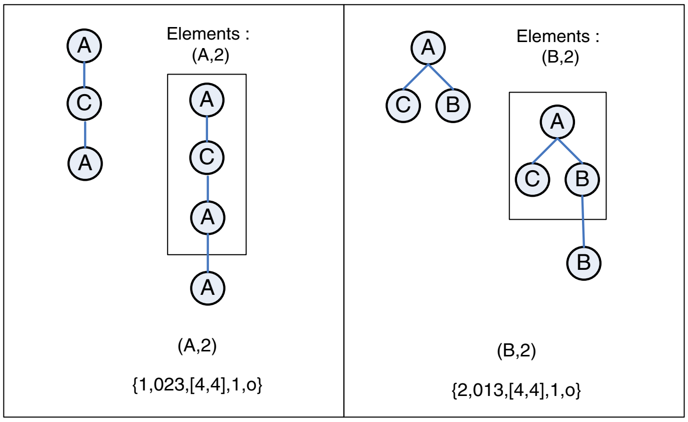


Fig. 9 Child extension of the elements in the class shown in Fig. 8 and the scope lists resulting from the in-scope join

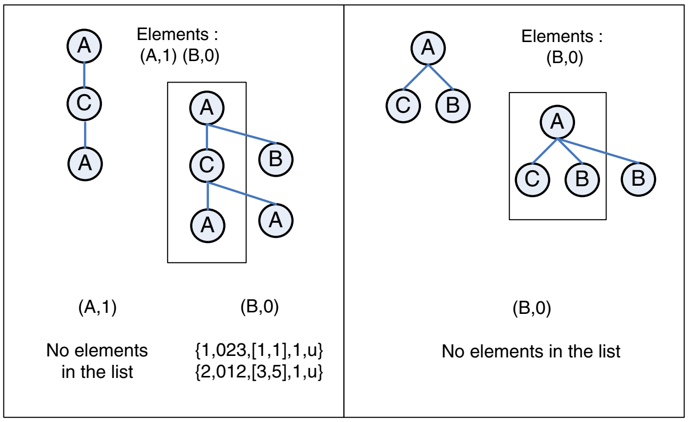


Fig. 10 Cousin extension of the elements in the class shown in Fig. 8 and the scope lists resulting from the out-scope join

5.2 Out-scope join

The out-scope join is used in conjunction with the cousin extension mechanism. It works as follows:
if

1. $t_x = t_y = t$ and
2. $m_x = m_y = m$ and
3. if the node is ordered and $s_x < s_y$ (i.e., $u_x < l_y$) or the node is unordered and either $s_x < s_y$ or $s_y < s_x$ (i.e., either $u_x < l_y$ or $u_y < l_x$),

then add $[t, m \bigcup \{l_x\}, s_y, d_y, \Theta_y]$ to the scope list of the generated candidate.

Figure 10 shows the candidates obtained by the cousin extension method from the elements in the class shown in Fig. 8.

Two elements are generated by cousin extension from the pattern ACA. The element $(A, 1)$ in $[ACA]$ is generated by the union of $(A, 1)$ in $[AC]$ with itself. Its scope list has no

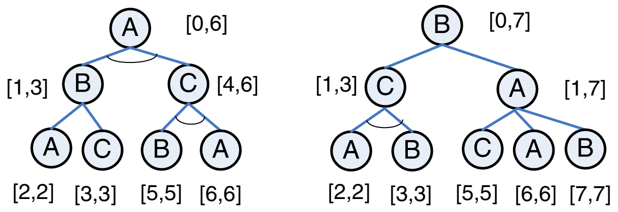


Fig. 11 Sample dataset containing two partially-ordered trees

elements because no pair of tuples satisfy the three conditions required by the out-scope join. The element $(B, 0)$ in $[ACA]$ is generated by the union of $(A, 1)$ with $(B, 0)$ in $[AC]$. We can join the first element of each list, (1,02,[3,4],1,u) and (1,02,[1,1],1,u), in order to generate (1,023,[1,1],1,u). The join of the third element in the scope list of $(A, 1)$, (2,01,[2,2],1,u), with the second element of the list of $(B, 0)$, (2,01,[3,5],1,u), results in (2,012,[3,5],1,u).

The cousin extension of the pattern $AC{\uparrow}B$ results in the element $(B, 0)$. In this case, the out-scope join does not generate any elements, and, therefore, the scope list of $(B, 0)$ in $[AC{\uparrow}B]$ is empty.

5.3 Counting the support of a pattern

Checking if a pattern is frequent consists of counting the elements in its scope list. The counting procedure is different depending on whether we consider the weighted support σ_w or not.

- If we count occurrences using the weighted support, all the tuples in the scope lists must be taken into account.
- If we are not using the weighted support, the support of a pattern is the number of different tree identifiers within the tuples in the scope list of the pattern.

This procedure is valid for counting embedded patterns. When counting induced patterns, we consider only the elements in the scope lists whose d parameter equals 1. It should be noted that d represents the distance between the last node in the pattern and its prefix m, hence $d = 1$ indicates the presence of an induced pattern provided that the scope lists were generated using the join operations discussed above.

6 Example

In this section, we present an example to illustrate how POTMiner works with partially-ordered trees. The dataset in Fig. 11 will be used as we identify all the induced subtrees that appear in both trees (i.e., the minimum support is 100%).

Figure 12 shows the vertical representation of the trees in Fig. 11, where each node is represented by its scope list. For example, the list corresponding to the node A contains six elements because A appears six times in the dataset. Each element in the scope lists has all the information corresponding to a single occurrence of a pattern as described in Sect. 5 (tree identifier, prefix, scope, depth parameter, order).

Since the three nodes are frequent, we build the classes shown in Fig. 13. These classes are built by child extension (Sect. 4.2) and the resulting scope lists are obtained using the in-scope join operation (Sect. 5.1).

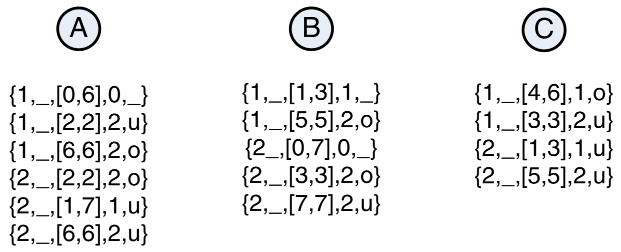


Fig. 12 Vertical representation of the trees in Fig. 11

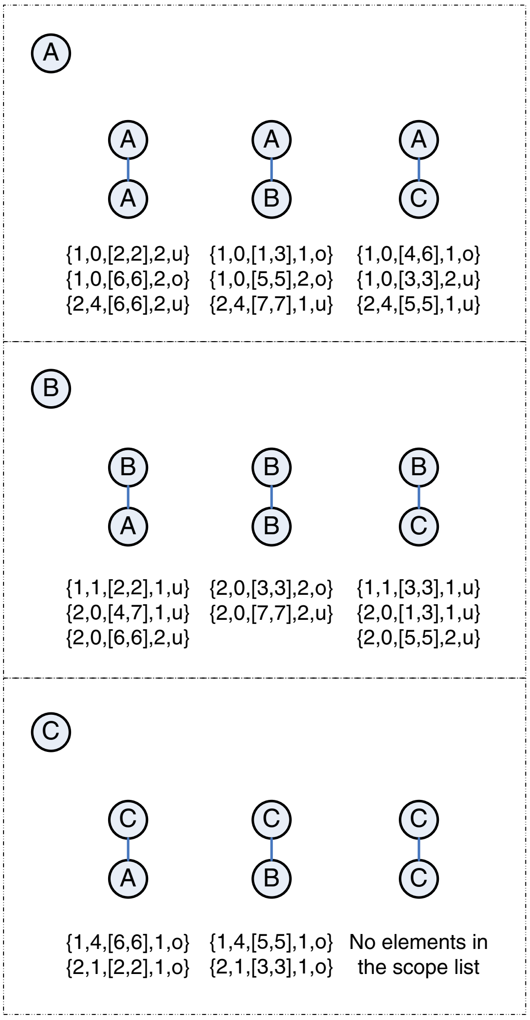


Fig. 13 Classes derived from the patterns of size 1 in Fig. 11

All the elements in the first class, which are derived from the node A, are frequent because there is at least one occurrence of each pattern in each tree. The other two classes contain two frequent patterns and an infrequent one. In class B, the pattern BB appears twice in the

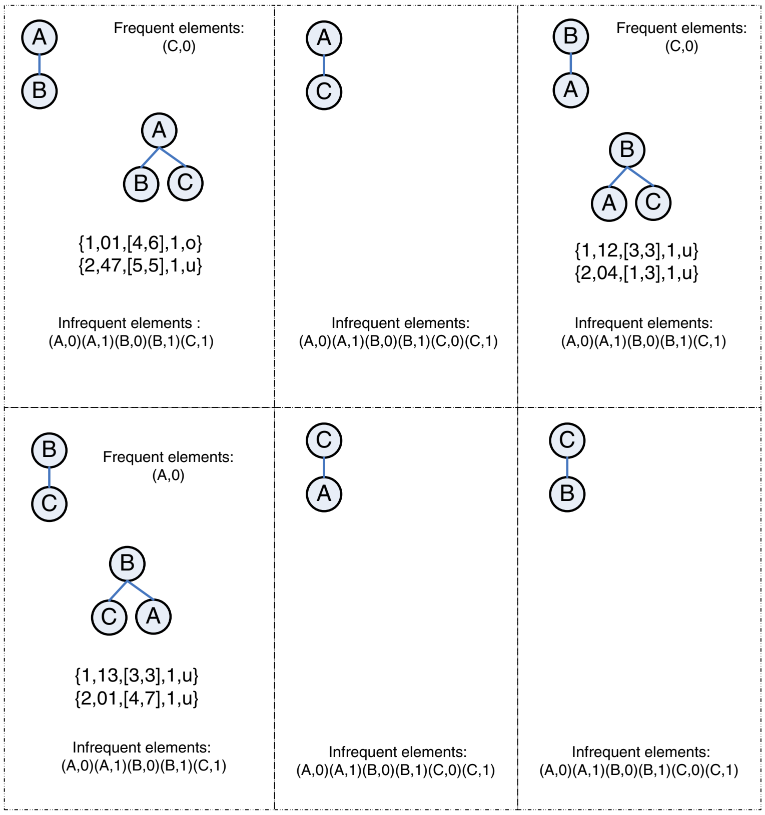


Fig. 14 Classes derived from the frequent patterns of size 2 in Fig. 13

second tree but it has no occurrences in the first tree. In class C, the pattern CC does not appear in any of the trees.

Since we are looking for induced patterns, it is necessary to check if the discovered patterns are actually frequent as induced patterns before we proceed with additional extensions. In this case, the first element of class A, i.e. AA, is not a frequent induced pattern because all its occurrences are embedded (their depth parameter is greater than 1). This element, therefore, will not be extended and it will not be returned as a frequent induced pattern. However, it must be taken into account for the extension of the other elements in the same class (i.e., AB and AC). These elements are frequent-induced patterns since, although there are elements with $d = 2$ in their scope lists, there is at least one occurrence of each pattern with $d = 1$ in each database tree. Likewise, patterns BA, BC, CA, and CB are frequent.

Figure 14 shows the result of the extension of the six frequent induced patterns of size 2 obtained in the previous iteration. Since we are working with partially-ordered trees, we have to address three different situations:

- *o–o* (*ordered-ordered*): The union of CA and CB does not produce any frequent pattern because both $CA{\uparrow}B$ and $CB{\uparrow}A$ appear only in one of the database trees. These patterns are built by cousin extension (Sect. 4.1) and the corresponding out-scope join operation (Sect. 5.2) detects that $CA{\uparrow}B$ is not in the first tree and $CB{\uparrow}A$ is not in the second one.

- *u–u* (*unordered-unordered*): The patterns $BA\uparrow C$, $BC\uparrow\uparrow A$ are both frequent. The reason is that they have been generated by the union of unordered elements and the union can be done in both directions.
- *u–o* (*unordered-ordered*): The pattern $AB\uparrow C$ is frequent, but the pattern obtained by changing the order of its sibling nodes, $AC\uparrow B$, is not frequent. $AB\uparrow C$ is an unordered subtree in the second tree of the dataset but it only appears as an ordered subtree in the first tree. Hence, both $AC\uparrow B$ and $AB\uparrow C$ occur in the first tree, but only $AB\uparrow C$ is in the second one.

The classes derived from the frequent patterns of size 3 contain no frequent elements. Therefore, no frequent patterns of size 4 are found in our tree dataset.

7 Implementation issues

In this section, we analyze the complexity of our algorithm and we discuss some implementation details that have important consequences in practice. In particular, we address the parallelization of our algorithm in order to reduce its running time and we also propose an alternative method to build scope lists in order to make POTMiner less memory consuming.

7.1 POTMiner complexity

POTMiner starts by computing the frequent patterns of size 1. This step is performed by obtaining the vertical representation of the tree database, i.e., the individual nodes that appear in the trees with their occurrences represented as scope lists. This representation is obtained in linear time with respect to the number of trees in the database by scanning it and building the scope lists for patterns of size 1. We then discard the patterns of size 1 that are not frequent. This results in L scope lists corresponding to the L frequent labels in the tree database and each frequent label leads to a candidate class of size 1.

Let $c(k)$ be the number of classes of size k, which equals the number of frequent patterns of size k, and $e(k)$ the number of elements that might belong to a particular class of size k (i.e., the number of patterns of size $k+1$ that might be included in the class corresponding to a given pattern of size k).

In POTMiner, each tree pattern grows only by adding a node as a child to a node in its rightmost path. In the worst case, when the tree is just a sequence of size k, the number of different trees of size $k+1$ that can be obtained by the extension of the tree of size k is $L * k$. Hence, the number of elements in a particular class, $e(k)$, is $O(L * k)$.

The number of classes of size 1 equals L, the number of frequent labels, i.e., $c(1) = L$. The classes of size $k+1$ are derived from the frequent elements in classes of size k. In the worst case, when all the $e(k)$ elements are frequent, $c(k+1) = c(k) * e(k)$. Solving the recurrence, we obtain $c(k+1) = c(k) * e(k) = O(L^{k+1} * k!)$, which can also be expressed as $c(k) = O(L^k * (k-1)!)$.

For each pattern considered of size $k+1$, POTMiner must perform a join operation to obtain its scope list from the scope lists of the two patterns of size k that led to it.

The size of the scope list for a pattern of size k is $O(t * e)$ while the cost of a scope-list join is $O(t * e^2)$, where t is the number of trees in the database and e is the average number of embeddings of the pattern in each tree [35].

In the worst case, when we are looking for embedded subtrees, the number of embeddings of a pattern of size $k-1$ in a tree of size n equals the number of subtrees of size $k-1$ within the tree of size n. This number, $s(k-1)$, is bounded by $\binom{n}{k-1} \leq n^k/(k-1)!$.

Hence, the cost of the join operation needed for obtaining the scope list of a pattern of size k, is $j(k) = O(t * s(k-1)^2) = O(t * (n^k/(k-1)!)^2)$.

The cost of obtaining all the frequent patterns of size k will be, therefore, $O(c(k) * j(k)) = O(L^k * (k-1)! * t * (n^k/(k-1)!)^2) = O(L^k * t * n^{2k}/(k-1)!) = O(t * (Ln^2)^k/(k-1)!)$.

The total cost of executing the POTMiner algorithm to obtain all the frequent patterns up to $k = \text{MaxSize}$ is $\sum_{k=1\ldots\text{MaxSize}} (t * (Ln^2)^k/(k-1)!)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e., $k = \text{MaxSize}$), POTMiner is $O(t * (Ln^2)^{\text{MaxSize}}/(\text{MaxSize} - 1)!)$.

Therefore, our algorithm is linear with respect to the number of trees in the tree database and its execution time is also proportional to the number of patterns considered.

7.2 Parallelization: making POTMiner faster

Parallelism can be used to improve the performance of data mining algorithms. The parallel implementation of these algorithms lets us exploit the architecture of modern multi-core processors and multiprocessors. In fact, different parallel and distributed association rule mining algorithms have been proposed in the literature, for instance [2,3,8,21,23,24].

POTMiner can also be parallelized. We have followed a candidate distribution approach [3]. In POTMiner, generating candidates in parallel simply involves partitioning the set of all candidate classes to be extended among the available processors.

The main idea behind the parallel version of POTMiner is that, in each step of the algorithm, the extension of each candidate class and the corresponding scope list join operations can be performed in parallel. Since these operations are independent from each other, each one can be assigned to a different processor without incurring into significant coordination costs.

The parallel version of POTMiner independently processes each class of size k to obtain candidates of size $k+1$ and, at the end of each iteration, the results of the independently-processed classes are combined to return all the candidate classes of size $k+1$. The pseudocode of the parallel version of POTMiner is shown in Fig. 15.

7.3 On-demand scope lists: reducing memory consumption

The candidate generation process is very memory consuming due to the huge amount of scope lists that have to be maintained. Moreover, the size of each scope lists is proportional to the number of embedded occurrences of each pattern, which can also be huge in large databases.

We have devised a variant of our algorithm, called LightPOTMiner, which computes scope lists on demand instead of storing all the scope lists in memory.

```
algorithm ParallelPOTMiner
    Obtain frequent nodes (frequent patterns of size 1)
    Build candidate classes $C_1$ from the frequent nodes
    for k=2 to MaxSize
        for each class $P \in C_{k-1}$
            Extend $P$ in parallel to obtain $P_{extended}$
        for each class $P \in C_{k-1}$
            $C_k = C_k \bigcup P_{extended}$
```

Fig. 15 Paralellization of the POTMiner algorithm

Fig. 16 On-demand recursive construction of scope lists in LightPOTMiner

```
algorithm $scopeList$ (Tree $t$): $s$
    // $t : n_1, n_2..n_{k-1}, n_k$

    if $k = 1$
    then
        $s$ = scope list of node $n_k$
    else
        $t_1 = t - n_k$
        $t_2 = t - n_{k-1}$
        $s_1 = scopeList(t_1)$
        $s_2 = scopeList(t_2)$
        if $n_k$.parent $= n_{k-1}$
        then
            // $t$ was obtained by child extension
            s = in-scope-join ($s_1$, $s_2$)
        else
            // $t$ was obtained by cousin extension
            s = out-scope-join ($s_1$, $s_2$)
```

POTMiner builds the scope lists of a new candidate by joining the scope lists of the patterns involved in its generation. LightPOTMiner recursively computes such scope lists directly from the scope lists of the frequent nodes, i.e., the tree patterns of size 1.

The key idea of these recursive process is that it is always possible to know if a given pattern was obtained by cousin extension or by child extension. Hence, we can infer which subtrees were used to generate the tree pattern and which join operation to perform on the corresponding scope lists, i.e., in-scope or out-scope join. The recursive algorithm employed by LightPOTMiner is shown in Fig. 16.

In LightPOTMiner, scope lists are calculated on-demand. The scope list for a pattern of size k is obtained by joining two scope lists of patterns of size $k-1$. Formally, the number of join operations needed to obtain a scope list for a pattern of size k is given by the following expression: $\text{join}(k) = 1 + 2 * \text{join}(k-1) = 2^k$. The cost of computing a scope list in LightPOTMiner is $j_{\text{light}}(k) = 2^k * j(k) = O(2^k * t * (n^k/(k-1)!)^2)$.

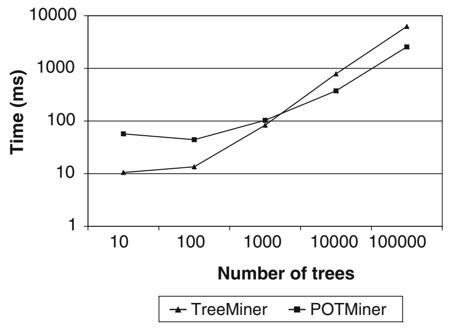
The cost of obtaining all the frequent patterns of size k in LightPOTMiner will be, therefore, $O(c(k) * j_{\text{light}}(k)) = O(L^k * (k-1)! * 2^k * t * (n^k/(k-1)!)^2) = O((2L)^k * t * n^{2k}/(k-1)!) = O(t * (2\text{Ln}^2)^k/(k-1)!)$.

The total cost of executing the LightPOTMiner algorithm to obtain all the frequent patterns up to $k = \text{MaxSize}$ is $\sum_{k=1...\text{MaxSize}} (t * (2\text{Ln}^2)^k/(k-1)!)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e., $k = \text{MaxSize}$), LightPOTMiner is $O(t * (2\text{Ln}^2)^{\text{MaxSize}}/(\text{MaxSize}-1)!)$.

In other words, we introduce a 2^{MaxSize} factor in the execution time of LightPOTMiner to reduce memory consumption. LightPOTMiner just needs to store L scope lists corresponding to the frequent patterns of size 1, while POTMiner had to store all the scope lists in memory, up to $c(\text{MaxSize}-1)$, the number of frequent patterns of size $\text{MaxSize}-1$ we might obtain, which is $L^{\text{MaxSize}-1} * (\text{MaxSize}-2)!$ in the worst case.

8 Experimental results

We have performed two series of experiments to evaluate POTMiner. First, we have performed some experiments with synthetic datasets in order to compare POTMiner with existing



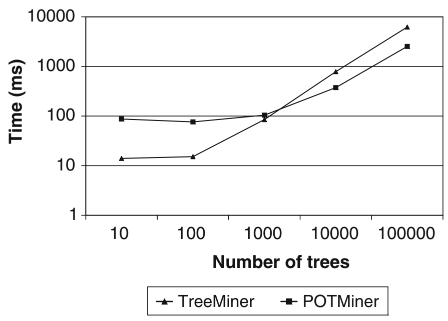


Fig. 17 POTMiner and TreeMiner execution times when identifying induced patterns (*left*) and embedded patterns (*right*) in completely-ordered trees

algorithms. Second, we have devised some experiments with the aim of analyzing POTMiner performance with real datasets.

8.1 POTMiner versus TreeMiner/SLEUTH

All the experiments described in this section have been performed on a 2 GHz Intel T7200 Dual Core processor with 2 GB of main memory running on Windows Vista. POTMiner has been implemented in Java using Sun Microsystems JDK 5, while Zaki's TreeMiner and SLEUTH C++ implementations were obtained from http://www.cs.rpi.edu/~zaki/.

The experiments were performed with five synthetic datasets generated by the tree generator available at http://www.cs.rpi.edu/~zaki/software/TreeGen.tgz. The datasets were obtained using the generator default values and varying the number of trees from 10 to 100,000. The labeled trees in these datasets contain 10 different labels, their maximum depth is 5, and their nodes maximum fanout is also 5.

8.1.1 Ordered trees

In our first experiments, we compare POTMiner to TreeMiner [36]. Since TreeMiner works on ordered trees, we consider that the trees in our synthetic datasets are completely-ordered.

Figure 17 shows POTMiner and TreeMiner execution times using a minimum support threshold of 20% to identify both induced and embedded patterns in our datasets.

POTMiner, when dealing with completely-ordered trees, works as TreeMiner. Therefore, the patterns identified by POTMiner and TreeMiner are exactly the same.

It should be noted that the charts in Fig. 17 use a logarithmic scale. The results show that both POTMiner and TreeMiner are efficient, scalable algorithms for mining induced and embedded subtrees in ordered trees.

For small datasets, POTMiner execution times are slightly higher than TreeMiner execution times. However, this difference disappears in larger datasets and POTMiner is even faster than TreeMiner. The observed differences are probably due to the different programming platforms used in the implementation of the two algorithms (Java for POTMiner, C++ for TreeMiner). The Java Virtual Machine used by our implementation of POTMiner introduces some start-up overhead with respect to the native C++ implementation of TreeMiner. This additional overhead is noticeable when dealing with small datasets, but quickly disappears on larger datasets. In larger datasets, POTMiner outperforms TreeMiner due to a more careful dynamic memory management.

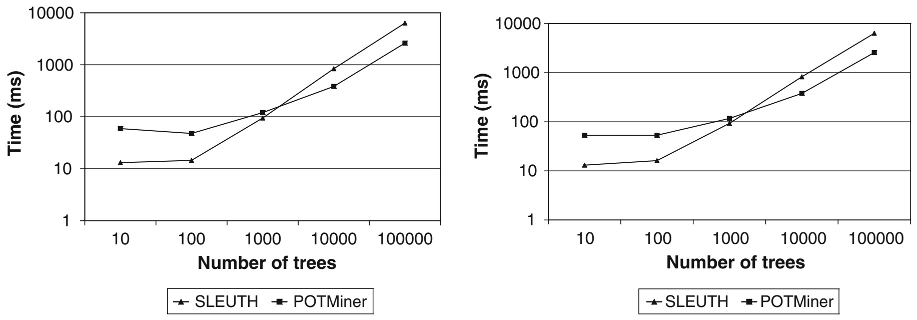


Fig. 18 POTMiner and SLEUTH execution times when identifying induced patterns (*left*) and embedded patterns (*right*) in completely-unordered trees

8.1.2 Unordered trees

In this series of experiments, we compare the performance of POTMiner with the algorithm proposed by Zaki for the identification of frequent patterns in unordered trees: SLEUTH [35]. The datasets we have used for these experiments contain the same trees we used to compare POTMiner and TreeMiner, but this time we regard them as completely-unordered trees.

The patterns obtained by POTMiner, which was designed to deal with partially-ordered trees, are always ordered. For unordered patterns, POTMiner returns all the frequent patterns obtained with different permutations for the unordered sibling nodes. In the case of completely-unordered trees, all such permutations will be frequent. Since SLEUTH only returns the canonical representation of each frequent pattern, the number of patterns that POTMiner returns is larger than the number of patterns returned by SLEUTH, even though both algorithms identify exactly the same unordered tree patterns.

Figure 18 shows the results we have obtained when comparing POTMiner and SLEUTH. POTMiner and SLEUTH are similar in efficiency and scalability. The differences that can be observed are, again, due to their different execution platforms and implementation details.

8.1.3 Partially-ordered trees

We have also performed some experiments with partially-ordered trees. Since TreeMiner [36] and SLEUTH [35] cannot be applied to partially-ordered trees, we have studied the behavior of POTMiner when dealing with this kind of trees.

Starting from the same datasets used in the previous experiments, we have randomly considered tree nodes as ordered or unordered. Figure 19 shows POTMiner execution times and the number of patterns discovered when varying the percentage of ordered nodes in our synthetic datasets.

It should be noted that the number of discovered patterns decrease when the number of trees increase. This is due to the fact that the minimum support threshold is a relative value (20% in our experiments) and the trees were randomly generated. Therefore, the probability of a given pattern reaching the minimum support threshold decreases as the number of trees is increased.

As we had expected, we found that execution times slightly decrease when the percentage of ordered nodes is increased, since ordered trees are easier to mine than unordered trees and the number of frequent patterns is smaller in ordered trees than in unordered ones.

	Induced subtrees									
	0% ordered nodes		25% ordered nodes		50% ordered nodes		75% ordered nodes		100% ordered nodes	
# Trees	Time	Patterns	Time	Patterns	Time	Patterns	Time	Patterns	Time	Patterns
10	59.2	35	94.5	32	43.0	31	44.1	30	57.0	28
100	47.8	18	103.0	18	79.8	17	62.7	17	44.2	16
1000	120.1	8	109.5	8	98.7	8	105.1	8	103.2	7
10000	383.0	7	447.8	7	452.6	7	410.1	7	373.5	6
100000	2608.5	7	2812.5	7	2774.0	6	2832.2	6	2557.0	6

	Embedded subtrees									
	0% ordered nodes		25% ordered nodes		50% ordered nodes		75% ordered nodes		100% ordered nodes	
# Trees	Time	Patterns	Time	Patterns	Time	Patterns	Time	Patterns	Time	Patterns
10	53.2	53	74.5	48	54.1	46	58.9	43	87.1	41
100	53.3	19	95.4	19	55.7	18	64.2	18	76.2	17
1000	117.1	11	104.6	10	110.4	10	126.1	10	104.2	9
10000	378.9	7	405.5	7	497.4	7	412.7	7	374.4	6
100000	2572.7	7	2804.5	7	2747.7	6	2794.5	6	2530.4	6

Fig. 19 POTMiner execution times and number of discovered patterns when varying the percentage of ordered nodes in partially-ordered trees

We have performed experiments to identify both induced and embedded subtrees in the five datasets of varying size we used in our prior series of experiments. For these randomly-generated datasets, no important differences have been observed between the time required for the identification of induced patterns and the time needed for the discovery of embedded patterns.

8.2 POTMiner on real datasets

We have also performed some experiments to study POTMiner behavior on real datasets. The datasets used in these experiments come from the Mutagenesis database. This multirelational database is frequently used as an ILP benchmark.

The Mutagenesis database contains four relations. We can derive a tree database from it by building a tree from each tuple in its target relation. The links from one relation to another within the database (i.e., the foreign keys in relational database terms) let us grow the different tree branches. This procedure has been used to obtain two different tree datasets:

- The first dataset, called Muta2, is built by following two links between Mutagenesis relations (*mole–molatm* and *moleatm–atom*). This dataset contains 188 trees with 138 nodes per tree (a total of 25,969 nodes).
- The second dataset, called Muta3, is built by following three links between Mutagenesis relations (*mole–molatm*, *moleatm–atom*, and *atom–bond*). This dataset contains 188 trees with 435 nodes per tree (81,898 nodes overall).

We have performed several experiments on these datasets to identify embedded subtrees of different sizes using several support thresholds. These experiments have been performed on an Intel 2.66 GHz Q6700 4-core processor with 4GB of main memory running on Windows Vista.

8.2.1 Discovered patterns

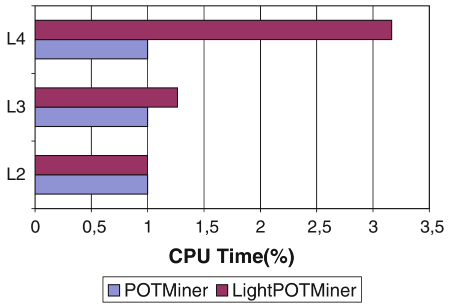
Figure 20 displays the number of patterns identified by POTMiner in the Muta2 and Muta3 datasets when varying the minimum support threshold and the maximum pattern size.

We have used three different minimum support thresholds in our experiments (5, 10, and 20%) to obtain all the frequent embedded patterns of size 2 (L2), size 3 (L3), and size 4 (L4) in the Muta2 and Muta3 datasets.

	Muta2		
	Minimum Support		
MaxSize	20%	10%	5%
L2	82	112	183
L3	857	1245	2438
L4	9136	14993	31241

	Muta3		
	Minimum Support		
MaxSize	20%	10%	5%
L2	115	145	216
L3	1922	2593	4618
L4	40571	58106	111446

Fig. 20 Number of patterns identified by POTMiner in the Muta2 and Muta3 datasets



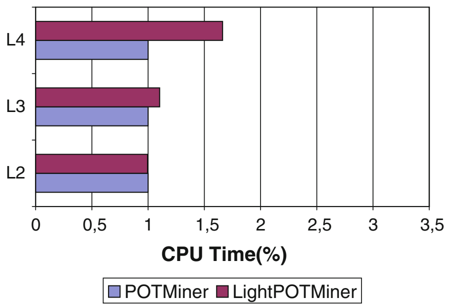


Fig. 21 Additional CPU time required by LightPOTMiner on the Muta2 (*left*) and Muta3 (*right*) datasets with respect to POTMiner

8.2.2 LightPOTMiner

LightPOTMiner is the variant of POTMiner that builds scope lists on demand, as described in Sect. 7.3, in order to avoid the need to store all the scope lists in memory.

LightPOTMiner trades space for time. Memory consumption is reduced at the cost of the additional CPU time required to build the scope lists as needed. Figure 21 displays the execution time overhead introduced by LightPOTMiner when identifying embedded patterns in the Muta2 and Muta3 datasets using a 20% minimum support threshold. In Muta2, LightPOTMiner is three times slower than POTMiner while, in Muta3, the overhead is lower than two times, a reasonable cost if we take into account the huge memory savings we obtain: POTMiner, as TreeMiner or SLEUTH, would need to store tens of thousands of long scope lists, while LightPOTMiner just requires a few dozens, for the frequent items in the database.

8.2.3 Parallel implementation of POTMiner and LightPOTMiner

We have also performed some experiments to evaluate the speed-up obtained by the parallel versions of POTMiner and LightPOTMiner. We have parallelized these algorithms as described in Sect. 7.2.

Figure 22 shows the actual running times (in s) required by the parallel implementation of POTMiner using four cores in a quad core processor. These results correspond to the time required by POTMiner to identify embedded subtrees in the Muta2 and Muta3 datasets for different minimum support thresholds and pattern sizes.

	Muta2		
	Minimum Support		
MaxSize	20%	10%	5%
L2	1,62	1,62	1,62
L3	1,98	2,11	2,22
L4	17,19	20,69	26,83

	Muta3		
	Minimum Support		
MaxSize	20%	10%	5%
L2	41,67	41,76	42,29
L3	44,55	44,90	45,61
L4	408,94	435,00	493,43

Fig. 22 Parallel POTMiner execution time (s) on the Muta2 and Muta3 datasets using four processors

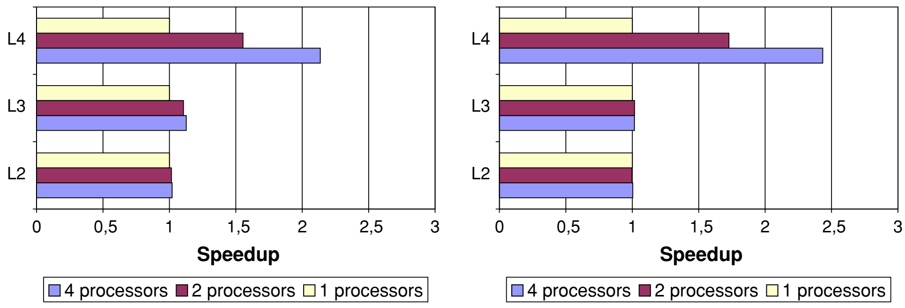


Fig. 23 Parallel POTMiner execution speedup on the Muta2 (*left*) and Muta3 (*right*) datasets

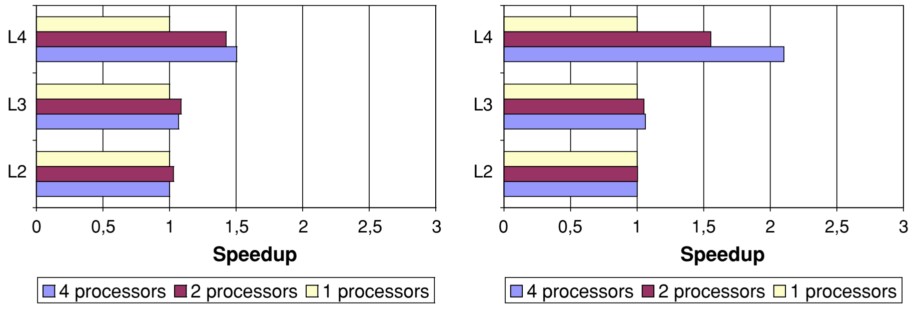


Fig. 24 Parallel LightPOTMiner speedup on the Muta2 (*left*) and Muta3 (*right*) datasets

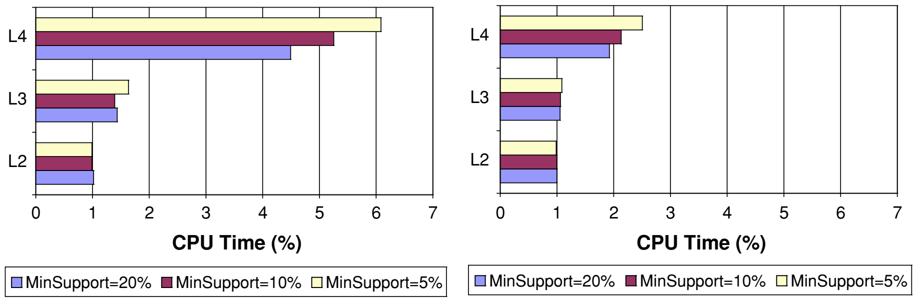


Fig. 25 Parallel LightPOTMiner versus parallel POTMiner for different minimum support thresholds on the Muta2 (*left*) and Muta3 (*right*) datasets

Figure 23 shows the performance improvement we have obtained with the parallel version of POTMiner. Using just two threads of control, the parallel POTMiner is 1.6 times faster than the sequential POTMiner in Muta2 and 1.7 times faster in Muta3. Using four processors, we obtain a 2.1× and 2.4× improvement on Muta2 and Muta3, respectively. That means that over 80% of POTMiner execution time is effectively parallelized.

LightPOTMiner is more CPU-bound than POTMiner, while POTMiner is more I/O-bound than LightPOTMiner. The parallel implementation of LightPOTMiner obtains similar results with respect its sequential implementation. The performance speed-up obtained by the parallel implementation of LightPOTMiner is shown in Fig. 24.

Finally, Fig. 25 compares the parallel implementations of POTMiner and LightPOTMiner using four processors in parallel. As in their sequential implementations, LightPOTMiner requires the additional CPU time introduced by the 2^{MaxSize} factor analyzed in Sect. 7.3.

9 Conclusions and future work

There are many different tree mining algorithms that work either on ordered or unordered trees, but none of them, to our knowledge, works with partially-ordered trees, that is, trees that have both ordered and unordered sets of sibling nodes. We have devised a new algorithm to address this situation that is as efficient and scalable as existing algorithms that exclusively work on either ordered or unordered trees.

Partially-ordered trees are important because they appear in different application domains. In the future, we expect to apply our tree mining algorithm to some of these domains. In particular, we believe that our algorithm for identifying frequent subtrees in partially-ordered trees can be useful in different applications:

- XML documents [18], due to their hierarchical structure, are directly amenable to tree mining techniques. Since XML documents can contain both ordered and unordered sets of nodes, partially-ordered trees provide a better representation model for them and POT-Miner is, therefore, better suited for mining them than previous tree mining techniques.
- In Software Engineering, it is usually acknowledged that mining the wealth of information stored in software repositories can "support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques" [14]. For instance, there are hierarchical program representations, such as dependence higraphs [6], that can be viewed as partially-ordered trees, hence the potential of tree mining techniques in software mining.
- Multi-relational data mining [13] is another emerging research area where tree mining techniques can be useful. Algorithms such as POTMiner can help improve existing multi-relational classification [33] and clustering [34] algorithms.

We also plan to extend POTMiner to deal with partial or approximate tree matching, a feature that would be invaluable in many real-world problems, from entity resolution in XML documents to program element matching in software mining.

Acknowledgment This work was partially supported by research project TIN2006-07262.

References

1. Abe K et al. (2002) Efficient substructure discovery from large semi-structured data. In: Proceedings of the 2nd SIAM international conference on data mining
2. Agarwal RC et al (2001) A tree projection algorithm for generation of frequent item sets. J Parallel Distrib Comput 61(3):350–371
3. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8:962–969
4. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of 20th international conference on very large data bases, 12–15 September, pp 487–499
5. Asai T et al (2003) Discovering frequent substructures in large unordered trees. In: Discovery science. Lecture Notes in Artificial Intelligence, vol 2843. Springer, Berlin, pp 47–61
6. Berzal F et al (2007) Hierarchical program representation for program element matching. In: IDEAL'07. Lecture Notes in Computer Science, vol 4881, pp 467–476
7. Bringmann B (2006) To see the wood for the trees: mining frequent tree patterns. In: Constraint-based mining and inductive databases, European workshop on inductive databases and constraint based mining. 11–13 March 2004, Hinterzarten, Germany. Revised Selected Papers. Lecture Notes in Computer Science, vol 3848. Springer, Berlin, pp 38–63
8. Cheung DW-L et al (1996) Efficient mining of association rules in distributed databases. IEEE Trans Knowl Data Eng 8(6):911–922
9. Chi Y et al (2005a) Frequent subtree mining—an overview. Fundam Inform 66(1–2):161–198

10. Chi Y et al (2005b) Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Trans Knowl Data Eng 17(2):190–202
11. Chi Y et al (2004) HybridTreeMiner: an efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: The 16th international conference on scientific and statistical database management, pp 11–20
12. Chi Y et al (2005c) Canonical forms for labelled trees and their applications in frequent subtree mining. Knowl Inform Syst 8(2):203–234
13. Džeroski S (2003) Multi-relational data mining: an introduction. SIGKDD Explor Newsl 5(1):1–16
14. Gall H et al (2007) 4th international workshop on mining software repositories (MSR 2007). In: ICSE COMPANION '07, pp 107–108
15. Hadzic F et al (2007) UNI3—efficient algorithm for mining unordered induced subtrees using TMG candidate generation. In: Computational intelligence and data mining, pp 568–575
16. Han J et al (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp 1–12
17. Hido S, Kawano H (2005) AMIOT: induced ordered tree mining in tree-structured databases. In: Proceedings of the 5th IEEE international conference on data mining, pp 170–177
18. Nayak R et al (2006) Knowledge discovery from XML documents. Lecture Notes in Computer Science, vol 3915. Springer, Berlin
19. Nijssen S, Kok JN (2003) Efficient discovery of frequent unordered trees. In: First international workshop on mining graphs, trees and sequences (MGTS2003), in conjunction with ECML/PKDD'03, pp 55–64
20. Nijssen S, Kok JN (2004) A quickstart in frequent structure mining can make a difference. In: Proceedings of the 10th ACM SIGKDD international conference on knowledge discovery and data mining, pp 647–652
21. Parthasarathy S et al (2001) Parallel data mining for association rules on shared-memory systems. Knowl Inform Syst 3(1):1–29
22. Rückert U, Kramer S (2004) Frequent free tree discovery in graph data. In: Proceedings of the 2004 ACM symposium on applied computing, pp 564–570
23. Schuster A et al (2005) A high-performance distributed algorithm for mining association rules. Knowl Inform Syst 7(4):458–475
24. Shen L et al (1999) New algorithms for efficient mining of association rules. Inform Sci 118(1–4):251–268
25. Tan H et al (2005a) X3-Miner: mining patterns from an XML database. In: The 6th international conference on data mining, text mining and their business applications. May 2005, Skiathos, Greece, pp 287–296
26. Tan H et al (2005b) MB3-Miner: mining eMBedded subTREEs using tree model guided candidate generation. In: Proceedings of the first international workshop on mining complex data, pp 103–110
27. Tan H et al (2006) IMB3-Miner: mining induced/embedded subtrees by constraining the level of embedding. In: Proceedings of the 10th Pacific-Asia conference on knowledge discovery and data mining, pp 450–461
28. Tatikonda S et al (2006) TRIPS and TIDES: new algorithms for tree mining. In: Proceedings of the 15th ACM international conference on information and knowledge management, pp 455–464
29. Termier A et al (2002) TreeFinder: a first step towards XML data mining. In: Proceedings of the 2nd IEEE international conference on data mining, pp 450–457
30. Termier A et al (2004) DRYADE: a new approach for discovering closed frequent trees in heterogeneous tree databases. In: Proceedings of the 4th IEEE international conference on data mining, pp 543–546
31. Wang C et al (2004) Efficient pattern-growth methods for frequent tree pattern mining. In: Proceedings of the 8th Pacific-Asia conference on knowledge discovery and data mining. Lecture Notes in Computer Science, vol 3056. Springer, Berlin, pp 441–451
32. Xiao Y et al (2003) Efficient data mining for maximal frequent subtrees. In: Proceedings of the 3rd IEEE international conference on data mining, pp 379–386
33. Yin X et al (2004) CrossMine: efficient classification across multiple database relations. In: International conference on data engineering, pp 399–410
34. Yin X et al (2005) Cross-relational clustering with user's guidance. In: Knowledge discovery and data mining, pp 344–353
35. Zaki MJ (2005a) Efficiently mining frequent embedded unordered trees. Fundam Inform 66(1–2):33–52
36. Zaki MJ (2005b) Efficiently mining frequent trees in a forest: algorithms and applications. IEEE Trans Knowl Data Eng 17(8):1021–1035
37. Zhang S, Wang JTL (2008) Discovering frequent agreement subtrees from phylogenetic data. IEEE Trans Knowl Data Eng 20(1):68–82

Author Biographies

Aída Jiménez is a researcher at the Intelligent Databases and Information Systems research group in the Department of Computer Science and Artificial Intelligence at the University of Granada, Spain. She received a B.E degree in Computer Engineering from the University of Granada in 2006 and a M.Sc. degree in Soft Computing and Artificial Intelligence from the University of Granada in 2008. Her research interests include database design and data mining.

Fernando Berzal is an associate professor in the Department of Computer Science and Artificial Intelligence at the University of Granada. Previously, he had been a visiting research scientist at the data mining research group led by Jiawei Han at the University of Illinois at Urbana-Champaign. His research interests include model-driven software development, software design, and the application of data mining techniques to software engineering problems. He received his PhD in Computer Science from the University of Granada in 2002 and he was awarded the Computer Science Studies National First Prize by the Spanish Ministry of Education in 2000. He is a senior member of the ACM and also a member of the IEEE Computer Society.

Juan-Carlos Cubero is a full professor in the Department of Computer Science and Artificial Intelligence at the University of Granada. He received his PhD in Computer Science from the University of Granada in 1994. He has lectured in several European Universities and he has published several books in Computer Science and more than 30 papers in JCR journals. His research interests include database design, data mining, and software modeling. He has served as PC member in about 50 international conferences and he has actively participated in the organization of different conferences and workshops. He has also been the leader of a Spanish consortium participating in a European Eureka project.

1.3 POTMiner improvements

The journal paper associated to this part of the dissertation is:

- A. Jiménez, F. Berzal, and JC Cubero, Mining Frequent Patterns from XML Data: Efficient Algorithms and Design Trade-Offs. **Submitted to Expert Systems with Applications**.
  - Status: **Submitted**.
  - Impact Factor (JCR 2009): 2.908.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 15 / 103.
    - Engineering, electrical & electronic. Ranking 16 / 246.
    - Operations research & management science. Ranking 3 / 73.

Manuscript Number:

Title: Mining Frequent Patterns from XML Data: Efficient Algorithms and Design Trade-Offs

Article Type: Full Length Article

Keywords: XML documents; data mining; frequent patterns;
partially-ordered trees; induced and embedded subtrees

Corresponding Author: Dr. Fernando Berzal, Ph.D.

Corresponding Author's Institution: University of Granada

First Author: Aida Jimenez

Order of Authors: Aida Jimenez; Fernando Berzal, Ph.D.; Juan Carlos Cubero, Ph.D.

Mining Frequent Patterns from XML Data: Efficient Algorithms and Design Trade-Offs

Aída Jiménez, Fernando Berzal, Juan-Carlos Cubero

Dept. Computer Science and Artificial Intelligence
ETSIIT - University of Granada, 18071 Granada, Spain
E-mail: aidajm@decsai.ugr.es, fberzal@decsai.ugr.es,jc.cubero@decsai.ugr.es

Mining Frequent Patterns from XML Data: Efficient Algorithms and Design Trade-Offs

Aída Jiménez, Fernando Berzal, Juan-Carlos Cubero

Dept. Computer Science and Artificial Intelligence
ETSIIT - University of Granada, 18071 Granada, Spain
E-mail: aidajm@decsai.ugr.es, fberzal@decsai.ugr.es,jc.cubero@decsai.ugr.es

Abstract

XML documents are now ubiquitous and their current applications are countless, from representing semi-structured documents to being the de facto standard for exchanging information. Viewed as partially-ordered trees, XML documents are amenable to efficient data mining techniques. In this paper, we describe how scalable algorithms can be used to mine frequent patterns from partially-ordered trees and discuss the trade-offs that are involved in the design of such algorithms.

Keywords: XML documents; data mining; frequent patterns; partially-ordered trees; induced and embedded subtrees;

1. Introduction

XML has become a popular format for the storage and exchange of data due to its flexible semi-structured nature, which allows the representation of a wide variety of databases as XML documents. XML data thus forms an important data mining domain [1].

Since XML documents are also text documents, a natural solution for XML data mining problems is the use of standard information retrieval techniques. Another approach for dealing with XML documents consists of just flattening their structure into a set, which allows the application of standard data mining algorithms. However, those approaches ignore a significant amount of structural information in the XML documents.

Structural XML data mining is a challenging problem. In the past few years, however, some novel techniques have been proposed for dealing with XML documents and their structure. Frequent discriminatory substructures within XML documents have been used to perform rule-based classification of XML data in XRules[1], an approach that outperforms standard associative classifiers. Substructures in XML documents have also been used in effective clustering algorithms for XML data [2]. In this case, the similarity within a cluster is defined in terms of the containment of particular frequent substructures which occur frequently in that particular segment of the data. These substructures are analogous to the projected dimensions used in subspace clustering.

In this paper, we will discuss the key problem of discovering the frequent patterns that later can be used to solve other problems in structural XML data mining. Apart from being at the heart of many interesting and challenging data mining problems, the discovery of frequent XML patterns is also useful in many other situations. For instance, the discovery of frequent XML query patterns in the history log of XML queries can be used to expedite XML query processing, as the answers to these queries can be cached and reused when the future queries "hit" such frequent patterns [3].

Discovering frequent tree patterns over tree-structured data has indeed become an important problem in data mining and many different tree pattern mining algorithms have been proposed [4]. In our paper, we will focus our discussion on POTMiner [5], an efficient parallel algorithm for mining partially-ordered trees. Such trees provide a general model for representing the structural information in XML documents. We will also analyze some design trade-offs that must be made in practice to deal with real-world data sets.

This paper is organized as follows. We introduce partially-ordered trees as well as some standard terms in Section 2. POTMiner is described in Section 3. Its parallelization is described in Section 4, while some trade-offs related to memory consumption are discussed in Section 5. Finally, we present some experimental results in Section 6 and our conclusions in Section 7.

2. Background

A **tree** is a connected and acyclic graph. A tree is rooted if its edges are directed and a special node, called root, can then be identified. The root is the node from which it is possible to reach all the other nodes in the tree. In contrast, a tree is said to be free if its edges have no direction, that is, when it is an undirected graph. A free tree, therefore, has no predefined root.

Rooted trees can be classified as **ordered trees**, when there is a predefined order within each set of sibling nodes in the tree, or **unordered trees**, when there is not such a predefined order among sibling nodes in the tree. **Partially-ordered trees** contain both ordered and unordered sets of sibling nodes within the same tree. Figure 1 depicts ordered, unordered, and partially-ordered trees (ordered sibling nodes are joined by an arc, while unordered sets of sibling nodes do not share such arc).

Figure 2 shows an XML document representing the purchase history of a particular customer and its corresponding partially-ordered tree. Partially-ordered trees are useful when the order within some sets of siblings is important but it is not necessary to establish an order relationship within all the sets of sibling nodes. In this case, taken from a typical scenario in market basket analysis, it is important to preserve the order among the different orders placed by the same customer, while the order among the items in the same order is irrelevant. In practice, XML schemata define the structure of XML documents and they determine which sets of sibling nodes are ordered and which ones are unordered.

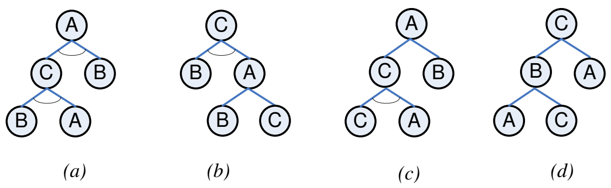


Figure 1: Different kinds of rooted trees (from left to right): *(a)* completely-ordered tree; *(b)* and *(c)*, partially-ordered trees; *(d)* completely-unordered tree.

```
<customer>
   <order>
      <item>milk</item>
      <item>bread</item>
   </order>
   <order>
      <item>beer</item>
      <item>diapers</item>
   </order>
</customer>
```

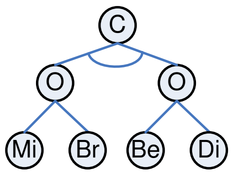


Figure 2: A simple XML document and its partially-ordered tree representation.

A subtree is a tree subgraph. Different kinds of subtrees can be defined depending on the way we define the matching function between the particular subgraph and the tree it derives from. A **bottom-up subtree** T' of T (with root v) can be obtained by taking one vertex v from T with all its descendants and their corresponding edges. An **induced subtree** T' can be obtained from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T. Finally, an **embedded subtree** T' can be obtained from a tree T by repeatedly removing nodes, provided that ancestor relationships among the vertices of T are not broken. Figure 3 shows a tree *(a)* wherein we have identified a bottom-up subtree *(b)*, an induced subtree *(c)* and an embedded subtree *(d)*.

3. Mining Partially-Ordered Trees

Tree mining algorithms cannot be directly applied to partially-ordered trees because they typically work either on completely-ordered or on completely-unordered trees. We have extended Zaki's TreeMiner [6] and SLEUTH [7] algorithms to mine partially-ordered trees. The algorithm we have devised is

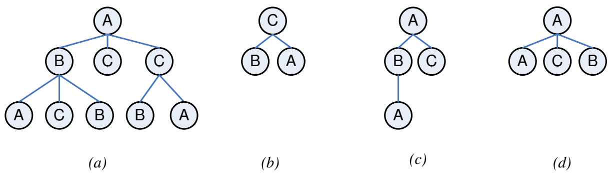


Figure 3: Different kinds of subtrees (from left to right): *(a)* original tree, *(b)* bottom-up subtree, *(c)* induced subtree, *(d)* embedded subtree.

```
**algorithm** *POTMiner*

Obtain frequent nodes (frequent patterns of size 1)
Build candidate classes $C_1$ from the frequent nodes
**for** k=2 **to** MaxSize
    **for each** class $P \in C_{k-1}$
        **for** each element $p \in P$.
            Compute the frequency of $p$
            **if** $p$ is frequent
            **then**
                Create a new class $P'$ from $p$.
                Add $P'$ **to** $C_k$
```

Figure 4: POTMiner algorithm pseudocode.

able to identify frequent subtrees, both induced and embedded, in ordered, unordered, and partially-ordered trees. Hence its name, POTMiner, which stands for *Partially-Ordered Tree Miner* [5].

Our algorithm is based on Apriori [8], just like TreeMiner and SLEUTH. Therefore, it follows an iterative pattern mining strategy. The k-th iteration of the algorithm mines the frequent subtrees of size k starting from the frequent subtrees of size $k-1$ discovered in the previous iteration. Each iteration, as in Apriori, is subdivided in two phases, candidate generation and support counting.

The pseudocode of the resulting algorithm is shown in Figure 4. The details of the candidate generation and support counting phases will be described in the following sections.

3.1. Candidate Generation for Partially-Ordered Trees

POTMiner employs Zaki's class-based extension method to generate candidates, one of the available strategies for enumerating candidate trees in tree pattern mining algorithms [4]. This method generates $(k+1)$-subtree candidates by joining two frequent k-subtrees with $k-1$ elements in common. Candidates are grouped into equivalence classes, each class containing all the trees that share the same prefix in their codification string.

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

TreeMiner [6] and SLEUTH [7] use the class-based extension method to mine ordered and unordered embedded subtrees, respectively. The main difference between TreeMiner and SLEUTH is that only those extensions that produce canonical subtrees are allowed in SLEUTH, which works with unordered trees. This constraint avoids the duplicate generation of candidates corresponding to different representations of the same unordered trees. However, since we must handle both ordered and unordered sets of sibling nodes in partially-ordered trees, all extensions must be allowed in POTMiner, as in TreeMiner.

Let (x, i) and (y, j) denote two elements in the same class $[P]$, and $[P^i_x]$ be the set of candidate trees derived from the tree that is obtained by adding the element (x, i) to P. Elements in the same equivalence class are joined to generate new candidates. This join procedure must consider two scenarios [7]: cousin extension and child extension.

3.1.1. Cousin extension

Cousin extension is performed when the father of (y, j) precedes (or is) the father of the element (x, i) in preorder, where the father of (y, j) is the node in position j and the father of (x, i) is in position i. Formally, **if** $j \leq i$ **and** $|P| = k - 1 \geq 1$, **then** $(y, j) \in [P^i_x]$. The first condition checks that only nodes on the rightmost branch of P are extended. The second one makes cousin extension possible only when patterns have more than one node.

3.1.2. Child extension

When (y, j) is a sibling node of (x, i), the child extension mechanism lets us add y as a child of the rightmost leaf of the tree. The formal definition of child extension is: **if** $j = i$ **then** $(y, k - 1) \in [P^i_x]$.

Child extension is used to make tree patterns grow in depth, while cousin extension lets them grow in width.

3.2. Support Counting for Induced and Embedded Subtrees

Once POTMiner has generated the potentially frequent candidates, it is necessary to determine which ones are actually frequent by counting the support of the potentially frequent candidates. The support counting phase in POTMiner follows the strategy of AprioriTID [8]. Instead of checking if each candidate is present in each database tree, which is a costly operation when dealing with trees, occurrence lists are used to preserve the occurrences of each pattern in the database, thus facilitating the support counting phase. Checking whether a pattern is frequent, therefore, consists of counting the elements in its occurrence list.

Several kinds of occurrence lists have been proposed in the tree mining literature: Standard occurrence lists [9] preserve the identifiers of the trees as well as the matching between the pattern nodes and the database tree nodes using a breadth-first codification scheme. Rightmost occurrence lists [10] only need to store the occurrences of the rightmost leaf of X in each database tree. Scope lists [7] preserve each occurrence of a pattern X in the database using a tuple which contains the scope of the last node of the pattern. The scope of a node is defined as a pair $[l,u]$ where l is the position of the node in depth-first order and u is the position of its rightmost descendant.

POTMiner employs scope lists to preserve the occurrences of a pattern in the tree database. The main difference between our scope lists and the ones proposed by Zaki in TreeMiner [6] and SLEUTH [7] is that we must add two

In-scope join procedure:
if

1. $t_x = t_y = t$ **and**
2. $m_x = m_y = m$ **and**
3. $d_x = 1$ when we are looking for induced patterns, **and**
4. $s_y \subset s_x$ (i.e., $l_x < l_y$ and $u_x \geq u_y$),

then add [t, m $\bigcup$ $\{l_x\}$, s_y, $d_y - d_x$, Θ_y] to the scope list of the generated candidate.

Out-scope join procedure:
if

1. $t_x = t_y = t$ **and**
2. $m_x = m_y = m$ **and**
3. **if** the node is ordered **and** $s_x < s_y$ (i.e. $u_x < l_y$) **or** the node is unordered **and** either $s_x < s_y$ or $s_y < s_x$ (i.e. either $u_x < l_y$ or $u_y < l_x$),

then add [t, m $\bigcup$ $\{l_x\}$, s_y ,d_y,Θ_y] to the scope list of the generated candidate.

Figure 5: Scope list join procedures in POTMiner.

new elements to the tuples in the scope lists, d and Θ, in order to deal with induced subtrees and partially-ordered trees, respectively.

Our scope lists, therefore, contain tuples (t, m, s, d, Θ) where t is the tree identifier, m stores which nodes of the tree match those of the (k-1) prefix of the pattern X in depth-first order, s=[l,u] is the scope of the last node in the pattern X, d is a depth-based parameter used for mining induced subtrees (not needed when mining embedded subtrees), and Θ indicates whether the last node of the pattern is ordered (Θ=o) or unordered (Θ=u) in the database tree.

When building the scope lists for patterns of size 1, m is empty and the element d is initialized with the depth of the pattern only node in the original database tree.

We obtain the scope list for a new candidate of size k by joining the scope lists of the two subtrees of size $k-1$ that were involved in the generation of the candidate. Let (t_x, m_x, s_x, d_x, Θ_x) and (t_y, m_y, s_y, d_y, Θ_y) be the scope lists of the subtrees involved in the generation of the candidate. The scope list for this candidate is built by a join operation that depends on the candidate extension method used to generate the candidate. The in-scope join procedure is used in conjunction with the child extension mechanism, while the out-scope join procedure is used in conjunction with the cousin extension mechanism (see Figure 5). Further details can be found in [5].

3.3. POTMiner Complexity

POTMiner starts by computing the frequent patterns of size 1. This step is performed by obtaining the vertical representation of the tree database, i.e., the individual nodes that appear in the trees with their occurrences represented as scope lists. This representation is obtained in linear time with respect to the number of trees in the database by scanning it and building the scope lists for patterns of size 1. We then discard the patterns of size 1 that are not frequent. This results in L scope lists corresponding to the L frequent labels in the tree database and each frequent label leads to a candidate class of size 1.

Let $c(k)$ be the number of classes of size k, which equals the number of frequent patterns of size k, and $e(k)$ the number of elements that might belong to a particular class of size k (i.e., the number of patterns of size $k+1$ that might be included in the class corresponding to a given pattern of size k).

In POTMiner, each tree pattern grows only by adding a node as a child to a node in its rightmost path. In the worst case, when the tree is just a sequence of size k, the number of different trees of size $k+1$ that can be obtained by the extension of the tree of size k is $L * k$. Hence, the number of elements in a particular class, $e(k)$, is $O(L * k)$.

The number of classes of size 1 equals L, the number of frequent labels, i.e. $c(1) = L$. The classes of size $k+1$ are derived from the frequent elements in classes of size k. In the worst case, when all the $e(k)$ elements are frequent, $c(k+1) = c(k) * e(k)$. Solving the recurrence, we obtain $c(k+1) = c(k) * e(k) = O(L^{k+1} * k!)$, which can also be expressed as $c(k) = O(L^k * (k-1)!)$.

For each pattern considered of size $k+1$, POTMiner must perform a join operation to obtain its scope list from the scope lists of the two patterns of size k that led to it.

The size of the scope list for a pattern of size k is $O(t * e)$ while the cost of a scope-list join is $O(t * e^2)$, where t is the number of trees in the database and e is the average number of embeddings of the pattern in each tree [7].

In the worst case, when we are looking for embedded subtrees, the number of embeddings of a pattern of size $k-1$ in a tree of size n equals the number of subtrees of size $k-1$ within the tree of size n. This number, $s(k-1)$, is bounded by $\binom{n}{k-1} \leq n^k/(k-1)!$.

Hence, the cost of the join operation needed for obtaining the scope list of a pattern of size k, is $j(k) = O(t * s(k-1)^2) = O(t * (n^k/(k-1)!)^2)$.

The cost of obtaining all the frequent patterns of size k will be, therefore, $O(c(k) * j(k)) = O(t * (Ln^2)^k/(k-1)!)$.

The total cost of executing the POTMiner algorithm to obtain all the frequent patterns up to k=MaxSize is $\sum_{k=1...MaxSize}(t*(Ln^2)^k/(k-1)!)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e. when k=MaxSize), POTMiner is $O(t*(Ln^2)^{MaxSize}/(MaxSize-1)!)$.

Therefore, our algorithm is linear with respect to the number of trees in the tree database and its execution time is also proportional to the number of patterns considered.

4. Making POTMiner faster: Parallelization

Parallelism can be used to improve the performance of data mining algorithms. The parallel implementation of POTMiner lets us exploit the archi-

```
algorithm ParallelPOTMiner

Obtain frequent nodes (frequent patterns of size 1)
Build candidate classes $C_1$ from the frequent nodes
for k=2 to MaxSize
    for each class $P \in C_{k-1}$
        Extend $P$ in parallel to obtain $P_{extended}$
    for each class $P \in C_{k-1}$
        $C_k = C_k \bigcup P_{extended}$
```

Figure 6: POTMiner algorithm parallelization.

tecture of modern multi-core processors and multiprocessors. In fact, many different parallel and distributed frequent pattern mining algorithms have been proposed in the association rule mining literature, e.g. [11] [12] [13] [14] [15].

POTMiner can also be parallelized using a candidate distribution approach [11]. The idea behind the parallel version of POTMiner is that, in each step of the algorithm, the extension of each candidate class and the corresponding scope list join operations can be performed in parallel. Since these operations are independent from each other, each one can be assigned to a different processor without incurring into significant coordination costs. Therefore, the parallel generation of candidate patterns simply involves partitioning the set of all candidate classes to be extended among the available processors.

The parallel version of POTMiner independently processes each class of size k to obtain candidates of size $k+1$ and, at the end of each iteration, the results of the independently-processed classes are combined to return all the candidate classes of size $k+1$. The pseudocode of this parallel version of POTMiner is shown in Figure 6.

5. Making POTMiner leaner: On-demand scope lists

The candidate generation process is very memory consuming due to the huge amount of scope lists that have to be maintained. Moreover, the size of each scope lists is proportional to the number of embedded occurrences of each pattern, which can also be huge in large databases.

We have devised two variants of the POTMiner algorithm that make different memory/CPU-time trade-offs to compute scope lists on demand, instead of keeping all of them in memory, which is impractical in many situations.

5.1. POTMiner Light

POTMiner builds the scope lists of a new candidate by joining the scope lists of the patterns involved in its generation. POTMiner Light recursively computes such scope lists directly from the scope lists of the frequent nodes, i.e., the tree patterns of size 1.

```
algorithm scopeList (Tree $t$): $s$
// $t : n_1, n_2..n_{k-1}, n_k$

if $k = 1$
then
    $s$ = scope list of node $n_k$
else
    $t_1 = t - n_k$
    $t_2 = t - n_{k-1}$
    $s_1 = scopeList(t_1)$
    $s_2 = scopeList(t_2)$
    if $n_k$.parent $= n_{k-1}$
    then // $t$ was obtained by child extension
        s = in-scope-join ($s_1$, $s_2$)
    else // $t$ was obtained by cousin extension
        s = out-scope-join ($s_1$, $s_2$)
```

Figure 7: Recursive construction of scope lists in POTMiner Light.

The key idea of these recursive process is that it is always possible to know if a given pattern was obtained by cousin extension or by child extension. Hence, we can infer which subtrees were used to generate the tree pattern and which join operation to perform on the corresponding scope lists, i.e. in-scope or out-scope join. The recursive algorithm employed by POTMiner Light is shown in Figure 7.

In POTMiner Light, therefore, scope lists are calculated on demand. The scope list for a pattern of size k is obtained by joining two scope lists of patterns of size $k-1$. Formally, the number of join operations needed to obtain a scope list for a pattern of size k is given by the following expression: $join(k) = 1 + 2 * join(k-1)$. Given that $join(2) = 1$, then $join(k) = 2^{k-1} - 1$. The cost of computing a scope list in POTMiner Light is, therefore, $j_{light}(k) = (2^{k-1} - 1) * j(k) \in O(2^k * t * (n^k/(k-1)!)^2)$.

The cost of obtaining all the frequent patterns of size k in POTMiner Light will be proportional to $O(c(k) * j_{light}(k)) = O(L^k * (k-1)! * 2^k * t * (n^k/(k-1)!)^2) = O((2L)^k * t * n^{2k}/(k-1)!) = O(t * (2Ln^2)^k/(k-1)!)$.

The total cost of executing the POTMiner Light algorithm to obtain all the frequent patterns up to $k = MaxSize$ is $\sum_{k=1...MaxSize}(t * (2Ln^2)^k/(k-1)!)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e. k=MaxSize), POTMiner Light is $O(t * (2Ln^2)^{MaxSize}$ / $(MaxSize - 1)!)$.

In other words, we introduce a $2^{MaxSize-1} - 1$ factor in the execution time of POTMiner Light to reduce memory consumption. POTMiner Light just needs to store L scope lists corresponding to the frequent patterns of size 1, while POTMiner had to store all the scope lists in memory, up to $c(MaxSize - 1)$, the number of frequent patterns of size $MaxSize - 1$ we might obtain, which is

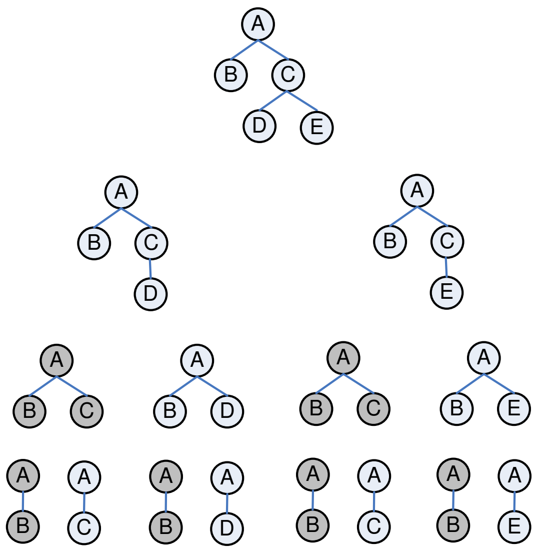


Figure 8: Recursive scope-list generation in POTMiner Light.

$L^{MaxSize-1} * (MaxSize - 2)!$ in the worst case.

5.2. POTMiner DP

The recursive on-demand computation of scope-lists in POTMiner Light repeats the generation of some of these scope-list. For patterns of size $k = 5$, the scope list of a candidate of size 3 is computed twice and the scope list of a candidate of size 2 is computed four times. Figure 8 shows an example of this time-consuming generation process. In general, for patterns of size k, there is a pattern of size $k - s$ (with $s \geq 2$) whose scope list is computed 2^{s-1} times using POTMiner Light recursive algorithm.

We can use dynamic programming in order to avoid the repeated computation of the same scope lists, which leads us to a new variant of our tree pattern mining algorithm: POTMiner DP. Our dynamic programming algorithms uses a bottom-up strategy to build scope lists and, by memoizing the scope lists that are recomputed by POTMiner Light, it trades CPU time for memory space (the space needed to store the scope lists that will be reused). The leftmost trees in Figure 8 are the ones whose scope lists are recomputed by POTMiner Light and they are the trees whose scope lists we will be kept in memory by POTMiner DP.

It should be noted, however, that we cannot know whether we should perform an in-scope or an out-scope join when building the scope lists bottom-up (that is, if we employed dynamic programming directly). For this reason, we have to decompose the candidate pattern in its constituent subpatterns before we compute its scope list. By keeping these subpatterns in memory, we can check

```
algorithm scopeListDP (Tree $t$): $s$
// $t : n_1, n_2..n_{k-1}, n_k$

for i=1 to k
    $list[i]$ = scope list of node $n_i$
    for j=1 to i-1
        if j=1
        then // pattern of size 2
            list[i] = in-scope-join ($list[j]$, $list[i]$)
        else
            $s$ = subtree[j+1][i] // $s : s_1, s_2..s_j, s_{j+1}$
            if $s_{j+1}$.parent = $s_j$
            then
                list[i] = in-scope-join($list[j]$, $list[i]$)
            else
                list[i] = out-scope-join ($list[j]$, $list[i]$)
return list[k]
```

Figure 9: Scope list generation in POTMiner DP.

whether the candidate grew by cousin extension or by child extension in $O(1)$, which is the information we need to decide between in-scope and out-scope join.

Therefore, the scope list computation for each pattern P in POTMiner DP will consist of the following two steps:

- Top-down subtree decomposition: The tree pattern is decomposed into its constituent subtrees, which are saved into a matrix that stores the subtree of size k whose last node is the n-th node in the P tree in $subtree[k][n]$.
- Bottom-up scope-list generation using dynamic programming: The scope list of a given pattern of size k is iteratively computed by reusing the scope lists of its leftmost subtrees, as shown in Figure 9.

Figure 10 shows the order in which operations are performed by POTMiner DP to build the scope list of a pattern of size 5 (the same pattern that was used in Figure 8 to illustrate the redundant computations in POTMiner Light). During the subtree generation step, subtree are generated from right to left and top-down following the hollow arrows. During the scope-list generation step, scope lists are generated from left to right and bottom-up following the solid arrows. It should be taken into account that only the scope lists of the patterns in the top row must be kept in memory while computing the scope list corresponding to the 5-node tree pattern shown at the top right corner.

In POTMiner DP, the number of join operations needed to obtain a scope list for a pattern of size k is given by the following expression: $join(k) = k*(k-1)/2$. The cost of computing a scope list in POTMiner PD is $j_{DP}(k) = (k*(k-1)/2)*j(k) = O(k^2 * t * (n^k/(k-1)!)^2)$.

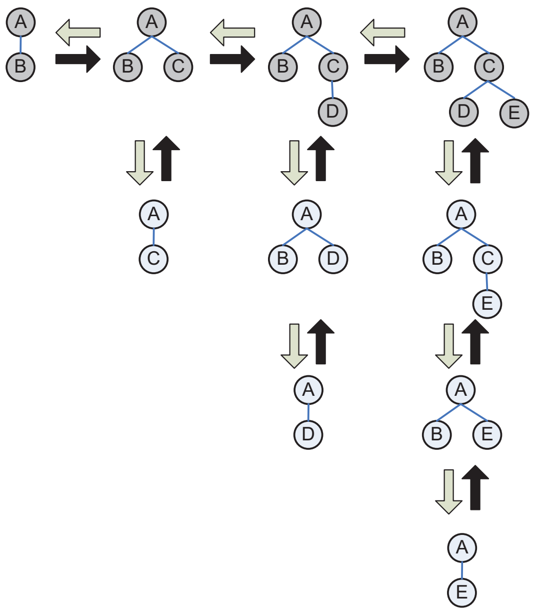


Figure 10: POTMiner DP computation of scope lists.

The cost of obtaining all the frequent patterns of size k in POTMiner DP will be, therefore, $O(c(k) * j_{DP}(k)) = O(t * k^2 * (Ln^2)^k/(k-1)!)$.

The total cost of executing the POTMiner DP algorithm to obtain all the frequent patterns up to $k = MaxSize$ is $\sum_{k=1...MaxSize}(t*k^2*(Ln^2)^k/(k-1)!)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e. $k = MaxSize$), POTMiner DP is $O(t * MaxSize^2 * (Ln^2)^k/(k-1)!)$.

In other words, we introduce a $(MaxSize * (MaxSize - 1)/2)$ factor in the execution time of our original POTMiner algorithm. It should be noted that this factor is much lower than the factor introduced by POTMiner Light $(2^{MaxSize-1} - 1)$, specially for large values of $MaxSize$. On the other hand, POTMiner DP needs to store L scope lists corresponding to the frequent patterns of size 1, as POTMiner Light, plus $k - 1$ ancillary list to speed up scope list computation using the dynamic programming algorithm described above.

6. Experimental results

We have performed two series of experiments to evaluate the different versions of POTMiner. First, we have tested our algorithms using a synthetic dataset to study their scalability. Second, we have devised some experiments with a real-world dataset in to test their behavior in practice.

All the experiments described in this section have been performed on an Intel 2.66GHz Q6700 4-core processor with 4GB of main memory running on

Table 1: Number of frequent induced patterns found in the synthetic dataset using a 5% minimum support threshold.

Pattern size	2	3	4	5	6	7
Number of patterns	24	65	123	174	201	209

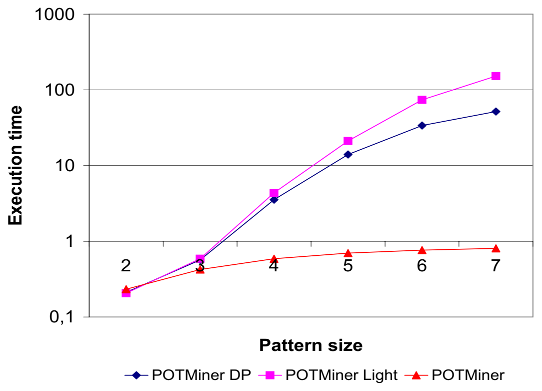


Figure 11: Comparison of POTMiner, POTMiner Light, and POTMiner DP execution times on the synthetic dataset for different pattern sizes.

Windows Vista. The three variants of POTMiner have been implemented in Java using Sun Microsystems JDK 6.

6.1. Experiments using a synthetic dataset

The experiments in this section were performed with a synthetic dataset generated by the tree generator available at *http://www.cs.rpi.edu/∼zaki/www-new/pmwiki.php/Software/Software#trees*. The 10000-tree dataset was obtained using the generator default values. The labeled trees in this dataset contain 10 different labels, their maximum depth is 5, and their nodes maximum fanout is also 5. The trees of this dataset have are 54686 nodes in total. Table 1 shows the numbers of frequent induced patterns that can be identified within this synthetic dataset using a 5% support threshold.

Figure 11 shows the performance of the different POTMiner variants on the synthetic dataset when identifying patterns from size 2 to 7 using a parallel implementation of the three variants that makes use of 4 parallel threads to take advantage of the 4-core processor the experiments have been performed on. Obviously, POTMiner is much faster that its memory-efficient variants, while POTMiner DP performance is better than its POTMiner Light counterpart.

6.2. Experiments using a real-world database

We have also performed some experiments to study the three POTMiner variants in a real-world scenario. The dataset we used in these experiments was

Table 2: Number of frequent induced patterns found in the Mutagenesis dataset using a 20% minimum support threshold.

Pattern size	2	3	4	5
Number of patters	82	857	9136	74922

obtained come from the Mutagenesis database, a multirelational database that is frequently used as an ILP benchmark.

The Mutagenesis database contains four connected relations that we can use to build a tree database. A tree is obtained from each tuple in its target relation. Next, the links from one relation to another within the multirelational database (i.e. the foreign keys in relational database parlance) let us grow the different tree branches.

The dataset used in our experiments was built by following 2 links between the Mutagenesis relations (*mole-molatm* and *moleatm-atom*). The resulting dataset contains 188 trees with 138 nodes per tree on average (i.e. a total of 25969 nodes). Table 2 shows the number of frequent patterns that were identified within the Mutagenesis dataset using a 20% of minimum support threshold for different pattern sizes. As can be seen in Table 2, the number of frequent patterns in a real-world database pattern is typically much larger than the number of frequent patterns in synthetic datasets (Table 1).

The large number of patterns makes POTMiner impractical, since memory is exhausted before all frequent itemsets are identified. In our experiments, POTMiner ran out of memory when discovering patterns of size 4 within the Mutagenesis database. While requiring more CPU time, both POTMiner Light and POTMiner DP are able to mine this database without problems.

Figure 12 shows the execution time corresponding to the different POTMiner variants when identifying frequent induced patterns within the Mutagenesis dataset. POTMiner DP execution time is slightly better than POTMiner Light execution time, and this difference is expected to grow for larger patterns. The original version of POTMiner is just unable to discover patterns of size 4 and 5 in this dataset because it quickly ran out of memory.

7. Conclusions

XML documents are amenable to efficient data mining when viewed as partially-ordered labeled trees. Many tree pattern mining algorithms have been proposed in the literature, but not all of them are useful in practice. Even though they might be scalable from a theoretical point of view and they work when dealing with synthetic datasets, their huge memory requirements when applied in more realistic settings make them impractical.

This paper has described how an efficient, scalable, and parallelizable tree pattern mining algorithm can be adjusted to reduce its memory requirements. The variants described here trade memory space for CPU time in order to surpass the limitations of a memory-bound algorithm such as POTMiner. The

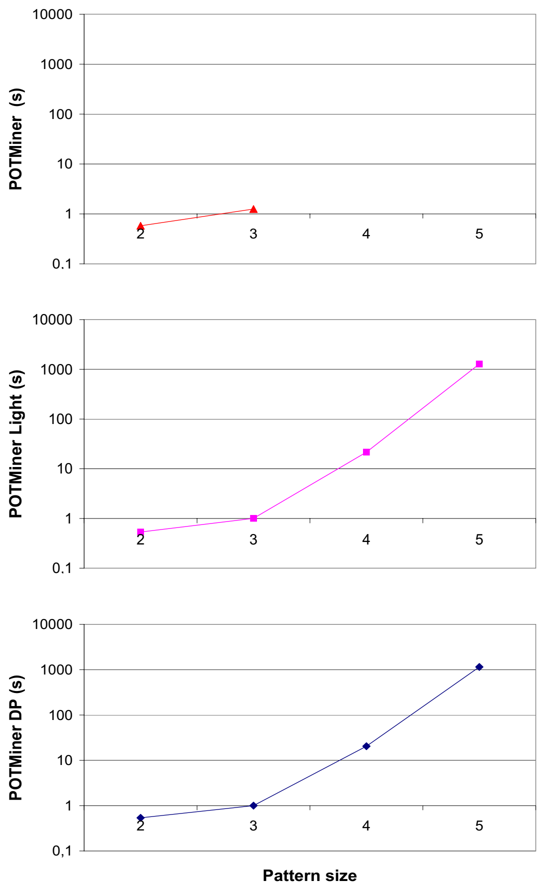


Figure 12: POTMiner (top), POTMiner Light (middle), and POTMiner DP (bottom) execution times on the Mutagenesis dataset for different pattern sizes.

recursive computation of occurrence lists in POTMiner Light provides an alternative to keeping all those lists in memory, while the dynamic programming approach provided by POTMiner DP renders another solution to the same problem that is asymptotically better when we are interested in discovering large frequent patterns in a tree database such as a set of XML documents.

In the future, we intend to apply our pattern mining algorithm on partially-ordered trees in different application domain, which will probably require us to push constraints into the tree pattern mining process in order to make the solution of real-world problems feasible.

Acknowledgments

This work was partially supported by the Spanish research project TIN2009-08296.

References

[1] M. J. Zaki, C. C. Aggarwal, XRules: An effective algorithm for structural classification of XML data, Machine Learning 62 (1-2) (2006) 137–170.

[2] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, M. J. Zaki, Xproj: a framework for projected structural clustering of XML documents, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 12-15, 2007, pp. 46–55.

[3] Y. Bei, G. Chen, L. Shou, X. Li, J. Dong, Bottom-up discovery of frequent rooted unordered subtrees, Information Sciences 179 (1-2) (2009) 70–88. doi:http://dx.doi.org/10.1016/j.ins.2008.08.020.

[4] A. Jiménez, F. Berzal, J. C. Cubero, Frequent tree pattern mining: A survey, Intelligent Data Analysis 14 (6), scheduled for publication.

[5] A. Jimenez, F. Berzal, J. C. Cubero, POTMiner: Mining ordered, unordered, and partially-ordered trees, Knowledge and Information Systems 23 (2010) 199–224.

[6] M. J. Zaki, Efficiently mining frequent trees in a forest: Algorithms and applications, IEEE Transactions on Knowledge and Data Engineering 17 (8) (2005) 1021–1035. doi:http://doi.ieeecomputersociety.org/10.1109/TKDE.2005.125.

[7] M. Zaki, Efficiently mining frequent embedded unordered trees, in: Fundamenta Informaticae, Vol. 66, IOS Press, Amsterdam, The Netherlands, 2005, pp. 33–52.

[8] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, pp. 487–499.

[9] Y. Chi, Y. Yang, R. R. Muntz, HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical form., in: The 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 11–20. doi:http://dx.doi.org/10.1109/SSDBM.2004.41.

[10] K. Abe, S. Kawasoe, T. Asai, H. Arimura, S. Arikawa, Efficient substructure discovery from large semi-structured data, in: Proceedings of the 2nd SIAM International Conference on Data Mining, 2002.

[11] R. Agrawal, J. C. Shafer, Parallel mining of association rules, IEEE Transactions on Knowledge and Data Engineering 8 (1996) 962–969.

[12] D. W.-L. Cheung, V. T. Y. Ng, A. W.-C. Fu, Y. Fu, Efficient mining of association rules in distributed databases, IEEE Transactions on Knowlegde and Data Engineering 8 (6) (1996) 911–922.

[13] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad, A tree projection algorithm for generation of frequent item sets, Journal of Parallel and Distributed Computing 61 (3) (2001) 350–371.

[14] S. Parthasarathy, M. J. Zaki, M. Ogihara, W. Li, Parallel data mining for association rules on shared-memory systems, Knowlegde and Information Systems 3 (1) (2001) 1–29.

[15] A. Schuster, R. Wolff, D. Trock, A high-performance distributed algorithm for mining association rules, Knowlegde and Information System 7 (4) (2005) 458–475.

2 Applications

2.1 Multirelational data mining

The journal paper associated to this part of the dissertation is:

- A. Jiménez, F. Berzal, and JC Cubero, Using trees to mine multirelational databases. **Submitted to Data Mining and Knowledge Discovery**
  - Status: **Submitted**.
  - Impact Factor (JCR 2009): 2.950.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 13 / 103.
    - Computer Science, Information Systems. Ranking 9 / 116.

Manuscript Number:

Title: Interestingness measures for association rules within groups

Article Type: Regular Paper

Keywords: Group association rules; interestingness measures

Corresponding Author: Fernando Berzal

Corresponding Author's Institution: University of Granada

First Author: Aida Jimenez

Order of Authors: Aida Jimenez;Fernando Berzal;Juan Carlos Cubero

Abstract: The study of association rules within groups of individuals in a database is interesting to analyze the characteristics and the behavior of such groups. In this paper, we define group association rules and we study interestingness measures for them. These
interestingness measures can be used to rank, not only groups of individuals, but also rules within each group. We also compare the rankings provided by those different interestingness measures in order to determine which one provides a better alternative depending on the kind of situations we wish to highlight within large databases with
many different (and overlapping) groups of individuals.

Under consideration for publication in Knowledge and Information Systems

Interestingness measures for association rules within groups

Aída Jiménez, Fernando Berzal and Juan-Carlos Cubero

Abstract.
The study of association rules within groups of individuals in a database is interesting to analyze the characteristics and the behavior of such groups. In this paper, we define group association rules and we study interestingness measures for them. These interestingness measures can be used to rank, not only groups of individuals, but also rules within each group. We also compare the rankings provided by those different interestingness measures in order to determine which one provides a better alternative depending on the kind of situations we wish to highlight within large databases with many different (and overlapping) groups of individuals.

Keywords: Group association rules; interestingness measures.

1. Introduction

The approach we propose in this paper intends to solve the second-order data mining problem that often arises in practice. Researchers in the data mining field have traditionally focused their efforts on obtaining fast and scalable algorithms in order to deal with huge amounts of data. When dealing with association rules, for instance, he overwhelming number of discovered rules, usually in the order of thousands or even millions, makes them of limited use in practice. The mere volume of these sets of rules causes the aforementioned second-order data mining problem (Berzal & Cubero, 2007).

Databases can naturally contain groups of individuals that share some characteristics (Plasse, et al., 2007). For example, within a census database, we can find many different groups of individuals (e.g. according to their sex, their marital status, whether they have children, or just by combining several of such features). Our proposal consists of automatically identifying potentially useful groups of related association rules and ranking the resulting group association rules so that expert users can more easily sift through vast amounts of association rules.

Association rules have been used to analyze co-occurrence relationships among frequent itemsets in transactional and relational databases. Let $I = \{I_1, I_2, ..., I_m\}$

be a set of items. Let D be a set of database transactions where each transaction T is a set of items such that $T \subseteq I$. Let S be a set of items. A transaction T is said to contain S if and only if $S \subseteq T$ (Han & Kamber, 2005). An **association rule** is an implication of the form $A \Rightarrow C$, where $A \subseteq I$, $C \subseteq I$, and $A \cap C = \emptyset$.

In this paper, we describe how association rules can be defined to study the features that individuals within a given group have in common. We define a **group** as a set of items $G = \{G_1, G_2, ..., G_n\}$ such that $G \subseteq I$. A **group association rule** $G : A \Rightarrow C$ is an association rule $A \Rightarrow C$ defined over the group G. In other words, a group association rule $G : A \Rightarrow C$ is equivalent to the classical association rule $GA \Rightarrow C$.

In this paper, we also adapt some of the interestingness measures that have been defined for association rules (Geng & Hamilton, 2006) (Berzal, et al., 2002) to group association rules and analyze how these tailored measures will help us rank the different groups in a database in order to highlight the most interesting ones. We also compare the group rankings obtained by different interestingness measures in order to study their differences.

Our paper is organized as follows. In Section 2, we describe some rule interestingness measures as proposed in the literature. Section 3 introduces interestingness measures for group association rules. In Section 4, we explain how to rank groups and group association rules within each group. Section 5 introduces criteria to compare alternative rankings. We analyze the experimental results we have obtained in Section 6. Finally, we end our paper with some conclusions in Section 7.

2. Interestingness measures for standard association rules

The classical measures used to characterize an association rule are its support and its confidence (Agrawal & Srikant, 1994)(Han & Kamber, 2005).

Definition **1.** The support of an itemset X in the database D is defined as the percentage of transactions that contain X, i.e.,

$$supp(X) = P(X).$$

Definition **2.** The rule $A \Rightarrow C$ holds in the transaction set D with **support** $supp(A \Rightarrow C)$, where $supp(A \Rightarrow C)$ is the percentage of transactions in D that contain $A \cup C$, i.e.,

$$supp(A \Rightarrow C) = supp(A \cup C) = P(A \cup C).$$

Definition **3.** The rule $A \Rightarrow C$ has **confidence** $conf(A \Rightarrow C)$ in the transaction set D, where $conf(A \Rightarrow C)$ is the percentage of transactions in D containing A that also contain C, i.e.,

$$conf(A \Rightarrow C) = P(C|A) = \frac{supp(A \Rightarrow C)}{supp(A)}$$

Confidence has some drawbacks, as we can see in the example shown in Figure 1, where we graphically represent two rules, $A \Rightarrow B$ and $A \Rightarrow C$. For the $A \Rightarrow B$ rule, we have the following support values for the intervening itemsets: supp(A)= 28%, supp(B)=38%, and supp($A \cup B$) = 21%. Therefore, the confidence of the $A \Rightarrow B$ rule is 75%. For the $A \Rightarrow C$ rule, even though the support of the consequent changes (supp(C)=85%), the confidence value of the $A \Rightarrow C$ rule is

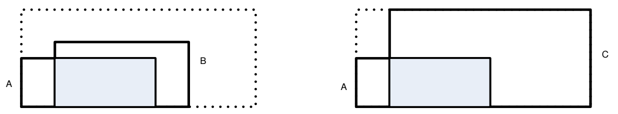


Fig. 1. Graphical depiction of two rules, $A \Rightarrow B$ and $A \Rightarrow C$, both with the same confidence but different consequent support.

still the same, 75%. In the first case, B was present in 38% of the transactions in the database and its presence increases to 75% in transactions where A is also present. In the second case, the presence of A reduces the presence of C, from 85% to 75%. However, the confidence measure does not let us distinguish between these two different situations.

In short, the confidence measure does not take into account the support of the rule consequent, hence it is not able to detect negative dependencies between items. Several measures have been proposed in the literature as alternative interestingness measures in order to avoid the limitations of the traditional support/confidence framework (Geng & Hamilton, 2006). In the following paragraphs, we describe some of them:

Definition **4.** The lift of the rule $A \Rightarrow C$, also known as interest, is defined as (Brin, et al., 1997a):

$$lift(A \Rightarrow C) = \frac{supp(A \Rightarrow C)}{supp(A)supp(C)}$$

The lift measure indicates how many times more often A and B occur together than expected if they where statistically independent. Values above 1 indicate positive dependence, while those below 1 indicate negative dependence. The lift values of the $A \Rightarrow B$ and $A \Rightarrow C$ rules in the the aforementioned example are $lift(A \Rightarrow B)$=4.2 and $lift(A \Rightarrow C)$=0.91. Here, $lift(A \Rightarrow B) > lift(A \Rightarrow C)$, which corresponds to our intuition that $A \Rightarrow B$ is more interesting than $A \Rightarrow C$.

Lift measures the degree of dependence between the itemsets. However, it only measures co-occurrence, but not the implication direction, since it is a symmetric measure, i.e., $lift(A \Rightarrow C)$=$lift(C \Rightarrow A)$.

Definition **5.** The conviction of the rule $A \Rightarrow C$ is defined as (Brin, et al., 1997b):

$$conv(A \Rightarrow C) = \frac{supp(A)supp(\neg C)}{supp(A \cup \neg C)}$$

The advantage of conviction with respect to the confidence measure is that it takes into account both the support of the antecedent and the support of the consequent of the rule. Conviction values in the (0,1) interval mean negative dependence, values above 1 mean positive dependence, and a value of 1 means independence, as happened with the lift measure. Unlike lift, conviction is not a symmetric measure, i.e. it measures the implication direction.

In the example from Figure 1, $supp(\neg B)$=0.62 and $supp(A \cup \neg B) = 0.07$. Therefore, the conviction of the $A \Rightarrow B$ rule is 2.48. For the $A \Rightarrow C$ rule, $supp(\neg C)$=0.15 and $supp(A \cup \neg C) = 0.07$. Therefore, $conv(A \Rightarrow C)$=0.6, which means negative dependence.

The main drawback of the conviction measure is that, as happened with lift, it is not bounded, i.e., its range is $[0,\infty)$. Therefore, it is difficult to establish a convenient conviction threshold in practice.

Supp(B)
0
1
Supp(C)
0
1

Fig. 2. Graphical examples illustrating the gain (and the certainty factor) of the rules derived from the scenarios represented in Figure 1: $A \Rightarrow B$ (left) and $A \Rightarrow C$ (right).

Let us now define an ancillary measure that will be useful in our discussion below: the gain of a rule as the difference between its confidence and the support of its consequent. Formally,

Definition **6.** The gain of a rule $A \Rightarrow C$ is defined as:

$$gain(A \Rightarrow C) = conf(A \Rightarrow C) - supp(C).$$

The gain values for the rules in Figure 1 are *gain* ($A \Rightarrow B$)=0.75-0.38=0.37 and *gain* ($A \Rightarrow C$)=0.75-0.85=-0.10. Figure 2 graphically shows these values. The lengths of the arrows represent the gain of the rules, i.e., the increase ($A \Rightarrow B$) or decrease ($A \Rightarrow C$) in the presence of the consequent given that the antecedent is present.

Definition **7.** The certainty factor of a rule $A \Rightarrow C$ is defined as (Shortliffe & Buchanan, 1975):

$$CF(A \Rightarrow C) = \frac{gain(A \Rightarrow C)}{1 - supp(C)} \text{ if } gain(A \Rightarrow C) \geq 0, \text{ and}$$

$$CF(A \Rightarrow C) = \frac{gain(A \Rightarrow C)}{supp(C)} \text{ if } gain(A \Rightarrow C) < 0.$$

The certainty factor is, therefore, the gain value normalized into the [-1,1] interval. The certainty factor can be interpreted as a measure of the variation of the probability that the consequent is in a transaction when we consider only those transactions where the antecedent occurs. More specifically, a positive CF measures the increase of the probability that the consequent is in a transaction, given that the antecedent is.

In the example from Figure 1, the CF of the $A \Rightarrow B$ rule is 0.37/(1-0.38)=0.60 while the CF for the $A \Rightarrow C$ rule is -0.10/0.85=-0.12.

3. Interestingness measures for group association rules

In the following paragraphs, we explain how to adapt the measures described in Section 2 to group association rules, as well as how these new measures can be useful for evaluating the interestingness of group association rules.

3.1. Group support

Definition **8.** The support of an itemset X in the group G is the percentage of transactions in G that contain X, i.e.,

$$supp_G(X) = \frac{P(XG)}{P(G)}.$$

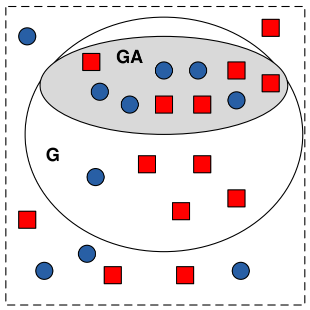


Fig. 3. Graphical representation of a group, G.

Figure 3 shows the representation of a group G within an example dataset. The support of *circles (●)* in the group G is $supp_G(\bullet)$= 6/15=0.4.

Property 1. The support of an itemset X in a group G is the confidence of the rule $(G \Rightarrow X)$, i.e.,

$$supp_G(X) = conf(G \Rightarrow X)$$

Proof. By Definition 8, $supp_G(X) = \frac{P(XG)}{P(G)}$. By Definition 3, $conf(G \Rightarrow X) = P(X|G) = \frac{P(XG)}{P(G)}$. Therefore, $supp_G(X)$=$conf(G \Rightarrow X)$ □

Definition **9.** The support of the group association rule $G : A \Rightarrow C$ is defined as:

$$supp_G(A \Rightarrow C) = supp_G(A \cup C) = \frac{P(GAC)}{P(G)}$$

In the previous example, the support of the group association rule $G : A \Rightarrow \bullet$ is $supp_G(A \Rightarrow \bullet) = 5/15 = 0.33$.

Property 2. The support of the rule $A \Rightarrow C$ in a group G is the confidence of the rule $G \Rightarrow AC$, i.e.,

$$supp_G(A \Rightarrow C) = conf(G \Rightarrow AC)$$

Proof. By Definition 9, $supp_G(A \Rightarrow C)=\frac{P(GAC)}{P(G)}$. By Definition 3, $conf(G \Rightarrow AC) = P(AC|G) = \frac{P(GAC)}{P(G)}$. Therefore, $supp_G(A \Rightarrow C)$=$conf(G \Rightarrow AC)$ □

3.2. Group confidence

Definition **10.** The confidence of the group association rule $G : A \Rightarrow C$ is defined as:

$$conf_G(A \Rightarrow C) = \frac{supp_G(A \Rightarrow C)}{supp_G(A)}.$$

The confidence of the rule $G : A \Rightarrow \bullet$ in Figure 3 is $conf_G(A \Rightarrow \bullet) = (5/15)$ / $(10/15) = 0.5$.

Property 3. The confidence of a rule $A \Rightarrow C$ in the group G is the confidence of the rule $GA \Rightarrow C$ in the database, i.e.,

$$conf_G(A \Rightarrow C) = conf(GA \Rightarrow C)$$

Proof. By Definition 10, $conf_G(A \Rightarrow C) = \frac{supp_G(A \Rightarrow C)}{supp_G(C)} = \frac{P(GAC)}{P(G)} \cdot \frac{P(G)}{P(GA)} = \frac{P(GAC)}{P(GA)}$. By Definition 3, $conf(GA \Rightarrow C) = \frac{P(GAC)}{P(GA)}$. Therefore, $conf_G(A \Rightarrow C) = conf(GA \Rightarrow C)$. $\square$

3.3. Group gain

Definition **11.** The gain of the group association rule $G : A \Rightarrow C$ is defined as:

$$gain_G(A \Rightarrow C) = conf_G(A \Rightarrow C) - supp_G(C)$$

The gain represents the difference between the confidence in the presence of the consequent when we know that the antecedent appears in the group, minus the support of the consequent within the group.

In Figure 3, the support of the *circles* in the group G is $supp_G(\bullet) = 6/15 = 0.4$ and the confidence of the $G : A \Rightarrow \bullet$ rule is $conf_G(A \Rightarrow \bullet) = 0.5$. Then, the gain of the rule is $gain_G(A \Rightarrow \bullet) = 0.5\text{-}0.4 = 0.1$. That means that, within the group G, finding a *circle* is 10% more likely when A holds.

Rules with high positive gain values help us describe subgroups (circles in the previous example) within the group G. On the other side, rules with high negative gain values help us find characteristics that are less frequent within the subgroup than in the overall group. For example, if the rule $A \Rightarrow \bullet$ had a negative gain value, that would mean that it would be more difficult to find a circle among those elements in GA than in G.

Property 4. The gain of the rule $G : A \Rightarrow C$ is the difference between the confidence of the $GA \Rightarrow C$ rule and the confidence of the $G \Rightarrow C$ rule , i.e.,

$$gain_G(A \Rightarrow C) = conf(GA \Rightarrow C) - conf(G \Rightarrow C)$$

Proof. By Definition 11, $gain_G(A \Rightarrow C) = conf_G(A \Rightarrow C)$ - $supp_G(C)$. By Properties 2 and 3, $supp_G(A \Rightarrow C) = conf(G \Rightarrow AC)$ and $conf_G(A \Rightarrow C) = conf(GA \Rightarrow C)$. Therefore, $gain_G(A \Rightarrow C) = conf(GA \Rightarrow C) - conf(G \Rightarrow C)$ $\square$

Property 5. The gain of the $G : A \Rightarrow C$ rule is the difference between the gain of the $GA \Rightarrow C$ rule and the gain of the $G \Rightarrow C$ rule , i.e.,

$$gain_G(A \Rightarrow C) = gain(GA \Rightarrow C) - gain(G \Rightarrow C)$$

Proof. By Definition 6, $gain(G \Rightarrow C) = conf(G \Rightarrow C) - supp(C)$. Then, we can solve for $conf(G \Rightarrow C)$ as $conf(G \Rightarrow C) = gain(G \Rightarrow C) + supp(C)$. If we replace the $conf(G \Rightarrow C)$ in Property 4, we obtain $gain_G(A \Rightarrow C) = conf(GA \Rightarrow C) - conf(G \Rightarrow C) = conf(GA \Rightarrow C) - supp(C) - gain(G \Rightarrow C)$. Finally, by Definition 6, $conf(GA \Rightarrow C) - supp(C) = gain(GA \Rightarrow C)$. Therefore, $gain_G(A \Rightarrow C) = gain(GA \Rightarrow C) - gain(G \Rightarrow C)$. $\square$

Theorem 1. Gain pseudo-commutativity. The difference between the gain of the $G : A \Rightarrow C$ rule in the G group and the gain of the $A \Rightarrow C$ rule in the database equals the gain of the rule $A : G \Rightarrow C$ in the group A minus the gain of the $G \Rightarrow C$ rule in the database , i.e.,

$$gain_G(A \Rightarrow C) - gain(A \Rightarrow C) = gain_A(G \Rightarrow C) - gain(G \Rightarrow C).$$

Proof. By Definition 6, $gain(G \Rightarrow C) = conf(G \Rightarrow C) - supp(C)$.

We can isolate $conf(G \Rightarrow C) = gain(G \Rightarrow C) + supp(C)$ and replace it in $gain_G(A \Rightarrow C) = conf(GA \Rightarrow C) - conf(G \Rightarrow C) = conf(GA \Rightarrow C) - (gain(G \Rightarrow C) + supp(C))$.

If we isolate $supp(C)$ from Definition 6 and replace it in the previous expression, we obtain: $gain_G(A \Rightarrow C) = conf(GA \Rightarrow C) - (gain(G \Rightarrow C) + supp(C)) = conf(GA \Rightarrow C) - gain(G \Rightarrow C) - (conf(A \Rightarrow C) - gain(A \Rightarrow C)) = conf(GA \Rightarrow C) - (conf(A \Rightarrow C) - gain(G \Rightarrow C) + gain(A \Rightarrow C))$.

By Definition 11, the first term can be expressed as $conf(GA \Rightarrow C) - conf(A \Rightarrow C) = gain_A(G \Rightarrow C)$. Then, we have $gain_G(A \Rightarrow C) = gain_A(G \Rightarrow C) - gain(G \Rightarrow C) + gain(A \Rightarrow C))$.

Therefore, $gain_G(A \Rightarrow C) - gain(A \Rightarrow C) = gain_A(G \Rightarrow C) - gain(G \Rightarrow C)$. □

3.4. Group gain normalization

The range of the gain measure for group association rules is $[-supp_G(C), 1 - supp_G(C)]$. In the following paragraphs, we propose several ways to normalize the group gain depending on the kind of information the user might be more interested in.

3.4.1. Group gain factor

We can normalize the gain into the $[-1, 1]$ interval to obtain a gain factor measure that corresponds to the certainty factor in the general association rule framework:

Definition **12.** The gain factor of the group association rule $G : A \Rightarrow C$ is defined as:

$GF_G\ (A \Rightarrow C) = \frac{gain_G(A \Rightarrow C)}{1 - supp_G(C)}$ if $gain_G(A \Rightarrow C) \geq 0$, and

$GF_G\ (A \Rightarrow C) = \frac{gain_G(A \Rightarrow C)}{supp_G(C)}$ if $gain_G(A \Rightarrow C) < 0$.

In our example, the gain factor of the rule $G : A \Rightarrow \bullet$ is $GF_G(A \Rightarrow \bullet)$= 0.1/(1-0.4) =0.17.

This measure is proportional to the group gain. When it is positive, it is also inversely proportional to the value $[1 - supp_G(C)]$. Therefore, all other things being equal, GF will be larger for subgroups of elements that were more common in the group G (i.e., those having a higher $supp_G(C)$). When GF is negative, it is inversely proportional to $supp_G(C)$: it will have a larger absolute value when the subgroup is less frequent in G, i.e. for small values of $supp_G(C)$.

3.4.2. Group variation

The group variation measure always normalizes the gain using the $supp_G(C)$ value, in order to highlight those consequents that are less frequent in our database.

Definition **13.** The variation of a group association rule $G : A \Rightarrow C$ is defined as:

$$\delta_G(A \Rightarrow C) = \frac{gain_G(A \Rightarrow C)}{supp_G(C)} = \frac{conf_G(A \Rightarrow C) - supp_G(C)}{supp_G(C)}$$

In contrast to the GF, variation is inversely proportional to $supp_G(C)$ when it is positive. It will have a higher value the less frequent C is in G. It should be noted that the variation equals the gain factor when the gain is negative. The variation of the rule $G : A \Rightarrow \bullet$ in Figure 3 is $\delta_G(A \Rightarrow \bullet)$= 0.1/(0.4)=0.25.

3.5. Group impact

In this section we present two new interestingness measures that are also based on the group gain. In these measures, we take into account the support of the group corresponding to the antecedent of the rule.

3.5.1. Impact

Definition **14.** The impact of the group association rule $G : A \Rightarrow C$ is defined as:

$$impact_G(A \Rightarrow C) = supp(GA) * gain_G(A \Rightarrow C)$$

The impact of a group association rule represents the number of individuals that are affected by the rule, i.e., the number of individuals that we did not expect to find in the transactions within the group G that contain A given what we knew about G in general.

The impact is proportional to $gain_G(A \Rightarrow C)$ and $supp(GA)$. It will be higher for those rules with a high gain and a frequent antecedent in the G group.

In our example from Figure 3, $impact_G(A \Rightarrow \bullet)$ = (10)*0.1=1. That impact means that there is one circle that we did not expect to find in GA when we only knew the support of circles in G. Since $supp_G(\bullet)$=0.4, we did expect four circles in GA but found five of them.

3.5.2. Impact ratio

The impact measure represents the number of individuals that are affected by the rule. However, in most cases, groups have different numbers of individuals and an absolute impact value might be misleading. The impact ratio might be useful in such situations, since it represents the ratio between the number of individual affected by the rule and the number of individuals within the group.

Definition **15.** The impact ratio of the group association rule $G : A \Rightarrow C$ is defined as:

$$IR_G(A \Rightarrow C) = \frac{impact_G(A \Rightarrow C)}{supp(G)}$$

The impact ratio of a group association rule represents the proportion of the impact of the rule within the group G with respect to the size of the group. The impact ratio is $IR_G(A \Rightarrow \bullet)$= 1/15=0.07 in the example from Figure 3.

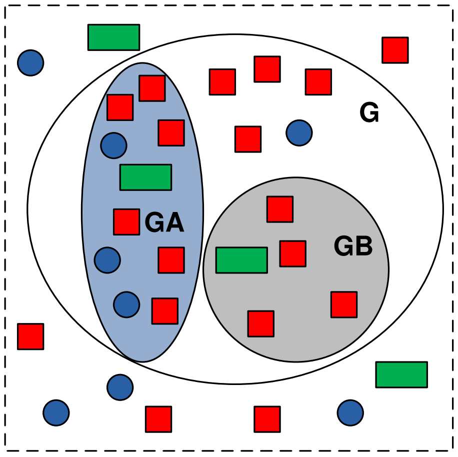


Fig. 4. Graphical representation of a group, G, and two rules within that group, $(A \Rightarrow \bullet)$ and $(B \Rightarrow \blacksquare)$.

4. Ranking groups and group association rules

All the different groups that can be identified in the database, as well as the rules within them, can be obtained in an unsupervised manner. However, the amount of rules and groups obtained in the rule mining process can be huge. As a consequence, a second-order data mining problem arises: it may be too difficult to extract useful information from so many rules and groups.

In this section, we explain how to rank the groups (and the rules within the groups) according to their potential interestingness. We will use the measures we described in previous sections to highlight those rules and groups that might be relevant to the user according to different criteria.

4.1. Ranking rules within a particular group

The use of each different interestingness measure provides us a different ordering relationship among the discovered association rules. In this section, we analyze how two rules within a group will have a different relative order within that group depending on the interestingness measure we use to evaluate them.

Figure 4 shows a group, G, within an example dataset. In this group, we have identified two rules, $A \Rightarrow \bullet$ and $B \Rightarrow \blacksquare$, whose values for the different interestingness measures we described in the previous sections are summarized in Table 1.

4.1.1. Characterizing subgroups within a group

If we are interested in obtaining those rules that characterize subgroups within a group (i.e., rules sharing their consequent), we should use the **gain** measure because a high gain increases our confidence in the presence of the consequent when we know that the antecedent holds.

In our example from Figure 4, the gain of both rules is 10 %, i.e., the confi-

	$A \Rightarrow \bullet$	$B \Rightarrow \blacksquare$
$supp_G(Y)$	$4/20 = 0.20$	$14/20 = 0.70$
$supp_G(X \Rightarrow Y)$	$3/20 = 0.15$	$4/20 = 0.20$
$conf_G(X \Rightarrow Y)$	$3/10 = 0.30$	$4/5 = 0.80$
$gain_G(X \Rightarrow Y)$	$0.3 - 0.2 = 0.10$	$0.8 - 0.7 = 0.10$
$GF_G(X \Rightarrow Y)$	$0.1/0.8 = 0.125$	$0.1/0.3 = 0.33$
$\delta_G(X \Rightarrow Y)$	$0.1/0.2 = 0.50$	$0.1/0.7 = 0.14$
$impact_G(X \Rightarrow Y)$	$0.1 * 10 = 10$	$0.1 * 5 = 0.50$
$IR_G(X \Rightarrow Y)$	$1/20 = 0.05$	$0.5/20 = 0.025$

Table 1. Values of the different interestingness measures for the ($A \Rightarrow \bullet$) and ($B \Rightarrow \blacksquare$) rules in Figure 4.

dence on the presence of $\bullet$ increases if also A holds, while the confidence on the presence of the $\blacksquare$ increases, in the same amount, when B is true.

Both are important rules within the group G but they give us different information as squares are more frequent in G than circles. Therefore, we should use other measures for distinguishing between them:

– If we want to highlight the most frequent subgroups, the **gain factor**, as inversely proportional to the interval $[1 - supp_G(C)]$, should be used.
  In our example, the GF_G value is higher for the $B \Rightarrow \blacksquare$ rule because squares where more frequent in G than circles.
– If we want to highlight anomalies, the **variation** measure is a better choice since, in contrast to the gain factor, it overweighs those subgroups that have a low support in the group. That is the case of the circles in G: the $A \Rightarrow \bullet$ rule has a larger value of δ_G.

4.1.2. Characterizing subgroups using frequent features

If we are interested, not only in the subgroups themselves, but also in discovering features that make them distinct in our database, we should use an interestingness measure that takes into account the frequency of the rule antecedent, e.g., the **impact** measure.

In our example, the impact of the $A \Rightarrow \bullet$ rule in G is larger than the impact of the $B \Rightarrow \blacksquare$ rule because the support of A is larger than the support of B.

This measure has the advantage that it can be easily interpreted: the number of individuals in G that are directly affected by the $A \Rightarrow C$ rule, i.e., those individuals that are not expected to be in GA given the overall support of C in the group.

In our example, the impact of the $A \Rightarrow \bullet$ rule is 1 because, given a 20% support for circles in G and 10 elements in the GA subgroup, we expected 2 circles in GA but found 3 of them.

It should be noted that the impact ratio measure gives us the same relative ordering among rules within the same group than the impact measure, since it takes into account the size of the group G, which is the same for all the rules within the same group.

4.2. Ranking groups within the database

Apart from ordering rules within a group, we can establish an order relationship among the groups in our database to highlight those groups that host the most interesting rules.

Given that groups can be described by many different association rules, if we intend to rank groups rather than individual rules, we must somehow compute an aggregate value that represents the overall interestingness of the group. We should average the impact of the n rules within a given group to indicate the overall interestingness of that group. However, as we have explained in Section 4.1, some rules are more interesting than others, hence they should not have the same importance when computing the overall group score.

Impact seems to be a good candidate for estimating the potential interestingness of the group, since it takes into account the number of individuals that are affected by each rule within the group. We can then define the weighted impact for the rules within a group using different interestingness measures. Formally, we define the weighted impact as:

$$\text{Weighted impact}(G) = \frac{\sum_{i=1}^{n} I_G(A \Rightarrow C) \cdot impact_G(A \Rightarrow C)}{\sum_{i=1}^{n} I_G(A \Rightarrow C)}$$

where $I_G(A \Rightarrow C)$ represents the value of any of the interestingness measures described in Section 3 and analyzed in Section 4.1, for each $A \Rightarrow C$ rule in the G group.

Large groups tend to have higher impact values for their rules because their impact depends on the support of their antecedent within the group and it is usually larger in large groups. Therefore, small groups are penalized in the overall group ranking if we use the impact measure to average the interestingness of the rules within the group. When we want to take into account the relative size of the groups, we should use the impact ratio measure instead, which gives us a more balanced ranking for groups of disparate size. Thus, we can also define a weighted impact ratio measure to rank groups within the database:

$$\text{Weighted IR}(G) = \frac{\sum_{i=1}^{n} I_G(A \Rightarrow C) \cdot IR_G(A \Rightarrow C)}{\sum_{i=1}^{n} I_G(A \Rightarrow C)}$$

5. Comparing alternative ranking criteria

Once we have proposed different interestingness measures that can be used to rank groups and group association rules, we are interested in comparing the rankings obtained by each measure in order to analyzed how similar (or different) they are.

Several measures have been proposed in the literature in order to compare two permutations σ_1 and σ_2 whose elements are in D. Two well-known measures are (Fagin, et al., 2003):

– **Kendall's tau:** For each pair $i, j \in P$ of distinct members of D, if i and j are in the same order in σ_1 and σ_2, then let $K_{i,j}(\sigma_1,\sigma_2) = 0$; and if i and j are in the opposite order, then $K_{i,j}(\sigma_1,\sigma_2) = 1$. Kendall's tau is given by $K(\sigma_1,\sigma_2) = \sum_{\{i,j\} \in P} K_{i,j}(\sigma_1,\sigma_2)$. The maximum value of $K(\sigma_1,\sigma_2)$ is $n(n-1)/2$, which occurs when σ_1 is the reverse of σ_2 (that is, when $\sigma_1(i) + \sigma_2(i) =$ n + 1 for each i). Kendall's tau turns out to be equal to the number of exchanges needed in a bubble sort to convert one permutation into the other. Kendall's tau can be computed in $O(nlogn)$ using a divide-and-conquer algorithm (Kleinberg & Tardos, 2005).

– **Spearman's footrule** is defined by $F(\sigma_1,\sigma_2) = \sum_{i=1}^{n} |\sigma_1(i) - \sigma_2(i)|$. The maximum value of $F(\sigma_1,\sigma_2)$ is $n^2/2$ when n is even, and $(n+1)(n-1)/2$ when n is odd. As with Kendall's tau, the maximum occurs when σ_1 is the reverse of σ_2. This measure can be easily computed in $O(n)$ given an inverse index for both rankings.

These measures let us compare two complete rankings. However, in many cases, the top K elements in the rankings are more important than others that appear at less prominent positions (think of a Web search engine, for instance). (Fagin et al., 2003) proposed an adaptation of the aforementioned measures to compare two top-K lists, τ_1 and τ_2:

– **Kendall's tau for top-K lists**: We have to consider 4 possible scenarios taking into account that not all the elements in the top K list τ_1 may be present in the top K list τ_2 and vice versa. For each pair $i, j \in D_{\tau_1} \cup D_{\tau_2}$:

  · Case 1: i, j are both present in τ_1 and τ_2. Then, if they are in the same order in both list, $K_{i,j}(\tau_1,\tau_2) = 0$ while, if they are in the opposite order, $K_{i,j}(\tau_1,\tau_2) = 1$.
  · Case 2: i, j are both present in τ_1 but only i is present in τ_2. Then, if i is ahead of j in τ_1, $K_{i,j}(\tau_1,\tau_2) = 0$. Otherwise, $K_{i,j}(\tau_1,\tau_2) = 1$.
  · Case 3: i is only present in τ_1 and j is only present in τ_2. Then, $K_{i,j}(\tau_1,\tau_2) = 1$.
  · Case 4: i, j are both present in one top k list (say τ_1) but neither i nor j appear in the other top k list (say τ_2). Then, we do not have information about the order of i and j in τ_2. Therefore, $K_{i,j}(\tau_1,\tau_2) = p$, where the value of p is usually 0.5 to be neutral.

– The **Spearman's footrule for top-K lists** is computed as $F^l(\tau_1,\tau_2) = \sum_{i \in D_{\tau_1} \cup D_{\tau_2}} |\tau_1' - \tau_2'|$, where $\tau_x'(i) = \tau_x(i)$ for $i \in D_{\tau_x}$, otherwise $\tau_x'(i) = l$. A natural choice for l is $k + 1$.

6. Experimental results

In order to study the performance of the proposed interestingness measures, we have performed a series of experiments using the `loan` multirelational database. The `loan` database was used in the PKDD CUP'09. Figure 5 shows the schema of this database.

The database contains eight relations with 75,982 tuples in total, while the relation we are going to use as target relation (`account`) contains 4500 tuples. This database was adapted by Yin et al. for their experiments with CrossMine (Yin, et al., 2004) and can be downloaded from: *http://research.microsoft.com/en-us/people/xyin/*.

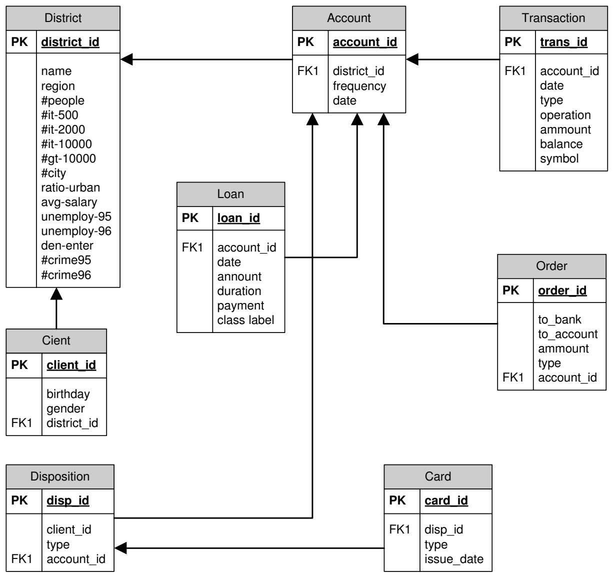


Fig. 5. Loan database schema.

6.1. Using trees to mine multirelational databases

A common approach to mine multirelational databases consists of joining all the relations in the database in order to obtain a single relation, usually called universal relation (Fagin, et al., 1982) (Maier & Ullman, 1983) (Maier, et al., 1984). Then, classical data mining techniques can be applied to this universal relation. However, join-based techniques such as the aforementioned one present a serious disadvantage: they do not preserve the proper support counts.

We have proposed two alternative representation schemes for multirelational databases. Our representation schemes are based on trees, so that we can apply existing tree mining techniques to identify frequent patterns in multirelational databases (Jiménez, et al., 2009).

The main idea behind our two representation schemes is building a tree from each tuple in the target relation and following the links between relations (i.e. foreign keys) to collect all the information related to each tuple in the target relation.

The key-based tree representation scheme is inspired by the concept of identity in the relational model while the object-based one is based on the concept of identity in object-oriented models. Relational databases rely on primary keys to ensure that each table row can be univocally referenced. Any unique field, or combination of fields, can be used as the primary key. In the object model, however, each object is already unique, and no specific key is needed. In an object database, each object is automatically assigned an unique ID. This does mean that you can create objects that have identical field values but are still different objects (Paterson, et al., 2006).

In our experiments, we have considered the `account` relation as our target relation in the `loan` database and we have built trees by collecting information from the `loan`, `disposition`, `order`, and `district` relations.

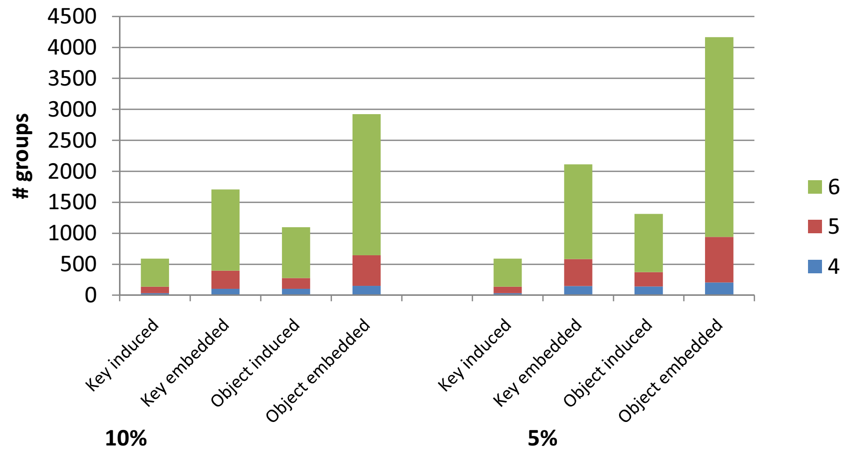


Fig. 6. Number of identified groups within the `loan` database, using patterns with varying size (from 4 to 6) and two different support thresholds (10% and 5%).

The tree database obtained from the `loan` multirelational database contains 4500 trees. Trees have an average of 34 nodes using the key-based representation scheme (153,102 nodes in total) and an average of 37 nodes using the object-based one (169,842 nodes in total).

6.2. Extracting group association rules from multirelational databases

We have transformed the `loan` database into two sets of trees (using both the key-based and object-based tree representation schemes). This way, tree mining algorithms can be used to extract patterns from this kind of databases, as described above. We can also define two different kinds of patterns to be identified within the trees: induced and embedded patterns. Therefore, four different kinds of patterns can be mined from a multirelational database using our approach, i.e., induced key-based patterns, embedded key-based patterns, induced object-based patterns, and embedded object-based patterns.

We have used the POTMiner tree pattern mining algorithm (Jimenez, et al., 2010) to identify these four different kinds of patterns within the `loan` database. Once we have identified the frequent tree patterns derived from the multirelational database, we have extracted association rules from them, using techniques that are analogous to the ones employed in a more traditional setting. Group association rules can then be derived from the discovered association rules just by clustering rules sharing parts of their antecedents.

Figure 6 shows the number of identified groups within the `loan` database for the four different kinds of patterns (induced key-based patterns, embedded key-based patterns, induced object-based patterns, and embedded object-based patterns) and two different minimum support thresholds (10% and 5%). As expected, the number of identified groups increases when the size of the patterns also increases, and also when the minimum support threshold decreases, since the more patterns are identified, the more group association rules can be extracted.

Figure 7 shows the execution time required by our algorithm to obtain all the

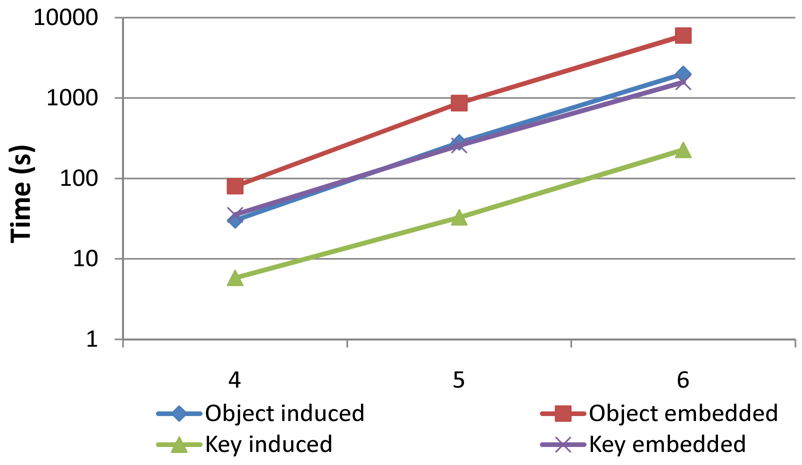


Fig. 7. POTMiner execution time required to extract groups from the `loan` multirelational database from frequent patterns of increasing size.

existing groups in the multirelational database, where time is displayed in seconds on a logarithmic scale. It should be noted that execution time includes the whole process required for identifying group association rules: the time required to identify the patterns, the time required to identify the association rules, and the time required to group the association rules into sets of group association rules. The identification of key-induced patterns is the fastest because there is a lower number of this kind of patterns. On the other hand, mining object-embedded patterns is the most costly alternative because more patterns are involved.

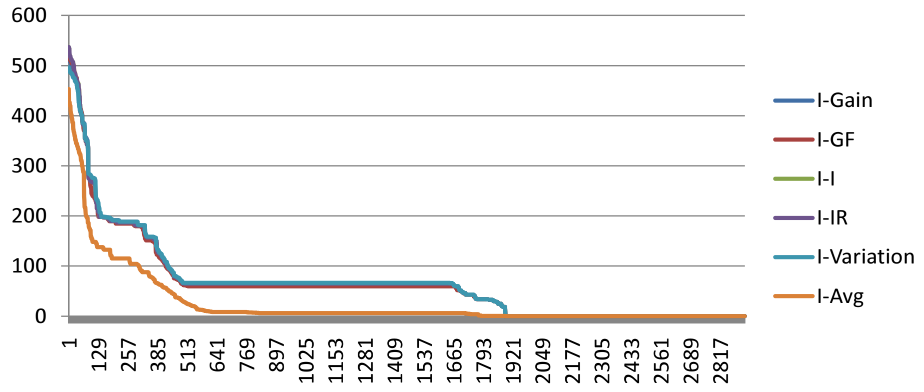
6.3. Ranking groups

In this section, we compare the group rankings provided by different interestingness measures in order to check how many groups are highlighted as the most interesting ones and analyze potential differences among them.

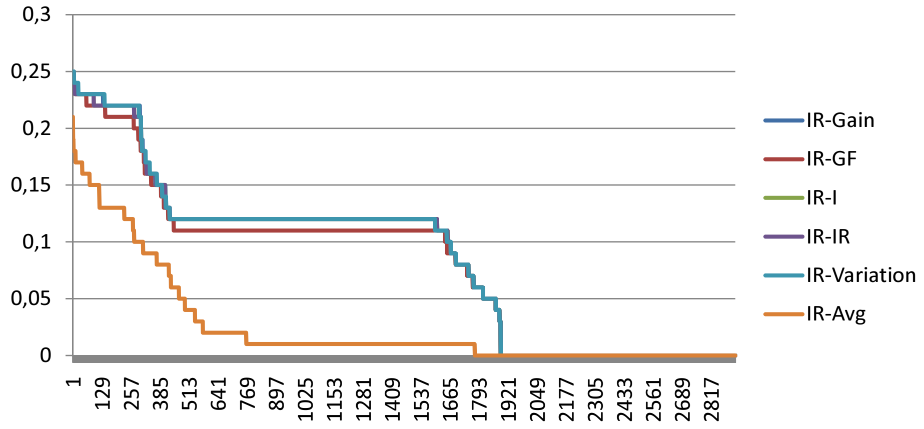
In these series of experiments, we have used the groups and rules obtained from the `loan` multirelational database, using embedded object-based patterns up to size 6 (i.e. maxsize=6) and a 10% minimum support threshold. Using these parameters, we obtained 2,924 groups and 7,394,860 group association rules.

We have ranked the resulting groups using the weighted impact and the weighted impact ratio measures. In order to weight those measures, we have tested five different interestingness measures: gain, gain factor, impact, impact ratio, and variation. Therefore, we have obtained 10 alternative rankings for the groups in our database. We have also included two additional rankings using the average impact and the average impact ratio measures, just to check that weighted measures give us different results than their plain averages (i.e., just to corroborate that taking rule interestingness into account is preferred over just aggregating rule impacts without taking that information into account).

Figures 8 a) and b) show the values of the weighted impact and the weighted impact ratio measures, respectively, for every group within the `loan` multirelational database (sorting the groups according to their weighted evaluation measure).



a) Weighted impact values.

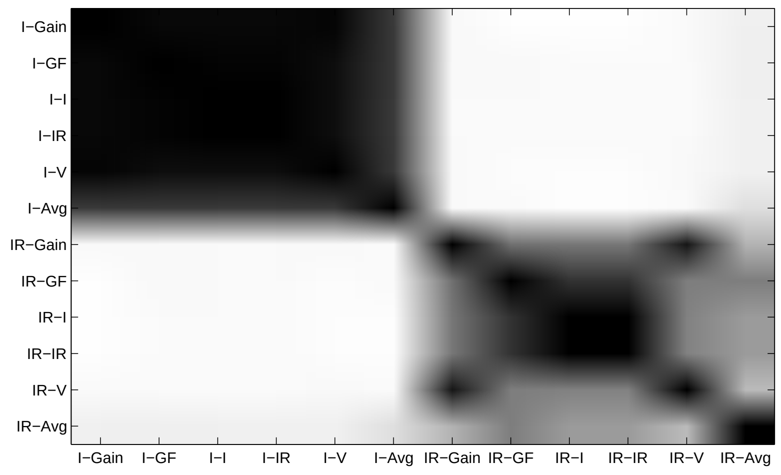


b) Weighted impact ratio values.

Fig. 8. Weighted impact and weighted impact ratio values for the 2924 groups identified within the `loan` multirelational database using six different interestingness measures.

As it can be seen in Figure 8 a), the weighted impact measures highlight about 130 of groups in the database. It should be noted that the weighted impact provides similar results for every different interestingness measure, i.e. the rankings provided by the different interestingness measures, used as weights in the weighted impact formula, are quite similar. The average values are lower, as expected, since less interesting rules contribute less to diminish the overall group score when using the weighted measure. The similarities among the particular rankings provided by different interestingness measures will be discussed later.

Figure 8 b) shows that the weighted impact ratio measures highlight about 400 of the groups in the database, more than the weighted impact measures. The weighted impact tends to highlight only large groups, since the impact of the rules depends on the absolute support of the antecedent in the group association rules and that support is typically larger in large groups. Using the weighted impact ratio, however, group size does not influence the final group score because impact ratio is a relative measure. Hence the larger number of



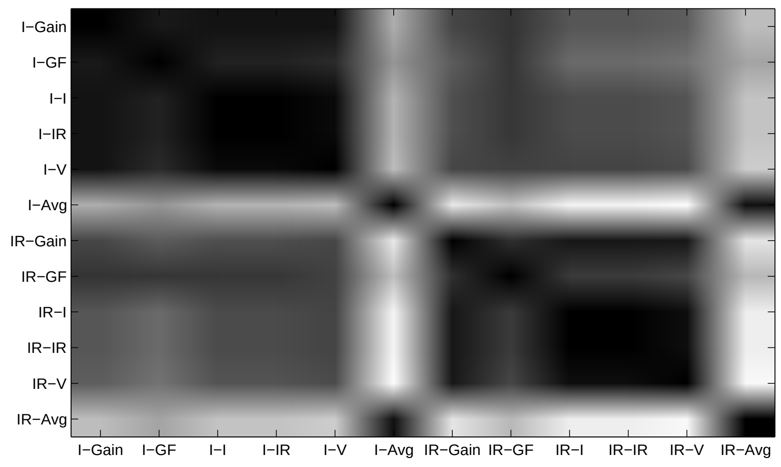


Fig. 9. Similarity matrices using Kendall's (top) and the Spearman's (bottom) measures to compare the rankings.

highlighted groups when using the weighted impact ratio: smaller groups are still highlighted when interesting rules affect a significant portion of them.

As happened with the impact measures, the results using different interestingness measures as weights for the weighted impact ratio are similar and raw averages provide lower scores. In the following section, we analyze the particular rankings provided by each alternative evaluation criterion.

6.4. Comparing rankings provided by different interestingness measures

In the previous section, we obtained twelve different rankings of the 2,924 groups within the `loan` database. In this section, we use the Kendall's and Spearman's measures we described in Section 5 to compare these rankings in order to discover their similarities.

We have compared every pair of rankings using both similarity measures. Figure 9 shows the resulting similarity matrices using Kendall's (top) and the Spearman's (bottom) measures. Black cells indicate that the rankings are almost indistinguishable, while white cells correspond the most dissimilar pairs of rankings within the matrix.

As it can be seen in Figure 9, we can easily observe two clearly-defined groups of different rankings, which correspond to those obtained using the weighted impact measures (upper left quadrant) and the ones obtained using the weighted impact ratio measures (lower right quadrant). When interpreting the results, it is important to recall that Kendall's measure is an order-based measure while Spearman's measure is a distance-based one.

In Figure 9, we are comparing the rankings for all the 2,924 groups within our database, hence there are some differences in the comparison between different variants of the same kind, a fact that is clearly visible for the weighted impact ratio results (lower right quadrant), especially for Kendall's order-based measure. Spearman's distance-based measure highlights some differences when using the gain factor as interestingness measure for individual rules within a group, as well as a surprising similarity between the average impact and the average impact ratio rankings.

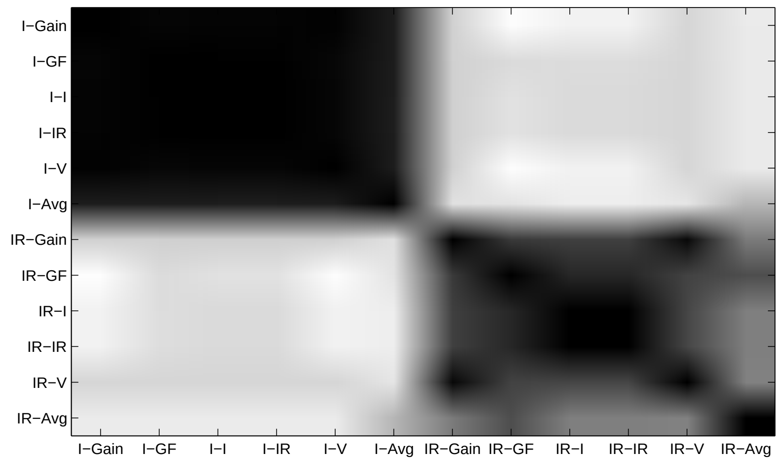
However, it should be noted that apparent differences do not have the same importance if they happen at the top of the rankings or at their bottom, since end users will not usually delve into the complete list of 2,924 groups. Therefore, a more realistic scenario consists of comparing the top elements in each ranking.

We have compared the top 100 and the top 10 groups in the rankings provided by each ranking criterion in order to obtain a more realistic comparison of ranking results.

The matrices shown in Figure 10 show the values for Kendall's and Spearman's measures when comparing the top 100 groups in each of the rankings. In this case, the matrices we have obtained using both Kendall's order-based and Spearman's distance-based similarity measures are remarkably similar. As happened when comparing the whole rankings, two groups are clearly visible: the ones corresponding to the weighted impact and those resulting from using the weighted impact ratio.

The different variants of weighted impact are almost undistinguishable, while some differences exist among the different rankings provided by the weighted impact ratio. Within the latter, using the impact (IR-I) and the impact ratio (IR-IR) as weights provide very similar results. Likewise, gain (IR-Gain) and variation (IR-V) are also similar. However, there are differences between gain factor (IR-GF) and variation (IR-V). Gain factor equals variation, but only for negative gain values. Here, we are considering the 100 most interesting groups, which typically contain many rules with positive gain values (it should be recalled that both impact and impact ratio are proportional to gain values).

Finally, we have compared the top 10 groups in the rankings as shown in Figure 11. For our particular database, we can again identify two clusters around



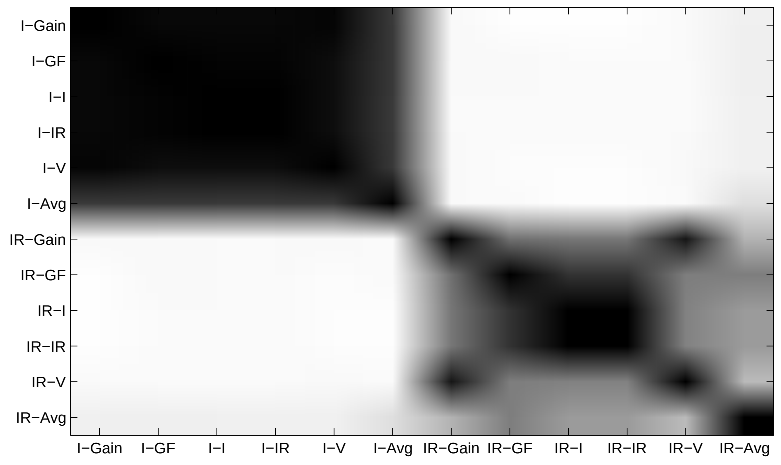


Fig. 10. Similarity matrices using Kendall's (top) and Spearman's (bottom) measures to compare the top 100 groups in each ranking.

weighted impact and weighted impact ratio measures. There are still some differences between the weighted measures and their plain averages, as well as between the variation variant of the weighted impact (I-V) and the rest of its variants (I-Gain, I-GF, I-I, I-IR). In this case, the I-V results are more similar to the I-Gain and the I-GF results (whose interestingness measures are closer to variation, which is just a method for normalizing rule gain, as the gain factor) than they are to the impact-related measures (I-I and I-IR).

When using the average impact of the rules within each group (I-Avg), large groups are favored. Almost all the groups in the top 10 have a 100% support and most of them are trivial and not really interesting. The average impact ratio (IR-Avg) is not so biased towards large groups.

Using the weighted impact measure (I-Gain, I-GF, I-I, I-IR, and I-V) provides a more balanced ranking. Albeit the support of the groups in their top 10 is still high (around 75%), end users can find some interesting groups, such as "accounts that have a monthly frequency of issuance of statements and an owner-type disposition".

The weighted impact ratio measure (IR-Gain, IR-GF, IR-I, IR-IR, IR-V) provides more interesting results. In this case, the support of the groups in the top 10 is around 25% and much more specific groups are highlighted, such as those "accounts that have a monthly frequency of issuance of statements, and a house-type permanent order, and are located in a district with one municipality with more than 10000 inhabitants".

In summary, weighted impact is biased towards large groups and it should be used when looking for the characterization of very large clusters within a database of individuals. However, users should resort to the weighted impact ratio if they are willing to discover smaller groups of potentially more interesting individuals (i.e. those who are not so common in the database but exhibit consistently different peculiar behaviors).

7. Conclusions

Databases naturally contain groups of individuals that share some of their features and some aspects of their behavior. In this paper, we have proposed group association rules, which are association rules that can be discovered within these groups of individuals.

We have adapted some of the standard interestingness measures for association rules to group association rules. We have also proposed new interestingness measures to evaluate this particular kind of association rules and we have studied some of the formal properties of these new measures.

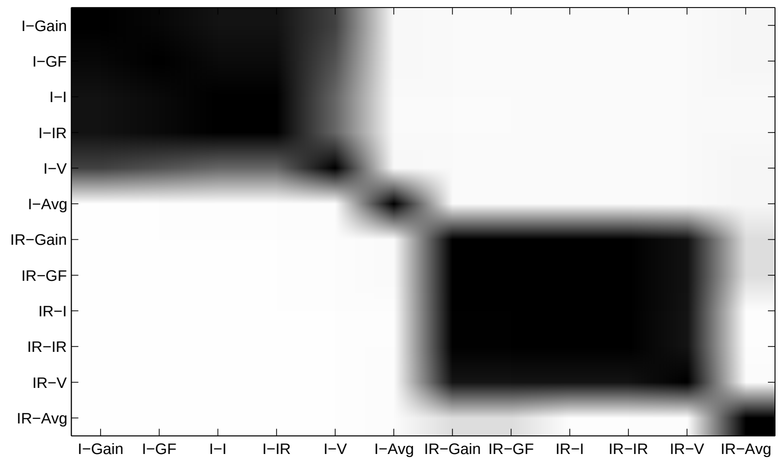
Finally, we have proposed an approach to rank groups within a database (and the rules within each group). Alternative rankings can be provided by employing alternative (and often complementary interestingness measures). Depending on their particular goals, users should choose which measure to employ in order to highlight the most potentially interesting groups. Our experiments on a real-world database corroborated our intuitions on the behavior of different ranking criteria and demonstrated that our approach can be useful in practice for ameliorating the second-order data mining problem that users must face when dealing with the huge amount of association rules that can be derived from real-world databases (more than seven million in our case study).

Acknowledgements

This work is partially supported by research projects TIN2006-07262 (Spanish Ministry of Science and Innovation), TIN2009-08296 (Spanish Ministry of Science and Innovation) and P07-TIC-03175 (Junta de Andalucía).

References

R. Agrawal & R. Srikant (1994). 'Fast Algorithms for Mining Association Rules in Large Databases'. In *20th International Conference on Very Large Data Bases*, pp. 487–499.



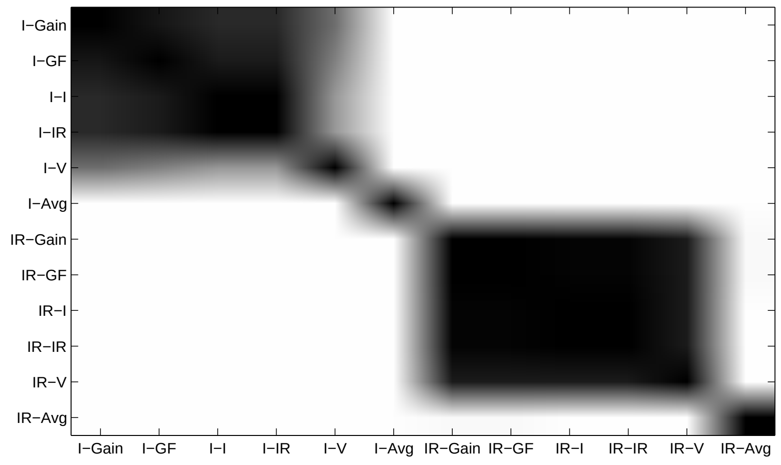


Fig. 11. Similarity matrices using Kendall's (top) and Spearman's (bottom) measures to compare the top 10 groups in each ranking.

F. Berzal, et al. (2002). 'Measuring the accuracy and interest of association rules: A new framework.'. *Intelligence Data Analysis* **6**(3):221–235.

F. Berzal & J. C. Cubero (2007). 'Guest editors' introduction'. *Data and Knowledge Engineering* **60**(1):1–4.

S. Brin, et al. (1997a). 'Beyond market baskets: generalizing association rules to correlations'. *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* **26**(2):265–276.

S. Brin, et al. (1997b). 'Dynamic itemset counting and implication rules for market basket data'. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, vol. 26, pp. 255–264. ACM.

R. Fagin, et al. (2003). 'Comparing top k lists'. In *Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pp. 28–36, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

R. Fagin, et al. (1982). 'A simplied universal relation assumption and its properties'. *ACM Transactions on Database Systems* **7**:343–360.

L. Geng & H. J. Hamilton (2006). 'Interestingness measures for data mining: A survey'. *ACM Computing Surveys* **38**(3):9.

J. Han & M. Kamber (2005). *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc.

A. Jiménez, et al. (2009). 'Frequent Itemset Mining in Multirelational Databases'. In *Proceedings of the 18th International Symposium on Foundations of Intelligent Systems*, pp. 15–24, Berlin, Heidelberg. Springer-Verlag.

A. Jimenez, et al. (2010). 'POTMiner: Mining Ordered, Unordered, and Partially-Ordered Trees'. *Knowlegde and Information System* **23**(2):199–224.

J. Kleinberg & E. Tardos (2005). *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

D. Maier & J. D. Ullman (1983). 'Maximal objects and the semantics of universal relation databases'. *ACM Transactions on Database Systems* **8**:1–14.

D. Maier, et al. (1984). 'On the foundations of the universal relation model'. *ACM Transactions on Database Systems* **9**:283–308.

J. Paterson, et al. (2006). *The Definitive Guide to db4o.* Apress.

M. Plasse, et al. (2007). 'Combined use of association rules mining and clustering methods to find relevant links between binary rare attributes in a large data set'. *Computational Statistics & Data Analysis* **52**(1):596–613.

E. H. Shortliffe & B. G. Buchanan (1975). 'A model of inexact reasoning in medicine'. *Mathematical biosciences* **23**:351–379.

X. Yin, et al. (2004). 'CrossMine: Efficient Classification Across Multiple Database Relations'. In *Proceedings of the 20th International Conference on Data Engineering*, pp. 399–410.

2.2 Group association rules

The journal paper associated to this part of the dissertation is:

- A. Jiménez, F. Berzal, and JC Cubero, Interestingness measures for association rules within groups. **Submitted to Knowledge and Information Systems**.
  - Status: **Submitted**.
  - Impact Factor (JCR 2009): 2.211.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 27 / 103.
    - Computer Science, Information Systems. Ranking 24 / 116.

Manuscript Number: DAMI1080R2

Title: Using trees to mine multirelational databases

Article Type: Manuscript

Keywords: Mutirelational databases; Frequent itemset mining;
Association rules; Tree pattern mining

Corresponding Author: Aida Jimenez

Corresponding Author's Institution:

First Author: Aida Jimenez

Order of Authors: Aida Jimenez;Fernando Berzal;Juan-Carlos Cubero

Abstract: This paper proposes a new approach to mine multirelational databases. Our approach is based on the representation of multirelational databases as sets of trees, for which we propose two alternative representation schemes. Tree mining techniques can thus be applied as the basis for multirelational data mining techniques, such as multirelational classification or multirelational clustering. In this paper, we discuss the differences between identifying induced and embedded tree patterns in the proposed tree-based representation schemes and we study the relationships among the sets of tree patterns that can be discovered in each case. This paper also describes how these frequent patterns can be used, for instance, to mine association rules in multirelational databases.

Data Mining and Knowledge Discovery manuscript No.
(will be inserted by the editor)

Using trees to mine multirelational databases

Aída Jiménez, Fernando Berzal and Juan-Carlos Cubero

the date of receipt and acceptance should be inserted later

Abstract This paper proposes a new approach to mine multirelational databases. Our approach is based on the representation of multirelational databases as sets of trees, for which we propose two alternative representation schemes. Tree mining techniques can thus be applied as the basis for multirelational data mining techniques, such as multirelational classification or multirelational clustering. We analyze the differences between identifying induced and embedded tree patterns in the proposed tree-based representation schemes and we study the relationships among the sets of tree patterns that can be discovered in each case. This paper also describes how these frequent tree patterns can be used, for instance, to mine association rules in multirelational databases.

Keywords Multirelational databases · Frequent itemset mining · Association rules · Tree pattern mining

Aída Jiménez
Dept. Computer Science and Artificial Intelligence,
ETSIIT - University of Granada, 18071, Granada, Spain
Tel.: +34-958-240599
Fax: +34-958-243317
E-mail: aidajm@decsai.ugr.es

Fernando Berzal
Dept. Computer Science and Artificial Intelligence,
ETSIIT - University of Granada, 18071, Granada, Spain
Tel.: +34-958-240599
Fax: +34-958-243317
E-mail: fberzal@decsai.ugr.es

Juan-Carlos Cubero
Dept. Computer Science and Artificial Intelligence,
ETSIIT - University of Granada, 18071, Granada, Spain
Tel.: +34-958-240597
Fax: +34-958-243317
E-mail: jc.cubero@decsai.ugr.es

basket

id	date	customer	zipcode
1	Thursday	Anna	60608
2	Saturday	Peter	60607
3	Thursday	John	60611

item

basket	product	qty	price
1	toner	2	75
1	printer	1	199
1	netbook	1	299
2	toner	3	75
3	toner	3	75
3	printer	2	199

Fig. 1 Simple multirelational database example.

1 Introduction

Data mining techniques have been developed to extract potentially useful information from databases. Classification, clustering, and association rules have been widely used. However, many existing techniques usually require all the interesting data to be in a single table.

Several techniques have been proposed in the literature to handle several tuples at once. Some algorithms explore tuples that are somehow related, albeit still in the same table (Tung et al, 2003) (Lee and Wang, 2007). Other algorithms, however, are able to extract information from multirelational databases, i.e., they take into account not just a single table, but all the tables that are connected to it (Džeroski, 2003). For instance, these algorithms have been used for multirelational classification (Yin et al, 2004) and multirelational clustering (Yin et al, 2005).

In this paper, we propose two alternative representation schemes for multirelational databases. Our representation schemes are based on trees, so that we can apply existing tree mining techniques to identify frequent patterns in multirelational databases. We also compare the proposed representation schemes to determine which one should be used depending on the information we would like to obtain from the database.

Figure 1 shows an instance of the prototypical multirelational database with two relations: `basket` and `item`. Our approach consists of transforming this database into a set of trees, using one tree for representing each tuple in a particular relation from the multirelational database. The result of this transformation, starting from the `basket` relation, is shown in Figure 2.

A typical solution to multirelational database mining problems consists of joining all the relations of our interest into a single relation, usually called universal relation (Fagin et al, 1982) (Maier and Ullman, 1983) (Maier et al, 1984) (Ullman, 1990). Then, classical data mining techniques can be applied

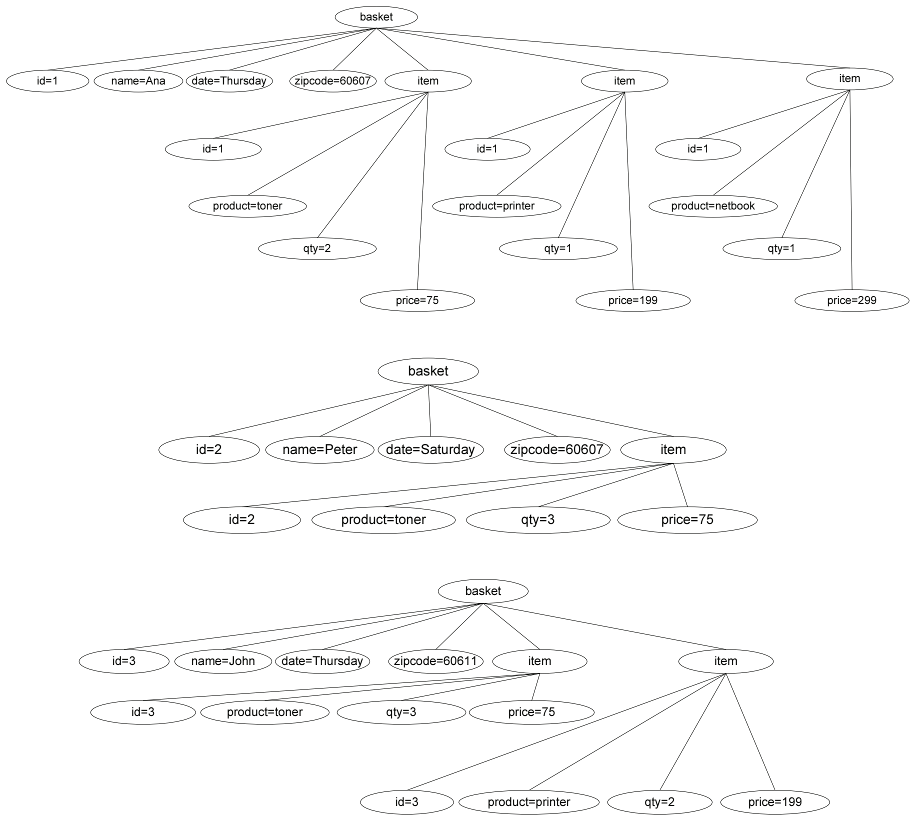


Fig. 2 Tree representation of the multirelational database in Figure 1.

item ⋈ basket

basket	product	qty	price	date	customer	zipcode
1	toner	2	75	Thursday	Anna	60608
1	printer	1	199	Thursday	Anna	60608
1	netbook	1	299	Thursday	Anna	60608
2	toner	3	75	Saturday	Peter	60607
3	toner	3	75	Thursday	John	60611
3	printer	2	199	Thursday	John	60611

Fig. 3 Joined 'universal' relation from the multirelational database in Figure 1.

to this universal relation. However, join-based techniques suffer from a serious disadvantage: they do not preserve the proper support counts. Figure 3 shows the result of joining the `basket` and `item` relations. In the resulting relation, the support of Thursday is 83% and the support of the 60608 zip code is 50%, whereas, if we look at the original `basket` relation, we can see that the actual support of Thursday is 66% and the actual support of the 60608 zip code is 33%. Using our tree-based proposal, we do not introduce such distortions in our support counts and we always preserve their original values.

Our paper is organized as follows. We introduce some standard terms and related work in Section 2. Section 3 describes two different schemes for representing multirelational databases as sets of trees. Some implementation issues when deriving trees from actual multirelational databases are addressed in Section 4. Section 5 studies the kinds of patterns that can be identified from such trees, while Section 6 explains how association rules we can extract association rules defined over the trees obtained from a multirelational database, as well as the constraints that can be used to improve the performance of the association rule mining process. Finally, we present some experimental results in Section 7 and we end our paper with some conclusions in Section 8.

2 Background

In this section, we introduce some basic concepts and published results on tree pattern mining and multirelational data mining.

2.1 Tree pattern mining

A **labeled tree** is a connected acyclic graph that consists of a vertex set V, an edge set $E \subseteq V \times V$, an alphabet Σ for vertex and edge labels, and a labeling function $L : V \cup E \rightarrow \Sigma \cup \varepsilon$, where ε stands for the empty label. The size of a tree is defined as the number of nodes it contains. Optionally, a tree can also have a predefined root, v_0, and a binary ordering relationship $\leq$ defined over its nodes (i.e. '$\leq$' $\subseteq V \times V$).

A **canonical tree representation** is an unique way of representing a labeled tree. This canonical representation makes the problems of tree comparison and subtree enumeration easier. Several alternatives have been proposed in the literature to represent trees as strings (Chi et al, 2003). In this paper, we will use a depth-first-based codification. In a depth-first codification, the string representing a tree is built by adding the label of each tree node in a depth-first order. A special symbol $\uparrow$, which is not in the label alphabet, is used when the sequence comes back from a child to its parent.

Many frequent tree pattern mining algorithms have been proposed in the literature (Jimenez et al, 2010a). These algorithms are usually derived from either Apriori (Agrawal and Srikant, 1994) or FP-Growth (Han et al, 2004). Most tree pattern mining algorithms follow the Apriori iterative pattern mining strategy, where each iteration is broken up into two distinct phases: candidate generation and support counting. Some of these algorithms are FreqT (Abe et al, 2002), TreeMiner (Zaki, 2005b), SLEUTH (Zaki, 2005a), and POTMiner (Jimenez et al, 2010b). Other algorithms follow the FP-Growth pattern growth approach and they do not explicitly generate candidates. For instance, the PathJoin algorithm (Xiao et al, 2003) uses compacted structures called FP-Trees to encode input data, while Chopper and XSpanner (Wang et al, 2004) codify trees as sequences and identify frequent subtrees by discovering frequent subsequences.

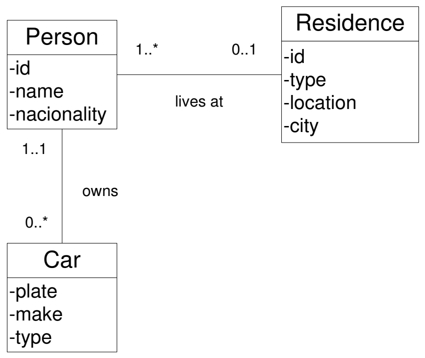


Fig. 4 Multirelational database schema in UML notation.

Person

id	name	nationality	houseID
1	Peter	US	5
…	…	…	

Residence

id	type	location	city
5	apartment	suburb	London
…	…	…	

Car

plate	make	type	ownerID
1234BCD	Toyota	SUV	1
5678JKL	Chevrolet	sedan	1
…	…	…	

Fig. 5 Multirelational representation of the database schema in Figure 4.

2.2 Multirelational data mining

Multirelational data mining techniques look for patterns that involve multiple relations (tables) in a relational database. In a relational database, a relation can be defined as $r = (A^r, K^r, FK^r)$ where $A^r = \{A_1^r, \ldots, A_n^r\}$ is a set of attributes, $K^r \subseteq A^r$ represents the primary key of the relation and $FK^r = \{FK_1^r, \ldots, FK_m^r\}$ is the set of foreign keys in r. Each foreign key can be defined as $FK_i^r = (F_i^r, s_i)$ where $F_i^r \subseteq A^r$ and s_i is the relation whose primary key K^{s_i} is referenced by FK_i^r.

Figure 4 depicts the conceptual schema of a multirelational database using the UML notation (Booch et al, 2005). Given such a conceptual schema, we can derive the suitable logical data model for a relational database (Garcia-Molina et al, 2008) (Silberschatz et al, 2001). The relations obtained from the conceptual model in Figure 4 are shown in Figure 5 with some example tuples.

Many different techniques have been proposed in the literature to deal with multirelational databases in a machine learning context. We have classified the proposed techniques into three broad categories: inductive logic programming, tuple ID propagation, and tree-based algorithms.

- Many multirelational data mining algorithms come from the field of **Inductive Logic Programming** (Džeroski, 2003).
  RELAGGS is a database-oriented approach based on aggregations that collects data from adjacent tables (Krogel and Wrobel, 2003). ACORA (Perlich and Provost, 2006) also employs aggregations to obtain a single table for classification problems.
  Another relevant proposal that deals with multirelational structures is WARMR (King et al, 2001). WARMR employs Datalog, a logic programming language (Ullman, 1988), and it is a level-wise algorithm like Apriori (Agrawal and Srikant, 1994). It has been used to discover frequent patterns in chemical compounds.
  Caraxterix (Turmeaux et al, 2003) was proposed to mine characteristic rules, where characterization consists of discovering properties that characterize objects by taking into account their properties and also properties of the objects linked to them.
- **Tuple ID propagation** is an efficient approach for virtually joining relations, as well as searching and propagating information among them. This technique propagates the IDs of target tuples to other relations, together with their associated class labels where appropriate. A classifier, dubbed CrossMiner (Yin et al, 2004), and a clustering algorithm, called CrossClus (Yin et al, 2005), have been proposed using this technique.
- **Tree-based algorithms** have also been used for multirelational data mining. FAT-miner (De Knijf, 2007), for instance, is a frequent tree pattern mining algorithm that seems to implicitly employ a representation similar to our object-based representation scheme, albeit the particular algorithm employed to derive the trees from a multirelational database is not explicitly described in (De Knijf, 2007) nor (De Knijf, 2006).

Trees have been used in many other data mining areas, although, to the best of our knowledge, our use of trees in this paper has not been formally addressed elsewhere. Decision trees, for example, have been applied for classification and clustering. TILDE (Blockeel and Raedt, 1998) is an ILP-based decision tree inducer. Another algorithm, MRDTL (Leiva et al, 2002), applies decision trees to multirelational databases. In the field of statistical learning, we can also find many other examples, such as probability trees (Neville et al, 2003) and spatiotemporal relational probability trees (McGovern et al, 2008). The former, as our proposal, uses foreign keys to mine multirelational databases for classification purposes.

3 Multirelational database tree representation

In this section, we describe how we can obtain a set of trees from a given multirelational database using two different representation schemes.

The analysis of multirelational databases typically starts from a particular relation. This relation, which we will call *target relation* (or target table), is selected by the end user according to her specific goals.

We introduce two different schemes for representing multirelational databases as sets of trees. The key-based tree representation scheme draws from the concept of identity in the relational model while the object-based representation scheme is based on the concept of identity in object-oriented models. Relational databases rely on primary keys to ensure that each tuple can be univocally referenced within a given relation. In the object model, however, each object is already unique and no specific key is needed. In object databases, each object is automatically assigned an unique ID, which means that you can create objects that have identical field values but are still different objects (Paterson et al, 2006).

The main idea behind our two representation schemes is building a tree from each tuple in the target table and following the links between tables (i.e. the foreign keys) to collect all the information related to each particular tuple in the target table. In both representation schemes, we will use the name of the target table as the label at the root of the trees.

From a database point of view, data can be classified into two types: atomic and compound. Atomic data cannot be decomposed into smaller pieces by the database management system (DBMS). Compound data, consisting of structured combinations of atomic data, can be decomposed by the DBMS (Codd, 1990). We will represent the value a_i of an atomic attribute A_i as $A_i = a_i$ in a tree node and, for compound attributes, as $(A_1, \ldots, A_n)= (a_1, \ldots, a_n)$. In the following sections, with the aim of clarifying the notation, we will suppose that all the attributes are atomic, assuming that, if they were compound, the notation $(A_1, \ldots, A_n)= (a_1, \ldots, a_n)$ should be used instead.

3.1 Key-based tree representation

The key-based tree representation scheme is based on the concept of identity in the relational model, i.e., each tuple is identified by its primary key. Therefore, the root node of the key-based tree representing a tuple in the target relation r will have, as its unique child, the value of the primary key of the tuple in the target relation that is represented by the tree. The children of this primary key node will be the remaining attribute values from the tuple in the target relation. The rest of the tree is then built by exploring the foreign keys that connect the target relation to other relations in the database. From a conceptual point of view, we have two different scenarios:

- When we have a one-to-one or many-to-one relationship between two relations, we will have a foreign key in the target table r pointing to another relation s. That results in a subtree with the data from the tuple in s.
- When we have a one-to-many relationship, we will have a foreign key in the table s that refers to the primary key of table r. In this case, many

tuples in s may point to the same tuple in r. A different subtree results from the data for each tuple in s that points to the target tuple in r.

Formally, the algorithm needed to build a key-based tree starting from each tuple in the target relation $r = (A^r, K^r, FK^r)$ is the following:

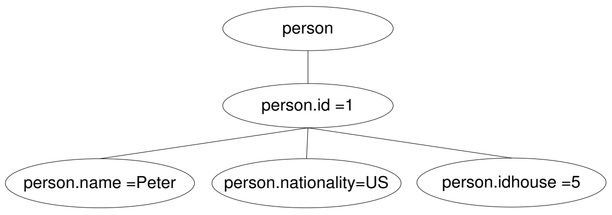
- Build the root node, whose label is the name of the target relation, i.e. r.
- Add a child node to the root corresponding to the primary key of the tuple in the target relation using the notation $r.K^r = k^r$.
- For each attribute $A_i^r \in A^r$, add a child node to the primary key node using the notation $r.A_i^r = a_i^r$
- For each foreign key $FK^r = (F^r, s)$ pointing to another relation, s, create a key-based tree representation for the tuple in s that is referenced by the tuple in r, i.e.:
  - Add a child node to the primary key node using the notation $r.F^r = f^r$.
  - For each attribute $A_i^s \in A^s$ from the s relation, add a child node to the $r.F^r = f^r$ node using the notation $r.F^r.A_i^s = a_i^s$.
- For each foreign key $FK^s = (F^s, r)$ in another relation, s, pointing to our target relation, r, create a key-based tree representation for each tuple in s that points to our tuple in r:
  - Add a child node to the primary key node with the name of the two relations, the foreign key, and the primary key of s using the notation: $r.s[F^s].K^s = k^s$.
  - For each attribute $A_i^s \in A^s$ in the s relation, add a child node to $r.s[F^s].K^s = k^s$ using the notation $r.s[F^s].A_i^s = a_i^s$.
- This algorithm is recursively applied, taking into account the foreign keys in s that point to other relations in the database, as well as the foreign keys pointing to s from other relations in the multirelational database.

Let us suppose that, in the multirelational database of Figure 4, our target table is `person`, its primary key is *id*, and its only attributes are as shown in Figure 5. If we have the tuple {1, Peter, US, 5}, its key-based tree representation will be the one we see in Figure 6 *a)*. In textual form (see Section 2.1), this tree can be represented as follows:

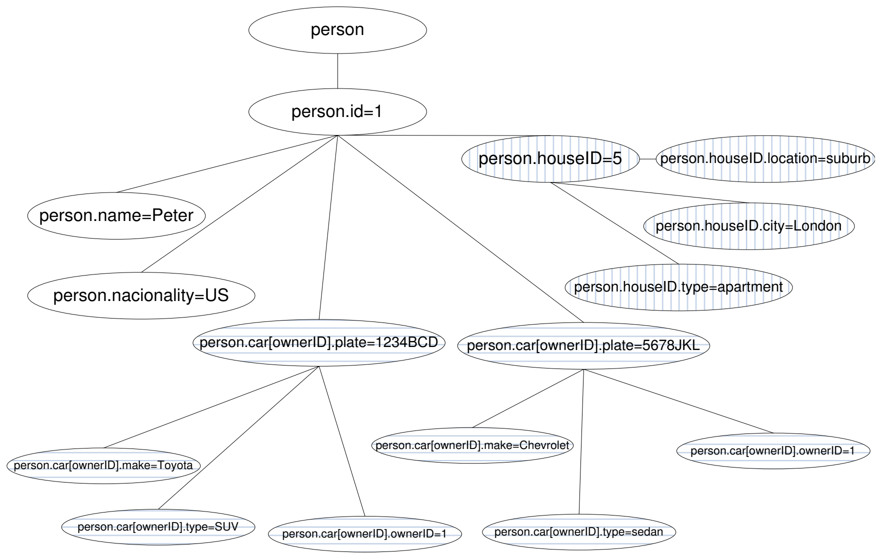
```
person
    person.id=1
        person.name=Peter ↑
        person.email=US ↑
        person.houseID=5 ↑↑
```

When we consider the links (foreign keys) between tables in our example database, we find the two situations we have described above:

- The relationship between `person` and `residence` is one-to-one. Therefore, we have a foreign key in the `person` table that refers to the `house` of each



a) Key-based tree representation of a tuple from the target relation.


person
person.id=1
person.houseID=5
person.houseID.location=suburb
person.houseID.city=London
person.houseID.type=apartment
person.name=Peter
person.nacionality=US
person.car[ownerID].plate=1234BCD
person.car[ownerID].plate=5678JKL
person.car[ownerID].make=Toyota
person.car[ownerID].type=SUV
person.car[ownerID].ownerID=1
person.car[ownerID].make=Chevrolet
person.car[ownerID].type=sedan
person.car[ownerID].ownerID=1

b) Key-based tree representation of data from different relations.

Fig. 6 Key-based tree representation of the multirelational database shown in Figure 5.

`person`. This relationship leads to the subtree in Figure 6 *b)* whose nodes are depicted with vertical lines in their background.

– The relationship between `person` and `car` is one-to-many. Therefore, the `car` table includes a foreign key that refers to the car owner. The key-based tree representation of this relationship is shown by the nodes shaded with horizontal lines in Figure 6 *b)*.

3.2 Object-based tree representation

The object-based tree representation scheme is based on the concept of object identity in an object-oriented model. In this representation scheme, we will use intermediate nodes as roots of the subtrees representing each tuple in the multirelational database. All the attribute values within the tuple, including the primary key attributes, will be children of the root node representing the tuple in the tree.

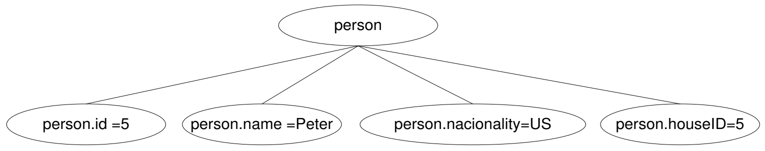
As in the previous section, the tree is built by exploring the relationships between the target relation and other relations in the multirelational database:

- The case of one-to-one and many-to-one relationships between two relations, r and s, where r has a foreign key pointing to s, is now addressed by adding the attributes of s as children of the intermediate node labeled with the names of the foreign key attributes.
- When the relationship is one-to-many, i.e. when the relation s has a foreign key that refers to the target relation r, a new subtree is built for each tuple in s that points to the same tuple in r. These subtrees will have root nodes labeled with the name of both tables and the foreign key involved in the relation, while their children will contain all the attribute values from the tuples in s.

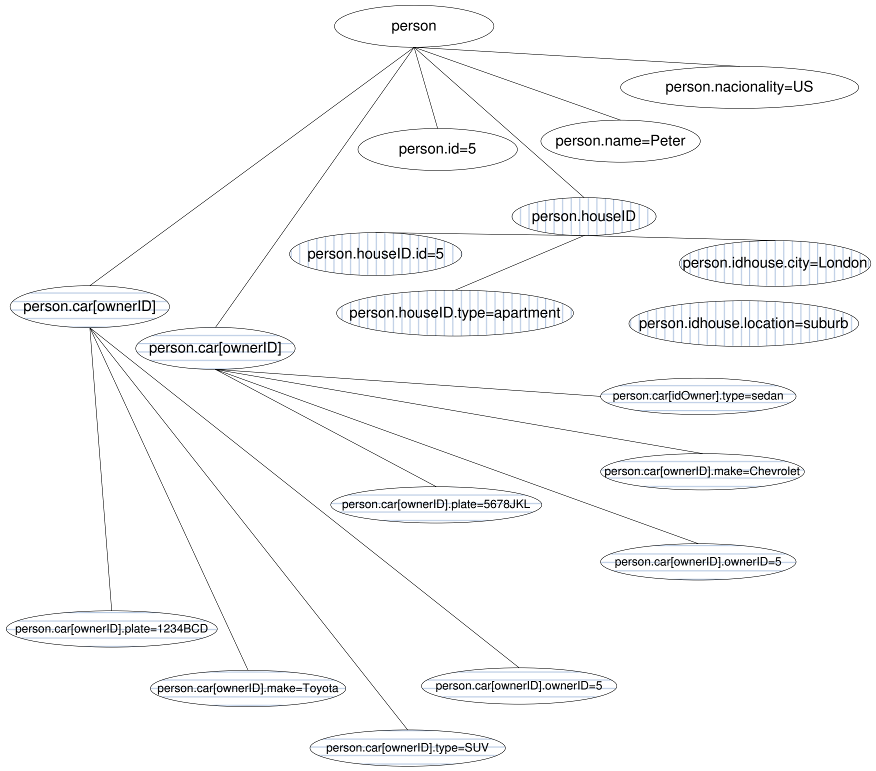
Formally, the algorithm to build a object-based tree starting from each tuple of the target relation $r = (A^r, K^r, F^r)$ is:

- Build the root node, whose label is the name of the target relation, i.e. r.
- For each attribute $A_i^r \in A_r$, including the primary key attributes, add a child node to the root using the notation $r.A_i^r = a_i^r$.
- For each foreign key $FK^r = (F^r, s)$ pointing to another relation, s, create a subtree with the data from the tuple in s that is referenced by the tuple in r:
  - Add a child node to the root using the notation $r.F^r$.
  - For each attribute $A_i^s \in A^s$ in the s relation, including primary key attributes, add a child node to the intermediate node we have just created using the notation $r.F^r.A_i^s = a_i^s$.
- For each foreign key $FK^s = (F^s, r)$ in another relation, s, pointing to our target relation, r, create a subtree for each tuple in s that points to the target tuple in r:
  - Add a child node to the root with the name of both relations and the foreign key attribute names using the notation: $r.s[F^s]$.
  - For each attribute $A_i^s \in A^s$ in the s relation, including the primary key attributes in K^s, add a child node to the $r.s[F^s]$ node that we have just created using the notation $r.s[F^s].A_i^s = a_i^s$.
- This procedure is recursively applied, taking into account the foreign keys in s that point to other relations in the database, as well as the foreign keys pointing to s from other relations in the multirelational database.

The example from Figure 6 is now displayed in Figure 7 using the object-based representation scheme. The nodes decorated with vertical lines in Figure 7 *b)* illustrate the many-to-one relationship between `person` and `residence`, while the one-to-many relationship between `person` and `car` is depicted by the nodes with horizontal lines in their background.



a) Object-based tree representation of a tuple from the target relation.


person
person.nacionality=US
person.name=Peter
person.id=5
person.houseID
person.houseID.id=5
person.idhouse.city=London
person.houseID.type=apartment
person.idhouse.location=suburb
person.car[ownerID]
person.car[ownerID]
person.car[idOwner].type=sedan
person.car[ownerID].make=Chevrolet
person.car[ownerID].plate=5678JKL
person.car[ownerID].ownerID=5
person.car[ownerID].plate=1234BCD
person.car[ownerID].make=Toyota
person.car[ownerID].ownerID=5
person.car[ownerID].type=SUV

b) Object-based tree representation of data from different relations.

Fig. 7 Object-based tree representation of the multirelational database shown in Figure 5.

The main difference between the object-based tree representation scheme and the key-based one is that, in the object-based representation scheme, primary key attribute values and non-prime attribute values appear at the tree level. Using the key-based representation scheme, however, non-prime attribute values within each tuple appear as children of the node representing the primary key.

It should also be noted that the object-based tree representation scheme generates trees with more nodes than the key-based one, since it introduces ancillary intermediate nodes, i.e., those nodes that do not contain values and just represent the start of a new tuple within the tree.

However, tree depth is typically lower in the object-based tree representation scheme than in the key-based one. When representing tuples from the target relation, no key nodes are needed in the object-based representation (i.e., the depth of the resulting tree is 2, as in Figure 6 *a)*). In the key-based representation, however, the primary-key node adds a new level to the tree (i.e., the depth of the resulting tree is 3, as shown in Figure 7 *a)*).

Section 5 will show how the use of these two different tree-based representation schemes will be useful to identify different kinds of patterns, but first we should address some implementation issues that arise in practice.

4 Deriving trees from a multirelational database

In this section, we discuss how we can traverse the foreign keys that connect individual relations in a multirelational database.

4.1 Exploration depth

The connections between relations in a given multirelational database can be represented as a graph whose nodes are relations and whose edges represent foreign keys connecting pairs of relations. Starting from the target relation, we can traverse such a graph. The tree resulting from this traversal will grow each time we visit a new graph node (relation) or revisit an already-discovered one. From the multirelational database perspective, the size and depth of the resulting tree depends on the number of links that we follow, starting from the target relation.

When the multirelational database is complex, the relations that are far away from the target relation might not always be interesting for us. Therefore, in practice, we should select the relations we want to represent in the resulting trees or, at the very least, bound the resulting tree depth.

We can define the **exploration depth** for the tree-based representation of multirelational databases as the length of the longest paths from the target relation to the other relations represented within the trees.

For example, in Figures 6 a) and 7 a), exploration depth is 0, since only the target relation is represented in those trees. In Figures 6 b) and 7 b), however, exploration depth is 1 because we have followed all the foreign keys that connect our target relation to other relations in our database.

4.2 Relationship traversal

When we are building the trees corresponding to each tuple in our target relation, all the foreign keys between the relations represented in the tree are traversed forward starting from the target relation. Apart from this, we must

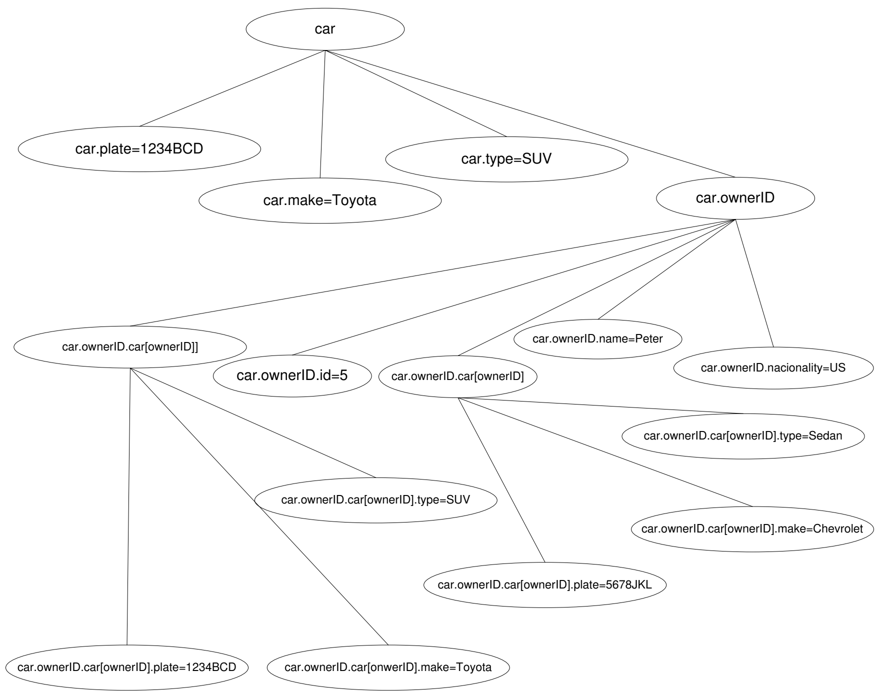

car
car.plate=1234BCD
car.make=Toyota
car.type=SUV
car.ownerID
car.ownerID.car[ownerID]]
car.ownerID.id=5
car.ownerID.car[ownerID]
car.ownerID.name=Peter
car.ownerID.nacionality=US
car.ownerID.car[ownerID].type=Sedan
car.ownerID.car[ownerID].type=SUV
car.ownerID.car[ownerID].make=Chevrolet
car.ownerID.car[ownerID].plate=5678JKL
car.ownerID.car[ownerID].plate=1234BCD
car.ownerID.car[onwerID].make=Toyota

Fig. 8 Object-based tree derived from the multirelational database in Figure 5 using `car` as the target relation.

consider whether it is interesting to go back through a foreign key that is already represented in the tree, i.e. whether to traverse it backwards or not.

To solve this problem, we focus on the relationships in our (high-level) conceptual database schema rather than on the (low-level) foreign keys that appear in our relational data model. When relationships are one-to-one or one-to-many, it is not necessary to traverse them backwards because we would just obtain the same data that we have already have included in the tree. However, if a relation is many-to-one or many-to-many, we should traverse that relationship backwards to obtain all the tuples that are connected to the tuple in our target relation that we are representing in tree format.

For example, consider the object-based tree in Figure 7 b). When we reach the node labeled $person.car[ownerID].ownerID = 5$, we have represented the information about Peter and his cars. Therefore, it is not necessary to go back through the `person-car` relationship because we would obtain the information about Peter that we already have in the tree. However, if the target table were `car`, as shown in Figure 8, we would first represent the information of the Toyota car. Next, we would traverse the `car-person` relationship to obtain the information about the owner of the car (Peter). Finally, we should go back through the `person-car` relationship to represent all the cars that Peter owns, not just the Toyota we started with.

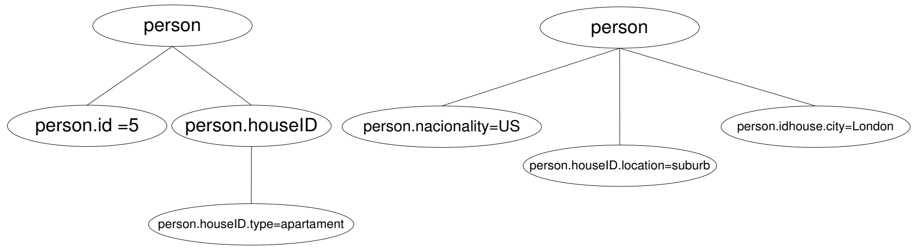

person
person.id =5
person.houseID
person.houseID.type=apartament
person
person.nacionality=US
person.houseID.location=suburb
person.idhouse.city=London

Fig. 9 An induced subtree *(left)* and an embedded subtree *(right)* from the tree shown in Figure 7 *(b)* .

5 Identifying frequent patterns in multirelational databases

The use of tree-based representation schemes for multirelational databases lets us apply tree mining techniques to identify frequent patterns in multirelational databases. Many algorithms have been proposed in the literature to identify frequent tree patterns, including TreeMiner (Zaki, 2005b), SLEUTH (Zaki, 2005a), and POTMiner (Jimenez et al, 2010b). Using these algorithms on the tree-based representation schemes we have introduced in Section 3, we will be able to discover different kinds of patterns. In this section, we will analyze them and we will study the relationships among the sets of patterns that can be discovered from a multirelational database.

5.1 Identifying different kinds of patterns

When working with tree pattern mining algorithms, different kinds of subtrees can be defined depending on the way we define the matching function between the pattern and the tree it derives from (Chi et al, 2005) (Jimenez et al, 2010a):

– A **bottom-up subtree** T' of T, with root v, can be obtained by taking one vertex v from T with all its descendants and their corresponding edges.
– An **induced subtree** T' can be obtained from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T.
– An **embedded subtree** T' can be obtained from a tree T by repeatedly removing nodes, provided that ancestor relationships among the vertices of T are not broken.

Bottom-up subtrees are a special case of induced subtrees. Likewise, induced subtrees are a special case of embedded subtrees. Frequent tree pattern mining algorithms usually focus on identifying induced or embedded subtrees as the ones shown in Figure 9. In the following subsections, we will examine the induced and embedded patterns we can discover when using both the key-based and the object-based tree representation schemes.

5.1.1 Induced key-based patterns

The key-based representation scheme uses primary keys as root nodes for the different subtrees that represent each tuple in the tree. Since those primary key nodes are not frequent in the tree database resulting from the multirelational database and induced patterns preserve all the nodes as in their original trees, no induced patterns starting at, or including, a primary key node will be identified using this representation scheme. However, it is certainly possible that we can identify induced patterns starting at foreign keys, since they might be frequent in the tree database. It should be noted, however, that all the information they contain will usually be from the same relation in our multirelational database.

For example, the `person.id=1` node in Figure 6 *b)* will not be frequent, since `person.id` is the primary key of the target table and it appears in just one database tree. Hence, no induced patterns starting at this node will be identified. It is possible, however, that induced patterns starting at node `person.houseID=5` may be frequent if there were more people sharing the same residence.

5.1.2 Embedded key-based patterns

If we are interested in obtaining patterns involving data from different relations in the key-based representation scheme, we will have to resort to embedded patterns. These patterns will miss some intermediate nodes, such as the primary key nodes, but they will be able to combine information from different relations in our original multirelational database.

For example, we could obtain the pattern in Figure 10 from the tree in Figure 6 *b)*. This pattern represents that people living in suburbs and owning a sedan car are frequent in our database.

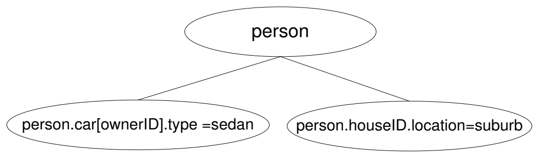

person
person.car[ownerID].type =sedan
person.houseID.location=suburb

Fig. 10 Embedded subtree obtained from the key-based tree in Figure 6 b).

5.1.3 Induced object-based patterns

The object-based representation scheme uses intermediate nodes to represent references to the different tuples represented within the trees. Using this representation scheme, it is possible to identify larger induced patterns than before: Induced object-based patterns can contain information from different relations, something that was not possible when we used the key-based representation scheme.

When appearing as part of frequent patterns, the intermediate nodes provide additional information. They tell us how many tuples in one relation are related to a given tuple in our target relation. For example, the tree pattern in Figure 11 a) shows that people living in suburbs and owning (at least) two cars are frequent in our database, without considering car details. It should be noted that this kind of patterns cannot be identified using the key-based representation scheme.

5.1.4 Embedded object-based patterns

When mining embedded patterns using the object-based representation scheme, we will obtain all the patterns that were identified when we used the key-based representation scheme, as well as those patterns that contain intermediate nodes.

As happened with induced object-based patterns, the use of intermediate nodes will let us identify patterns that were not discoverable using the key-based representation scheme, such as the one shown in Figure 11 b), which is extracted from the tree in Figure 7 b).

However, it should be noted that number of patterns including intermediate nodes could be high because these nodes are typically frequent in the trees representing the multirelational database. Hence, their discovery comes at a price that we will analyze in our experiments in Section 7.

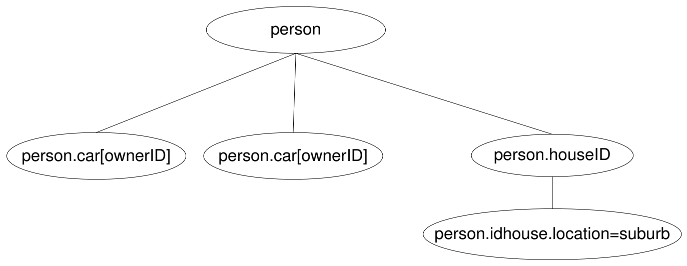
5.2 Induced vs. embedded patterns

Induced and embedded patterns provide us different information about the multirelational database.

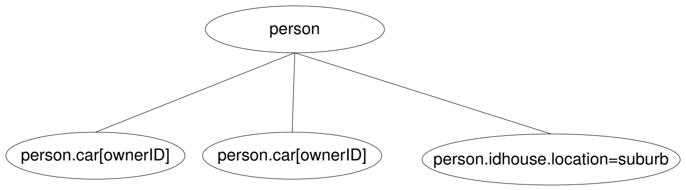
In some sense, induced patterns describe the database in fine detail. Induced patterns preserve the structure of the original trees in the database by maintaining the relationships among all their nodes as they appear in the tree-based representation of the multirelational database. This causes that, in order to obtain useful information from the multirelational database, the identification of large patterns is often necessary. Unfortunately, this need to unveil large patterns might involve a large computational effort.

Embedded patterns are typically smaller than the induced patterns required to represent the same kind of information. However, if we use embedded patterns, some of the relationships among the nodes in the original trees are not preserved. In other words, we might not be able to rebuild the original tree structure from an embedded tree pattern.

For example, Figure 12 shows some patterns obtained from the object-based tree representation scheme of the multirelational database in Figure 4. The induced pattern shown on the left tells us that some people in our database have a Toyota *and* a sedan car, while the induced pattern on the right tells



a) Induced object-based subtree.



b) Embedded object-based subtree.

Fig. 11 Subtrees from the object-based tree in Figure 7 b).

us that people in our database have a Toyota *that* is also a sedan car. The embedded pattern shown in the same figure illustrates that some people have a Toyota car and a sedan car, but we do not know if it is the same car (a Toyota sedan) or they own two cars (a Toyota and a sedan car, which is different from the Toyota). We cannot be sure of the original tree that led to the discovered embedded pattern. In other words, embedded patterns can introduce some ambiguity in their interpretation.

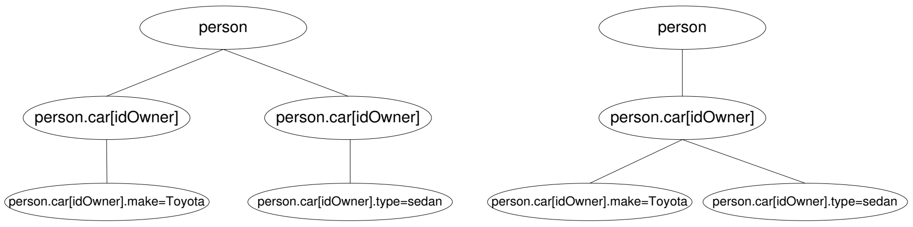
5.3 Key-based vs. object-based patterns

The key-based and the object-based tree representation schemes also provide us different information about the multirelational database.

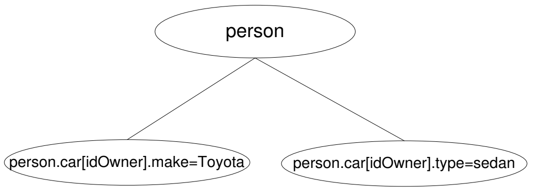
When we use the key-based representation scheme, no induced patterns with information about the target table can be identified. Induced patterns might contain, however, information about tuples in other tables, those that are frequently related to the tuples in the target table.

When we use the object-based representation scheme, induced patterns with information about the target table can now be obtained. Therefore, the object-based representation scheme is our only choice if we are interested in induced patterns, which preserve the original structure of the trees in the database.

On the other hand, using the object-based representation scheme to discover embedded patterns is useful only if we are interested in patterns that


person
person.car[idOwner]
person.car[idOwner]
person.car[idOwner].make=Toyota
person.car[idOwner].type=sedan
person
person.car[idOwner]
person.car[idOwner].make=Toyota
person.car[idOwner].type=sedan

Induced patterns.


person
person.car[idOwner].make=Toyota
person.car[idOwner].type=sedan

Embedded pattern.

Fig. 12 Embedded and induced patterns from the object-based tree representation of the multirelational database in Figure 5.

show that a particular object in a given relation is related to at least n objects in another relation. An example of this kind of pattern is shown in Figure 11 b). That pattern indicates that people living in suburbs with two cars ($n = 2$ in this example) are frequent in our database, without any references to particular car features. This kind of pattern cannot be identified using the key-based representation scheme, since all the nodes in a key-based tree necessarily involve attribute values.

In the object-based representation scheme, however, the presence of intermediate nodes increases the number of identified patterns and, therefore, the computational effort needed to discover them. Hence, we should only resort to the object-based representation scheme when we are interested in patterns similar to the one in Figure 11 b). Otherwise, the key-based representation scheme provides faster results in the discovery of embedded patterns.

5.4 Relationships between kinds of patterns

Once we have discussed the kind of patterns we can obtain using each representation scheme, we will study the relationships between the different sets of patterns that we can identify within a multirelational database.

First of all, we will define an equivalence relationship between key-based trees and object based-trees when they represent the same information from a logical point of view. Let $prefix$ be a substring $t_1 \dots t_m$ of the string representing the label of a node $t_1 \dots t_n$ where $m \leq n$.

Definition 1 *Equivalence between key-based and object-based trees.*

We consider two scenarios to define the equivalence relationship between key-based and object-based trees:

a) When the relation r contains a foreign key that points to the relation s, which might correspond to a one-to-many or a many-to-one relationship, we will say that a key-based tree T_1 =

$$\begin{array}{l} prefix.F^r = K^s \\ \quad prefix.F^r.A_1^s = a_1^s \uparrow \\ \quad prefix.F^r.A_2^s = a_2^s \uparrow \ldots \\ \quad prefix.F^r.A_n^s = a_n^s \uparrow\uparrow \end{array}$$

is ***equivalent*** *($=_{eq}$) to the object-based tree T_3=*

$$\begin{array}{l} prefix.F^r \\ \quad prefix.F^r.K^s = k^s \uparrow \\ \quad prefix.F^r.A_1^s = a_1^s \uparrow \\ \quad prefix.F^r.A_2^s = a_2^s \uparrow \ldots \\ \quad prefix.F^r.A_n^s = a_n^s \uparrow\uparrow \end{array}$$

b) When a foreign key in s points to r, we will say that a key-based tree T_2 =

$$\begin{array}{l} prefix.s[F^s].K^s = k^s \\ \quad prefix.s[F^s].A_1^s = a_1^s \uparrow \\ \quad prefix.s[F^s].A_2^s = a_2^s \uparrow \ldots \\ \quad prefix.s[F^s].A_n^s = a_n^s \uparrow\uparrow \end{array}$$

is ***equivalent*** *($=_{eq}$) to the object-based tree T_4=*

$$\begin{array}{l} prefix.s[F^s] \\ \quad prefix.s[F^s].K^s = k^s \uparrow \\ \quad prefix.s[F^s].A_1^s = a_1^s \uparrow \\ \quad prefix.s[F^s].A_2^s = a_2^s \uparrow \ldots \\ \quad prefix.s[F^s].A_n^s = a_n^s \uparrow\uparrow \end{array}$$

Definition 2 *Equivalence inclusion for sets of tree patterns.*

We say that $A \subseteq_{eq} B$ if and only if, for each element $a \in A$, there exists an element $b \in B$ such that $a =_{eq} b$.

As we have seen before, we can identify four different sets of patterns using either induced or embedded subtrees with both representation schemes: induced key-based patterns (IK), embedded key-based patterns (EK), induced object-based patterns (IO), and embedded object-based patterns (EO). We can identify some relationships among those four sets of patterns by using the following list of properties, whose proofs can be found in the Appendix:

1. All induced key-based patterns are embedded key-based patterns, i.e.,
   Induced key-based patterns $\subseteq$ Embedded key-based patterns.
2. All induced object-based patterns belong to the set of embedded object-based pattern, i.e.,
   Induced object-based patterns $\subseteq$ Embedded object-based patterns.
3. Every induced key-based pattern is equivalent to one pattern that belongs to the set of induced object-based patterns, i.e.,
   Induced key-based patterns $\subset_{eq}$ Induced object-based patterns.
4. Every embedded key-based pattern is equivalent to one pattern that belongs to the set of induced object-based patterns, i.e.,
   Embedded key-based patterns $\subset_{eq}$ Induced object-based patterns.
5. Every embedded key-based pattern is equivalent to one pattern that belongs to the set of embedded object-based patterns., i.e.,
   Embedded key-based patterns $\subset_{eq}$ Embedded object-based patterns.
6. Every induced key-based pattern is equivalent to one pattern that belongs to the set of embedded object-based patterns, i.e.,
   Induced key-based patterns $\subset_{eq}$ Embedded object-based patterns.

In short, we can summarize the relationships among induced key-based, embedded key-based, induced object-based, and embedded object-based patterns as:

$$IK \subseteq EK \subset_{eq} IO \subseteq EO$$

6 Extracting association rules from tree patterns

An association rule is defined as follows for transactional databases: Let $I = i_1, i_2, \ldots, i_m$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. We say that a transaction T contains X, a set of some items in I, if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence c if $c\%$ of transactions in D that contain X also contain Y. The rule $X \Rightarrow Y$ has support s in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$ (Agrawal and Srikant, 1994).

In multirelational databases, we define D as a set of trees. We say that a tree contains X when X is a subtree of T. Let P be a frequent pattern in D. An association rule in a multirelational database is an implication of the form $X \Rightarrow P$, where X is a subtree of P.

6.1 Tree rules

In this section, we explain how to obtain association rules from frequent tree patterns. The kind of rules we obtain (and, therefore, the knowledge they provide us) will depend on the kind of patterns we mine and the tree-based representation scheme we choose, as we analyzed in Sections 5.2 and 5.3.

In association rule mining, once all frequent itemsets are identified, we iterate through all their subitemsets in order to enumerate all potentially-interesting association rules. For each subitemset S of a frequent itemset I of size n, a rule $S \Rightarrow I - S$ can be obtained.

When dealing with tree patterns, we have to enumerate all the subtrees of each frequent tree pattern:

- If we are working with embedded patterns, its embedded subtrees can be obtained by repeatedly removing nodes from the original pattern (all but the root node because, if we removed the root node, we would break the tree).
- In the case of induced patterns, only leaf nodes can be removed in the process, since we must guarantee that the resulting subtree is also induced.

It should be noted that, when working with tree patterns, we cannot represent rules as in transactional databases ($S \Rightarrow I - S$) because:

- $I - S$ may not be a tree (for example, when S includes the root node), and
- we may not know how to match the S subtree with the $I - S$ subtree (in a similar vein to our discussion on matching trees in Figure 12 a)).

Therefore, we will represent the whole pattern I in the consequent of the association rule, albeit its meaning is still analogous to $I - S$ in the traditional sense, since the presence of the pattern I involves the presence of the pattern $I - S$. This way, we obtain rules where both the antecedent and the consequent are trees. Figure 13 shows an example rule that can be obtained from the second induced pattern in Figure 12.

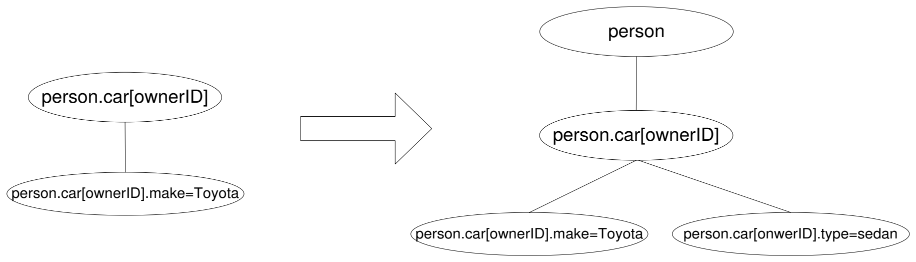

person.car[ownerID]
person.car[ownerID].make=Toyota
person
person.car[ownerID]
person.car[ownerID].make=Toyota
person.car[onwerID].type=sedan

Fig. 13 Rule obtained from the second induced pattern in Figure 12.

6.2 Rule mining constraints

The number of rules that can be obtained from a multirelational database can be huge and most of those rules might not be useful for the end user. In this section, we study how to apply constraints (Pei and Han, 2002) to tree rules in order to reduce the number of rules to be considered.

6.2.1 Rule-specific constraints

Rule-specific constraints (Bayardo, 2004) are based on the measures employed to determine the interestingness of an association rule, like confidence, lift, or certainty factor (Berzal et al, 2002).

A threshold can be established for one or several of these interestingness measures to reduce the number of resulting rules. This constraint lets us reject those rules that do not reach the established thresholds during the rule generation process. Accepting only those rules with confidence above 0.7 (i.e. 70%) is a typical example of this kind of constraint.

6.2.2 Item constraints

An item constraint specifies which groups of items must be present (or not) in the patterns we are looking for. They are typically used to prune the rules so that we obtain only those rules that have a predefined attribute (or attribute value) in their consequent. The use of these constraints in conjunction with length constraints, for instance, is common in associative classification models (Berzal et al, 2004).

Item constraints can also be employed to prune the set of candidate patterns in Apriori-based pattern mining algorithms. When we know which items must be present in every pattern, we can discard the patterns that do not contain the required items (Srikant et al, 1997).

When we apply item constraints at the end of the rule mining process, we can also reduce the number of rules by specifying the items to be present in the antecedent or the consequent of the rules. For example, in order to obtain rules involving the make of a car from the multirelational database in Figure 4, we could prune the rule set by considering only those rules that have the attribute `person[ownerID].car.make` in their consequent.

A special case of item constraints does not only consider one item but rather its relationships to other items: model-based constraints. These special constraints guide our search for patterns that are sub or super-patterns of some given patterns. For instance, in the database from Figure 5, we could look for frequent patterns related to Toyota sedan cars.

6.2.3 Length constraints

A length constraint specifies the number of items in the patterns. It can also be applied, in a more restrictive way, by specifying the number of elements in the consequent of the resulting rules. Length constraints can be useful, for instance, when association rules are used for building classification models, where they should have a single element in their consequent.

Maximum length constraints can be applied during the pattern mining phase to reduce the number of generated patterns. Antecedent or consequent maximum length constraints, however, can only be applied during the rule generation process.

7 Experimental results

In this section, we present some experimental results that we have obtained using both the key-based and the object-based tree representations schemes. In these experiments, we have used both synthetic and actual datasets to study the feasibility and performance of our approach to multirelational database mining.

The synthetic datasets have been created using the database generator provided by Yin at his web page: *http://research.microsoft.com/en-us/people/xyin/*. The parameters of his generator are the number of relations in the database (r), the expected number of tuples in each relation (t), and the expected number of foreign keys in each relation (f). Given particular values for those parameters, the generator produces a relation schema of r relations, one of them being the target relation. The number of attributes in each relation obeys an exponential distribution whose expectation is 5, with a minimum of 2 attributes. The number of different values for each attribute obeys an exponential distribution whose expectation is 10, with a minimum of 2 different attribute values. The number of foreign keys in each relation also obeys an exponential distribution, with expectation f. Finally, the target relation has exactly t tuples and the number of tuples in each nontarget relation obeys an exponential distribution with expectation t and a minimum of 50 tuples (Yin et al, 2004).

For the experiments with actual datasets, we have used three multirelational databases: `mutagenesis`, `loan`, and `genes`. Figures 14 and 15 depict their schemata.

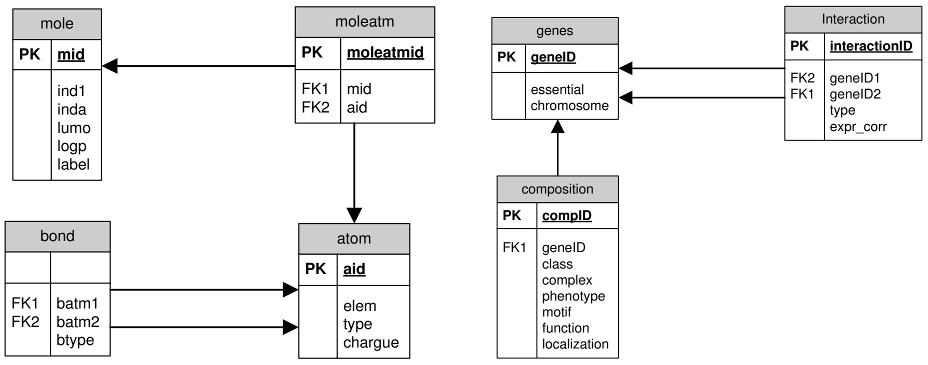


Fig. 14 Schemata of the mutagenesis (left) and genes (right) databases.

The `mutagenesis` database (Srinivasan et al, 1994) is a frequently-used ILP benchmark. It contains four relations and 15,218 tuples. The target relation (`mole`) contains 188 tuples.

The `loan` database was used in the PKDD CUP'09. The database contains eight relations with 75,982 tuples in total. The target relation (`loan`) contains 400 tuples.

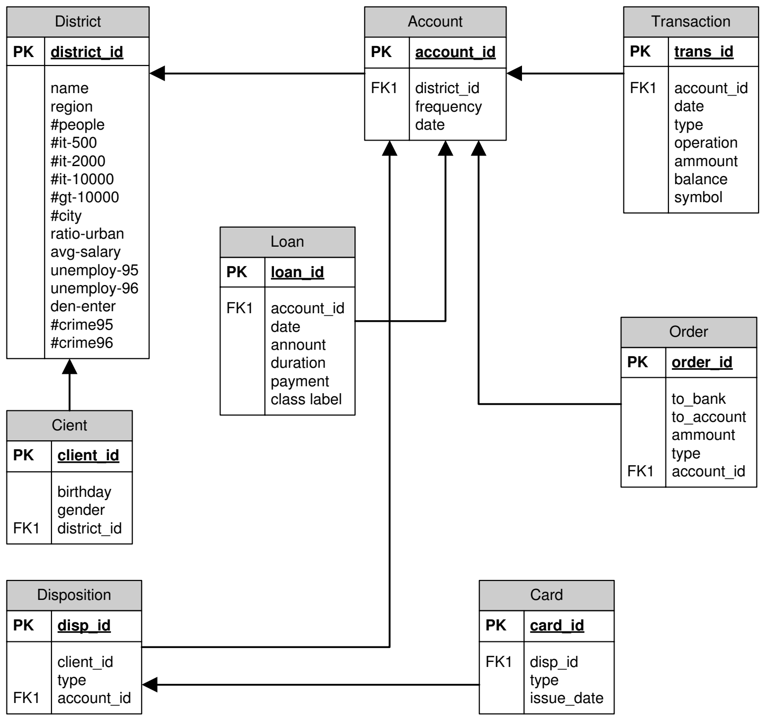


Fig. 15 Schema of the loan database.

The `mutagenesis` and `loan` databases were adapted by Yin et al. for their experiments with CrossMine (Yin et al, 2004) and can be downloaded from: *http://research.microsoft.com/en-us/people/xyin/*

The `genes` database was proposed in the KDD CUP'01. This database can be used to predict the function of genes, thus we have chosen `function` as its class label for our experiments. The target table in the `genes` database is `composition`, which includes 4,636 tuples.

The original `genes` database contained only two relations called `genes` and `interaction`, but the `genes` relation was not normalized, i.e., the table contained 862 different genes but there were several rows in the table for some genes. The normalization of the database was achieved, as proposed by Leiva et al. (Leiva et al, 2002), by creating two tables as follows: attributes in the `genes-relation` table that did not have unique values for each gene were placed in the `composition` table and the rest of the attributes were placed in the `gene` table. The `gene_id` attribute is the primary key in the `gene` table and a foreign key in the `composition` table. The `interaction` relation has not changed with respect to the original one. The difference between our normalization and the one proposed by Leiva et al. is that we found that the attribute `localization` has two different values for the gene with `gene_id`=G239017. Therefore, we have placed this attribute in the `composition` table instead of keeping it in the `genes` table.

Using the proposed tree-based representation schemes for multirelational databases, tree databases can be obtained from each multirelational database. When the exploration depth is 2, we obtain the following tree databases from the original multirelational databases:

– The `mutagenesis` tree database contains 188 trees. Trees have an average of 163 nodes using the key-based representation scheme (30,674 nodes in total) and an average of 189 nodes per tree using the object-based one (35,567 nodes in total).

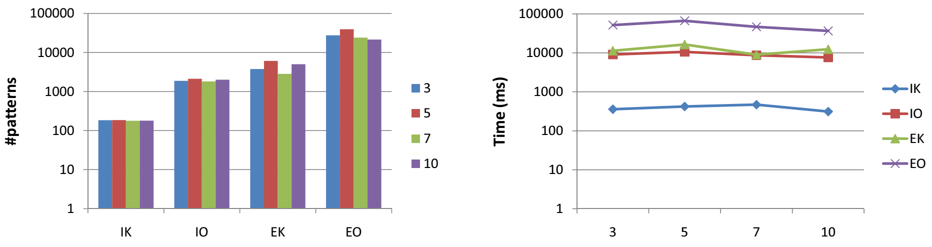


Fig. 16 Number of identified patterns (left) and POTMiner execution time (right) when varying the number of relations in the synthetic database.

- The tree database obtained from the `loan` database contains 400 trees. Trees have an average of 166 nodes using the key-based representation scheme (66,466 nodes in total) and an average of 186 nodes using the object-based one (75,785 nodes in total).
- The tree database obtained from the `genes` database contains 4,636 trees. Trees have an average of 113 nodes using the key-based representation scheme (491,315 nodes in total) and an average of 127 nodes using the object-based one (555,192 nodes in total).

7.1 Identifying induced and embedded patterns

In this section, we present the experiments we have performed to identify frequent patterns in both synthetic and actual databases. We have used the POTMiner tree pattern mining algorithm (Jimenez et al, 2010b) to discover induced and embedded subtrees from the tree-based representation of multirelational databases.

7.1.1 Identifying induced and embedded patterns in synthetic databases

In order to test the performance and scalability of our algorithm, we have performed some series of experiments by varying the parameters of our algorithm (exploration depth, minimum support, and maximum pattern size), as well as the parameters of the synthetic multirelational databases themselves (i.e., number of relations, number of tuples, and number of foreign keys). We have used the following base configuration: #relations=5, #tuples=200, #foreign keys=1, exploration depth=2, support=10%, and maxsize (maximum pattern size)=4.

For each experiment series, we graphically depict the number of identified patterns (up to maxsize), both induced (I) and embedded (E), for each representation scheme, i.e. key-based (K) and object-based (O). We also indicate POTMiner execution time for each case. It should be noted that a logarithmic scale has been used for the Y-axis in all the figures within this section.

First, we have performed some experiments by varying the number of relations: four databases were created with 3, 5, 7, and 10 relations, respectively. Figure 16 shows the results obtained in these experiments. As it can be seen, the number of identified patterns, as well as POTMiner execution time, is more or less independent of the number of relations. The variation in the number of relations does not affect POTMiner execution time because we are not changing the number of trees, which is constant, nor their average size.

We have also performed some experiments by varying the number of tuples in each relation: four databases were created with 100, 200, 300, and 400 tuples, respectively. As we increase the number of tuples, while keeping the number of attribute values constant, the number of frequent patterns exponentially increases as shown in Figure 17. It should also be noted that the number of trees depends on the number of tuples in the target relation of the multirelational database and POTMiner execution time is asymptotically linear with respect to the number of trees in the tree database (Jimenez et al, 2010b).

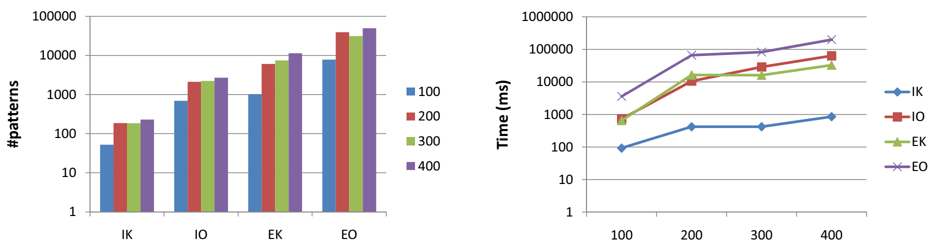


Fig. 17 Number of identified patterns (left) and POTMiner execution time (right) when varying the number of tuples in the relations of the synthetic database.

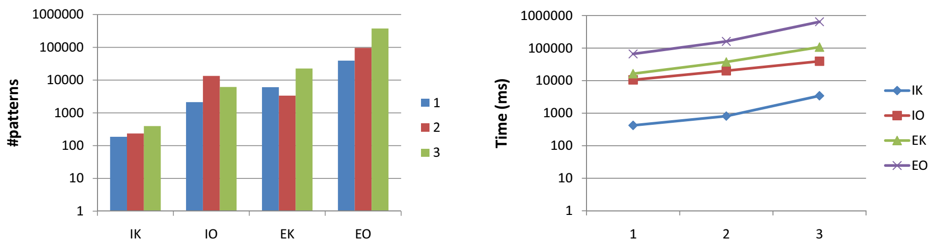


Fig. 18 Number of identified patterns (left) and POTMiner execution time (right) when varying the number of foreign keys in the relations of the synthetic database.

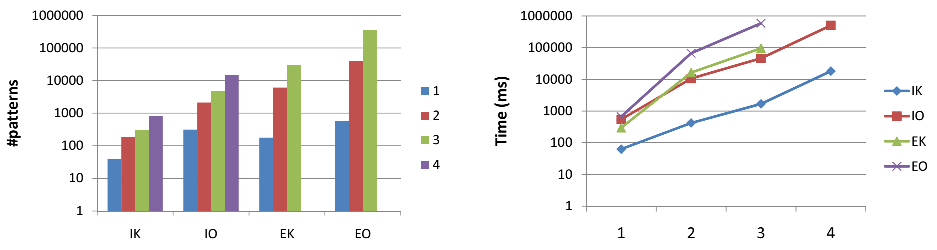


Fig. 19 Number of identified patterns (left) and POTMiner execution time (right) when varying the exploration depth in the synthetic database.

In our experiments varying the number of foreign keys in each relation, shown in Figure 18, three databases were created with 1, 2, and 3 expected foreign keys for each relation. When increasing the number of foreign keys in each relation, tree size increases and more patterns are identified. Then, as in a more traditional frequent pattern mining setting, POTMiner time is proportional to the number of identified patterns.

With respect to the parameters of our algorithm, we first vary the exploration depth (see Section 4.1). We have generated four databases with exploration depths 1, 2, 3, and 4. As shown in Figure 19, the number of identified patterns increases exponentially when we increase the exploration depth because the size of the trees is also increased (we are including information about more relations within them). As in the previous experiment involving the number of foreign keys, POTMiner execution time is proportional to the number of examined patterns (and, hence, exponential with respect to the exploration depth, since the number of patterns exponentially grows with respect to the exploration depth).

The results of our experiments varying the minimum support of frequent patterns are shown in Figure 20. We have extracted patterns using 20%, 10%, and 5% minimum support thresholds. As expected, the number of identified patterns increases when the support decreases and POTMiner execution time is proportional to the increase in the number of identified patterns.

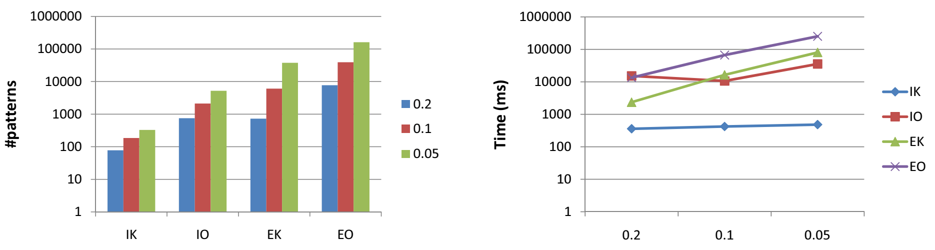


Fig. 20 Number of identified patterns (left) and POTMiner execution time (right) when varying the minimum support threshold for the frequent patterns in the synthetic database.

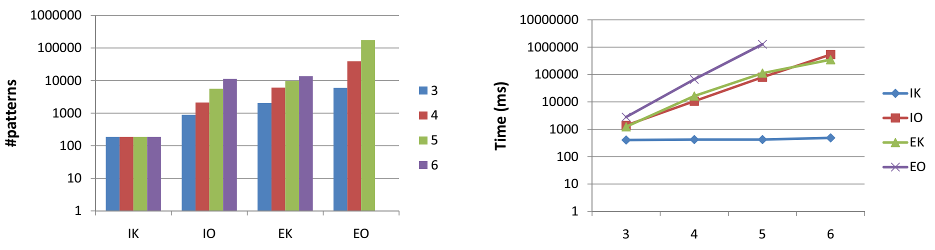


Fig. 21 Number of identified patterns (left) and POTMiner execution time (right) when varying the maximum size of the identified patterns within the synthetic database.

Finally, we have performed a series of experiments by varying the maximum size of the identified patterns. Figure 21 shows the results of these experiments including the number of patterns up to maxsize, which goes from 3 to 6. It should be noted that, in the case of induced key-based patterns, no patterns of size greater than 1 were identified. In the remaining cases, the number of identified patterns and, consequently, POTMiner execution time exponentially increases with the pattern size.

POTMiner execution time is linear with respect to the number of trees in the database and also proportional to the number of identified patterns (Jimenez et al, 2010b). In the case of multirelational databases, the number of tuples in the target relation determines the number of trees in the database, hence POTMiner is linear with respect to the number of tuples in the target relation. Therefore, our algorithm is asymptotically optimal for the problem of mining frequent patterns from multirelational databases using tree-based representation schemes.

However, as in the classical frequent pattern mining problem, the number of identified patterns within a multirelational database can be exponential. The number of foreign keys, the exploration depth, the minimum support threshold, and the maximum pattern size can generate a combinatorial explosion in the number of frequent patterns. Since POTMiner execution time is proportional to the number of identified patterns (Jimenez et al, 2010b), our algorithm execution time can be exponential with respect to the aforementioned parameters. Care should be taken while setting the right values for those parameters in a real-world situation.

7.1.2 Identifying induced and embedded patterns in actual databases

In our experiments with three actual databases (`mutagenesis`, `loan`, and `genes`), we have identified induced and embedded patterns including up to six nodes, i.e., $maxsize = 6$ in POTMiner.

Figure 22 shows the number of induced patterns discovered using different minimum support thresholds for the three datasets, each one obtained with exploration depths 1,

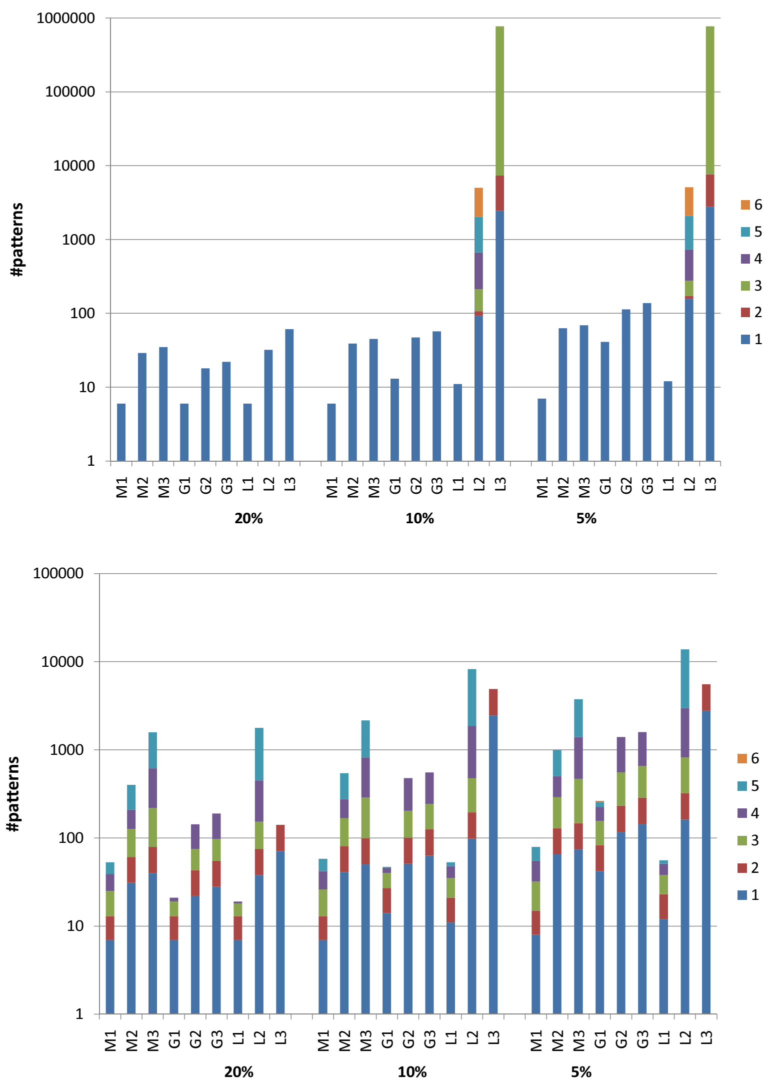
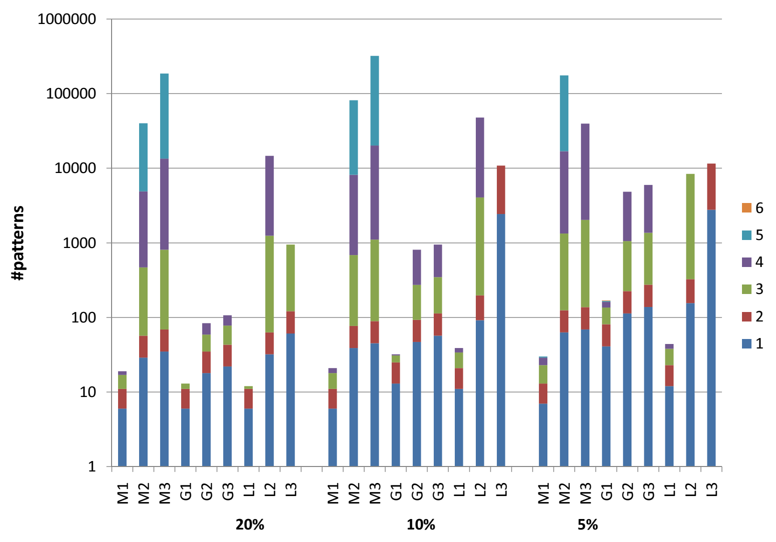


Fig. 22 Number of induced patterns using the key-based (top) and the object-based (bottom) representation schemes for the muta (M), loan (L), and genes (G) databases.

2, and 3, for both the key-based (top) and the object-based (bottom) tree representation schemes.

It should be noted that the number of key-based induced patterns is low and, in most cases, only patterns of size 1 are identified. This is due to the use of primary keys as internal nodes within the trees, which are rarely frequent, as we mentioned in Section 5.3.

The number of embedded frequent patterns in the three databases is shown in Figure 23. The number of discovered patterns using the object-based tree representation scheme is larger than the number of identified patterns using the key-based representation scheme. This is mainly due to the use of intermediate nodes to represent each tuple from the database and the fact that these intermediate nodes are usually frequent.



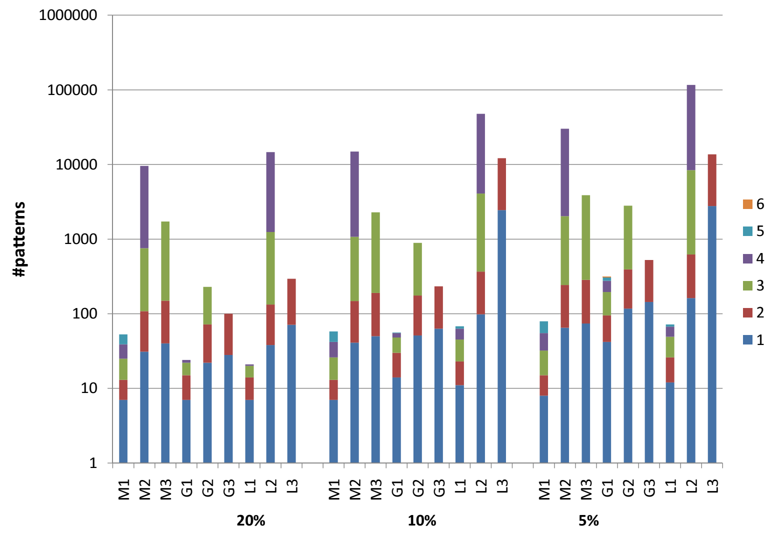


Fig. 23 Number of embedded patterns using the key-based (top) and the object-based (bottom) representation schemes for the muta (M), loan (L), and genes (G) databases.

Figure 24 shows two patterns that have been identified in those databases. The induced object-based pattern at the top is from the `genes` database: 21% of the genes in this database are considered to be *Non-Essential* yet they are involved in *Physical* interactions. The embedded object-based pattern at the bottom of Figure 24 is from the `mutagenesis` database: 47% of the molecules in this database include, at least, three oxygen atoms (in this particular database, all chemical compounds include at least two).

Figure 25 compares the time required to identify induced and embedded patterns using the object-based representation scheme. As we explained in the previous section, POTMiner execution time is proportional to the number of patterns that are examined (Jimenez et al, 2010b). The discovery of induced patterns is faster than the discovery of embedded patterns

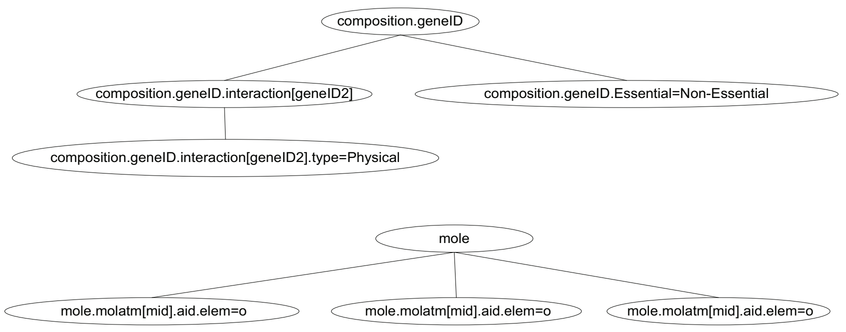

composition.geneID
composition.geneID.interaction[geneID2]
composition.geneID.Essential=Non-Essential
composition.geneID.interaction[geneID2].type=Physical
mole
mole.molatm[mid].aid.elem=o
mole.molatm[mid].aid.elem=o
mole.molatm[mid].aid.elem=o

Fig. 24 Patterns identified in the genes (top) and mutagenesis (bottom) databases.

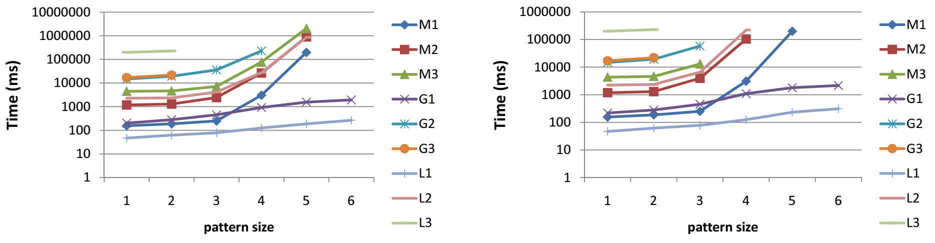


Fig. 25 POTMiner execution time for identifying induced (left) and embedded (right) patterns using the object-based representation scheme for the muta (M), loan (L), and genes (G) databases.

because there is a lower number of them. Likewise, mining object-based patterns requires more execution time because a greater number of patterns is considered, as we analyzed in Section 5.2.

7.2 Extracting rules from frequent tree patterns

In these series of experiments, we have identified induced and embedded frequent patterns of size 4 using both the key-based and the object-based representation schemes, exploration depths from 1 to 3, and a 10% minimum support threshold. We have then used those frequent patterns to extract association rules by varying the minimum confidence threshold (using 0.7, 0.8, and 0.9 as threshold values).

7.2.1 Extracting rules from frequent tree patterns in synthetic databases

In these experiments, we have used the patterns identified in the synthetic database using the base configuration described in the previous section.

Figure 26 shows the number of discovered rules and highlights the rules involving the `class` attribute in the target relation. We compare the number of rules that include the `class` attribute in their consequent with respect to the total number of discovered rules. The `only class` value shows the number of rules that have the `class` attribute as their unique consequent with respect to the number of rules that include the status attribute in their consequent but not necessarily alone. These constraints, as we explained in Section 6.2, are useful when we use association rules to build classification models.

The number of rules obtained from induced patterns in the object-based representation is similar for all the exploration depths in the synthetic datasets. Since induced patterns

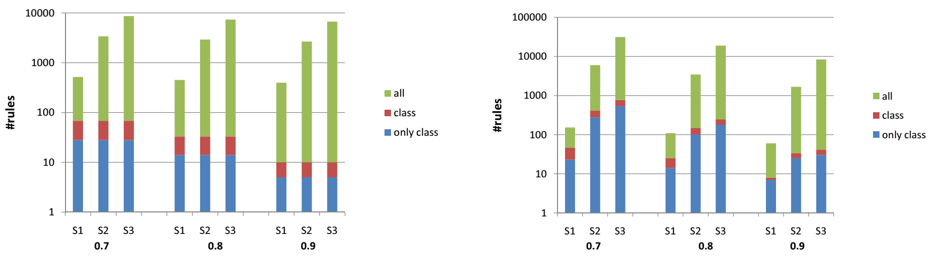


Fig. 26 Number of association rules from the synthetic database using induced object-based patterns (left) and embedded key-based patterns (right) with different minimum confidence thresholds.

preserve the original structure of the database trees, the discovered patterns of size 4 are more or less the same regardless of the chosen exploration depth and, therefore, the resulting rules are also similar. No rules were obtained from induced key-based patterns because all of them were of size 1. More rules were obtained from embedded patterns than from induced patterns, since embedded patterns are a superset of induced patterns.

Albeit not shown in the figures, it should be noted that, once the frequent patterns are identified, the process of extracting rules from them is very fast, just a few seconds for a complete multirelational database.

7.2.2 Extracting rules from frequent tree patterns in actual databases

In our experiments with actual datasets, we have extracted rules from the `mutagenesis`, `loan`, and `genes` databases. We have considered `label`, `status`, and `function`, respectively, as their class attribute.

Figure 27 shows the number of rules resulting from the induced object-based patterns and the embedded key-based patterns derived from the actual databases. Albeit not shown in the figure, as happened with the synthetic datasets, no rules were obtained from the induced key-based patterns for the `mutagenesis` and `genes` databases because no pattern of size greater than 1 was obtained (see Section 7.1). Only a few rules, which did not contain the class attribute, were obtained from the induced key-based representation of the `loan` database.

Figure 28 a) depicts one rule obtained from induced object-based patterns in the `loan` database. This rule, with 65% support and 82% confidence, can be interpreted as: "If we have a `loan` whose associated `account` has a monthly `frequency` of issuance of statements, then the `loan` usually has no problems (i.e. `status`=1)". In other words, of all the loans in our database that have an associated `account` with a monthly `frequency` of issuance of statements, 82% of them have no problems.

Rules derived from embedded patterns provide information that cannot be obtained from the induced patterns. For instance, Figure 28 b) shows a rule from the embedded key-based patterns in the `loan` database. This rule, with 20% support and 82% confidence, states that "if we have a `loan` whose `account` has a `permanent order` of *house* type and its district has one municipality with more than 10000 inhabitants (`num_gt_10000` = 1), then the `loan` usually has no problems (`status`=1)". Hence, it is less frequent in our database to find loans without problems associated to accounts in a district with more than 10000 inhabitants with a `permanent order` of *house* type than loans without problems that have an associated account with monthly `frequency`. However, our confidence about not having problems with those loans is the same in both situations.

The number of association rules from object-based patterns is 3 to 10 times higher than the number of rules derived from key-based patterns. Since, every embedded key-based pattern has an equivalent embedded object-based pattern, as discussed in Section 5.4), all embedded key-based rules have their counterpart in the set of embedded object-based rules.

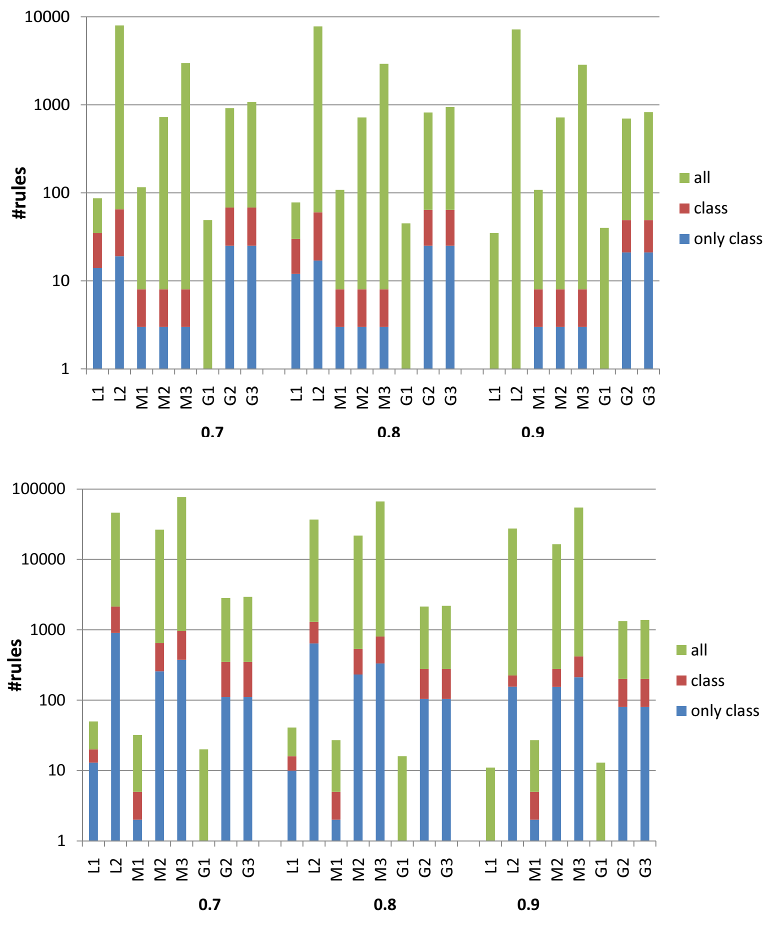


Fig. 27 Number of association rules from the mutagenesis, loan, and genes databases using induced object-based patterns (top) and embedded key-based patterns (bottom) with different minimum confidence thresholds.

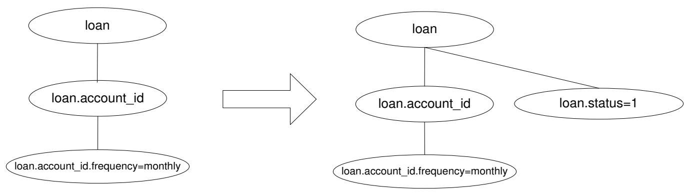
Therefore, a rule equivalent to the rule in Figure 28 b) will be found in the set of embedded object-based rules.

Object-based rules will be specially useful if we are interested in the kind of knowledge we described with the example in Figure 11 from Section 5.1.3. In our experiments, we have obtained a similar rule from the `loan` database, as shown in Figure 28 c): “Within the set of `accounts` with an associated `loan` and two `orders`, 83% of them have a monthly `frequency` of issuance of statements” (support=67%, confidence=83%).

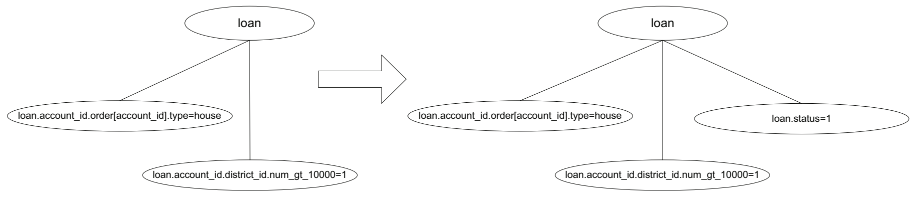
8 Conclusions

This paper proposes a new approach to mine multirelational databases. Our approach is based on representing multirelational databases as sets of trees.

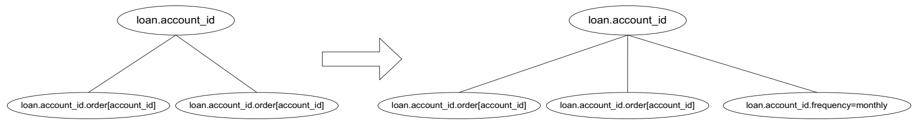
We have designed two alternative tree representation schemes for multirelational databases. The main idea behind both of them is building a tree representing each tuple in the


loan
loan.account_id
loan.account_id.frequency=monthly
loan
loan.account_id
loan.status=1
loan.account_id.frequency=monthly

a) Rule obtained from an induced object-based pattern.


loan
loan.account_id.order[account_id].type=house
loan.account_id.district_id.num_gt_10000=1
loan
loan.account_id.order[account_id].type=house
loan.status=1
loan.account_id.district_id.num_gt_10000=1

b) Rule obtained from an embedded key-based pattern.


loan.account_id
loan.account_id.order[account_id]
loan.account_id.order[account_id]
loan.account_id
loan.account_id.order[account_id]
loan.account_id.order[account_id]
loan.account_id.frequency=monthly

c) Rule obtained from an embedded object-based pattern.

Fig. 28 Some rules obtained during our experiments.

target table (i.e., the most interesting table for the user) by following the foreign keys that connect tables in the relational databases.

The key-based representation scheme uses primary keys at the roots of the subtrees representing each tuple. In contrast, the object-based representation scheme uses generic intermediate nodes to include new tuples in the trees.

We have identified frequent patterns in the trees derived from multirelational databases and we have studied the differences that arise from identifying induced or embedded patterns in both the key-based and the object-based representation schemes. As we explain in Section 5.4 and prove in the appendix, every key-based pattern is equivalent to a pattern that belongs to the set of object-based patterns.

We have also described how tree patterns can be employed to discover association rules from multirelational databases. The performance of this rule mining process can be improved by the use of constraints, which are also useful for reducing the number of rules that are returned to the user.

Our experiments with synthetic and actual datasets show that our approach is feasible in practice, since it can rest on our previous tree pattern mining algorithm, POTMiner, which is linear with respect to the number of trees in the database and whose execution time is proportional to the number of examined patterns, an asymptotically optimal solution for the frequent tree pattern mining problem.

The discovery of induced patterns combined with the object-based representation scheme is often enough to mine multirelational databases. Embedded patterns, when used with the key-based representation scheme, let us reach data that is farther from the target relation, although such patterns do not always preserve the structure of the original database trees and can introduce some ambiguities in their interpretation. Embedded object-based patterns, the most general case we have studied, can discover situations where the number of related tuples is important, regardless of their particular attribute values.

Acknowledgements We would like to thank the anonymous referees for their valuable comments and suggestions, which gave us the chance to improve the quality of this manuscript. This work has been partially supported by research projects TIN2006-07262 (Spanish Ministry of Science and Innovation), TIN2009-08296 (Spanish Ministry of Science and Innovation), and P07-TIC-03175 (Junta de Andalucía).

Appendix: Relationships between kinds of patterns

Property 1 *All induced key-based patterns are embedded key-based patterns, i.e.,*
Induced key-based patterns $\subseteq$ *Embedded key-based patterns.*

Proof *a)* $\forall p \in IK \Rightarrow p \in EK$. By definition, an induced pattern is an embedded pattern where all the parent-children relationships are preserved. Therefore, $\forall p \in IK, p \in EK$, i.e., $IK \subseteq EK$.

b) For patterns of size ≤ 2, $EK = IK$. All key-based patterns of size ≤ 2 have to preserve their only parent-children relationship, when it exists. Therefore $\forall q \in EK$ of size$(q) \leq 2$, $q \in IK$, i.e., $EK = IK$.

c) For patterns of size > 2, $EK \not\subset IK$. Let $q \in EK$ be an embedded key-based pattern that does not preserve all the parent-children relationships. Then, $q \in EK$ but $q \notin IK$. Therefore, $q \in EK \not\Rightarrow q \in IK$.. Hence, $EK \not\subset IK$.

Therefore, $IK \subseteq EK$.

Property 2 *All induced object-based patterns belong to the set of embedded object-based pattern, i.e.,*
Induced object-based patterns $\subseteq$ *Embedded object-based patterns.*

Proof Analogous to the proof of Property 1. Therefore, $IO \subseteq EO$.

Property 3 *Every induced key-based pattern is equivalent to one pattern that belongs to the set of induced object-based patterns, i.e.,*
Induced key-based patterns $\subset_{eq}$ *Induced object-based patterns.*

Proof
a) $IK \subset IO$: $\forall p \in IK \Rightarrow \exists p' \in IO, p =_{eq} p'$.

By induction on the number of foreign keys in the tree:

1. Suppose that the pattern p does not contain any foreign keys, i.e., $\#FK(p) = 0$. Then, $p \in IK$ is of the form

$$\begin{array}{l} r \\ \quad r.K^r = k^r \\ \qquad r.A_1^r = a_1^r \uparrow \dots \\ \qquad r.A_n^r = a_n^r \uparrow\uparrow\uparrow \dots \end{array}$$

and $p' \in IO$ is of the form

$$\begin{array}{l} r \\ \quad r.K^r = k^r \uparrow \\ \quad r.A_1^r = a_1^r \uparrow \dots \\ \quad r.A_n^r = a_n^r \uparrow\uparrow \dots \end{array}$$

Therefore, by Definition 1, $p =_{eq} p'$.

2. Suppose that $q \in IK$ contains k foreign keys, i.e., $\#FK(q) = k$, and $\exists q' \in IO$, $q =_{eq} q'$. Let q be a subtree of p, which contains $k+1$ foreign keys; i.e., $\#FK(p) = k+1$. Then, we have to consider two scenarios depending on the kind of foreign key added to q in order to obtain the pattern p.
   (a) If the foreign key is in the relation r pointing to another relation s, then a subtree of the form T_1 will be added to a node with label $prefix$ in q in order to generate the pattern p. Let $p' \in IO$ be a pattern generated by adding a subtree with the form of T_3 to the pattern q' at the node with label $prefix$. As $T_1 =_{eq} T_3$, $q =_{eq} q'$, and T_1 is attached to q at the same node where T_3 is attached to q', we have $p =_{eq} p'$.
   (b) If the foreign key is in the relation s pointing to our relation, then a tree of the form T_2 will be added to a node with label $prefix$ in q in order to generate the pattern p. Let $p' \in IO$ be a pattern generated by adding a subtree with the form of T_4 to the pattern q' at the node with label $prefix$. As $T_2 =_{eq} T_4$, $q =_{eq} q'$, and T_2 is attached to q at the same node where T_4 is attached to q', we have $p =_{eq} p'$.

   Therefore, $\forall p \in IK$, $\exists p' \in IO$, $p =_{eq} p'$. Hence, $IK \subset IO$.

b) $IO \not\subset_{eq} IK$. As a counterexample, the special object-based patterns that let us know how many tuples are related to a given one cannot be identified using the key-based representation (as the example shown in Figure 11 a)). Hence, $IO \not\subset_{eq} IK$.

Therefore, $IK \subset_{eq} IO$.

Property 4
Every embedded key-based pattern is equivalent to one pattern that belongs to the set of induced object-based patterns, i.e.,
Embedded key-based patterns $\subset_{eq}$ *Induced object-based patterns.*

Proof
a) $EK \subset IO$: $\forall p \in EK \Rightarrow \exists p' \in IO, p =_{eq} p'$.
By induction on the number of foreign keys in the tree:

1. Suppose that the pattern p does not contain any foreign keys, i.e., $\#FK(p) = 0$. Then, $p \in EK$ is of the form

```
r
    r.K^r = k^r
        r.A_1^r = a_1^r ↑ ...
        r.A_n^r = a_n^r ↑↑↑ ...
```

   and $p' \in IO$ is of the form

```
r
    r.K^r = k^r ↑
    r.A_1^r = a_1^r ↑ ...
    r.A_n^r = a_n^r ↑↑↑ ...
```

   Therefore, by Definition 1, $p =_{eq} p'$.
2. Suppose that $q \in EK$ contains k foreign keys, i.e., $\#FK(q) = k$, and $\exists q' \in IO$, $q =_{eq} q'$. Let q be a subtree of p, which contains $k+1$ foreign keys, i.e., $\#FK(p) = k+1$. Then, we have to consider two scenarios depending on the kind of foreign key added to q in order to obtain the pattern p. Furthermore, as p is an embedded pattern, we have to consider whether the subtree f corresponding to the foreign key conserves its root node or note.
   – If the foreign key is in the relation r pointing to another relation s.
     (a) When the subtree f conserves its root node in p (i.e., the node $prefix.A_{FK} = k^s$), then f is of the form T_1 and it will be added to a node with label $prefix$ in q to generate the pattern p. Let $p' \in IO$ be a pattern generated by adding a subtree with the form of T_3 to the pattern q' at the node with the label $prefix$. As $T_1 =_{eq} T_3$, $q =_{eq} q'$, and T_1 is attached to q at the same node that T_3 is attached to q', then $p =_{eq} p'$.

(b) When the subtree f has no root node in p (i.e., we add only some leaf nodes with their attribute values), then:

$prefix.F^r.A^s_1 = a^s_1 \uparrow$
$prefix.F^r.A^s_2 = a^s_2 \uparrow \ldots$
$prefix.F^r.A^s_n = a^s_n \uparrow\uparrow$

are the nodes that have been added as children to the node with the label $prefix$ in q to form p. Then, if we add the same leaf nodes as children to the node with label $prefix$ in q', we will obtain a pattern $p' \in IO$. Therefore, $p =_{eq} p'$.

– If the foreign key is in the relation s pointing to the relation r:

(a) When the subtree f conserves its root node in p (i.e., the node $prefix.s[F^s].K^s = k^s$), then f is of the form T_2 and it will be added to a node with label $prefix$ in q to generate the pattern p. Let $p' \in IO$ be a pattern generated by adding a subtree with the form of T_4 to the pattern q' at the node with the label $prefix$. As $T_2 =_{eq} T_4$, $q =_{eq} q'$, and T_2 is attached to q at the same node that T_4 is attached to q', then $p =_{eq} p'$.

(b) When the subtree f has no root node in p (i.e., we add only some leaf nodes with their attribute values), then:

$prefix.s[F^s].A^s_1 = a^s_1 \uparrow$
$prefix.s[F^s].A^s_2 = a^s_2 \uparrow \ldots$
$prefix.s[F^s].A^s_n = a^s_n \uparrow\uparrow$

are the nodes that have been added as children to the node with the label $prefix$ in q to form p. Then, if we add the same leaf nodes as children to the node with the label $prefix$ in q', we will obtain a pattern $p' \in IO$. Therefore, $p =_{eq} p'$.

Hence, $\forall p \in EK$, $\exists p' \in IO$, $p =_{eq} p'$. Therefore, $EK \subset IO$.

b) $IO \not\subset_{eq} EK$: As a counterexample, the induced object-based patterns that let us know how many tuples are related to a given one cannot be identified using the key-based representation (as the example shown in Figure 11 a)) . Hence, $IO \not\subset_{eq} EK$.

Therefore, $EK \subset_{eq} IO$.

Property 5 *Every embedded key-based pattern is equivalent to one pattern that belongs to the set of embedded object-based patterns., i.e.,*
Embedded key-based patterns $\subset_{eq}$ Embedded object-based patterns.

Proof
a) $EK \subset_{eq} EO$: We have proved that $EK \subset_{eq} IO$ in Property 4 and that $IO \subseteq EO$ in Property 2. Therefore, $EK \subset_{eq} IO \subseteq EO$. By transitivity, $EK \subset_{eq} EO$.
b) $EO \not\subset_{eq} EK$: As a counterexample, the embedded object-based patterns that let us know how many tuples are related to a given one cannot be identified using the key-based representation (as the example shown in Figure 11 b)). Hence, $EO \not\subset_{eq} EK$.

Therefore, $EK \subset_{eq} EO$.

Property 6 *Every induced key-based pattern is equivalent to one pattern that belongs to the set of embedded object-based patterns, i.e.,*
Induced key-based patterns $\subset_{eq}$ Embedded object-based patterns.

Proof
a) $IK \subset_{eq} EO$: We have proved that $IO \subseteq EO$ in Property 2 and also that $IK \subset_{eq} IO$ in Property 3. Therefore, $IK \subset_{eq} IO \subseteq EO$. By transitivity, $IK \subset_{eq} EO$.
b) $EO \not\subset_{eq} IK$: As a counterexample, the embedded object-based patterns that let us know how many tuples are related to a given one cannot be identified using the key-based representation (as the example shown in Figure 11 b)). Hence, $EO \not\subset_{eq} IK$.

Therefore, $IK \subset_{eq} EO$.

References

Abe K, Kawasoe S, Asai T, Arimura H, Arikawa S (2002) Efficient substructure discovery from large semi-structured data. In: Proceedings of the 2nd SIAM International Conference on Data Mining, pp 158–174

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, September 12-15, pp 487–499

Bayardo RJ (2004) The hows, whys, and whens of constraints in itemset and rule discovery. In: Constraint-Based Mining and Inductive Databases, Lecture Notes in Artificial Intelligence, pp 1–13

Berzal F, Blanco I, Sánchez D, Vila MA (2002) Measuring the accuracy and interest of association rules: A new framework. Intelligent Data Analysis 6(3):221–235

Berzal F, Cubero JC, Sánchez D, Serrano JM (2004) ART: A hybrid classification model. Machine Learning 54(1):67–92

Blockeel H, Raedt LD (1998) Top-down induction of first-order logical decision trees. Artifcial Intelligence 101(1-2):285–297

Booch G, Rumbaugh J, Jacobson I (2005) The Unified Modeling Language User Guide (2nd Edition). Addison-Wesley Professional

Chi Y, Yang Y, Muntz RR (2003) Indexing and mining free trees. In: Proceedings of the 3rd IEEE International Conference on Data Mining, pp 509–512

Chi Y, Muntz RR, Nijssen S, Kok JN (2005) Frequent subtree mining - an overview. Fundamenta Informaticae 66(1-2):161–198

Codd EF (1990) The Relational Model for Database Management, Version 2. Addison-Wesley

De Knijf J (2006) FAT-miner: Mining frequent attribute trees. Tech. Rep. UU-CS-2006-053, Department of Information and Computing Sciences, Utrecht University

De Knijf J (2007) FAT-miner: Mining frequent attribute trees. In: Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, New York, NY, USA, pp 417–422

Džeroski S (2003) Multi-relational data mining: An introduction. SIGKDD Explorations Newsletter 5(1):1–16

Fagin R, Mendelzon AO, Ullman JD (1982) A simplied universal relation assumption and its properties. ACM Transactions on Database Systems 7:343–360

Garcia-Molina H, Ullman JD, Widom J (2008) Database Systems: The Complete Book. Pearson Education

Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery 8(1):53–87

Jimenez A, Berzal F, Cubero JC (2010a) Frequent tree pattern mining: A survey. Intelligent Data Analysis 14(6):603–622

Jimenez A, Berzal F, Cubero JC (2010b) POTMiner: Mining ordered, unordered, and partially-ordered trees. Knowlegde and Information System 23(2):199–224

King RD, Srinivasan A, Dehaspe L (2001) Warmr: a data mining tool for chemical data. Journal of Computer-Aided Molecular Design 15(2):173–181

Krogel MA, Wrobel S (2003) Facets of aggregation approaches to propositionalization. In: Horvath T, Yamamoto A (eds) Work-in-Progress Track at the Thirteenth International Conference on Inductive Logic Programming

Lee AJT, Wang CS (2007) An efficient algorithm for mining frequent inter-transaction patterns. Information Sciences 177(17):3453–3476

Leiva HA, Gadia S, Dobbs D (2002) MRDTL: A multi-relational decision tree learning algorithm. In: Proceedings of the 13th International Conference on Inductive Logic Programming, Springer-Verlag, pp 38–56

Maier D, Ullman JD (1983) Maximal objects and the semantics of universal relation databases. ACM Transactions on Database Systems 8:1–14

Maier D, Ullman JD, Vardi MY (1984) On the foundations of the universal relation model. ACM Transactions on Database Systems 9:283–308

McGovern A, Hiers NC, Collier M, II DJG, Brown RA (2008) Spatiotemporal relational probability trees: An introduction. In: Proceedings of the 8th IEEE International Conference on Data Mining, IEEE Computer Society, pp 935–940

Neville J, Jensen D, Friedland L, Hay M (2003) Learning relational probability trees. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 625–630

Paterson J, Edlich S, Hörning H, Hörning R (2006) The Definitive Guide to db4o. Apress

Pei J, Han J (2002) Constrained frequent pattern mining: A pattern-growth view. SIGKDD Explorations Newsletter 4(1):31–39

Perlich C, Provost F (2006) Distribution-based aggregation for relational learning with identifier attributes. Machine Learning 62:65–105

Silberschatz A, Korth HF, Sudarshan S (2001) Database Systems Concepts. McGraw-Hill

Srikant R, Vu Q, Agrawal R (1997) Mining association rules with item constraints. In: Proceedings of the 3rd International Conference of Knowledge Discovery and Data Mining, pp 63–73

Srinivasan A, Muggleton SH, King R, Sternberg M (1994) Mutagenesis: ILP experiments in a non-determinate biological domain. In: Proceedings of the 4th International Workshop on Inductive Logic Programming, volume 237 of GMD-Studien, pp 217–232

Tung AKH, Lu H, Han J, Feng L (2003) Efficient mining of intertransaction association rules. IEEE Transactions on Knowlegde and Data Engeneering 15(1):43–56

Turmeaux T, Salleb A, Vrain C, Cassard D (2003) Learning characteristic rules relying on quantified paths. In: Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases, pp 471–482

Ullman JD (1988) Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems. Computer Science Press, Inc.

Ullman JD (1990) Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies. W. H. Freeman & Co., New York, NY, USA

Wang C, Hong M, Pei J, Zhou H, Wang W, Shi B (2004) Efficient pattern-growth methods for frequent tree pattern mining. In: Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, Lecture Notes in Computer Science, vol 3056, pp 441–451

Xiao Y, Yao JF, Li Z, Dunham MH (2003) Efficient data mining for maximal frequent subtrees. In: Proceedings of the 3rd IEEE International Conference on Data Mining, pp 379–386

Yin X, Han J, Yang J, Yu PS (2004) CrossMine: efficient classification across multiple database relations. In: Proceedings of the 20th International Conference on Data Engineering, pp 399–410

Yin X, Han J, Yu PS (2005) Cross-relational clustering with user's guidance. In: Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining, pp 344–353

Zaki MJ (2005a) Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae 66(1-2):33–52

Zaki MJ (2005b) Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE Transactions on Knowledge and Data Engineering 17(8):1021–1035

2.3 Identifying transposed motifs in music

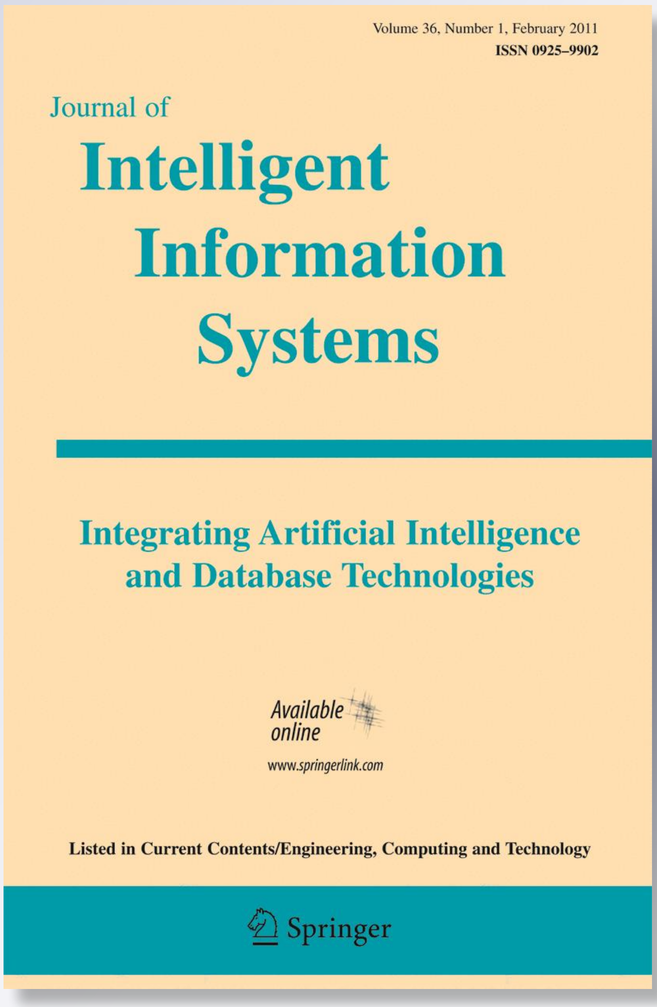
- A. Jiménez, M. Molina-Solana, F. Berzal and W. Fajardo, Mining transposed motifs in music. Journal of Intelligent Information Systems 36:99-115, 2011, DOI 10.1007/s10844-010-0122-7
  - Status: **Published**.
  - Impact Factor (JCR 2009): 0.980.
  - Subject Category:
    - Computer Science, Artificial Intelligence. Ranking 73 / 103.
    - Computer Science, Information Systems. Ranking 72 / 116.

Mining transposed motifs in music

Journal of Intelligent Information Systems
Integrating Artificial Intelligence and Database Technologies

ISSN 0925-9902
Volume 36
Number 1

J Intell Inf Syst (2010)
36:99-115
DOI 10.1007/s10844-010-0122-7



Volume 36, Number 1, February 2011
ISSN 0925–9902

Journal of
Intelligent
Information
Systems

Integrating Artificial Intelligence
and Database Technologies

Available online
www.springerlink.com

Listed in Current Contents/Engineering, Computing and Technology

Springer

Springer

J Intell Inf Syst (2011) 36:99–115
DOI 10.1007/s10844-010-0122-7

Mining transposed motifs in music

Aída Jiménez · Miguel Molina-Solana · Fernando Berzal · Waldo Fajardo

Received: 27 November 2009 / Revised: 23 February 2010 / Accepted: 21 March 2010 /
Published online: 13 April 2010

Abstract The discovery of frequent musical patterns (motifs) is a relevant problem in musicology. This paper introduces an unsupervised algorithm to address this problem in symbolically-represented musical melodies. Our algorithm is able to identify transposed patterns including exact matchings, i.e., null transpositions. We have tested our algorithm on a corpus of songs and the results suggest that our approach is promising, specially when dealing with songs that include non-exact repetitions.

Keywords Musical mining · Motifs · Frequent pattern mining

1 Introduction

The discovery of frequent musical patterns (motifs) is a relevant problem in musicology. In music, we can find several entities that can be repeated such as notes, intervals, rhythms, and harmonic progressions. In other words, music can be seen as a string of musical entities such as notes or chords on which pattern recognition techniques can be applied.

We can define a music motif as *the smallest meaningful melody element*. As a rule, motifs are groups of notes no longer than one measure. In human speech, a motif is a word. In the same way that sentences consist of words, motifs form musical phrases. A melody is formed by several main motifs, which are repeated, developed, and opposed one against another within the melody evolution.

A. Jiménez (✉) · M. Molina-Solana · F. Berzal · W. Fajardo
Centro de Investigación en Tecnologías de la Información y las Comunicaciones,
University of Granada, Granada, Spain
e-mail: aidajm@ugr.es

M. Molina-Solana
e-mail: miguelmolina@ugr.es

F. Berzal
e-mail: berzal@acm.org

W. Fajardo
e-mail: aragorn@ugr.es

When analyzing a music work, musicians carry out a deep analysis of the musical material. This analysis includes motif extraction as a basic task. Musician studies include contextual information (such as the author, the aim, or the period) but also morphological data from the music itself. Looking for the motifs that build the whole work is the first step that a musician takes when faced with a music sheet.

Audio-thumbnailing (i.e., summarizing or abstracting) is another interesting application in the musical domain that is related to motif extraction. It provides the user with a brief excerpt of a song that (ideally) contains the main features of the work. Before hearing or purchasing a whole song, it would be useful to hear a representative thumbnail of the whole work. This technique is also important for indexing large datasets of songs, which can be browsed more quickly and searched more efficiently if indexed by those small patterns instead of being indexed by the whole song.

One of the most fundamental ways to classify MIR methods is to divide them into those that process audio signals using signal processing methods and those that process symbolic representations. We have decided to work with a symbolic representation instead of an audio one because it is closer to the original sheet of music. In other words, the main difficulty with audio representation is that the transformation from audio signals to symbolic data is far from being accurate. This fact makes the pattern recognition problem much more difficult and it requires completely new techniques to deal with signals.

Using the algorithm we present in this paper, we are able to find frequent melodic and rhythmical patterns in music starting from the *MusicXML* representation of the song (www.wikifonia.org). We first transform this symbolic representation into a sequence of notes. These notes are defined at their lowest level (i.e., pitch and duration) and in an absolute, not relative, way.

According to the above considerations, we have developed a TreeMiner-based (Zaki 2005b) algorithm to discover frequent subsequences in music files. Our algorithm is able to identify sequences even when they are transposed. It can be used to find common motifs in several songs and also find repetitions within a song. In this paper, we present its application to the discovery of long motifs that are repeated within a single song. Our hypothesis is that those patterns probably correspond to the chorus or the more significant part of the song.

Our paper is an extended version of a paper presented at the ISMIS'09 conference (Berzal et al. 2009) and is organized as follows. In Section 2, we provide some background on musical data mining and introduce some relevant terms. Section 3 formally defines our sequence pattern mining problem and describes the algorithm we have devised to solve it. In Section 4, we explain the way our algorithm works by means of a particular example. Some experimental results are presented in Section 5, whereas in Section 6 we draw some conclusions.

2 Background

Although it is almost impossible to be exhaustive in analyzing the state of the art in musical pattern identification, we survey the most relevant works in this field in Section 2.1. As our approach is based on sequences, we introduce some standard terms and review some sequence mining algorithms proposed in the literature in Section 2.2.

2.1 Data mining in music

Pattern processing techniques have been applied to musical strings. A complete overview can be found, for instance, in the paper by Cambouropoulos et al. (2001). Those algorithms can be divided into those that deal with audio signals (using signal processing methods) and those that use symbolic representations.

2.1.1 Dealing with audio signals

There are several researchers that have addressed the problem of pattern induction in an acoustic signal. For instance, Aucouturier and Sandler (2002) proposed an algorithm to find repeated patterns in an acoustic signal by focusing on timbre; whereas Chu and Logan (2002) proposed a method to find the most representative pattern in a song using Mel-spectral features.

Recently, some works have gone further in this direction by trying to identify the sectional form of a musical piece from an acoustic signal. For example, Paulus and Klapuri (2009) address this task using a probabilistic fitness measure based on three acoustic features; whereas Levy and Sandler (2008) use clustering methods to extract this sectional structure.

Solving the problem of identifying the structure of a musical piece is key for audio-thumbnailing (i.e., finding a short and representative sample of a song). Zhang and Samadani (2007) addressed this problem by detecting paragraphs in the song with repeated melody in a first step and then identifying vocal portions in the song. With such information, the structure of the song is derived. Another approach, by Bartsch and Wakefield (2005), developed a chroma-based system that searches for structural redundancy within a given song with the aim of identifying something like a chorus or refrain.

2.1.2 Using symbolic representations

There are certain similarities in the use of text and musical data which also allow the application of text mining methods to process musical data. Both have a hierarchical structure and the relative order among the elements is of importance. For that reason, researchers have proposed many different meaningful ways of representing a piece of music as a string, but all of them use either event strings (where each symbol represents an event) or interval strings (where each symbol represents the transformation between events).

Most of the proposed techniques start from a symbolic transcription of music. For example, Hsu et al. (1998) used a dynamic programming technique to find repeating factors in strings representing monophonic melodies; whereas Rolland (1998) recursively computed the distances between large patterns from the distances between smaller patterns. Meredith et al. (2002) proposed a geometric approach to repetition discovery in which the music is represented as a multidimensional dataset. Pienimäki (2002) introduced a text mining based indexing method for symbolic representation of musical data that extracts maximal frequent phrases from musical data and sorts them by their length, frequency and personality.

Finally, the paper by Grachten et al. (2004) is of particular relevance because it represents melodies at a higher level than notes but lower enough to capture the essence of the melody. This level is the 'Narmour patterns' level, based on Narmour's I/R model (1992), which is well-known in musicology.

Our algorithm is also based in a symbolic representation of the song: MusicXML.

2.2 Sequence mining

In our approach for musical motif extraction, we first transform a song into a sequence of notes. There is a rich variety of sequence types, ranging from simple sequences of letters to complex sequences of relations.

A *sequence* over an element type τ is an ordered list $S = s_1 ... s_m$, where:

- each s_i (which can also be written as $S[i]$) is a member of τ, and is called an element of S;
- m is referred to as the length of S and is denoted by $|S|$;
- each number between 1 and $|S|$ is a position in S.

$T = t_1 .... t_n$ is called a *subsequence* of the sequence $S = s_1 ... s_m$ if there exist integers $1 < j_1 < j_2 < ... < j_n < m$ such that $t_1 = s_{j_1}$, $t_2 = s_{j_2}$, and in general, $t_n = s_{j_n}$.

Sequences have been used to solve different problems in the literature (Dong and Pei 2007; Han and Kamber 2005):

- *String matching problems*: Several sequence mining techniques have been used, for instance, in Bioinformatics to find some structures in a DNA sequence (Böckenhauer and Bongartz 2007):
  - Exact string matching: Given two strings, finding the occurrence of one as a substring of the other one.
  - Substring search: Finding all the sequences in a sequence database that contain a particular string as a subsequence.
  - Longest common substring: Finding the substring with maximum length that is common to all the sequences in a given set.
  - String repetition: Finding substrings that appear at least twice in a sequence.
- *Periodic pattern discovery*: A traditional periodic pattern consists of a tuple of k components, each of which is either a literal or '*', where k is the period of the pattern and '*' can be substituted for any literal and is used to enable the representation of partial periodicity (Wang et al. 2001; Yang et al. 2001).
- *Sequence motifs:* A motif is essentially a short distinctive sequential pattern shared by a number of related sequences. There are four main problems in this area (Dong and Pei 2007): motif representation (i.e., designing the proper motif representation for the different applications), motif finding (i.e., finding the motifs shared by several sequences), sequence scoring (i.e., computing the probability of a sequence to be generated by a motif—using Markov models, for example), and sequence explanation (i.e., given a sequence and a motif with hidden states, providing the most likely state path that produced that sequence).
- *Sequential pattern mining in transactional databases:* Sequential patterns have been used for predicting the behavior of individual customers. Each customer is typically modeled by a sequence of transactions containing the set of items he has bought. Several algorithms address this kind of problems, the most common being AprioriAll (Agrawal and Srikant 1994), SPADE (Zaki 2001) GSP (Srikant and Agrawal 1996), and PrefixSpan (Pei et al 2001).

- *Sequential pattern mining in sequence databases*: Some algorithms, such as the one developed by Jiang and Hamilton (2003), look for subsequences that have a larger frequency (or number of repetitions) than an user-defined threshold, which is established beforehand. Jiang's algorithm, for example, uses a tree-based structure called *trie*, that preserves the number of times a subsequence is present in the sequence. Three different versions of his algorithm can be devised:
  - A breadth-first search algorithm passes K times through the data sequence, counting the sequences of size i in its i-th iteration. At the end of each iteration, infrequent subsequences are pruned.
  - A depth-first search algorithm passes just one time through the data using a window of size K which is moved one position at a time. The sequence in the window is preserved in the *trie* structure as well as its prefix. The way this *trie* structure is completely built is very memory consuming without pruning.
  - A heuristic-first search algorithm is a variation of the depth-first algorithm. The number of occurrences of the prefixes of a subsequence is compared to the threshold before inserting the subsequence in the *trie*. If any prefix of the subsequence has not yet been shown to be frequent, then occurrences of the subsequence itself are not counted. This algorithm is more efficient in time and space but it is not able to find all the frequent subsequences.

Our problem of motif extraction in a piece of music could be seen as a particular case of the 'sequential pattern mining in sequence databases' problem. However, the algorithms proposed by Jiang et al. have the drawback that they are limited by the size of the alphabet (i.e., the element type τ according to our notation). This value can be very high in our particular domain, provided that we take different note pitches and durations into account.

Furthermore, we consider the presence of similar sequences (transposed motifs in our problem) that should be counted as if they were exact repetitions. In our case, it is also interesting to know where these repetitions appear in the sequence, specially when considering these similar repetitions. This information would not be given by Jiang's algorithms, as they only count the number of repetitions.

In the following section, we propose a novel TreeMiner-based algorithm to find motifs in a sequence in order to solve the problem of motif extraction in a piece of music. It should be noted, however, that our algorithm can be applied to several sequences at a time, as well as to different kinds of sequence databases (not just musical ones).

3 Our sequence pattern mining algorithm

The goal of frequent sequence pattern mining is the discovery of all the frequent subsequences in a large database of sequences D or in an unique large sequence.

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

We also consider the occurrences of a pattern that approximately match (i.e., those occurrences that are very similar but are not exactly the same). We define the *exact support* of a subsequence as the number of occurrences that are exactly equal to the pattern, whereas the *transposed support* includes both exact and similar occurrences.

We say that a subsequence S is *frequent* if its support is greater than or equal to a predefined minimum support threshold. We define L_k as the set of all frequent k-subsequences (i.e., subsequences of size k).

3.1 SSMiner

Our algorithm, called SSMiner (Similar Sequence Miner), is based on the POTMiner (Jimenez et al. 2009) frequent tree pattern mining algorithm, a TreeMiner-like algorithm for discovering frequent patterns in trees (Zaki 2005b). POTMiner and its antecessor follow the Apriori (Agrawal and Srikant 1994) iterative pattern mining strategy, where each iteration is broken up into two distinct phases:

– *Candidate Generation*: A candidate is a potentially frequent subsequence. In Apriori-like algorithms, candidates are generated from the frequent patterns discovered in the previous iteration. Most Apriori-like algorithms, including ours, generate candidates of size $k + 1$ by merging two patterns of size k having $k - 1$ elements in common.
– *Support Counting*: Given the set of potentially frequent candidates, this phase consists of determining their actual support and keeping only those candidates whose support is above the predefined minimum support threshold (i.e., those candidates that are actually frequent).

The pseudo-code of our algorithm is shown in Fig. 1 and its implementation details will be discussed in Sections 3.2 through 3.4.

The sequence of a song is scanned twice by our algorithm, in the process of obtaining the frequent elements of size 1. The first scan is needed to save the occurrences of each note and the second one is employed to detect the transposed occurrences of each note. Then, the infrequent notes are pruned and we are ready to apply the two phases of the SSMiner algorithm without checking the original sequence any more.

Fig. 1 SSMiner: our sequence mining algorithm

```
algorithm
    Obtain frequent elements (frequent patterns of size 1)
    Build candidate classes $C_1$ from the frequent elements
    for $k$=2 to MaxSize
        for each class $P \in C_{k-1}$
            for each element $p \in P$.
                Compute the frequency of $p$
                if $p$ is frequent
                then
                    Create a new class $P'$ from $p$.
                    Add $P'$ to $C_k$
```

3.2 Candidate generation

We use an equivalence class-based extension method to generate candidates (Zaki 2005a). This method generates $(k+1)$-subsequence candidates by joining two frequent k-subsequences with $k-1$ elements in common.

Two k-subsequences are in the same equivalence class $[P]$ if they share the same prefix string until their $(k-1)$th element. Each element of the class can then be represented as x, where x is the k-th element label.

Elements in the same equivalence class are joined to generate new candidates. This join procedure, called extension in the literature, works as follows. Let (x) and (y) denote two elements in the same class $[P]$, and $[P_x]$ be the set of candidate sequences derived from the sequence that is obtained by adding the element (x) to P. The join procedure results in attaching the element (y) to the sequence generated by adding the element (x) to P, i.e, $(y) \in [P_x]$. Likewise, $(x) \in [P_y]$

3.3 Occurrence lists

Once we have generated the potentially frequent candidates, it is necessary to determine which ones are actually frequent.

The support counting phase in our algorithm follows the strategy of AprioriTID (Agrawal and Srikant 1994). Instead of checking the presence of each candidate in the sequence (which would entail $O(|S|)$ operations), special lists are used to preserve the occurrences of each pattern in the database, thus facilitating the support counting phase.

Each occurrence list contains tuples (t, m, p, d, Θ) where t is the sequence identifier, m stores the elements of the sequence which match those of the $(k-1)$ prefix of the pattern X, p is the position of the last element in the pattern X, d is a position-based parameter used for guaranteeing that elements in the pattern are contiguous within the sequence and Θ indicates the similarity between the occurrence and the original pattern.

When building the scope lists for patterns of size 1, m is empty and the element d is initialized with the position of the pattern only element in the original database sequence. In the first pass through the sequence, exact patterns of size 1 are collected, being its Θ parameter initialized as "=". When similar pattern occurrences are collected in the second pass through the sequence, the parameter Θ is initialized with a value that indicates the similarity between the original pattern and the actual occurrence.

We obtain the occurrence list for a new candidate of size k by joining the lists of the two subsequences of size $k-1$ that were involved in the generation of the candidate. Let $(t_x, m_x, p_x, d_x, \Theta_x)$ and $(t_y, m_y, p_y, d_y, \Theta_y)$ be two tuples to be joined. The join operation proceeds as follows:

if

1. $t_x = t_y = t$ **and**
2. $m_x = m_y = m$ **and**
3. $d_x = 1$ (only if $k \neq 2$) **and**
4. $p_x < p_y$ **and**
5. $\Theta_x = \Theta_y$

then add $[t, m \bigcup \{p_x\}, p_y, d_y - d_x, \Theta_y]$ to the occurrence list of the generated candidate.

3.4 Support counting

Checking if a pattern is frequent consists of counting the elements in its occurrence list. The counting procedure is different depending on whether the weighted support σ_w is considered or not.

- If we count occurrences using the weighted support, all the tuples in the lists must be taken into account.
- If we are not using the weighted support, the support of a pattern is the number of different sequence identifiers within the tuples in the occurrence list of the pattern.

It should be noted that d represents the distance between the last node in the pattern and its prefix m. Therefore, we only have to consider the elements in the scope lists whose d parameter equals 1 for guaranteeing that elements in the pattern are contiguous within the sequence. It is important to remark that the remaining elements in the lists cannot be eliminated because they are needed to build the occurrence lists of larger patterns.

3.5 Representative patterns

Our algorithm returns all the frequent patterns of the maximum size indicated by the user (or smaller ones if there are no patterns of such size). As musical motifs are generally no longer than a measure, a value of ten is typically used by default. Nevertheless, this limit can be easily modified since our algorithm can return all the frequent patterns that exist in the song regardless of their size. The resulting output will be the set of frequent patterns that represent the song. The algorithm also returns the positions of the different occurrences of the patterns within the song (including transposed occurrences if needed).

3.6 SSMiner complexity

SSMiner starts by computing the frequent patterns of size 1. This step is performed by obtaining the vertical representation of the sequence database, i.e., the individual notes that appear in the sequences with their occurrences represented as scope lists. This representation is obtained in linear time with respect to the number of sequences in the database just by scanning it and building the scope lists for patterns of size 1. We then discard the patterns of size 1 that are not frequent. This results in L scope lists corresponding to the L frequent notes in the sequence database and each frequent label leads to a candidate class of size 1.

Let $c(k)$ be the number of classes of size k, which equals the number of frequent patterns of size k, and $e(k)$ the number of elements that might belong to a particular class of size k (i.e., the number of patterns of size $k+1$ that might be included in the class corresponding to a given pattern of size k).

In SSMiner, each sequence pattern grows only by adding an element at the end of the sequence pattern. The number of different sequences of size $k+1$ that can be obtained by the extension of a sequence of size k is $L \cdot k$. Hence, the number of elements in a particular class, $e(k)$, is $O(L)$.

The number of classes of size 1 equals L, the number of frequent labels, so that $c(1) = L$. The classes of size $k+1$ are derived from the frequent elements in classes of size k. In the worst case, when all the $e(k)$ elements are frequent, $c(k+1) = c(k) \cdot e(k)$. Solving the recurrence, we obtain $c(k+1) = c(k) \cdot L = O(L^k)$.

For each considered pattern of size $k+1$, SSMiner must perform a join operation to obtain its scope list from the scope lists of the two patterns of size k that led to it.

The size of the scope list for a pattern of size k is $O(t \cdot e)$ while the computational cost of a scope-list join operation is $O(t \cdot e^2)$, where t is the number of sequences in the database and e is the average number of embeddings of the pattern in each sequence (Zaki 2005a).

In the worst case, the number of embeddings $s(k-1)$ of a pattern of size $k-1$ in a sequence of size S equals the number of subsequences of size $k-1$ within the sequence of size S. This number is bounded by $S-k+1$.

Hence, the cost of the join operation needed for obtaining the scope list of a pattern of size k, is $j(k) = O(t \cdot s(k-1)^2) = O(t \cdot (S-k+1)^2)$.

The cost of obtaining all the frequent patterns of size k will be, therefore, $O(c(k) \cdot j(k)) = O(L^k \cdot t \cdot (S-k+1)^2)$

The total cost of executing the SSMiner algorithm to obtain all the frequent patterns up to $k = MaxSize$ is $\sum_{k=1\ldots MaxSize}(L^k \cdot t \cdot (S-k+1)^2)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e., $k = MaxSize$), SSMiner is $O(L^{MaxSize} \cdot t \cdot (S - MaxSize + 1)^2)$.

Therefore, our algorithm execution time is proportional to the number of sequences in the sequence database ($t = 1$ in our motifs identification problem), and to the number of patterns than can be identified (L^k). Finally, its execution time is quadratic with respect to the size of the sequences (S).

4 An example

In this section, we present an example to help the reader understand the way our algorithm identifies frequent subsequences in a sequence. In order to facilitate the understanding of the procedure, we are not considering the duration of notes. Furthermore, we only take into account those transpositions of fifth.

We will use in this paper the scientific pitch notation which combines a letter-name, accidentals (if any) and a number identifying the pitch's octave. This notation is the most common in English written texts.

Let's suppose we have the following piece of a song: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4 (see Fig. 2), and we want to extract those subsequences that appear at least four times in it.

The first step of our algorithm is scanning the sequence to obtain all the occurrences of each note. Then, the occurrence lists of each note are built as indicated in Section 3.3. Results are shown in Fig. 3.

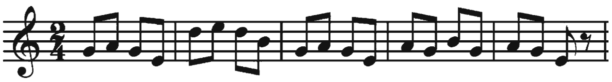


Fig. 2 Sample piece: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4

The first element is 1 in all the tuples because we only have one sequence (i.e., only one song) in our example. The second one is the prefix of the substring (empty in patterns of size 1). The third element indicates the position of the last element of the pattern in the sequence. The fourth element is the distance between the last element of the pattern and its prefix (or the position of the element when there is no prefix, as this is the case). Finally, the last element indicates if the occurrence is exactly equal to the pattern ('=') or if it is transposed.

In this example, only those transpositions of up one fifth are being taken into account, so that '+5' is the only alternative for this element in our example. It should be noted, however, that all possible transpositions—distances between two notes—could be taken into account. In any case, we need only to compare notes in one direction, as we will always find at least a version of the pattern that summarizes all its transpositions.

Going back to our example, the note G4 is transposed up one fifth as D5. Therefore, there are 9 tuples in the occurrence list of G4: 7 as itself, 2 as D5.

The next step is checking if all the notes are frequent. In this case, only G4, A4, and E4 have at least four occurrences. Therefore, only these patterns will be kept.

Figure 4 shows the extension of the element G4. This element is extended with all the frequent patterns of size 1 including itself, and the occurrence lists of each candidate pattern of size 2 are obtained by joining the lists of the elements that generated it, as explained in Section 3.3.

Figure 4 shows, with bold letters, the tuples where $d = 1$. That means that these are contiguous occurrences of the pattern. In our example, only the patterns G4 A4 and G4 E4 appear as contiguous subsequences in our song. Furthermore, they have at least four occurrences—our minimum support threshold—and they will be extended to generate candidates of size 3. It should be noted that the pattern G4 G4

G4	**A4**	**E4**	**D5**	**E5**	**B4**
{1,_,1,1,=}	{1,_,2,2,=}	{1,_,4,4,=}	{1,_,5,5,=}	{1,_,6,6,=}	{1,_,8,8,=}
{1,_,3,3,=}	{1,_,10,10,=}	{1,_,12,12,=}	{1,_,7,7,=}		{1,_,15,15,=}
{1,_,9,9,=}	{1,_,13,13,=}	{1,_,19,19,=}			
{1,_,11,11,=}	{1,_,17,17,=}	{1,_,8,8,+5}			
{1,_,14,14,=}	{1,_,6,6,+5}	{1,_,15,15,+5}			
{1,_,16,16,=}					
{1,_,18,18,=}					
{1,_,5,5,+5}					
{1,_,7,7,+5}					

Fig. 3 Occurrence lists of the elements of the following sequence: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4

Prefix: G4

G4 G4

{1,1,3,2,=}	{1,1,9,8,=}
{1,1,11,10,=}	{1,1,14,13,=}
{1,1,16,15,=}	{1,1,18,17,=}
{1,3,9,6,=}	{1,3,11,8,=}
{1,3,14,11,=}	{1,3,16,13,=}
{1,3,18,15,=}	{1,9,11,2,=}
{1,9,14,5,=}	{1,9,16,7,=}
{1,9,18,9,=}	{1,11,14,3,=}
{1,11,16,5,=}	{1,11,18,7=}
{1,14,16,2,=}	{1,14,18,4,=}
{1,16,18,2,=}	{1,5,7,2,+5}

G4 A4

{1,1,2,1,=}	{1,1,10,9,=}
{1,1,13,12,=}	{1,1,17,16,=}
{1,3,10,7,=}	{1,3,13,10,=}
{1,3,17,14,=}	**{1,9,10,1,=}**
{1,9,13,4,=}	{1,9,17,8,=}
{1,11,13,2,=}	{1,11,17,6,=}
{1,14,17,3,=}	**{1,16,17,1,=}**
{1,5,6,1,+5}	

G4 E4

{1,1,4,3,=}	{1,1,12,11,=}
{1,1,19,18,=}	**{1,3,4,1,=}**
{1,3,12,9,=}	{1,3,19,16,=}
{1,9,12,3,=}	{1,9,19,10,=}
{1,11,12,1,=}	{1,11,19,8,=}
{1,14,19,5,=}	{1,16,19,3,=}
{1,18,19,1,=}	{1,5,8,3,+5}
{1,5,15,10,+5}	**{1,7,8,1,+5}**
{1,7,15,8,+5}	

Fig. 4 Extension of the element G4 in Fig. 3

is not contiguous and will not be extended. However, it is preserved to perform the extension of G4 A4 with G4 E4.

After two more extensions, which are done in the same way, we obtain the pattern G4 A4 G4 E4 with a *support* of 4 and an *exact support* of 3. This is be the pattern we would use to characterize our example song.

5 Experiments

We have tested our algorithm using a corpus of 44 songs. This set includes songs from a wide variety of authors. The first column in Table 1 shows the songs used in our experiments.

We have performed 4 experiments with different constraints:

- Exact pitch and duration (**pitch-duration**)
- Exact pitch and any rhythm (**pitch**)
- Transpositions but exact duration (**transposition-duration**)
- Transpositions and any duration (**transposition**)

Pitch-duration is the most restrictive one, whereas **transposition** is the experiment with a lower number of constraints. All these configurations are representative when looking for musical motifs, as they can be modified in tempo or in pitch. Unlike the example in the former section, all the possible transpositions are taken into account in these experiments.

Table 1 summarizes the results of our experiments. Each row corresponds to one of the songs in the corpus. The second column (*'notes'*) indicates the number of notes in each song.

The *'Max Size'* column indicates the size of the longest pattern(s) found in each song. Patterns of this size are the only ones that are finally returned to the user. As our aim in this paper is searching for repeating motifs, and not whole repeating sections, we have to introduce an upper limit to the size of the patterns. Preliminary

Table 1 Corpus of songs and results for each song in our four series of experiments

Song	Notes	Pich-duration			Pich			Transposition-duration				Transposition			
		Max S	# pat	r_sup	Max S	# pat	r_sup	Max S	# pat	r_sup	t_sup	Max S	# pat	r_sup	t_sup
Alfie	192	7	1 / 41	4	7	1 / 45	5	7	1 / 120	4	4	7	1 / 135	5	5
All good things	248	9	2 / 97	[4–5]	9	2 / 103	[4–5]	9	2 / 151	[4–5]	[4–5]	9	2 / 183	[4–5]	[4–5]
Angels	388	8	1 / 98	4	10	3 / 195	4	8	1 / 232	4	4	10	3 / 267	4	4
Angie	530	6	2 / 106	[4–5]	10	4 / 188	4	6	2 / 401	[4–5]	[4–5]	10	4 / 455	4	4
Apologize	384	10	9 / 217	4	10	10 /209	4	10	9 / 318	4	4	10	10 / 283	4	4
Bach	182	5	2 / 38	4	5	2 / 44	4	10	2 / 178	2	5	9	1 / 189	2	4
Ballade pour Adeline	453	10	33 / 376	[4–12]	10	36 / 353	[4–30]	10	33 / 566	[4–18]	[4–27]	10	36 / 528	[4–36]	[4–52]
Beat it	171	4	5 / 42	[4–5]	6	1 / 51	4	4	5 / 62	[4–5]	[4–6]	7	1 / 68	2	4
Beautiful	332	9	1 / 82	4	10	1 / 112	4	9	1 / 331	4	4	10	1 / 265	4	4
Beautiful that way	268	10	20 / 243	4	10	20 / 235	[4–5]	10	20 / 316	4	4	10	20 / 356	4	4
Bleeding love	539	9	1 / 148	4	9	2 / 185	[4-5]	9	1 / 277	4	4	9	2 / 320	[4–5]	[4–5]
Brown eyed girl	138	3	3 / 25	[4–7]	5	1 / 34	4	4	1 / 54	3	4	5	2 / 71	[3–4]	[4–5]
Crazy	307	5	2 / 81	4	6	2 / 85	4	6	2 / 285	[1–3]	4	6	3 / 201	[3–4]	4
Don’t speak	334	9	1 / 97	4	10	2 / 145	4	9	1 / 244	4	4	10	2 / 217	4	4
Every breath you take	295	6	1 / 56	7	6	1 / 68	4	8	1 / 241	3	5	8	4 / 265	[3–4]	[4–5]
Everybody hurts	271	4	1 / 46	4	8	3 / 83	4	4	2 / 121	[1–4]	4	6	1 / 149	4	4
Feel	394	10	9 / 159	[5–6]	10	13 / 172	4	10	9 / 393	[5–6]	[5–6]	10	13 / 364	[4–7]	[4–7]
Fever	121	8	1 / 31	4	8	1 / 37	[4–7]	8	1 / 45	4	4	8	1 / 54	4	4
Fur Elise	267	10	16 / 202	[4–6]	6	16 / 77	4	10	16 / 396	[4–6]	[4–6]	10	16 / 391	[4–6]	[4–6]
Hero	259	5	1 / 56	4	10	1 / 141	5	6	1 / 220	1	4	6	1 / 214	5	5
How to save a life	448	7	1 / 102	5	10	3 / 100	5	7	1 / 251	5	5	10	1 / 226	5	5
I hate this part	231	10	3 / 94	4	10	6 / 103	[4–7]	10	6 / 301	[1–4]	[4–5]	10	6 / 333	[1–4]	[4–5]
I say a little prayer	172	4	2 / 32	4	4	3 / 39	4	5	2 / 95	3	[4–5]	5	1 / 92	2	5
Imagine	188	5	1 / 44	5	6	1 / 49	[4–5]	6	1 / 93	1	4	10	6 / 150	[4–5]	[4–5]

Let it be	179	5	1 / 32	4	10	1 /49	4	5	1 / 84	4	4	6	1 / 129	4	4
Livin la Vida Loca	434	10	19 / 342	[4–6]	5	22 /345	[4–8]	10	19 / 507	[4–6]	[4–6]	10	23 / 455	[2–8]	[4–8]
Love me tender	53	2	2 / 7	[4–6]	4	1 / 33	4	5	1 / 28	1	4	5	2 / 33	[1–4]	4
Paint it black	129	2	3 / 19	[4–6]	10	16 / 205	5	3	4 / 51	2	[4–6]	5	1 / 48	2	4
Quizas quizas quizas	156	8	1 / 61	6	9	1 / 70	[4–6]	8	1 / 146	6	6	9	1 / 135	6	6
Roxanne	396	10	18 / 213	[4–10]	10	20 / 226	6	10	18 / 270	[4–10]	[4–10]	10	20 / 256	[4–10]	[4–10]
Smile	105	4	2 / 30	[4–6]	4	4 / 33	[4–10]	10	4 / 211	2	4	10	1 / 91	2	4
Sunshine of my life	352	10	26 / 308	[5–6]	10	27 / 305	[5–6]	10	26 / 419	[5–6]	[5–6]	10	27 / 409	[5–6]	[5–6]
The nearness of you	125	4	1 / 24	4	5	3 / 36	4	5	1 / 82	1	4	5	4 / 90	[1–4]	4
Torn	388	10	11 / 194	[4–5]	10	11 / 200	[4–5]	10	11 / 275	[4–5]	[4–5]	10	11 / 271	[4–5]	[4–5]
Under the bridge	541	10	9 / 246	4	10	14 / 291	4	10	9 / 661	4	4	10	14 / 889	[4–6]	[4–6]
Underneath your clothes	260	5	1 / 55	4	5	3 / 67	[4–6]	5	1 / 206	4	4	7	1 / 224	2	4
Viva la vida	444	10	1 / 165	4	10	13 / 235	[4–5]	10	1 / 366	4	4	10	13 / 476	[4–5]	[4–5]
We're the champions	303	5	1 / 62	4	8	1 / 93	4	5	1 / 206	4	4	8	2 / 256	[2–4]	4
What a wonderful world	179	7	1 / 54	4	8	1 / 61	6	7	1 / 149	4	4	8	1 / 109	6	6
When I'm sixtyfour	185	6	1 / 30	4	7	1 / 41	4	7	1 / 116	1	4	7	2 / 186	[1–4]	[4–5]
With or without you	236	5	1 / 44	4	5	1 / 57	4	5	1 / 204	4	4	5	2 / 165	[1–4]	[4–5]
Wonderwall	184	9	1 / 68	4	9	1 / 75	4	9	1 / 121	4	4	9	1 / 135	4	4
Your song	236	5	1 / 43	4	5	1 / 51	4	5	1 / 118	4	4	5	1 / 96	4	4
Zombie	239	10	3 / 92	4	10	3 /100	4	10	3 / 130	4	4	10	3 / 141	4	4

tests with our song corpus have shown that a value of ten is adequate for this parameter providing that musical motifs are generally no longer than a measure.

The *'Patterns'* column indicates the number of patterns our algorithm finds. The first number is the amount of patterns of maximum size (i.e. the size indicated on the previous column); whereas the second number is the total amount of patterns, regardless of their size. As the reader can see, the total amount of patterns is pretty high. However, many of these patterns are not musically relevant since they are part of bigger ones. Even more, patterns of size 1 and 2, which can hardly be considered as motifs, are also included in this set.

The *'exact support'* column is the exact support of the returned patterns. Intervals appear in some cases due to the fact that not all the returned patterns necessarily appear the same number of times.

Finally, *'transposed support'* indicates the support of patterns including transpositions. This column is only relevant for the experiments **transposition-duration** and **transposition**, when transpositions are taken into account. As expected, values in this column are always equal or greater than those in the corresponding *'exact support'* column.

For us, the minimum support required for a pattern to be considered frequent is four. Any pattern with a lower number of repetitions will be deleted. This value has been manually set regarding the size of the evaluating songs and some preliminary tests. However, it can be easily adjusted when new datasets require it.

The reader can also notice that, in some cases (namely, when transpositions are taken into account), patterns with *exact support* lower than four can be found. Those patterns do not have enough exact repetitions as themselves, but they are frequent when transpositions are taken into account. Hence, transpositions are important because, without considering them, some patterns would have not been discovered, as they do not reach the minimum support threshold just by exact repetitions. That happens, for instance, with the 'Crazy' and 'Hero' songs.

Regarding the length of the excerpts that have been tested, three notes can hardly represent any meaningful motif. However, almost anyone could identify Beethoven's Fifth Symphony by just four notes. Hence, a minimum length of four seems adequate.

It should be noted that, in some situations, there is another transposition of the pattern that has greater *exact support* than the one returned by our algorithm as described in Section 3. As we mentioned earlier, our algorithm looks for transposed motives only in one direction and this suffices to guarantee that it will find all the relevant occurrences; however, the returned patterns are not necessarily the most frequent exact ones. Given that our algorithm keeps track of the occurrences of a given pattern and all of its transpositions, it is trivial to obtain the most frequent exact pattern just by looking at the corresponding scope list. This pattern will correspond to the most common Θ in the scope list.

Table 2 Percentage of songs that include at least one identified pattern within their chorus

	Pitch-duration	Pitch	Transposition-duration	Transposition
%Yes	63.64	68.18	63.64	72.73
%No	29.55	25.00	29.55	20.45
%Without chorus	6.82	6.82	6.82	6.82

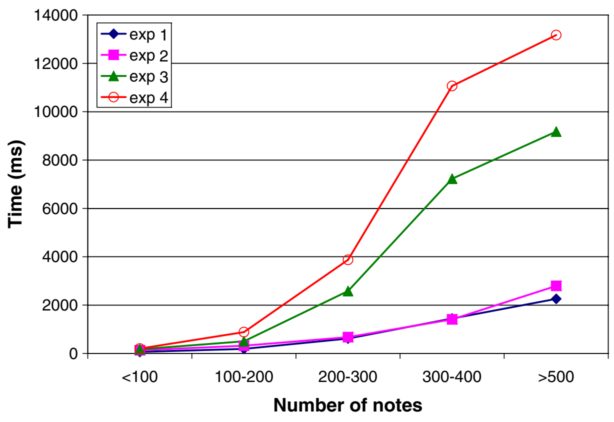


Fig. 5 SSMiner execution time

It should also be observed that, in some songs, the number of patterns of *MaxSize* elements is pretty high (e.g. 'Ballade pour Adeline' or Beethoven's 'Für Elise'). This is due to the fact that many of those are still subpatterns of bigger ones. For instance, a pattern of size 15 includes six sequential subpatterns of size 10 (starting from the 1st, 2nd, 3rd, 4th, 5th and 6th note, respectively).

In order to evaluate the goodness of our method, we have checked whether or not the discovered frequent patterns belong to the chorus of the song—in our experiments, 6.82% of the songs do not have a clear chorus. Table 2 shows the percentage of songs which have at least one identified pattern within their chorus. As can be seen, above 60% of the songs fulfill this requirement. Also, it is remarkable that not considering the rhythm results in more patterns belonging to the chorus of the songs. This fact indicates that patterns are not always exactly repeated as themselves, but slightly modified. Although the chorus-belonging criterion appears to be a valid and obvious one, it should be noted that some songs are better identified by patterns which do not belong to the chorus.

Regarding SSMiner computation time, Fig. 5 shows the time consumed in each experiment with respect to the number of notes in the melody. The chart groups the songs into five groups according to their lengths and displays the average execution time for each subset of melodies. These execution times are quadratic with respect to the number of notes in the melodies, as explained in Section 3.6.

6 Conclusions

We have presented the application of frequent pattern mining to the discovery of musical motifs in a piece of music. *MusicXML* files, which can be easily collected, are transformed into sequences of notes, defined at their lower level. Our algorithm, SSMiner, is able to efficiently identify frequent subsequences in a sequence.

The matching between the patterns does not need to be exact. Our algorithm is able to identify transposed patterns including exact matchings, i.e., null transpositions. Our experiments suggest that our approach performs well in a set of randomly-selected songs.

In the future, we intend to employ interval strings to represent notes rather than the absolute pitches we have used in the experiments reported in this paper. We will also consider more abstract representations of melodies, such as the one proposed by Narmour. Finally, we plan to study the parallelization of our algorithm implementation in order to improve its execution time, which is already asymptotically optimal.

Acknowledgements F. Berzal and A. Jiménez are supported by the projects TIN2006-07262 and TIN2009-08296, whereas W. Fajardo and M. Molina-Solana are supported by the research project TIN2006-15041-C04-01.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th int. conf. on very large data bases* (pp. 487–499).

Aucouturier, J. J., & Sandler, M. (2002). Finding repeating patterns in acoustic musical signals: Applications for audio thumbnailing. In *Audio engineering 22nd int. conf. on virtual, synthetic and entertainment audio (AES22)* (pp. 412–421).

Bartsch, M., & Wakefield, G. (2005). Audio thumbnailing of popular music using chroma-based representations. *IEEE Transactions on Multimedia, 7*(1), 96–104.

Berzal, F., Fajardo, W., Jiménez, A., & Molina-Solana, M. (2009). Mining musical patterns: Identification of transposed motives. In *18th Int. symposium of foundations of intelligent systems*. Lecture Notes in Computer Science, vol. 5722, pp. 271–280.

Böckenhauer, H. J., & Bongartz, D. (2007). *Algorithmic aspects of bioinformatics*. New York: Springer.

Cambouropoulos, E., Crawford, T., & Iliopoulos, C. S. (2001). Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities, 35*(1), 9–21.

Chu, S., & Logan, B. (2002). Music summary using key phrases. In *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP-00)* (pp. 749–752).

Dong, G., & Pei, J. (2007). *Sequence data mining (advances in database systems)*. New York: Springer.

Grachten, M., Arcos, J. L., & de Mantaras, R. L. (2004). Melodic similarity: Looking for a good abstraction level. In *5th Int. Conf. on Music Information Retrieval (ISMIR 2004)* (pp. 210–215).

Han, J., & Kamber, M. (2005). *Data mining: Concepts and techniques*. Denver: Morgan Kaufmann.

Hsu, J. L., Liu, C. C., & Chen, A. (1998). Efficient repeating pattern finding in music databases. In *ACM 7th int. conf. on information and knowledge management* (pp. 281–288).

Jiang, L., & Hamilton, H. J. (2003). Methods for mining frequent sequential patterns. In *Advances in artificial intelligence, Lecture of Notes in Computer Sciences* (Vol. 2671/2003, pp. 486–491). Berlin: Springer.

Jimenez, A., Berzal, F., & Cubero, J. C. (2009). Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. *Knowledge and Information Systems, 4994/2008*, 111–120. doi:10.1007/s10115-009-0213-3.

Levy, M., & Sandler, M. (2008). Structural segmentation of musical audio by constrained clustering. *IEEE Transactions on Audio, Speech, and Language Processing, 16*(2), 318–326.

Meredith, D., Lemström, K., & Wiggins, G. A. (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research, 31*(4), 321–345

Narmour, E. (1992). *The analysis and cognition of melodic complexity: The implication realization model*. Chicago: Univ. Chicago Press.

Paulus, J., & Klapuri, A. (2009). Music structure analysis using a probabilistic fitness measure and a greedy search algorithm. *IEEE Transactions on Audio, Speech, and Language Processing, 17*(6), 1159–1170.

Pei, J., Han, J., Asl, M. B., Pinto, H., Chen, Q., Dayal, U., et al. (2001). Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *5th int. conf. on extending database technology* (pp. 215–224).

Pienimäki, A. (2002). Indexing music databases using automatic extraction of frequent phrases. In *3rd int. conf. on music information retrieval* (pp. 25–30).

Rolland, P. Y. (1998). Discovering patterns in musical sequences. *Journal of New Music Research, 28*(4), 334–350

Srikant, R., & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. *Extending Database Technology, 1057*, 3–17.
Wang, W., Yang, J., & Yu, P. S. (2001). Meta-patterns: Revealing hidden periodic patterns. In *IBM research report* (pp. 550–557).
Yang, J., Wang, W., & Yu, P. S. (2001). Infominer: mining surprising periodic patterns. In *7th ACM int. conf. on knowledge discovery and data mining (SIGKDD)* (pp. 395–400). New York: ACM
Zaki, M. J. (2001). Spade: an efficient algorithm for mining frequent sequences. *Machine Learning, 42*, 31–60.
Zaki, M. J. (2005a) Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae, 66*(1–2), 33–52
Zaki, M. J. (2005b). Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering, 17*(8), 1021–1035.
Zhang, T., & Samadani, R. (2007). Automatic generation of music thumbnails. In *Proceedings of the 2007 IEEE int. conf. on multimedia and expo* (pp. 228–231).

Bibliography

[1] Kenji Abe, Shinji Kawasoe, Tatsuya Asai, Hiroki Arimura, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 158–174, 2002.

[2] Charu C. Aggarwal, Na Ta, Jianyong Wang, Jianhua Feng, and Mohammed Javeed Zaki. Xproj: a framework for projected structural clustering of XML documents. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 12-15*, pages 46–55, 2007.

[3] Rakesh Agrawal and John C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8:962–969, 1996.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases, September 12-15*, pages 487–499, 1994.

[5] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin ichi Nakano. Discovering frequent substructures in large unordered trees. In *Discovery Science*, volume 2843 of *Lecture Notes in Artificial Intelligence*, pages 47–61. Springer, 2003.

[6] Mark A. Bartsch and Gregory H. Wakefield. Audio thumbnailing of popular music using chroma-based representations. *IEEE Transactions on Multimedia*, 7(1):96–104, 2005.

[7] Yijun Bei, Gang Chen, Lidan Shou, Xiaoyan Li, and Jinxiang Dong. Bottom-up discovery of frequent rooted unordered subtrees. *Information Sciences*, 179(1-2):70–88, 2009.

[8] Fernando Berzal, Ignacio J. Blanco, Daniel Sánchez, and María Amparo Vila. Measuring the accuracy and interest of association rules: A new framework. *Intelligence Data Analysis*, 6(3):221–235, 2002.

[9] Fernando Berzal and Juan C. Cubero. Guest editors' introduction. *Data and Knowledge Engineering*, 60(1):1–4, 2007.

[10] Fernando Berzal, Juan-Carlos Cubero, and Aida Jimenez. Hierarchical program representation for program element matching. In *IDEAL'07: 8th International Conference on Intelligent Data Engineering and Automated Learning*, volume 4881 of *Lecture Notes in Computer Science*, pages 467–476, 2007.

[11] Fernando Berzal, Juan Carlos Cubero, Daniel Sánchez, and José-María Serrano. ART: A hybrid classification model. *Machine Learning*, 54(1):67–92, 2004.

[12] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional, 2005.

[13] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[14] Emilio Cambouropoulos, Tim Crawford, and Costas S. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities*, 35(1):9–21, 2001.

[15] Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. Frequent subtree mining - an overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.

[16] Yun Chi, Yirong Yang, and Richard R. Muntz. Indexing and mining free trees. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 509–512, 2003.

[17] Yun Chi, Yirong Yang, and Richard R. Muntz. HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 11–20, 2004.

[18] Diane J. Cook and Lawrence B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[19] Luis Cuevas, Nicolás Marín, Olga Pons, and María Amparo Vila. pg4db: A fuzzy object-relational system. *Fuzzy Sets and Systems*, 159(12):1500–1514, 2008.

[20] Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, 1991.

[21] Miguel Delgado, Nicolás Marín, Daniel Sánchez, and María Amparo Vila. Fuzzy association rules: general model and applications. *IEEE Transactions on Fuzzy Systems*, 11:214–225, 2003.

[22] Mukund Deshpande, Michihiro Kuramochi, and George Karypis. Automated approaches for classifying structures. In *Proceedings of the 2nd Workshop on Data Mining in Bioinformatics*, pages 11–18, 2002.

[23] Peter J. Diggle, Kung-Yee Liang, and Scott L. Zeger. *Analysis of longitudinal data*. Number 13 in Oxford statistical science series. Clarendon Press, 1994.

[24] Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman. A simplied universal relation assumption and its properties. *ACM Transactions on Database Systems*, 7:343–360, 1982.

[25] N. M. Laird G. M. Fitzmaurice and J. H. Ware. *Applied Longitudinal Analysis*. Wiley Series in Probability and Statistics, 2004.

[26] Harald Gall, Michele Lanza, and Thomas Zimmermann. 4th International Workshop on Mining Software Repositories. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 107–108, 2007.

[27] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson Education, 2008.

[28] Liqiang Geng and Howard J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3):9, 2006.

[29] Fedja Hadzic, Henry Tan, and Tharam S. Dillon. Uni3 - efficient algorithm for mining unordered induced subtrees using tmg candidate generation. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining*, pages 568–575, 2007.

[30] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques (2ed)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[31] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[32] Shohei Hido and Hiroyuki Kawano. AMIOT: induced ordered tree mining in tree-structured databases. In *Proceedings of the 5th IEEE International Conference on Data Mining*, pages 170–177, 2005.

[33] Jia-Lien Hsu, Arbee L. P. Chen, and C.-C. Liu. Efficient repeating pattern finding in music databases. In *Proceedings of the 7th International conference on Information and Knowledge Management*, pages 281–288, 1998.

[34] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Interestingness measures for association rules within groups. *Submitted to Knowledge and Information Systems.*

[35] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Mining frequent patterns from XML data: Efficient algorithms and design trade-offs. *Submitted to Expert Systems with Applications.*

[36] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Using trees to mine multirelational databases. *Submitted to Data Mining and Knowledge Discovery.*

[37] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 14:603–622, 2010.

[38] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. POTMiner: Mining ordered, unordered, and partially-ordered trees. *Knowlegde and Information System*, 23(2):199–224, 2010.

[39] Aída Jiménez, Miguel Molina-Solana, Fernando Berzal, and Waldo Fajardo. Mining transposed motifs in music. *Journal of Intelligent Information Systems*, 36:99–115, 2011.

[40] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

[41] David Maier and Jeffrey D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems*, 8:1–14, 1983.

[42] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Transactions on Database Systems*, 9:283–308, 1984.

[43] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[44] David Meredith, Kjell Lemström, and Geraint A. Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.

[45] Eugene Narmour. *The Analysis and Cognition of Melodic Complexity: The Implication Realization Model.* Chicago, IL: Univ. Chicago Press, 1992.

[46] Richi Nayak and Mohammed Javeed Zaki(eds). Knowledge discovery from XML documents. volume 3915 of *Lecture Notes in Computer Science*. Springer, 2006.

[47] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 647–652, 2004.

[48] Claudia Perlich and Foster Provost. Distribution-based aggregation for relational learning with identifier attributes. *Machine Learning*, 62:65–105, 2006.

[49] Anna Pienimäki. Indexing music databases using automatic extraction of frequent phrases. In *3rd International Conference on Music Information Retrieval*, pages 25–30, 2002.

[50] Marie Plasse, Ndeye Niang, Gilbert Saporta, Alexandre Villeminot, and Laurent Leblond. Combined use of association rules mining and clustering methods to find relevant links between binary rare attributes in a large data set. *Computational Statistics & Data Analysis*, 52(1):596–613, 2007.

[51] Pierre-Yves Rolland. Discovering patterns in musical sequences. *Journal of New Music Research*, 28(4):334–350, 1998.

[52] Ulrich Rückert and Stefan Kramer. Frequent free tree discovery in graph data. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.

[53] E. H. Shortliffe and B. G. Buchanan. A model of inexact reasoning in Medicine. *Mathematical biosciences*, 23:351–379, 1975.

[54] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts.* McGraw-Hill, 2001.

[55] HHerbert A Simon. *The Sciences of the Artificial (3rd Edition).* The MIT Press, 1996.

[56] Henry Tan, Tharam S. Dillon, Ling Feng, Elizabeth Chang, and Fedja Hadzic. X3-Miner: Mining Patterns from an XML Database. In *Proceedings of the 6th International Conference on Data Mining, Text Mining and their Business Applications*, pages 287–296, 2005.

[57] Henry Tan, Tharam S. Dillon, Fedja Hadzic, Elizabeth Chang, and Ling Feng. IMB3-Miner: Mining induced/embedded subtrees by constraining the level of embedding. In *Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 450–461, 2006.

[58] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (1st Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[59] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient pattern-growth methods for frequent tree pattern mining. In *Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 3056 of *Lecture Notes in Computer Science*, pages 441–451. Springer, 2004.

[60] Jason Tsong-Li Wang, Kaizhong Zhang, Karpjoo Jeong, and Dennis Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6:559–571, 1994.

[61] Wikifonia. Wikifonia foundation: www.wikifonia.org.

[62] Yongqiao Xiao, Jenq-Foung Yao, Zhigang Li, and Margaret H. Dunham. Efficient data mining for maximal frequent subtrees. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 379–386, 2003.

[63] Xiaoxin Yin, Jiawei Han, Jiong Yang, and Philip S. Yu. CrossMine: efficient classification across multiple database relations. In *Proceedings of the 20th International Conference on Data Engineering*, pages 399–410, 2004.

[64] Xiaoxin Yin, Jiawei Han, and Philip S. Yu. Cross-relational clustering with user's guidance. In *Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining*, pages 344–353, 2005.

[65] Osmar R. Zaiane and Jiawei Han. Resource and knowledge discovery in global information systems: A preliminary design and experiment. In *Proceedings of the First International Conference On Knowledge Discovery and Data Mining, Montreal, Canada*, pages 331–336, 1995.

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

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

[68] Mohammed Javeed Zaki and Charu C. Aggarwal. XRules: An effective algorithm for structural classification of XML data. *Machine Learning*, 62(1-2):137–170, 2006.

[69] Tong Zhang and Ramin Samadani. Automatic generation of music thumbnails. In *Proceedings of the 2007 IEEE International Conference on Multimedia and Expo*, pages 228–231, 2007.