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.

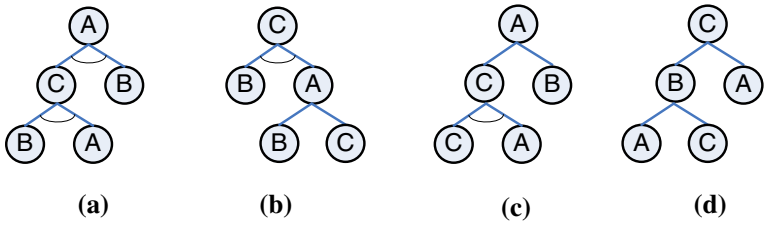

Fig. 1 Different kinds of rooted trees (from *left* to *right*): **a** completely-ordered tree, **b**, **c** partially-ordered trees, **d** completely-unordered tree

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).

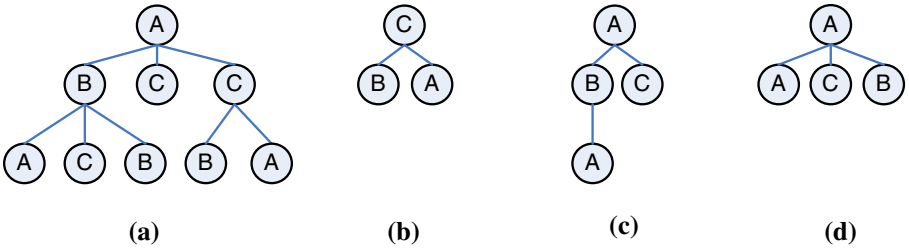

Fig. 2 Different kinds of subtrees (from *left* to *right*): **a** original tree, **b** bottom-up subtree, **c** induced subtree, **d** embedded subtree

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 $\delta_T(S)$ 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

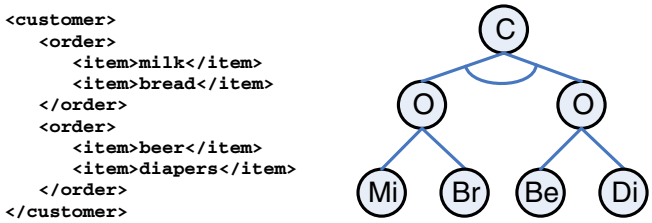

Fig. 3 An XML document and its partially-ordered tree representation

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.

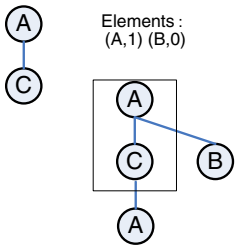

Fig. 5 Equivalence class with two elements, ACA and $AC\uparrow B$, that share the prefix AC

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 $(B, 0)$

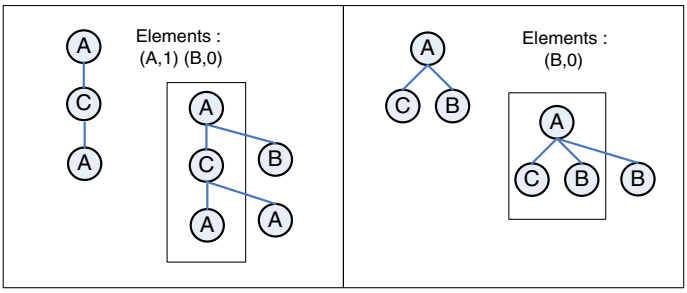

Fig. 6 Cousin extension applied to the elements of the class in Fig. 5

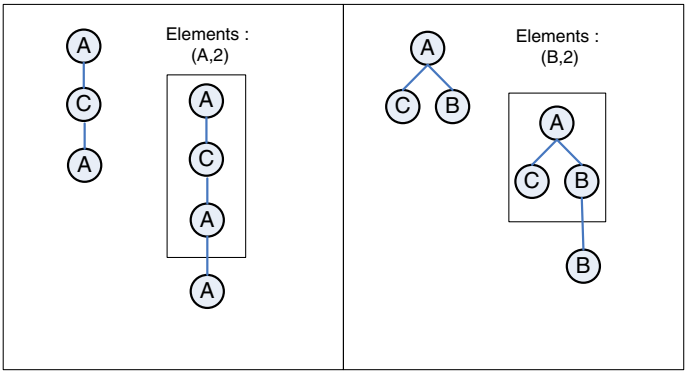

Fig. 7 Child extension applied to the elements of the class in Fig. 5

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.

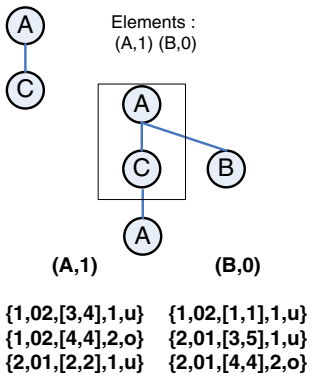

Fig. 8 Class with two elements, $(A, 1)$ and $(B, 0)$, and their scope lists

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).

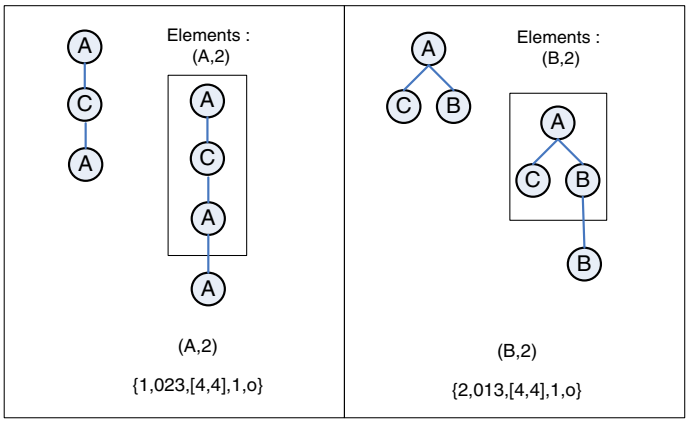

Fig. 9 Child extension of the elements in the class shown in Fig. 8 and the scope lists resulting from the in-scope join

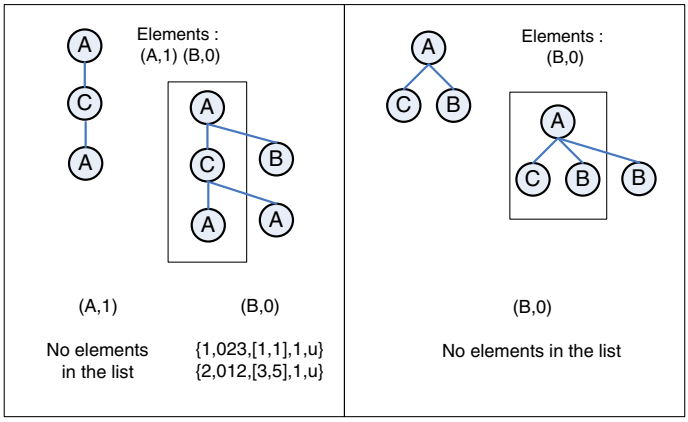

Fig. 10 Cousin extension of the elements in the class shown in Fig. 8 and the scope lists resulting from the out-scope join

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

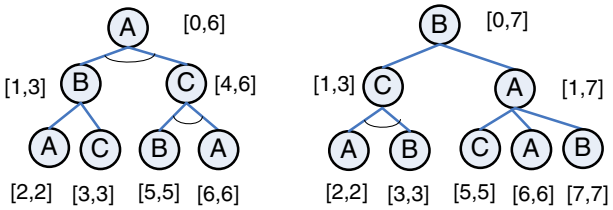

Fig. 11 Sample dataset containing two partially-ordered trees

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).

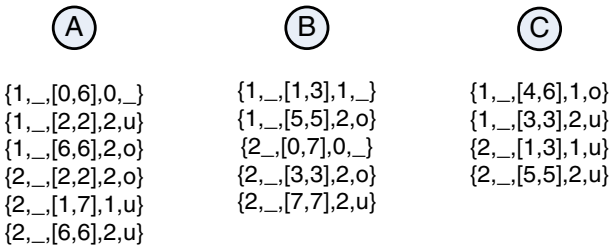

Fig. 12 Vertical representation of the trees in Fig. 11

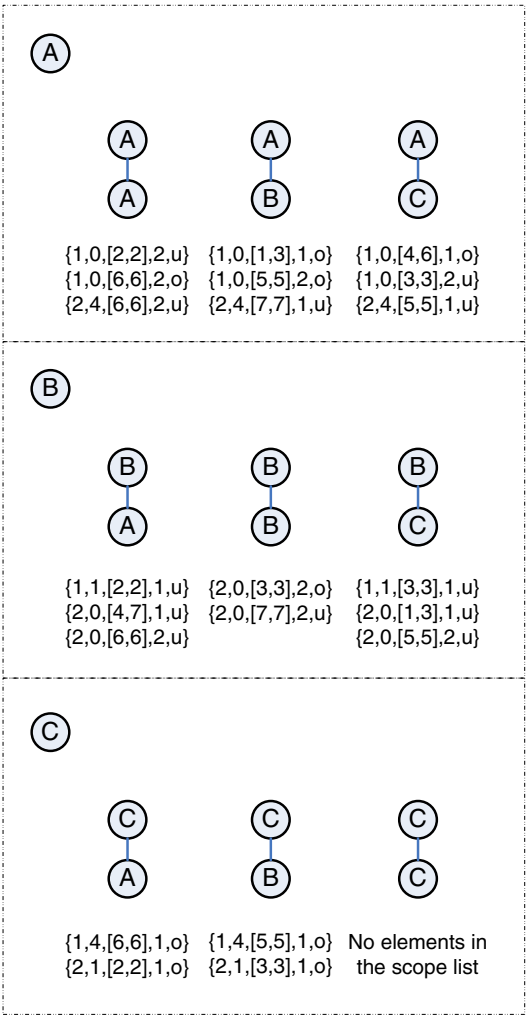

Fig. 13 Classes derived from the patterns of size 1 in Fig. 11

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

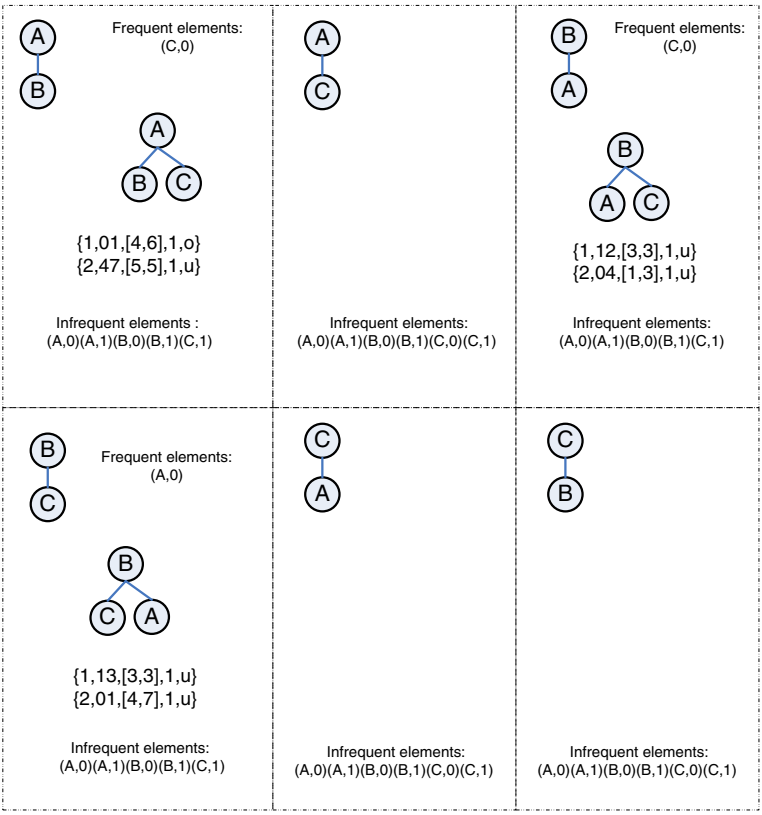

Fig. 14 Classes derived from the frequent patterns of size 2 in Fig. 13

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 $\sum_{k=1...\text{MaxSize}}(t * (Ln^2)^k/(k-1)!)$. 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 ∈ C_{k-1}
            Extend P in parallel to obtain P_extended
        for each class P ∈ C_{k-1}
            C_k = C_k ⋃ 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 $\sum_{k=1\ldots\text{MaxSize}}(t*(2\text{Ln}^2)^k/(k-1)!)$. 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

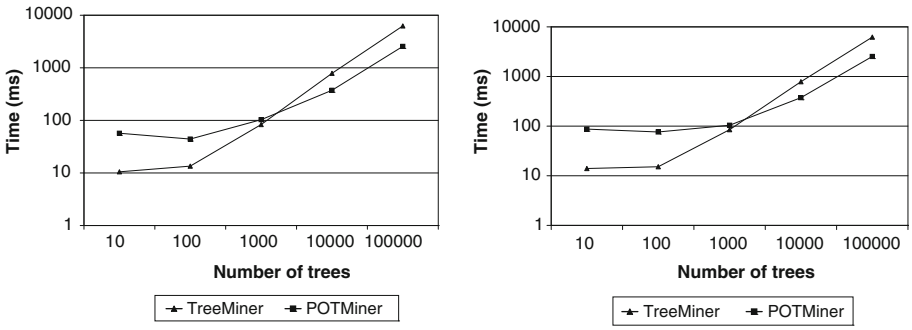

Fig. 17 POTMiner and TreeMiner execution times when identifying induced patterns (*left*) and embedded patterns (*right*) in completely-ordered trees

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.

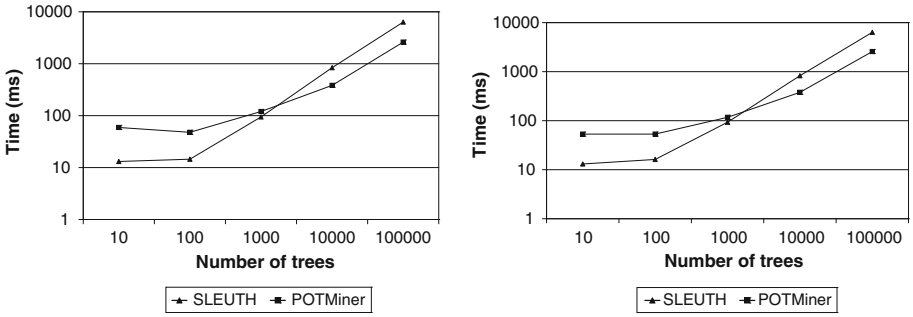

Fig. 18 POTMiner and SLEUTH execution times when identifying induced patterns (*left*) and embedded patterns (*right*) in completely-unordered trees

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

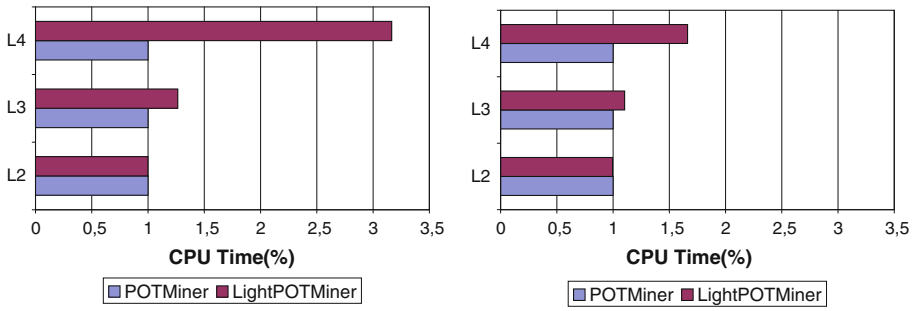

Fig. 21 Additional CPU time required by LightPOTMiner on the Muta2 (*left*) and Muta3 (*right*) datasets with respect to POTMiner

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

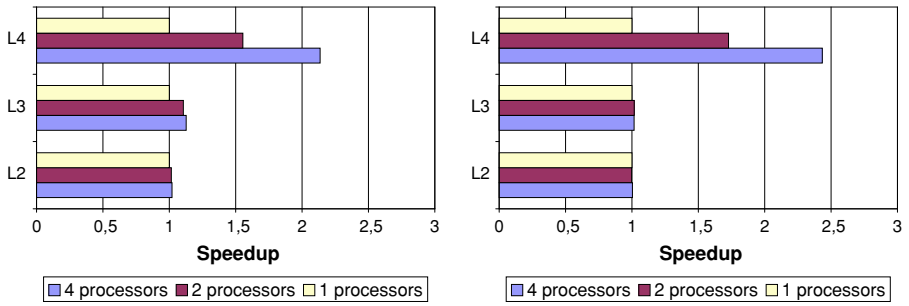

Fig. 23 Parallel POTMiner execution speedup on the Muta2 (*left*) and Muta3 (*right*) datasets

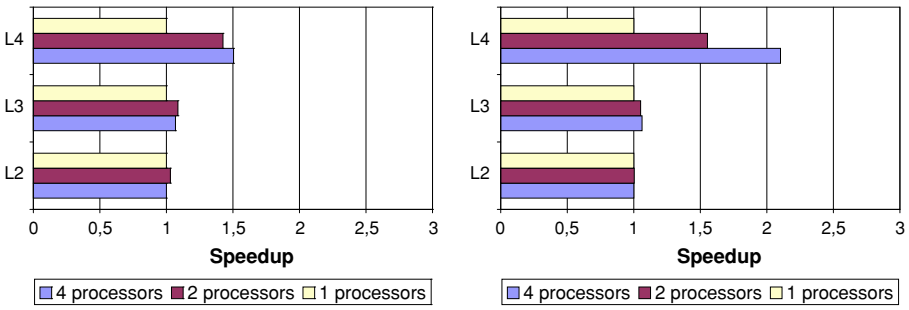

Fig. 24 Parallel LightPOTMiner speedup on the Muta2 (*left*) and Muta3 (*right*) datasets

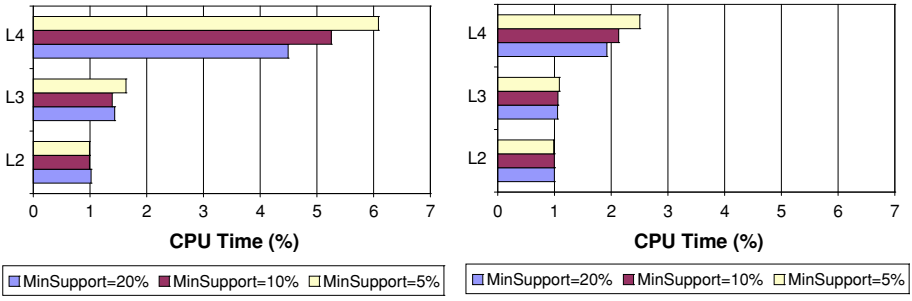

Fig. 25 Parallel LightPOTMiner versus parallel POTMiner for different minimum support thresholds on the Muta2 (*left*) and Muta3 (*right*) datasets

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 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" [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.