DRYADEPARENT, an efficient and robust closed attribute tree mining algorithm

Article *in* IEEE Transactions on Knowledge and Data Engineering · October 2007

DOI: 10.1109/TKDE.2007.190695 · Source: IEEE Xplore

6 authors, including:

Alexandre Termier
University of Grenoble
44 PUBLICATIONS **460** CITATIONS
SEE PROFILE

Marie-Christine Rousset
University Joseph Fourier - Grenoble 1
144 PUBLICATIONS **2,369** CITATIONS
SEE PROFILE

Michèle Sebag
Laboratoire de Recherche en Informatique
252 PUBLICATIONS **3,243** CITATIONS
SEE PROFILE

Kouzou Ohara
Aoyama Gakuin University
68 PUBLICATIONS **630** CITATIONS
SEE PROFILE

DRYADEPARENT, An Efficient and Robust Closed Attribute Tree Mining Algorithm

Alexandre Termier, Marie-Christine Rousset, Michèle Sebag, Kouzou Ohara, *Member*, *IEEE*, Takashi Washio, *Member*, *IEEE Computer Society*, and Hiroshi Motoda, *Member*, *IEEE Computer Society*

Abstract—In this paper, we present a new tree mining algorithm, DRYADEPARENT, based on the hooking principle first introduced in DRYADE. In the experiments, we demonstrate that the branching factor and depth of the frequent patterns to find are key factors of complexity for tree mining algorithms, even if often overlooked in previous work. We show that DRYADEPARENT outperforms the current fastest algorithm, CMTreeMiner, by orders of magnitude on data sets where the frequent tree patterns have a high branching factor.

Index Terms—Data mining, mining methods and algorithms, mining tree structured data.

1 INTRODUCTION

In the last 10 years, the frequent pattern discovery task of data mining has expanded from simple item sets to more complex structures, for example, sequences [1], episodes [2], trees [3], or graphs [4], [5]. In this paper, we focus on *tree mining*, that is, finding frequent tree-shaped patterns in a database of tree-shaped data. Tree mining can lead to many practical applications in the areas of computer networks [6], bioinformatics [7], [8], and XML documents databases mining [9], [10] and hence have received a lot of attention from the research community in recent years. Most of the well-known algorithms use the same generate-and-test principle that made the success of frequent item set algorithms. The main adaptation to the tree case is the design of efficient candidate tree enumeration algorithms in order to avoid generating redundant candidates and to enable efficient pruning. However, the search space of tree candidates is huge, particularly when the frequent trees to find have both a high depth and a high branching factor. Especially, the high branching factor case has received very little attention in the tree mining community. However, the performances of existing algorithms are dramatically affected by the branching factor of the tree patterns to find, as shown in our experiments.

Starting from this observation, we have developed the DRYADEPARENT algorithm. This algorithm is an adaptation of our earlier algorithm DRYADE [11]. DRYADE is based on a more general tree inclusion definition appropriate for mining highly heterogeneous collections of tree data. DRYADEPARENT follows the same principles of DRYADE but uses a standard inclusion definition [12], [13] to make possible performance comparisons with other existing systems based on different principles. We will show in this paper that DRYADEPARENT outperforms the up-to-date CMTreeMiner algorithm [13] and conduct a thorough study on the influence of structural characteristics of the tree patterns to find, like depth and branching factor, on the computation time performance of both algorithms.

The paper is outlined as follows: Section 2 introduces the notations and definitions used throughout the paper. Section 3 presents and discusses the state of the art in tree mining. Section 4 gives an overview of the DRYADEPARENT algorithm. Section 5 reports detailed comparative experiments, both on real and artificial data sets, as well as an application example with XML data. In Section 6, we conclude and give some directions for future work.

2 FORMAL BACKGROUND

Intuitively, the objective task of the DRYADEPARENT algorithm that we present in this paper is, given a set of trees and an arbitrary threshold ε, to discover the biggest tree substructures common to at least ε trees of the input set of trees. This is illustrated in the example in Fig. 1. The substructure CS containing the nodes B, C, and D appears in T_1 and T_2, that is, two trees of the input: For a support threshold of $\varepsilon = 2$, it is the only desired result. In this section, we give the graph theory background necessary to formally define the task described before. We will first formally define what a tree is. Then, we will show how to define a tree substructure of a tree (*tree inclusion* definition)

- *A. Termier and M.-C. Rousset are with the Laboratoire d'Informatique de Grenoble, University of Grenoble, 681 rue de la Passerelle, BP 72, 38402 St. Martin d'Heres Cedex, France. E-mail: {alexandre.termier, marie-christine.rousset}@imag.fr.*
- *M. Sebag is with the Laboratoire de Recherche en Informatique (LRI), Université Paris-Sud, Bat 490, 91405 Orsay, France. E-mail: michele.sebag@lri.fr.*
- *K. Ohara and T. Washio are with the Institute of Scientific and Industrial Research, Osaka University, 8-1 Mihogaoka, Ibaraki, Osaka, 567-0047 Japan. E-mail: {ohara, washio}@ar.sanken.osaka-u.ac.jp.*
- *H. Motoda is with the Asian Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, 7-23-17 Roppongi, Minato-ku, Tokyo 106-0032, Japan. E-mail: hiroshi.motoda.JP@aoard.af.mil or motoda@ar.sanken.osaka-u.ac.jp.*

Manuscript received 19 Jan. 2006; revised 16 Feb. 2007; accepted 13 Sept. 2007; published online 1 Oct. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0021-0106.

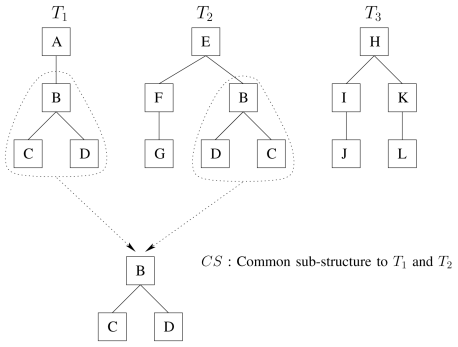

Fig. 1. A set of trees and their common substructure for $\varepsilon = 2$.

and under which conditions a given tree substructure is considered common to several other trees (*frequent trees* definition). Last, we will characterize the "biggest" of these tree substructures (*closed frequent trees* definition).

2.1 Trees

Let $L = \{l_1, \ldots, l_n\}$ be a set of labels. A *labeled tree* $T = (N, A, root(T), \varphi)$ is an acyclic connected graph, where N is the set of nodes, $A \subset N \times N$ is a binary relation over N defining the set of edges, $root(T)$ is a distinguished node called the *root*, and φ is a labeling function $\varphi : N \mapsto L$ assigning a label to each node of the tree. We assume without loss of generality that edges are unlabeled: As each edge connects a node to its parent, the edge label can be considered as part of the child node label. A tree is an *attribute tree* if φ is such that two sibling nodes cannot have the same label (more details on attribute trees can be found in [12]). Let $u \in N$ and $v \in N$ be two nodes of a tree. If there exists an edge $(u, v) \in A$, then v is a *child* of u, and u is the *parent* of v. For two nodes $u \in N$ and $v \in N$, if there exists a set of nodes $\{u_1, \ldots, u_k\}$ such that

$$(u, u_1) \in A, (u_1, u_2) \in A, \ldots, (u_k, v) \in A,$$

then $\{u, u_1, \ldots, u_k, v\}$ form a *path* in T. The *length of the path* $\{u, u_1, \ldots, u_k, v\}$ is $|\{u, u_1, \ldots, u_k, v\}| - 1$. If there exists a path from u to v in the tree, then v is a *descendant* of u, and u is an *ancestor* of v. Let $u \in N$ be a node of a tree T. The length of the path from $root(T)$ to u is the *depth* of u, denoted by $depth(u)$.

Tree truncation. Our DRYADEPARENT algorithm has the specificity to discover its objective trees one level of depth at a time. Consider, for example, the tree T in Fig. 2 and suppose that it is the objective of DRYADEPARENT: Each iteration will discover one more of its depth level, discovering first $T_{|0}$ and $T_{|1}$ (the first iteration discovers the depth levels 0 and 1), then $T_{|2}$ in the second iteration, and $T_{|3} = T$ in the last iteration. To characterize these intermediate levels $T_{|0}$, $T_{|1}$, and $T_{|2}$, we introduce the *tree truncation* concept: The truncation of a tree at a given depth level consists only of the nodes of that tree having a lesser or equal depth level and the corresponding edges. The formal definition is given as follows: Let $T = (N, A, root(T), \varphi)$ be a tree and d be an integer such that $d \leq depth(T)$. The truncation of T at the depth level d is the tree $T_{|d} = (N_{|d}, A_{|d}, root(T), \varphi)$ such that $N_{|d} = \{n \in N \mid depth(n) \leq d\}$ and $A_{|d} = \{(u, v) \in A \mid u, v \in N_{|d}\}$.

2.2 Tree Inclusion

The essential problem for discovering frequent patterns is to be able to determine if a given pattern appears or not in the input data. In the case of tree mining, this means determining if a pattern tree is included in any tree of the data. There are many different ways to define such a *tree inclusion*; the interested reader is referred to [14] for an extensive study. In this paper, we use the following definition, which is the basis of many other tree mining algorithms.

Let $AT = (N_1, A_1, root(AT), \varphi_1)$ be an attribute tree and $T = (N_2, A_2, root(T), \varphi_2)$ be a tree. AT is included in T if there exists an injective mapping $\mu : N_1 \mapsto N_2$ such that

1. μ preserves the labels: $\forall u \in N_1\ \varphi_1(u) = \varphi_2(\mu(u))$.
2. μ preserves the parent relationship:

$$\forall u, v \in N_1 (u, v) \in A_1 \Leftrightarrow (\mu(u), \mu(v)) \in A_2.$$

This relation will be written as $AT \sqsubseteq T$. In the tree mining literature, AT is also said to be an *induced subtree* of T when using the inclusion definition stated above. Fig. 3 shows the inclusion of an attribute tree AT in a tree T, along three possible mappings μ_1, μ_2, and μ_3.

If we have $AT \sqsubseteq T$ and $T \not\sqsubseteq AT$, then we say that AT is *strictly included* into T, and we denote it by $AT \sqsubset T$. If $AT \sqsubseteq T$, the set of mappings supporting the inclusion is denoted by $\mathcal{EM}(AT, T)$. In the example, we have $\mathcal{EM}(AT, T) = \{\mu_1, \mu_2, \mu_3\}$. The set of *occurrences* of AT in T, denoted by $Locc(AT, T)$, is the set of nodes of T onto which the root of AT is mapped by a mapping of $\mathcal{EM}(AT, T)$. In the example, $Locc(AT, T) = \{3, 11\}$, this corresponds to the identifiers of nodes labeled by A mapped by mappings μ_1, μ_2, and μ_3.

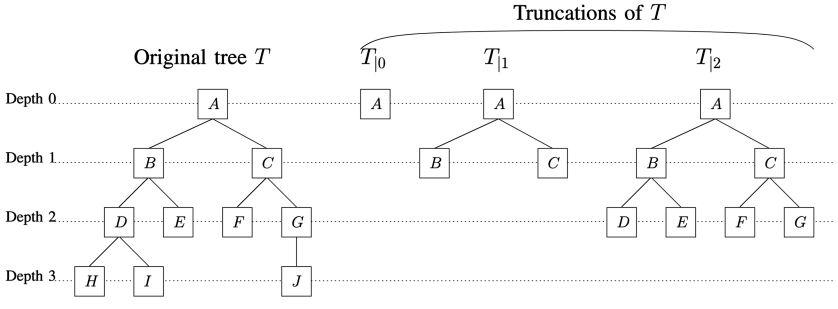

Fig. 2. A tree T and its truncations.

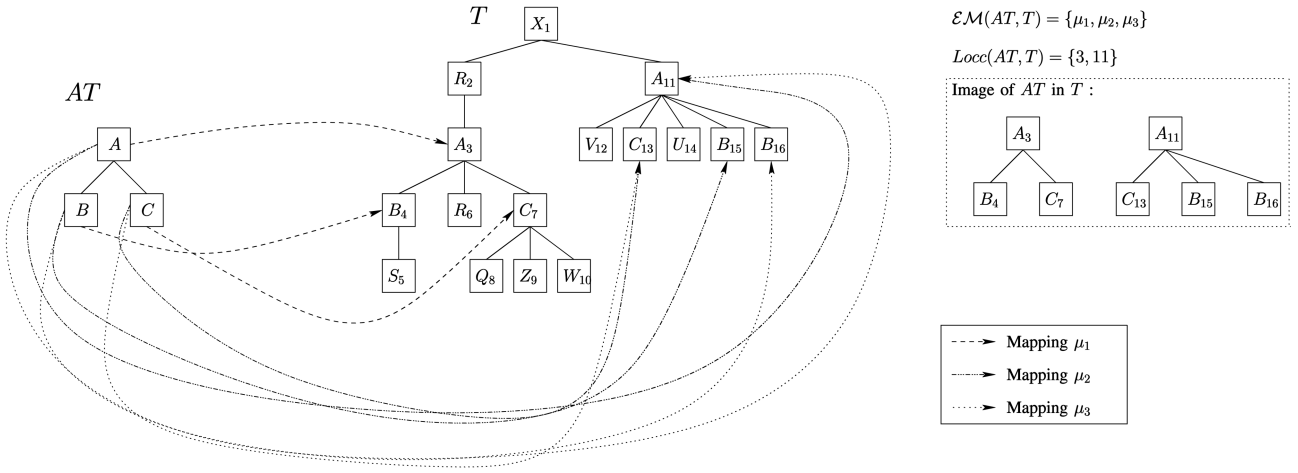

Fig. 3. Tree inclusion example (node identifiers are subscripts of node labels in T).

We also introduce the notion of *image* of an attribute tree AT in a tree T. The set of images of AT into T is the set of (attribute) trees obtained by mapping AT onto T by applying the mappings from $\mathcal{EM}(AT, T)$. In the example, we can see that the image of AT in T consists of the nodes of T mapped from AT by μ_1, μ_2, and μ_3.

2.3 Frequent Attribute Trees

We can now define the problem of finding *frequent attribute trees* in a tree database. Let $TD = \{T_1, \ldots, T_m\}$ be a tree database. The *datatree* D_{TD} is the tree whose root is an unlabeled node, having the trees $\{T_1, \ldots, T_m\}$ as its direct subtrees. Such a datatree is shown in Fig. 4a, where $TD = \{T_1, T_2\}$.

The *support* of an attribute tree AT in the datatree can be defined in two ways:

- $support_d(AT) = \sum_{i=1}^{m} \sigma_d(AT, T_i)$, where $\sigma_d(AT, T_i) = 1$ if $AT \sqsubseteq T_i$ and 0 otherwise *(document support)*.
- $support_o(AT) = \sum_{i=1}^{m} \sigma_o(AT, T_i)$, where $\sigma_o(AT, T_i) = |Locc(AT, T_i)|$ *(occurrence support)*.

In this paper, we are interested in finding attribute trees frequent by document support. The term *support* will now be used for document support. However, for the sake of completeness, our algorithm needs to keep track of all frequent occurrences and will use the occurrence support for processing.

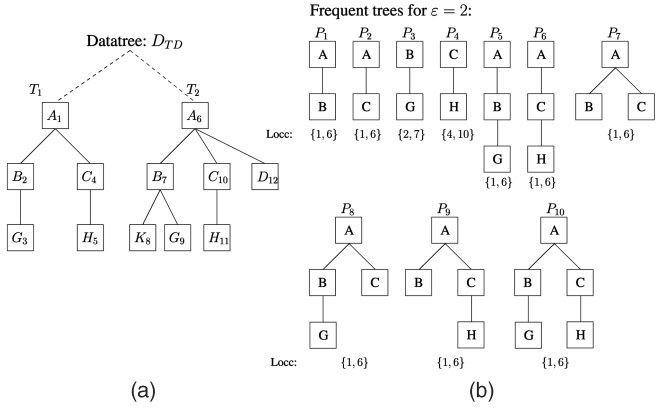

Fig. 4. (a) A datatree with two trees. (b) All the frequent trees for $\varepsilon = 2$.

Let ε be an absolute frequency threshold. AT is a *frequent attribute tree* of D_{TD} if $support_d(AT) \geq \varepsilon$. The set of all frequent attribute trees is denoted by $\mathcal{F}(D_{TD}, \varepsilon)$, and by the abuse of notation, we will only denote it as $\mathcal{F}$ in the rest of this paper.

The example in Fig. 4 shows all the frequent attribute trees for a support threshold of $\varepsilon = 2$.

2.4 Closed Trees

The problem with frequent trees is that usually, there are many of them, which implies long computation time. Moreover, lots of these frequent trees contain redundant information. For example, consider Fig. 4: Trees $P_1, P_2, \ldots, P_9$ are frequent, but this is just a byproduct of the fact that tree P_{10} is frequent (if a tree is frequent, all its subtrees are also frequent). When examining the mappings, we can see that the mappings of $P_1, P_2, \ldots, P_9$ are included in the corresponding mappings of P_{10}: Trees $P_1, P_2, \ldots, P_9$ do not bring any new information compared to P_{10}. Therefore, if we could characterize trees such as P_{10} and only compute those trees without generating trees like $P_1, P_2, \ldots, P_9$, a lot of computation time would be saved.

Such a characterization exists and has been pioneered by Pasquier et al. [15] for frequent item sets and by Chi et al. [13] for trees. It is based on the *closure property*: P_{10} is a *closed frequent tree*; intuitively, this means that for its set of mappings, it is the maximal tree according to inclusion. Formally, we have the following definition:

Definition 1. *A frequent attribute tree $AT \in \mathcal{F}$ is closed if either*

- *$AT \in \mathcal{F}$ is not included into any other frequent attribute tree $AT' \in \mathcal{F}$ or*
- *AT is included into a frequent attribute tree $AT' \in \mathcal{F}$, in which case there exists a mapping in $\mathcal{EM}(AT, D_{TD})$ that is not in the mappings of $\mathcal{EM}(AT', D_{TD})$.*

We will denote the set of all closed frequent attribute trees as $\mathcal{C}$, with the same abuse of notation as before.

2.5 Closed Set of Trees

Let $\mathcal{S} \subseteq \mathcal{F}$. The set $\mathcal{S}$ is said to be *closed* if all the trees of $\mathcal{S}$ are closed relative to the other trees of $\mathcal{S}$, that is, in Definition 1, $\mathcal{F}$ is replaced by $\mathcal{S}$.

2.6 Tree Mining Problem

The tree mining problem we are interested in is to find all the closed frequent attribute trees for a given datatree and support threshold. The merit of this problem is that the number of closed frequent attribute trees is much smaller than the number of all frequent attribute trees, but the amount of information is the same in both cases: All of the frequent attributes trees can be easily deduced from the closed frequent attribute trees. Thus, finding such closed trees enables faster mining without loss of information.

3 Related Work

In this section, we will first recall the seminal works about frequent item set mining and show how they have been extended to perform frequent tree mining.

3.1 Item Set Mining

The pioneering works for the mining of frequent item sets have been made by Agrawal and Srikant, who introduced the Apriori algorithm for mining frequent item sets in a propositional database [16]. The settings are much simpler than the problem in this paper: The data consists of *transactions*, which are sets of *items*. The problem is to find frequent *item sets*, that is, the sets of items that occur frequently in the data. To find these frequent item sets, Apriori uses a *generate-and-test* method, which means that it will proceed by generating candidate item sets and then test these candidate item sets against the data to check if they are frequent or not. The enumeration of these candidate item sets is done in a *levelwise* manner: First, the candidate item sets of size 1 are generated, then the candidate item sets of size 2, and so on and so forth. The candidate item sets of size $i+1$ are generated by combining together the item sets of size i that passed the frequency test. To prune the search space and hence improve the performances, the algorithm uses an *antimonotonicity property*: If a candidate item set I_1 is found to be infrequent, then it is not necessary to build a bigger candidate item set I_2 such that $I_1 \subset I_2$, as by definition, this candidate will necessarily be also infrequent.

Fig. 5 shows an example execution of the Apriori algorithm. The data is first transformed into a matrix representation, which is easier to use for counting frequency. In the first iteration, the candidates of size 1 are generated (all the single items), and their support is computed. The frequency threshold being set to 2, only item E is not frequent and does not make it to the next iteration. All the other candidates of size 1 are frequent item sets and are combined together in an iteration to make candidate item sets of size 2. The frequency of these candidates is computed, and it is found that only $\{B, D\}$ is not frequent. The other candidates are frequent item sets and are combined together in the third and last iteration to give the candidates of size 3. Note that even if $\{B, C\}$ and $\{C, D\}$ are frequent, candidate $\{B, C, D\}$ is not constructed. This comes from the fact that $\{B, D\} \subset \{B, C, D\}$, and $\{B, D\}$ is known to be infrequent. Therefore, necessarily, $\{B, C, D\}$ is also infrequent and need not be generated. The support of the candidates of size 3 is evaluated, and $\{A, B, D\}$ is eliminated as infrequent. The other candidates

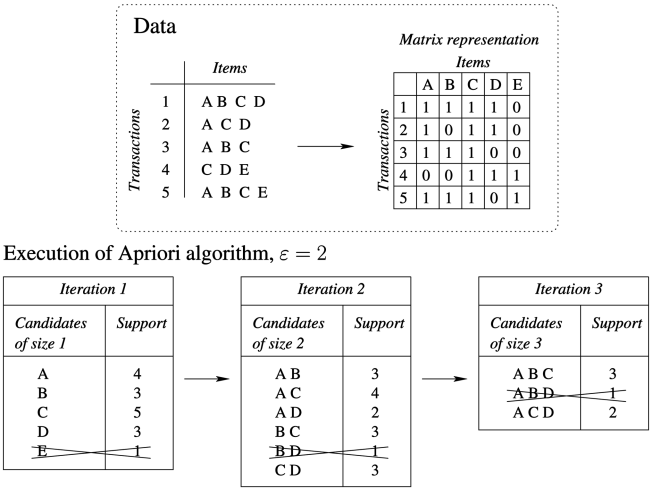

Fig. 5. An example of Apriori execution.

are frequent, and there are no ways to combine them for a fourth iteration: the algorithm stops.

Among the many improvements to this algorithm, Pasquier et al. [15] were the first to design an algorithm for discovering only the *closed* frequent item sets and showed performance improvements of around one order of magnitude. These results were improved by Zaki and Hsiao's CHARM algorithm [17]. Today, the fastest algorithm for discovering closed frequent item sets is LCM2 [18], the winner of the Second Workshop on Frequent Itemset Mining Implementations (FIMI '04) contest.

3.2 Tree Mining

Most tree mining algorithms are adaptations of the Apriori principle to tree-structured data. They usually deal with finding *all* the frequent subtrees from a collection of trees. One pioneering work is Asai et al.'s Freqt algorithm [3], which discovers all frequent subtrees with the preservation of the order of the siblings. The other pioneering work is Zaki's *TreeMiner* [19], which uses a more relaxed inclusion definition where the order still has to be preserved, but instead of the parent relationship, the mapping has only to preserve the ancestor relationship.

Both of these algorithms, like the Apriori algorithm described before, are levelwise generate-and-test algorithms and make use of the antimonotonicity property. The size of a candidate tree is expressed as its number of nodes, so these algorithms first generate all the candidate trees with one node, then, from those of these candidates that are frequent, generate the candidate trees with two nodes, and so on and so forth. Each candidate's frequency has to be assessed by testing its inclusion in all the trees of the data, which is a very computation-time expensive operation. Another difficult part is the candidate enumeration method. Unlike in the case of item sets, here, the extensions of two different candidates of size i can lead to the same candidate of size $i+1$, as seen in Fig. 6: there are two different candidates of size 2, $A-B$ and $A-C$. To create candidates of size 3, one possibility is to join $A-C$ to $A-B$; the other is to join $A-B$ to $A-C$. Obviously, these two possibilities lead to the same candidate of size 3. This introduces redundancies in the enumeration process, which must be

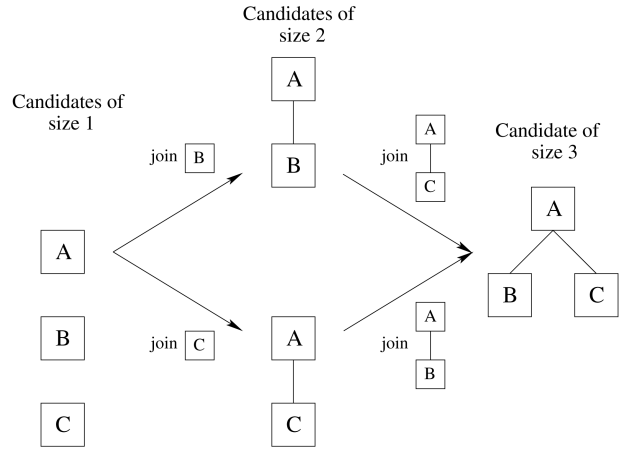

Fig. 6. Three steps of candidate generation. The two candidates of size 2 lead to a single candidate of size 3.

avoided at all costs as testing the frequency for one candidate or testing for duplicates inside the candidates set are computationally expensive operations.

The authors of the two previous papers prevent this by setting an order on the generation of candidates, which imposes to add new nodes only on the *rightmost branch* of the frequent tree of size i used as a basis. This enumeration strategy avoids duplicates, thus enabling a better efficiency than naive methods. It is illustrated in Fig. 7.

The second generation of tree mining algorithms has been designed to get rid of the order preservation constraint. This was realized by basing the enumeration procedures on canonical forms, one canonical form representing all the trees that are isomorphic except for the order of siblings. Such work includes the Unot algorithm by Asai et al. [20], the work of Nijssen and Kok [21], the PathJoin algorithm [22], and the recent Sleuth algorithm by Zaki [23].

There are still very few algorithms mining closed frequent trees. We already mentioned our DRYADE algorithm [11], which relies on a very general tree inclusion definition and a

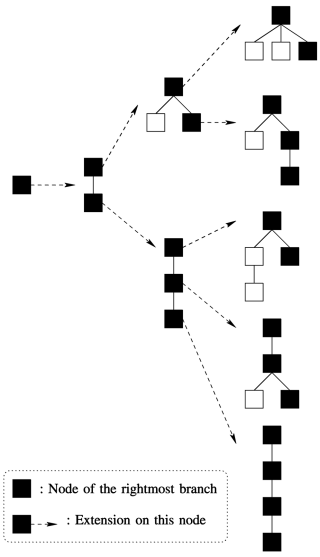

Fig. 7. Candidate generation via the rightmost branch enumeration method.

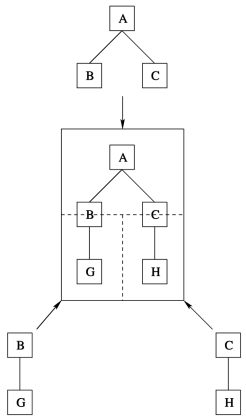

Fig. 8. A closed frequent tree and its tiles.

new *hooking* principle. The only algorithm mining closed frequent *induced* subtrees is the CMTreeMiner algorithm of Chi et al. [13]. It uses the same generate-and-test principle as other tree mining algorithms, extended to handle closure. This algorithm has shown excellent experimental results. Recently, Arimura and Uno proposed the CLOTT algorithm [12] for mining closed frequent attribute trees, in the same settings as those in this paper. This algorithm has a proved output-polynomial time complexity, which should also give excellent performances. Up to now, there is not yet an implementation available.

It is clear that the generate-and-test method used by all these algorithms (except DRYADE) has an efficiency that depends heavily on the structure of the tree patterns to find. In case of big tree patterns with a high depth and a high branching factor, many edge-adding steps are needed to find these tree patterns, and each step can be computationally expensive because of the number of possible expansions and of the necessary frequency testing.

4 The DryadeParent Algorithm

4.1 Idea of the Algorithm

Before going into the details of the DRYADEPARENT algorithm, we will first explain the intuition behind our method. For sake of readability, we will use the term *closed frequent tree* to designate the closed frequent attribute trees that the DRYADEPARENT algorithm discovers.

Briefly stated, the principle of our algorithm is to discover parts of the frequent trees and then assemble these parts together to get the frequent trees. The parts that we are interested in are the closed frequent trees *of depth 1*. The interesting characteristic of these closed frequent trees of depth 1 with respect to the final result is that either

- they are closed frequent trees as is or
- they represent one node and its children in one or more closed frequent trees (a formal proof will come later in Lemma 1), for example, in Fig. 8, the closed frequent trees of depth 1 and of roots A, B, and C assembled together make a single tree of depth two, which is the closed frequent tree to find.

It is quite simple to find these closed frequent trees of depth 1 by using a standard closed frequent item set

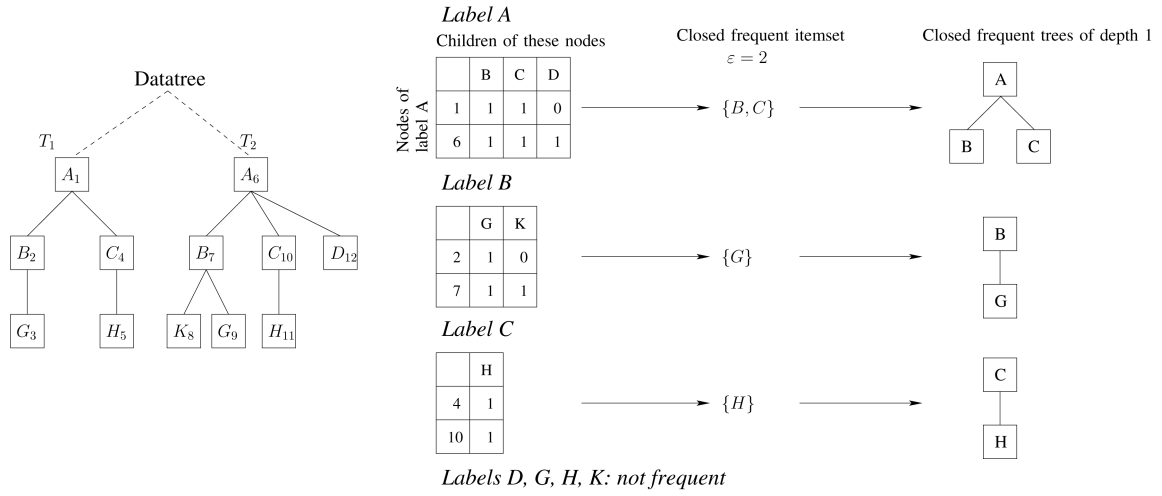

Fig. 9. Example of discovery of closed frequent trees of depth 1.

algorithm: for any label $x \in L$, create a matrix whose transactions are the nodes of labels x in the trees of the data and whose items are the labels of the children of these nodes. The resulting closed frequent item sets will be sets of edges $\{(x, y_1), \ldots, (x, y_n)\}$ rooted on the same node, that is, closed frequent trees of depth 1. By iterating on all the labels x, all the closed frequent trees of depth 1 can be found by this method. An example of discovery of closed frequent trees of depth 1 is shown in Fig. 9.

That is why, from now on, we will call the closed frequent trees of depth 1 with the shorter name of *tiles*, as like in mosaics or in puzzles, they are the small parts that are assembled together to make a closed frequent tree of $\mathcal{C}$.

Remark. Another advantage of the tiles is that they follow the dynamic programming as defined in [24], in the sense that they are solutions to subproblems of the main problem, which are computed only once and can then be reused any number of times. This allows for better performances, especially in the cases where the closed frequent trees share many common tiles.

The most obvious hint to determine how to combine the tiles together is to look at their labels. If a leaf label of a tile Ti_1 matches with the root label of a tile Ti_2, then it is possible for these two tiles to be "hooked" together and create a bigger tree. This is shown in Fig. 10. However, nothing guarantees that in the mappings of Ti_1 and Ti_2 in the data, the leaf of Ti_1 and the root of Ti_2 are the same node. If this is not the case in at least ε trees of the data, then

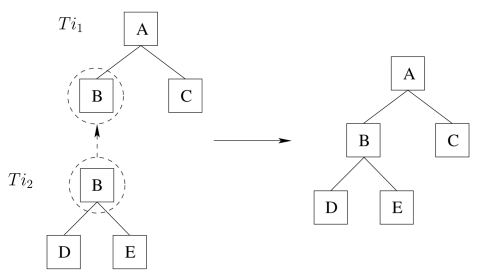

Fig. 10. A simple hooking between two tiles and the resulting tree.

the tree constructed by combining Ti_1 and Ti_2 will not be frequent and so cannot be part of the final result. For example, consider Fig. 11. The tiles are the same tiles Ti_1 and Ti_2 as those in Fig. 10, so from the labels, they can be hooked. By analyzing the mappings, we can see that in T_1, the nodes for B in Ti_1 and Ti_2 are the same (node 3), so this mapping supports the hooking of Ti_2 on Ti_1. However, in T_2, the nodes for B are different: node 9 for Ti_1 and node 11 for Ti_2. Therefore, the mapping from T_2 does not support the hooking. The hooking being supported in only one tree and the frequency threshold being $\varepsilon = 2$, the hooking is not frequent and so must not be done.

Ensuring that the mappings of the data support the tile combinations is a necessary step. However, this is not sufficient. There can be many tiles $Ti_2, \ldots, Ti_n$ whose root node label matches a leaf node label of Ti_1, such matching being supported by mappings in the data. Thus, many new trees can be constructed: combining Ti_1 with Ti_2, Ti_1 with Ti_3, or even Ti_1 with Ti_2 and $Ti_3 \ldots$ This is illustrated on the example in Fig. 12, where three tiles $\{Ti_2, Ti_3, Ti_4\}$ can hook on Ti_1.

However, few of these combinations correspond to what can actually be found in a closed frequent attribute tree of the result. In fact, the tiles $Ti_2, \ldots, Ti_n$ combined with Ti_1 do not only need to verify a frequency criterion but also need to verify a *closure* criterion. This means that we will hook on T_1 only the closed frequent sets of tiles of $\{T_2, \ldots, T_n\}$ whose combination with T_1 to make a new tree is supported by the data. We will show later that this corresponds exactly to what is found in the closed frequent attribute trees. We call the operation consisting of finding the closed frequent sets of tiles hooking on other tiles and creating new trees from them a *hooking*. This is the basis of our algorithm. Such operations allow for a simple level-by-level breadth-first strategy:

1. Find the tiles that represent the top level of the closed frequent trees; they will be called *root tiles*.
2. For each of these tiles, iteratively perform hookings to grow them by one level at each iteration.

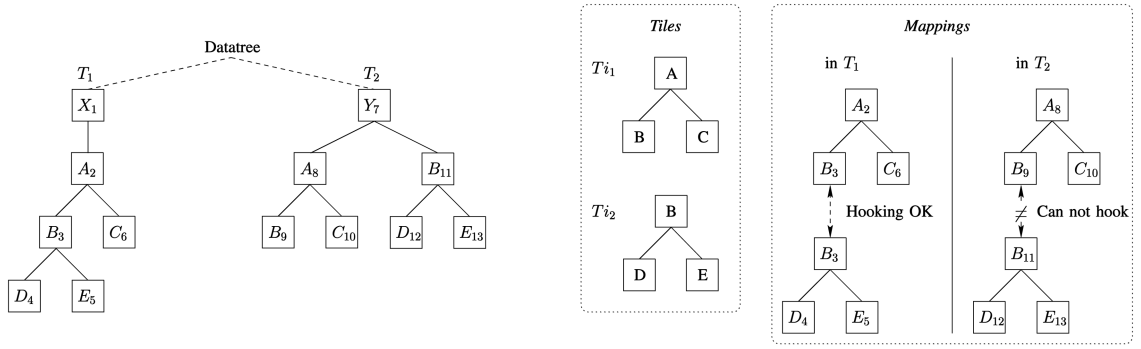

Fig. 11. A case where the hooking of tiles is not backed up by the mappings.

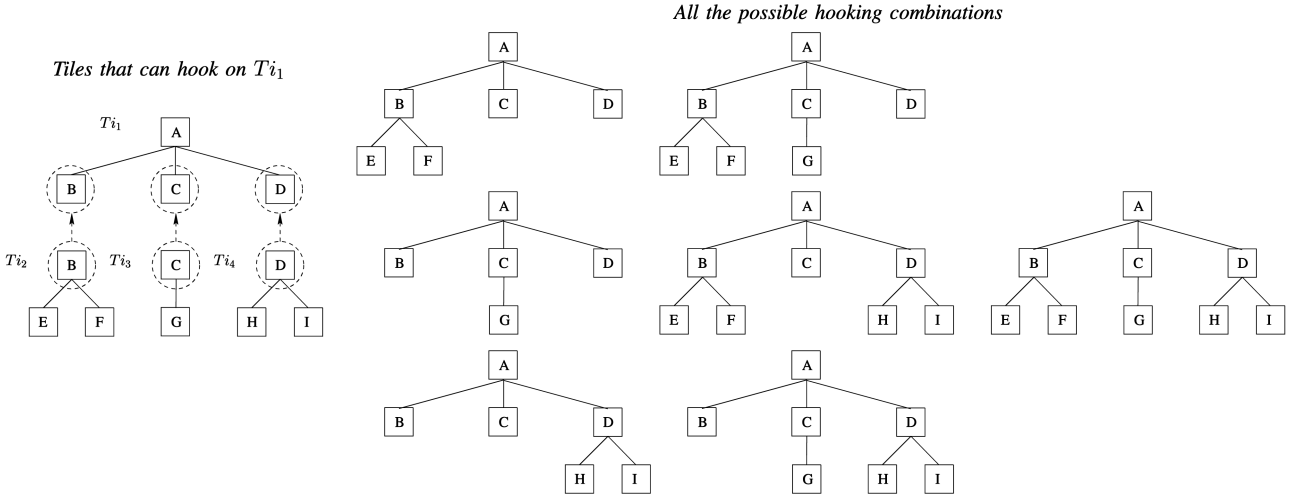

Fig. 12. Multiple hooking possibilities on a tile Ti_1 and the resulting trees.

4.2 Algorithm Details

Until now, we have given an intuitive overview of our method. We now give thorough explanations over the concepts of tiles and hookings, as well as the detailed pseudocode of our algorithm. As a running example, we use the datatree in Fig. 13 with a support threshold of $\varepsilon = 2$. The closed frequent attribute trees to find (that is, the elements of $\mathcal{C}$) are also represented in this figure as P_1, P_2, P_3, and P_4, along with their occurrences in the datatree.

The whole algorithm is summed up in Algorithm 1. Note that in Algorithms 1 and 2, *closed_frequent_itemset_algorithm* is a general algorithm mining closed frequent item sets; it can be any closed frequent item set miner. We assume that this closed frequent item set miner is sound and complete. In our implementation, we use the algorithm LCM2 [18].

Algorithm 1. The DRYADEPARENT algorithm
Input: A datatree D_{TD} and an absolute frequency threshold ε
Output: The set $\mathcal{C}_{Dryade}$ of all the closed frequent trees in D_{TD} with frequency $\geq \varepsilon$

$\mathcal{TI}(\mathcal{C}) \leftarrow$ Computation of all the tiles
$\mathcal{RP}_0 \leftarrow$ initial root tiles of D_{TD}
$i \leftarrow 0$; $\mathcal{C}_{Dryade} \leftarrow \emptyset$
$HookingBase \leftarrow \emptyset$
while $\mathcal{RP}_i \neq \emptyset$ **do**
  $\mathcal{RP}_{i+1} \leftarrow \emptyset$
  for all $RT \in \mathcal{RP}_i$ **do**
    if no hooking is possible on RT **then**
      $\mathcal{C}_{Dryade} \leftarrow \mathcal{C}_{Dryade} \cup RT$
    else
      $\mathcal{RP}_{i+1} \leftarrow \mathcal{RP}_{i+1} \cup Hookings(RT, HookingBase)$
    end if
  end for
  $\mathcal{RP}_{i+1} \leftarrow \mathcal{RP}_{i+1} \cup DetectNewRootTiles(\mathcal{TI}(\mathcal{C}), HookingBase)$
  $i \leftarrow i + 1$
end while
Return $\mathcal{C}_{Dryade}$

Algorithm 2. The $Hookings$ function
Input: A closed frequent attribute tree AT, hooking database $HookingBase$
Output: All the new closed frequent attribute trees found by hooking tiles on the leaves of AT

$Result \leftarrow \emptyset$
$M \leftarrow$ matrix whose transactions are the occurrences of AT, and whose columns are the tiles that can be hooked on AT.
$FIS \leftarrow closed\_frequent\_itemset\_algorithm(M)$
for all $(f, O) \in FIS$ **do**
  if $\nexists HK \in HookingBase$ st $(AT, f, O) \subseteq HK$ **then**
    $Result \leftarrow Result \cup$ new attribute tree resulting from the hooking of the tiles of f on AT
    Add (AT, f, O) to $HookingBase$

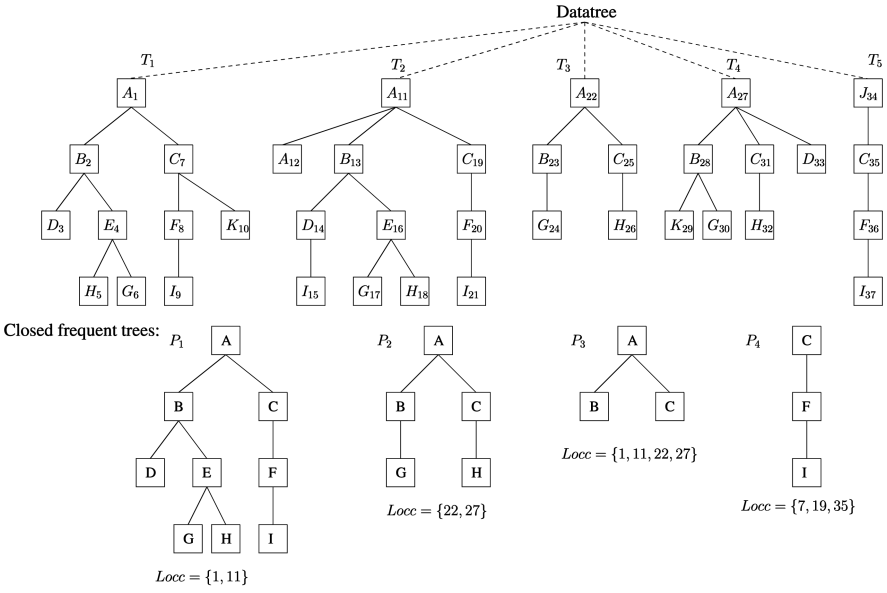

Fig. 13. Datatree example (node identifiers are subscripts of node labels) and closed frequent trees for $\varepsilon = 2$.

8: **if** $\exists\{HK_1, \ldots, HK_x\} \in HookingBase$ st $\forall i \in [1, x]\ HK_i \subseteq (AT, f, O)$ **then**
9: Suppress $\{HK_1, \ldots, HK_x\}$ from $HookingBase$, as well as the corresponding attribute trees in $\mathcal{RP}_i$ or $\mathcal{C}_{Dryade}$
10: **end if**
11: **end if**
12: **end for**
13: **Return** $Result$

4.2.1 Computation of the Tiles

The definition of a tile is given as follows:

Definition 2 (tile). *A tile is a frequent attribute tree made from a node of a closed frequent tree of $\mathcal{C}$ and all its children. The set of all tiles for the closed frequent trees of $\mathcal{C}$ is noted $\mathcal{TI}(\mathcal{C})$.*

We have seen before that we can use a closed frequent item set mining algorithm to compute these tiles. We will now detail how and prove that this method actually computes the tiles of $\mathcal{TI}(\mathcal{C})$.

For a given label l, let us consider the subproblem of finding all the tiles of the closed frequent trees of $\mathcal{C}$ whose root is labeled by l. We note the set of these tiles $\mathcal{TI}(\mathcal{C})_l$. Because these tiles come from closed frequent trees of $\mathcal{C}$, they are frequent in the datatree D_{TD}. We can also infer that the set of tiles $\mathcal{TI}(\mathcal{C})_l$ is closed; if it was not the case, it would contradict the closure of $\mathcal{C}$ (see the proof of the following lemma for more details). As all of these tiles share the same root label, we have to find the sets of children labels and the occurrences.

This problem can be reformulated as a propositional closed frequent item set discovery problem (as in Section 3.1) as follows: Consider a transaction matrix M_l whose transactions are the nodes of D_{TD} of label l and whose items are the labels of the children of these nodes (we remind the reader that as defined in Section 3.1, in a transaction matrix, the transactions are the rows, and the items are the columns). A "1" in the cell of the row corresponding to the node o (of label l) and of the column corresponding to the label x indicates that the node o of label l has a child of label x. For example, in the datatree in Fig. 13, M_B is

Occurence of B	D	E	G	K
2	1	1	0	0
13	1	1	0	0
23	0	0	1	0
28	0	0	1	1

The closed frequent item sets for matrix M_l are noted $CFIS(M_l)$. All of these closed frequent item sets satisfy the occurrence frequency constraint defined before. Since we are interested in document-frequent results, we suppress from $CFIS(M_l)$ all item sets whose occurrences appear in less than ε documents to get the set $CFIS_{doc}(M_l)$. From each item set f of $CFIS_{doc}(M_l)$, a tile of root l is built whose children are the items of f and whose occurrences are the transactions supporting f. The set of such tiles is noted $\mathcal{TI}(CFIS_{doc}(M_l))$.

Lemma 1. *For any label $l \in L$, we have*

$$\mathcal{TI}(\mathcal{C})_l = \mathcal{TI}(CFIS_{doc}(M_l)).$$

Proof. $(\mathcal{TI}(\mathcal{C})_l \subseteq \mathcal{TI}(CFIS_{doc}(M_l)))$. Consider a tile $T \in \mathcal{TI}(\mathcal{C})_l$. Let H denote the set of the labels of the leaves of T and O be the set of the occurrences of T. We have to show that H appears in $CFIS_{doc}(M_l)$. By definition, the tile is frequent and so has at least ε occurrences in D_{TD}. All of these occurrences appear in M_l, so H is frequent by document frequency, with support O. Hence, to show that H appears in $CFIS_{doc}(M_l)$, we only have to show that H is closed (intuitively, H is closed if it is the maximal for its set of occurrences. We refer the interested reader to [15] for a formal definition of closed item sets). Seeking a contradiction, suppose that H is not closed; then, there would be, for the occurrences of O, an item set H' such that $H \subset H'$. From H', we can build a tile T' that has the same occurrences as T but more leaves. Considering the closed frequent tree of $\mathcal{C}$ from which T

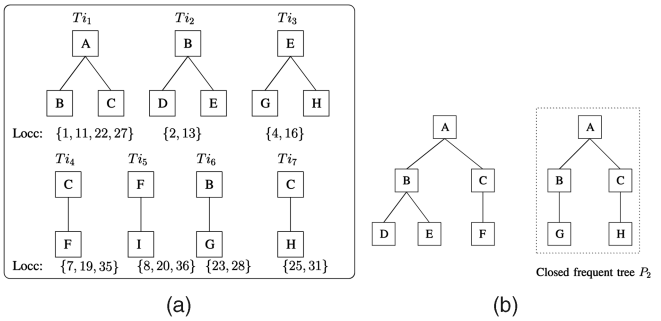

Fig. 14. Tiles and hookings. (a) Tiles of our example. (b) Hookings at iteration 1.

was extracted, it means that this closed frequent tree can be replaced with a closed frequent tree including T', so it means that $\mathcal{C}$ was not closed. This contradicts the hypothesis, so we proved by negation that H is closed.

$(\mathcal{TI}(\mathcal{C})_l \supseteq \mathcal{TI}(CFIS_{doc}(M_l)))$ Consider f a document-frequent closed frequent item set of M_l. It has at least ε different occurrences O, so a tile T rooted by l and having at least the labels of f as children exists in a closed frequent tree of $\mathcal{C}$; the occurrences of T include those of O. If in the closed frequent tree of $\mathcal{C}$ the root of the considered subtree had one more children than in f, then this would be reflected in M_l, and f would not be closed. Hence the labels of the leaves of T are exactly the labels in f. In the same way, if T had more occurrences than O, then these occurrences would appear in M_l with exactly the items of f, which is impossible as the only occurrences for the item set f are those of O. □

By iterating on the labels of L with the method previously shown, all the tiles of $\mathcal{TI}(\mathcal{C})$ can be computed. This is the first operation of our algorithm, so it is done on line 1 of Algorithm 1.

In the example, from the matrix M_B, the closed frequent item sets $\{D, E\}$ and $\{G\}$ are extracted, with respective occurrences {2, 13} and {23, 28}. Both these item sets are document frequent, the corresponding tiles appear in Fig. 14a as Ti_2 and Ti_6, along with all the other tiles for the datatree in Fig. 13.

4.2.2 Hooking the Tiles

Having found the tiles, the goal of DRYADEPARENT is to compute efficiently all the closed frequent trees through the hookings of these tiles. As stated before, we have chosen a levelwise strategy, where each iteration computes the next depth level for the closed frequent trees being constructed.

Initial root tiles. To begin with, the tiles that correspond to the depth levels 0 and 1 of the closed frequent trees must be found in the set of tiles. Such tiles are called *root tiles* for they are the top level of the closed frequent trees of $\mathcal{C}$. They are the starting point of our algorithm.

As these tiles represent the top level of the closed frequent trees, one naive way to discover them is to discover the tiles that cannot be hooked on any other tile, that is, which are never under any other tile whatever the mappings. This method works partially and can discover easily a subset of the root tiles, which we call *initial root tiles*. This is done in line 2 of Algorithm 1. In our example, Ti_1 is the only initial root tile because its occurrences 1, 11, 22, and 27 are not leaves of any other tile.

Notations. In the following, we will denote by $\mathcal{RP}_i$ the frequent trees that are the starting points for the algorithm's ith iteration ($\mathcal{RP}_0$ being the initial root tiles) and by $\mathcal{C}_{\mathcal{RP}_i}$ the closed frequent trees that will be obtained by successive hookings on the frequent trees of $\mathcal{RP}_i$ at the end of the algorithm. $\mathcal{C}_{\mathcal{RP}_i}$ is for illustration purposes and is not actually constructed by the algorithm. In the example, $\mathcal{RP}_0 = \{Ti_1\}$, and $\mathcal{C}_{\mathcal{RP}_0} = \{P_1, P_2, P_3\}$ in Fig. 13.

Hooking. The initial root tiles are the entry point to the main iteration of DRYADEPARENT. In iteration i, for each element T of $\mathcal{RP}_i$, the algorithm will discover all the possible ways to add one depth level to T with respect to the closed frequent trees to get. This is done via the **hooking** operation:

Definition 3 (hooking). *For an integer i, let T be an element of $\mathcal{RP}_i$ and $C \in \mathcal{C}_{\mathcal{RP}_i}$ such that $\exists q \leq i$ such that $T = C_{|q}$ (T is the truncation of C at depth q). The* hooking *operation consists of constructing a new frequent tree T' by hooking a set of hooking tiles $\{Ti_1, \ldots, Ti_k\}$ on the leaves of T such that the occurrences $\{o_1, \ldots, o_p\}$ of T' include those of C and $T' = C_{|q+1}$.*

Such a hooking will be denoted by

$$HK(T, T') = (T, \{Ti_1, \ldots, Ti_k\}, \{o_1, \ldots, o_p\}).$$

The subtle point is to find all the frequent hooking tile sets for an element T of $\mathcal{RP}_i$. The potential hooking tiles on T are all the tiles whose root is mapped to a leaf node of T. In our example, the potential hooking tiles on Ti_1 are $\{Ti_2, Ti_4, Ti_6, Ti_7\}$. Among all of these potential hooking tiles, we want to find those that frequently appear together according to the occurrences of T. This is a propositional closed frequent item set discovery problem, and we can solve it by creating a matrix M where each line k corresponds to an occurrence o_k of T and each column j corresponds to a potential hooking tile Ti_j. $M[i, j] = 1$ if and only if for the occurrence o_k of T, a leaf of T is mapped to the same node as the root of Ti_j. Applying a closed frequent item set discovery algorithm on M enables discovering efficiently all the closed frequent hooking tile sets. This is done in lines 2 and 3 of Algorithm 2. The frequent trees discovered must be inserted into $\mathcal{RP}_{i+1}$ for further expansion in the next iteration.

In our example, the matrix M for Ti_1 is

Occurence of Ti_1	Ti_2	Ti_4	Ti_6	Ti_7
1	1	1	0	0
11	1	1	0	0
22	0	0	1	1
27	0	0	1	1

We deduce that the frequent hooking tile sets on Ti_1 are $\{Ti_2, Ti_4\}$ and $\{Ti_6, Ti_7\}$. These hookings are illustrated in Fig. 14b. It can be seen that the closed frequent tree P_2 has been discovered.

Closure checking. However, in some cases, hooking can lead to frequent trees that are not closed. Consider the example in Fig. 15. Both tiles Ti'_1 and Ti'_2 are initial root tiles, but *Hooking 2* on tile Ti'_2 produces a frequent tree that

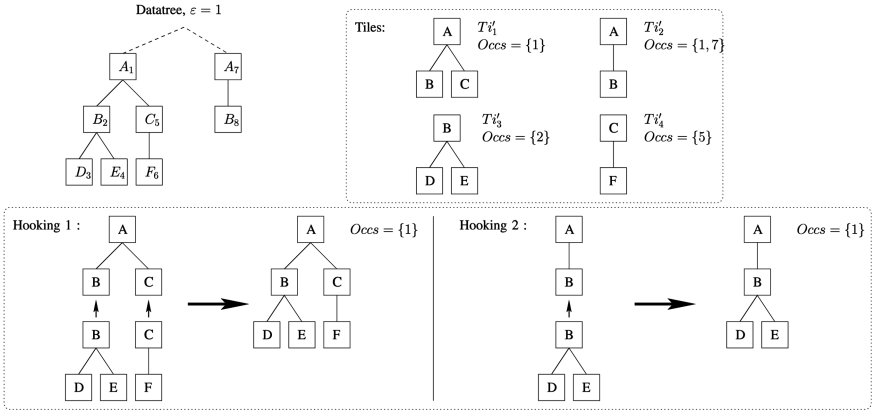

Fig. 15. Example of the generation of an unclosed frequent tree.

is included in the frequent tree produced by *Hooking 1* on tile Ti'_1, thus being unclosed.

Such cases can be detected quickly by analyzing the hookings already made in the previous iterations. For this purpose, the hookings performed so far are stored in a database denoted by $HookingBase$. Each hooking is represented by a triplet

$$(root\ frequent\ tree, hooking\ tiles, occurrences),$$

where $root\ frequent\ tree$ is the root attribute tree of the hooking, $hooking\ tiles$ are the tiles hooking on $root\ frequent\ tree$ for this hooking, and $occurrences$ are the occurrences of $root\ frequent\ tree$ considered in this hooking. As shown in lines 4-12 of Algorithm 2, when a new hooking is proposed, the function $Hookings$ checks that this new hooking satisfies the closure property with respect to the hookings of the database. Two nonclosure cases can arise: 1) the new hooking is included into an existing hooking, and then, the new hooking is discarded (line 5), and 2) the new hooking includes an existing hooking and, then, the existing hooking and the corresponding closed frequent tree are erased from the database, and a new closed frequent tree is created from the new hooking, which is registered into the hooking database (lines 8-9).

Preparing the next iteration. In the first iteration, the seeds of the closed frequent trees to be discovered are the initial root tiles, grouped into $\mathcal{RP}_0$. The frequent trees grown by hooking tiles on these root tiles are inserted into $\mathcal{RP}_1$ and will be used as the seed for the next iteration (line 10 of Algorithm 1). However, this is not enough to discover all of the closed frequent trees of $\mathcal{C}$. We have seen before that only a fraction of all the root tiles could be discovered at the beginning of the algorithm; these were the initial root tiles. The problem is that a tile T can both be the root tile of a closed frequent tree P and a nonroot tile of another closed frequent tree P'. Therefore, for the mappings of T corresponding to P', T will be hooked on other tiles, preventing it from satisfying the same conditions as the initial root tiles. In the example, Ti_4 is both a subtree in P_1 and the root tile of P_4. The problem is that if we look at the mappings of Ti_4, this tile does not hook on any other tile, only for the mapping rooted at occurrence 35: Its "root" status does not appear frequent with so few information. Therefore, for all these root tiles that are not initial root tiles, their discovery is delayed to later iterations, at a moment where we will have enough information to determine if this tile was only the subtree of one or more closed frequent trees or if it can also be the root tile of some other closed frequent trees. Therefore, after our hooking step, we have to analyze the hooked tiles to see if they belong to the category of tiles that will always be hooked somewhere or if they can become root tiles at the next iteration. This is done in the $DetectNewRootTiles$ function (Algorithm 3). In line 2 of Algorithm 3, the tiles T that have been hooked on other tiles in the current iteration (and so appear in $HookingBase_i$) are iterated over. In line 3, these tiles T are tested: the left part of the AND checks that there does not exist any unknown hooking between these tiles and a given tile T'; this is done for all the occurrences of T. If this left part is true, then we are assured to know everything about the hookings of T. Here comes the "root" part verification: In the right part of the AND, we check that there exists at least one occurrence of T where T does not hook on any other tile. If this part is also true, then T can be not only a subtree of other closed frequent trees but also a root tile. This is recorded in line 4. In our example, Ti_4 is one of these candidates to be a root tile, it has been hooked on Ti_1 for occurrences 7 and 19. There are no other tiles where it can hook (left part of the AND of line 3 satisfied), and for occurrence 35, it does not hook on any other tile (right part of the AND also satisfied). Therefore, Ti_4 becomes a new root tile; this will allow the discovery of closed frequent tree Pi_4 in the next iteration.

Algorithm 3. The $DetectNewRootTiles$ function
Input: Set of tiles $\mathcal{TI}(\mathcal{C})$, hooking database $HookingBase$ where $HookingBase_j$ are the hookings performed in iteration j
Output: Tiles of $\mathcal{TI}(\mathcal{C})$ that have become root
1: $Result \leftarrow \emptyset$
2: **for all** $T \in \mathcal{TI}(\mathcal{C})$ st $T \in HT$ where $(*, HT, *) \in HookingBase_i$ **do**
3: **if** [$\forall o \in Locc(T, D_{TD})$ $\nexists T'$ st T can hook on T' and $((T', \{\ldots, T, \ldots\}, *) \notin HookingBase)$] AND [$\exists o \in Locc(T, D_{TD})$ st T cannot hook on any other tile for o] **then**

4: $Result \leftarrow Result \cup T$
5: **end if**
6: **end for**
7: **Return** $Result$

4.3 Soundness and Completeness

Theorem 1. *The algorithm DRYADEPARENT is sound and complete, that is,* $\mathcal{C}_{Dryade} = \mathcal{C}$.

Proof. *Completeness.* Let $P \in \mathcal{C}$ be a closed frequent tree. We want to prove that P is found by DRYADEPARENT. Let us prove by induction on the depth levels of P that for every depth level d, $P_{|d}$ is found at some iteration of DRYADEPARENT.

For depth level 1, $P_{|1}$ is by definition a closed frequent tree of depth 1, that is, a tile. Therefore, it is found in the first step of DRYADEPARENT.

For depth level d, let us suppose that the induction property is true, that is, that there exists an iteration i of DRYADEPARENT where $P_{|d}$ is found as an element of $\mathcal{RP}_{i+1}$. Let us show that $P_{|d+1}$ is found in a later iteration of DRYADEPARENT.

By the definition of the tiles, all the tiles corresponding to the direct subtrees of $P_{|d}$ in P have been found in the first step of DRYADEPARENT, so all these tiles appear as columns of M in the $Hookings$ procedure. Let S denote this set of tiles. Because P occurs in at least ε documents, P has at least ε occurrences, so the closed frequent item set algorithm in the $Hookings$ finds a set of tiles f, where at least $f \supseteq S$. Let us show that we cannot have $f \supset S$. Suppose that f has one more tile T than S for the same occurrences. This means that T can also be hooked on $P_{|d}$ with the other tiles of S, with occurrences that include the occurrences of P. Therefore, for all the mappings of P, new $P + T$ mappings can be found. This contradicts the fact that P is closed. Hence, $f = S$.

We must now show that the test on line 5 of the $Hookings$ function (Algorithm 2) is evaluated to $true$, that is, that there are no hookings in the hooking base that includes the hooking of the tiles of f on $P_{|d}$ (else, no frequent trees would be built from the hookings of f). In the same way as we did previously, it is easy to show by negation that if there was such a hooking, then P would not be closed.

Hence, the closed frequent tree $P_{|d+1}$, resulting from the hookings of the tiles of f on $P_{|d}$, is correctly constructed.

It is inserted into $\mathcal{RP}_{i+2}$; hence, the induction property holds.

Therefore, DRYADEPARENT is complete.

Soundness. Let P be a frequent tree outputted by DRYADEPARENT. We want to show that we have $P \in \mathcal{C}$, that is, P is frequent, and P is closed with respect to the set of all frequent trees.

Frequency. Suppose by negation that P is not frequent. It means that either a tile of P is not frequent or that there exists a depth level of P where the set of tiles for this depth is not frequent. In both cases, it means that the closed frequent item set algorithm gave a nonfrequent result. It contradicts the soundness of $closed\_frequent\_itemset\_algorithm$. Hence, P is frequent.

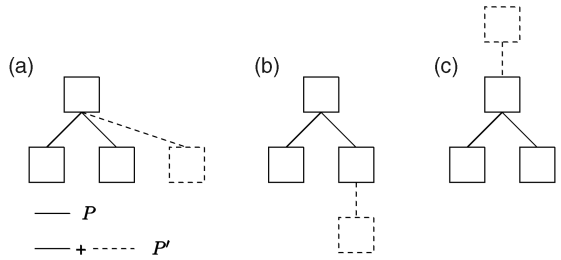

Fig. 16. Three possible inclusion cases.

Closedness. Suppose by negation that P is not closed, that is, there exists a closed frequent tree P' in which P is included for all its occurrences. We consider all the possible inclusion cases, as shown in Fig. 16:

1. *One more sibling node*. This case would mean that the corresponding tile was not closed and, hence, that the closed frequent item set gave a nonclosed result. Once again, it contradicts the soundness of the closed frequent item set mining algorithm.
2. *One more leaf child node*. This case would mean that a tile hooking has not been discovered or not been done. Because all the tiles are correctly found thanks to Lemma 1 and the filling of the hooking discovery matrix is trivial, it would mean that either the closed frequent item set algorithm was not complete, which contradicts the completeness of $closed\_frequent\_itemset\_algorithm$, or the hooking was found but later dismissed. Such a dismissal could only be done by the closure checking mechanism and only if there is a bigger hooking for the same occurrences at the same place. This would mean that P' itself is unclosed, which contradicts the hypothesis.
3. *One more root parent node*. Let T be the root tile of P as found by DRYADEPARENT. In this case, the root tile of P' (containing P) is a tile $T' \neq T$, and T hooks on T'. Suppose that there is such a tile T'. By definition, it cannot be an initial root tile (or DRYADEPARENT would have found it), and neither can it be T (because it hooks on T'). Because it was never considered as a root tile, the hookings of T on T' have not been found and do not appear in the hooking database. Therefore, the condition on line 3 of $DetectNewRootTiles$ cannot be satisfied for all the occurrences of T, and so, T cannot be detected as a root tile.

   By the definition of the root tile detection procedure, this case cannot occur.

Hence, P is closed, and we can conclude that the algorithm DRYADEPARENT is sound. □

4.4 Complexity

We estimate the time complexity of the DRYADEPARENT algorithm according to the following parameters:

- $\|D_{TD}\|$, the number of nodes of the input database,
- $|\mathcal{C}|$, the number of closed frequent trees to find,
- d, the average depth of a closed frequent tree of $\mathcal{C}$, and
- $|\mathcal{TI}(\mathcal{C})|$, the total number of tiles in the closed frequent trees of $\mathcal{C}$.

Computation of tiles. The tiles are computed with the LCM2 algorithm [18], whose time complexity is linear with the number of closed frequent item sets to find. Therefore, the time complexity of the tile computation step is linear with the number of tiles:

$$Complexity(Tile\_computation) \simeq O(|\mathcal{TI}(\mathcal{C})|).$$

Computing the initial root tiles. To determine which tile is an initial root tile, all the occurrences of all the tiles are checked. This simple step hence has a time complexity of

$$Complexity(Initial\_root\_tiles) \simeq O(\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|).$$

Main iteration. The first step of the main iteration is a loop repeated as many times as there are elements in $\mathcal{RP}_i$. These elements are truncations of closed frequent trees of $\mathcal{C}$, so we have $|\mathcal{RP}_i| = \alpha.|\mathcal{C}|$, where α is a constant:

- **"if" of line 7**. Determining if there are hookings on an element, $RT \in \mathcal{RP}_i$ comes to check all of its occurrences; the time complexity is

$$Complexity(Check\_if\_hookings) \simeq O(\|D_{TD}\|).$$

- *Hookings* **procedure**. Building the transaction matrix and running the LCM2 algorithm has a time complexity of $O(\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|)$. The hooking base must then be checked; the time complexity of this search operation is linear with the number of hookings. An upper bound for the number of hookings is the number of tiles. Hence,

$$\begin{aligned} Complexity(Hookings) &\simeq O(\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})| \\ &\quad + |\mathcal{TI}(\mathcal{C})|) \\ &\simeq O(\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|). \end{aligned}$$

The overall time complexity of the for loop is then

$$\begin{aligned} Complexity(for\_loop) &\simeq O(|\mathcal{C}| \times (\|D_{TD}\| + \|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|)) \\ &\simeq O(|\mathcal{C}|.\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|). \end{aligned}$$

Then, we have to compute the complexity of the $DetectNewRootTiles$ procedure. For each tile, there is a search in the hooking base on line 2 and, then, on line 3, there is a search on all the occurrences of the tile, which needs another search in the hooking base. This gives an overall complexity of

$$\begin{aligned} Complexity(DetectNewRootTiles) &\simeq O(|\mathcal{TI}(\mathcal{C})|.|\mathcal{TI}(\mathcal{C})|. \\ &\quad \|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|) \\ &\simeq O(\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|^3). \end{aligned}$$

The main iteration is repeated $\beta.d$ times (with β as a constant), so its time complexity is

$$\begin{aligned} Complexity(Iterations) &\simeq d.(Complexity(for\_loop) \\ &\quad + Complexity \\ &\quad (DetectNewRootTiles)) \\ &\simeq O(d.(|\mathcal{C}|.\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})| + \|D_{TD}\|. \\ &\quad |\mathcal{TI}(\mathcal{C})|^3)) \\ &\simeq O(d.\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|. \\ &\quad (|\mathcal{TI}(\mathcal{C})|^2 + |\mathcal{C}|)). \end{aligned}$$

The overall time complexity of the whole DRYADEPARENT algorithm is then

$$\begin{aligned} Complexity(DryadeParent) &\simeq Complexity(Tile\_computation) \\ &\quad + Complexity \\ &\quad (Initial\_root\_tiles) \\ &\quad + Complexity(Iterations) \\ &\simeq O(|\mathcal{TI}(\mathcal{C})| + \|D_{TD}\|.|\mathcal{TI}(\mathcal{C})| \\ &\quad + d.\|D_{TD}\|.|\mathcal{TI}(\mathcal{C})|.(|\mathcal{TI}(\mathcal{C})|^2 \\ &\quad + |\mathcal{C}|)) \\ &\simeq O(|\mathcal{TI}(\mathcal{C})|.(1 + \|D_{TD}\| + d. \\ &\quad \|D_{TD}\|.(|\mathcal{TI}(\mathcal{C})|^2 + |\mathcal{C}|))) \\ &\simeq O(|\mathcal{TI}(\mathcal{C})|.d.\|D_{TD}\|.(|\mathcal{TI}(\mathcal{C})|^2 \\ &\quad + |\mathcal{C}|)). \end{aligned}$$

We have given our complexity formula in terms of the number of tiles $|\mathcal{TI}(\mathcal{C})|$. This number of tiles can be approximated by the number of internal nodes in the closed frequent trees. With this, we can reformulate the complexity in terms of $\|\mathcal{C}\|$, the number of nodes in the closed frequent trees, and b, the average branching factor of the closed frequent trees of $\mathcal{C}$. Let $IN(\mathcal{C})$ be the internal nodes of the closed frequent trees of $\mathcal{C}$. We have

$$b = \frac{\text{number of edges in } \mathcal{C}}{\text{number of internal nodes } in\ \mathcal{C}}.$$

The number of edges in a single tree T with N nodes is $N-1$, we deduce that for the set of trees $\mathcal{C}$

$$b = \frac{\|\mathcal{C}\| - |\mathcal{C}|}{\|IN(\mathcal{C})\|}.$$

Therefore,

$$\|IN(\mathcal{C})\| = \frac{\|\mathcal{C}\| - |\mathcal{C}|}{b}.$$

Hence,

$$|\mathcal{TI}(\mathcal{C})| \simeq \|IN(\mathcal{C})\| = \frac{\|\mathcal{C}\| - |\mathcal{C}|}{b}.$$

The complexity formula can now be written as

$$\begin{aligned} Complexity(DryadeParent) \simeq O\Bigg(&\frac{\|\mathcal{C}\| - |\mathcal{C}|}{b}.d.\|D_{TD}\|. \\ &\left(\frac{(\|\mathcal{C}\| - |\mathcal{C}|)^2}{b^2} + |\mathcal{C}|\right)\Bigg). \end{aligned}$$

From the above formulas, we can conclude that the complexity of DRYADEPARENT is polynomial in the

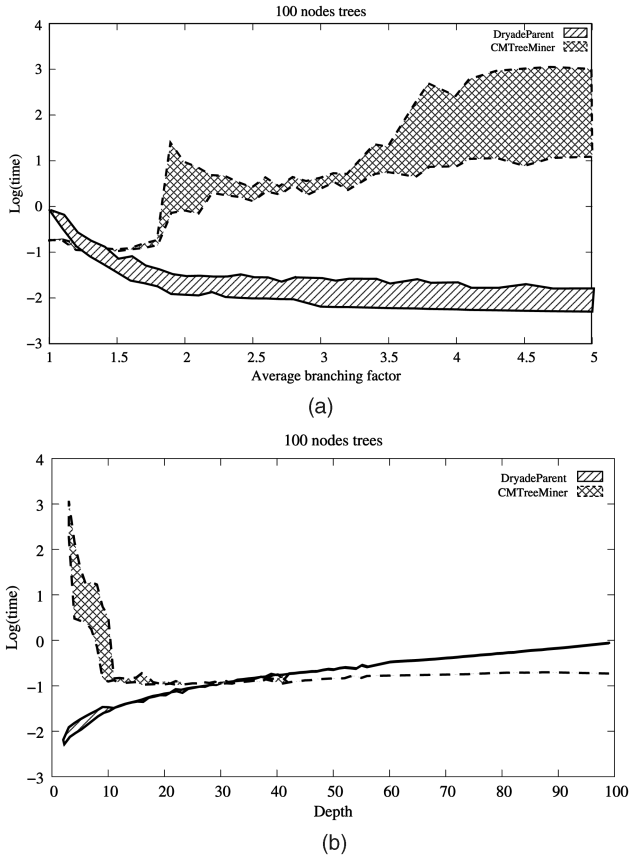

Fig. 17. Random trees with 100 nodes. (a) Log(time)/average branching factor. (b) Log(time)/depth.

number of tiles, polynomial on the number of nodes in the closed frequent trees, inversely proportional to the square of the average branching factor, linear with the size of the data, and linear with the average depth of the closed frequent trees. Such characteristics should allow good scale-up properties; this will be investigated in the next section.

5 Experiments

This section reports on the experimental validation of DRYADEPARENT on artificial and real-world data sets, as well as an application example on real XML data. The DRYADEPARENT algorithm will be compared with the state-of-the-art closed tree mining algorithm, CMTreeMiner [13], using the original C++ implementation of its authors. All runtimes are measured on a 2.8-GHz Intel Xeon processor with 2 Gbytes of memory (Rocks 3.3.0 Linux). DRYADEPARENT is written in C++, involving the closed frequent item set algorithm LCM2 [18], kindly provided by Takeaki Uno. The reported results are wall-clock runtimes, including data loading and preprocessing.

5.1 Artificial Data Sets

In the usual tree mining algorithms studies, at most, the length (that is, the number of nodes) of the found closed frequent trees is reported, without any information about the structure of these closed frequent trees. However, the branching factor and the depth of the closed frequent trees intervene directly in the candidate generation process, so

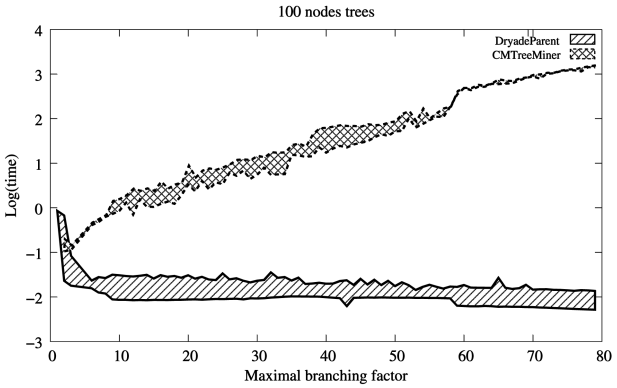

Fig. 18. Random trees with 100 nodes, log(time) with respect to the maximal branching factor.

they are likely to play a major role with respect to the computation time. To ascertain this hypothesis, we wrote a random tree generator that can generate trees with a given node number N and a given average branching factor b. Nodes are labeled with their preorder identifier, so there are no couples of nodes with the same label in a tree. We generated trees with $N = 100$ nodes and $b \in [1.0; 5.0]$, b increasing by an increment of 0.1. For each value of b, 10,000 trees were generated. Let T be such a tree. For each T, a data set D_T was generated, consisting simply of 200 identical copies of T (we perform this 200-time duplication of each T to increase the processing time for D_T and so reduce the error rate on time measurement). Each D_T was processed by both algorithms, with a support threshold of 200 (hence, the closed frequent tree to find is the tree T), and the processing time was recorded. Eventually, for each value of b, we regrouped the trees by their depth d and got a point (b, d) by averaging the processing times for all the trees of the average branching factor b and depth d. Fig. 17a shows the logarithms of these averaged time values with respect to the average branching factor b, and Fig. 17b shows the logarithms of these averaged time values with respect to the depth d.

Fig. 17a shows that DRYADEPARENT is orders of magnitude faster than CMTreeMiner as long as the branching factor exceeds 1.3, which is the case in most of the experiments' space. For lower branching factor values, CMTreeMiner has a small advantage. Closed frequent trees with such a low branching factor necessarily have a high depth, this is confirmed in Fig. 17b. This figure shows that DRYADEPARENT exhibits a linear dependency on the depth of the closed frequent trees. This is not surprising: each iteration of DRYADEPARENT computes one more depth level of the closed frequent trees, so very deep closed frequent trees will need more iterations.

CMTreeMiner, on the other hand, shows a dependency on the average branching factor, but for a given value of b, the computation time varies greatly, being especially high for low depth values. Because of the constraints on the random tree generator, a tree that has a low depth with a high average branching factor will necessarily have some nodes with a very large branching factor. We plotted in Fig. 18 a new curve, showing the computation time with respect to the *maximal* branching factor.

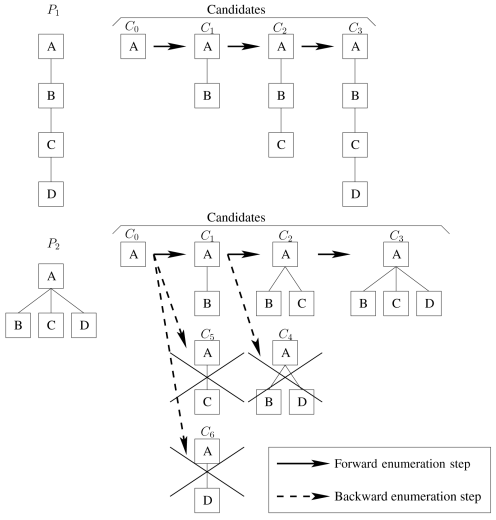

Fig. 19. CMTreeMiner candidate enumeration for a linear tree and for a flat tree.

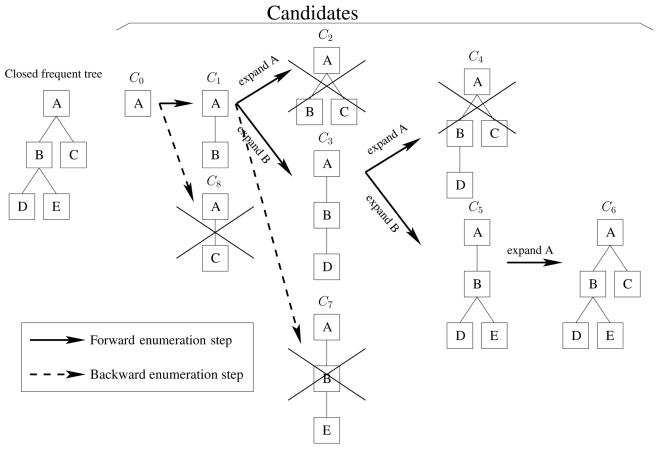

Fig. 20. CMTreeMiner enumeration for a left-balanced closed frequent tree.

DRYADEPARENT is nearly unaffected by the maximal branching factor, but the computation time of CMTreeMiner depends strongly on this parameter. In order to understand how much the behavior of CMTreeMiner and DRYADEPARENT differ, we analyze below the reasons of the dependency to the branching factor of CMTreeMiner and of the variability of its performances in general.

We give a brief reminder of the candidate enumeration technique of CMTreeMiner, the rightmost branch expansion. To generate candidates with k nodes from a frequent tree with $k-1$ nodes, CMTreeMiner tries to add a new edge connecting to a node of known frequent label and starting at a node of the rightmost branch of the $k-1$ node tree. All the nodes of the rightmost branch are explored successively in a top-down fashion, from the root to the rightmost leaf.

1. **Branching factor leads CMTreeMiner to generate more unclosed candidates by backtracking**. For a node with a high branching factor, finding correctly the set of its frequent children is a classical frequent item set mining problem, and the highly combinatorial nature of this problem often leads to the generation of useless candidates. CMTreeMiner is no exception to this rule: its top-down rightmost branch expansion technique finds very quickly all the children of a node but then needs to systematically backtrack to check for frequent subsets of these children. In most cases, this leads to the generation of nonclosed candidates. For example, compare the two closed frequent trees in Fig. 19. The linear tree P_1 is found without generating any unclosed candidates. However, the flat tree P_2 is found after the generation of three unclosed candidates, so according to our experiments finding P_2 needs 7 percent more time than finding P_1 in this simple setting with four nodes and 100 percent more time in a similar setting with 11 nodes.

   DRYADEPARENT also has to confront such a combinatorial problem in high-branching-factor cases, but it does so by using the LCM2 closed frequent item set mining algorithm, which provides, as of now, the most efficient way to explore the search space of closed frequent item sets. Furthermore, by discovering the tiles once and for all at the beginning of the algorithm, DRYADEPARENT avoids to repeat these complex computations if the same tile appears more than once in the closed frequent trees.

   In this problem, CMTreeMiner could probably be improved by modifying its enumeration technique in order to use LCM2 for sibling enumeration. Such a modified algorithm should be similar to the recent CLOTT algorithm by Arimura and Uno, which is an extension of the LCM2 principles to the closed attribute tree case.

2. **Candidate generation asymmetry**. The previous problem explains partly why CMTreeMiner is slower than DRYADEPARENT in most cases. As we have seen, this problem can theoretically be overcome. However, another problem remains, that cannot be overcome easily, and this problem is essential to the superior performances of our hooking strategy over any algorithm based on rightmost branch expansion.

   Consider the simple closed frequent tree in Fig. 20. As it can be seen, during candidate enumeration, unwanted candidates are generated, because the rightmost leaf expansion technique has to test "blindly" all the potential expansions on the rightmost branch but can only grow good candidates for certain expansions. For example, the candidate C_2 contains correct information: it corresponds to the first level of the closed frequent tree to find. However, as some expansions must be made on the node labeled B, which is not on the rightmost branch of C_2, C_2 is eliminated. In the same way, C_4 is computed for nothing. The children with label C of the root node will have to be recomputed in candidate C_6, even if it could have been discovered much earlier.

   This behavior is not only suboptimal, it also undermines the robustness of CMTreeMiner. Consider the two closed frequent trees in Fig. 21. Except for the names of the labels, both these closed frequent trees exhibit the same tree structure, so it is expected that they are discovered in exactly the

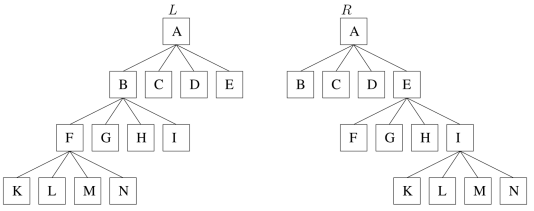

Fig. 21. L: left-balanced closed frequent tree. R: right-balanced closed frequent tree.

TABLE 1
Computation Time for Finding Closed Frequent Trees R and L

Closed Frequent Tree	R	L
CMTreeMiner	0.0010 s	0.0015 s
DRYADEPARENT	0.0013 s	0.0013 s

same amount of time. However, assuming that the sibling processing order is the ascending order of labels (this is the case in the actual implementation of CMTreeMiner), closed frequent tree R, which is right-balanced, is the ideal case for enumeration by rightmost tree expansion. CMTreeMiner will check 43 candidates to discover it. On the other hand, the left-balanced closed frequent tree L is the worst case, and CMTreeMiner will require to check 79 candidates for its discovery. The computation times reflect this difference in candidate checking: the time for finding L is 50 percent higher than the time for finding R, as shown in Table 1.

On the other hand, thanks to its tree-orientation neutral hooking technique, DRYADEPARENT requires exactly the same amount of time for processing these two closed frequent trees. For both L and R, DRYADEPARENT will generate three candidates: 1) the initial tile with root A, 2) a candidate generated by the hooking of a tile on respectively B or E, and 3) the closed frequent tree L or R by the hooking of another tile on, respectively, F or I.

Last, we compared the scalability of DRYADEPARENT and CMTreeMiner in both time and space in Fig. 22. The data set consists of 1,000 to 10,000 copies of a unique perfect binary tree of depth 5. We can see that in both time and space, DRYADEPARENT scales linearly. The memory usage is higher for DRYADEPARENT, but here, the reason is mostly implementation specific: for example, the DRYADEPARENT integer type is "integer," whereas CMTreeMiner's one is "short," which is four times smaller on our 64-bit machine. Moreover, DRYADEPARENT's internal representation for trees is based on trees of pointers, which uses the most memory, especially on a machine where the pointers are 8 bytes long.

Complexity issues. Here, we evaluate the validity of our complexity analysis in Section 4 when compared to the actual results for the artificial data set.

Fig. 23 compares the logarithm of the processing time for the real algorithm with the logarithm of the complexity formula in Section 4 with respect to 1) the number of tiles and 2) the average branching factor (the linear behavior of the algorithm with respect to the depth has already been ascertained in Fig. 17b). For a given number of tiles

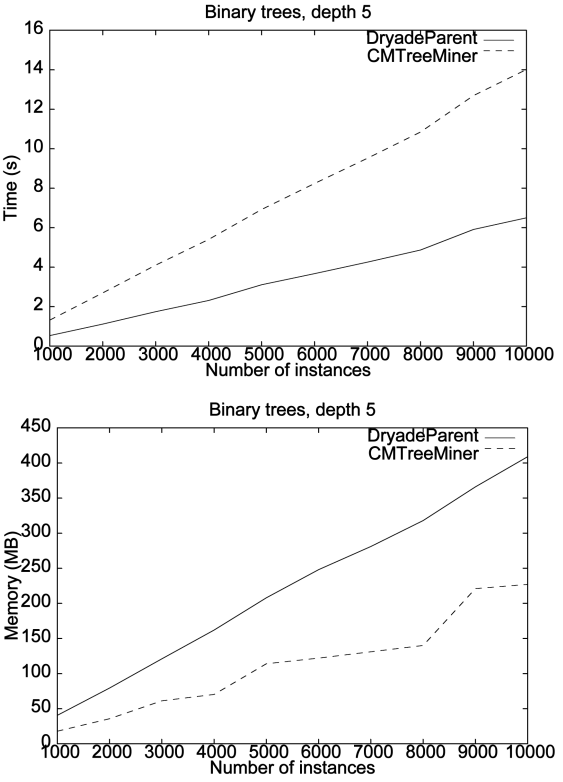

Fig. 22. Scalability tests on binary trees (time-memory).

(Fig. 23a) or average branching factor (Fig. 23b), there are several trees with different shapes satisfying this constraint, leading to different processing times or estimates. The shaded area in the figures represents all of these processing times (for the real algorithm) or estimates (for the complexity estimate).

In Fig. 23a, the estimated curve and the real times match well for more than 50 tiles, but for a lesser number of tiles, the real-time curve presents a gentler slope than the complexity estimate. For a high number of tiles, DRYADEPARENT spends most of its time on hooking tiles, with a lot of iterations. The start-up time needed for loading the data and creating all the needed data structures is negligible compared to the total time in these cases. However, for lower numbers of tiles, there are fewer iterations, and DRYADEPARENT is very fast at completing them. Therefore, the start-up time is no longer negligible compared to the total times in such cases. Such start-up processings are not taken into account in the complexity formula; hence, the difference occurs between the two curves.

The same behavior can be observed in Fig. 23b, for a high branching factor cases the real algorithm performs fewer hookings and hence is limited by the start-up time, which is not reflected in the complexity estimate.

One can also note a visible discontinuity on the curves for the complexity estimate. This discontinuity reflects the behavior of our artificial data generator. For average branching factors above 1.9, the generator is allowed to produce nodes with a very high branching factor, whereas it is not possible for branching factors below it. This allows the efficient generation of artificial trees satisfying the given constraints, at the price of smoothness. The curves for

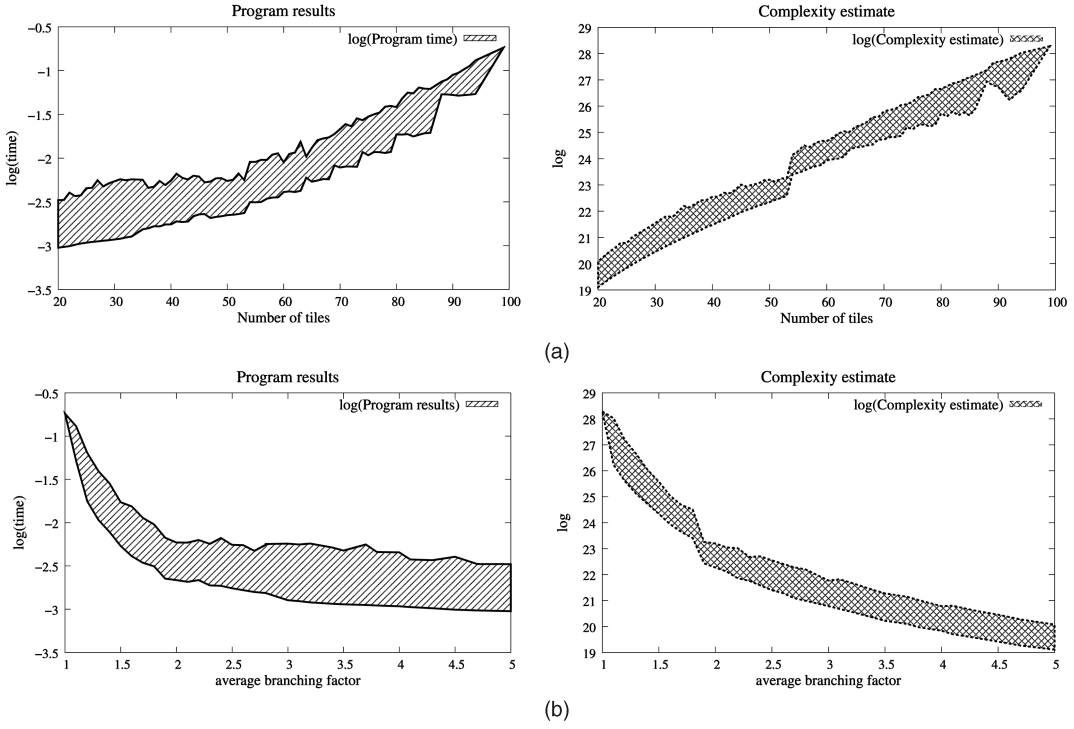

Fig. 23. Comparing real processing time and estimated complexity. (a) Log(time)/number of tiles. (b) Log(time)/average branching factor.

DRYADEPARENT also present this discontinuity, although it is less visible.

In conclusion, the complexity estimates that we provided seem to capture well the behavior of the DRYADEPARENT algorithm, especially when the algorithm has enough hooking work to do.

5.2 Real Data Sets

In the tree mining literature, two real-world data sets are widely used for testing: the NASA data set sampled by Chi et al. from multicast communications during a shuttle launch event [25] and the CSLOGS data set consisting of Web logs collected over one month at the Computer Science Department of Rensselaer Institute [19].

The runtimes obtained for various frequency thresholds for both DRYADEPARENT and CMTreeMiner are displayed in Fig. 24.

DRYADEPARENT is more than twice faster than CMTreeMiner on the CSLOGS data set. For the NASA data set, the performances are similar for high and medium support values, DRYADEPARENT having a distinct advantage for the lowest support values. Note that we obtained similar results with simplified CSLOGS and NASA data sets consisting only of attribute trees. We were interested to know why DRYADEPARENT and CMTreeMiner have a bigger performance difference on the CSLOGS data set than on the NASA data set. Analyzing the structure of the computed closed frequent trees in both cases, we found that in the CSLOGS data set, for the support value 0.003 (lowest value tested), there are 924 closed frequent trees, with three nodes on the average, and an average branching factor of 1.6. For the NASA data set, the picture is different: at the support value 0.1, there are 737 closed frequent trees, with 42 nodes on the average, an average depth of 12, and an average branching factor of 1.2.

Discussion. Our artificial experiments have shown that the structure of the closed frequent trees to find, especially their branching factor, is a crucial performance factor. The closed tree mining algorithm CMTreeMiner, based on candidate enumeration by rightmost branch expansion, has performances that vary considerably with the branching

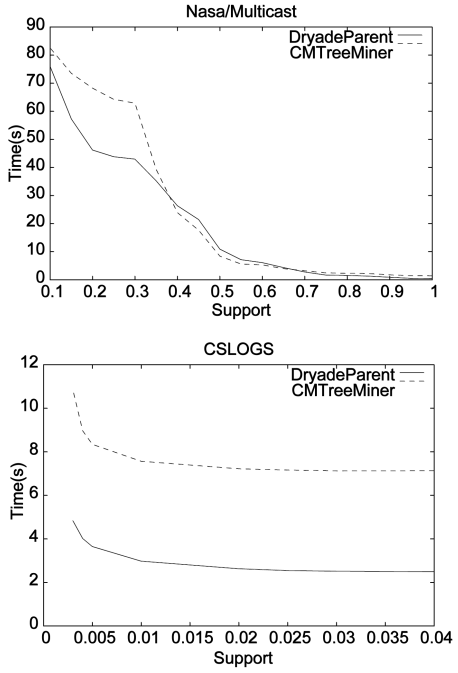

Fig. 24. Runtime with respect to support for the NASA/Multicast and CSLOGS data sets.

factor of the closed frequent trees and even with their balance. The fact that CMTreeMiner and DRYADEPARENT have similar performances on the NASA data set, with closed frequent trees having a quite low branching factor, and that CMTreeMiner is slower than DRYADEPARENT on the CSLOGS data set, with closed frequent trees having a higher branch factor, is consistent with our experiments on artificial data.

Experiments have shown that the new method for finding closed frequent attribute trees of our DRYADEPARENT algorithm is not only computation-time efficient but also robust with respect to the tree structure, delivering good performances with most tree structure configurations. Such robustness is a desirable feature for most applications, especially the applications that deal with trees having a great diversity of structure, for which the typical structure of closed frequent trees cannot be predicted.

5.3 XML Application Example

In this last series of experiments, we show the analysis of a corpus of real XML data. This corpus comes from the *XML Mining Challenge*, compiled by Denoyer [26]. We used the "Movie" corpus, initially designed for a mapping task. The training part of this corpus has the advantage to contain well-formed XML documents with meaningful tags, each document describing one movie.

Preprocessing. We preprocessed this corpus in the following way:

- To all leaf tags corresponded a *PCDATA string* giving the value associated to this tag (for example, a tag "name" could have as its associated PCDATA "John Wayne"). All the PCDATA of these tags were processed to get rid of punctuation signs and convert the text into lowercases. In case of strings with spaces, like in "John Wayne," the spaces were replaced by underscores, with a prefixing underscore, like in "_john_wayne" (so that the Perl parser we used could handle numeral strings like "_1941"). These normalized strings were used as labels of new nodes added as children of the labeled nodes that the original strings were values of.
- We made minor alterations to the structure in order to convert the original trees to attribute trees. For this, tags that represented list items were replaced with their children, that is, by their actual content. For example, each actor in a movie was represented by a tag named "entry" under a main "cast" tag, and inside this "entry" tag were a "name" and one or more "roles" tags. We suppressed these intermediary tags and instead created a new tag with the actual name of the actor, which became a child of the "cast" tag. The roles of this actor became children of the tag bearing the name of the actor.
- There are two tags, "synopsis" and "review," whose data is a short text respectively describing the movie and reviewing it. Each text was cut into words; the stopwords like "the," "and," etc., were suppressed from this list of words. For each word, only one of its occurrences was kept, and all the remaining words became new tags added as leaves of the "synopsis" and "review" tags.

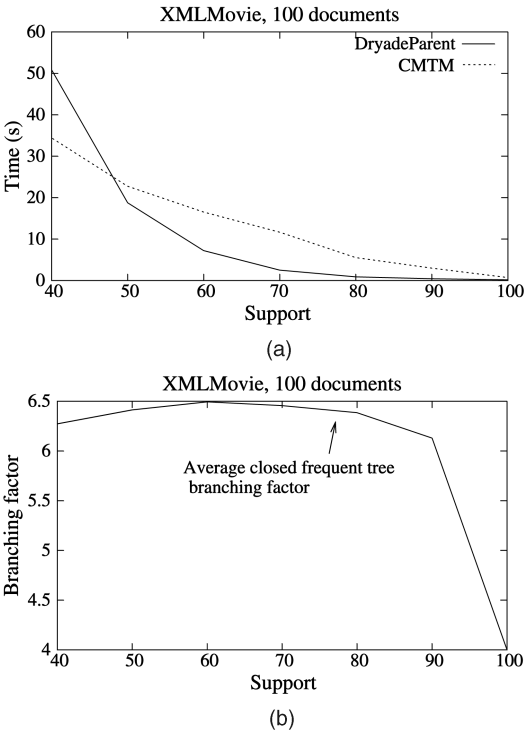

Fig. 25. Comparative results for DRYADEPARENT and CMTreeMiner with the XMLMovie data set. (a) Computation time. (b) Average closed frequent tree branching factor.

Performances. In the first experiment, we preprocessed 100 documents out of the 693 from the collection and fed them as input to DRYADEPARENT and CMTreeMiner in order to analyze the computation time performances. The results are given in Fig. 25a, with the average branching factor of the closed frequent trees given in Fig. 25b. There are no results under a support value of 40 percent, as in this case, DRYADEPARENT saturated the memory.

The closed frequent trees have a high branching factor: as expected from the previous experiments, for high support values, DRYADEPARENT largely outperforms CMTreeMiner. Surprisingly, for lower support values, the contrary happens. To understand why this was happening, we analyzed carefully the time spent by DRYADEPARENT in its various tasks. We found that for a support value of 40 percent, it was spending 74 percent of its computation time making closure tests (which corresponds to line 5 of Algorithm 2). The problem is that, as we stated in the introduction, DRYADEPARENT has been designed with heterogeneous data sets in mind, that is, data from various organizations about the same topics. Because of the very nature of such data sets, they are currently very difficult to find. The publicly available data sets, like the one we use here, usually come from the same organization and so are very homogeneous. The consequence is that a lot of closed frequent trees are nearly identical, and so, a lot of hookings also resemble each other, with subtle differences. Our closure test has been written in a rather naive way: it first looks for exact hooking matches in the database and then for bigger (the current hooking is included in a bigger

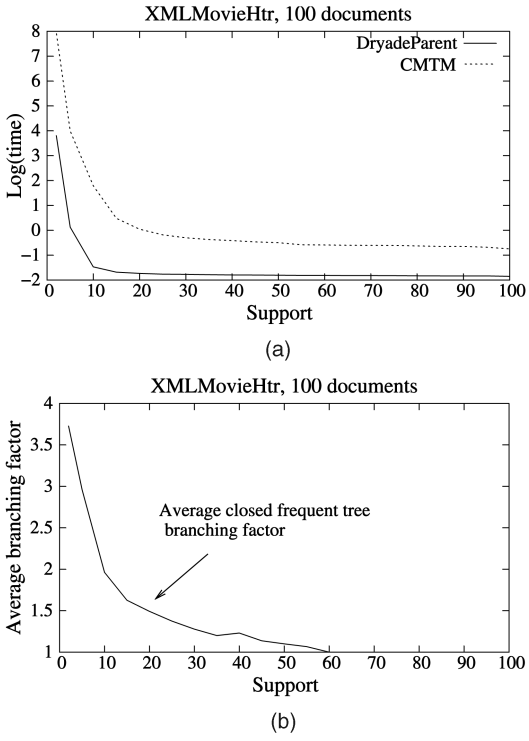

Fig. 26. Comparative results for DRYADEPARENT and CMTreeMiner with the XMLMovieHtr data set. (a) Logarithm of computation time. (b) Average closed frequent tree branching factor.

hooking of the database) and smaller matches (the current hooking includes a smaller hooking of the database), iterating on all the possible cases of bigger/smaller matches. Usually, this is very fast because there are not that many cases to examine, but with a very homogeneous data set such as XMLMovie, it became a bottleneck. On the other hand, the edge-adding strategy of CMTreeMiner performs better here: the fact that all the closed frequent trees resemble each other means that it has fewer candidates to expand, so what is a bad case for DRYADEPARENT is a good case for CMTreeMiner.

To evaluate the behavior of both algorithms on more heterogeneous data, we derived a new data set from XMLMovie. The XMLMovie documents are all rooted with the "movie" tag. We divided our 100 documents into 10 groups $\{G_1, \ldots, G_{10}\}$, chose 10 arbitrary tags $\{t_1, \ldots, t_{10}\}$, and, in each group G_i, for all the documents of this group, made the tag t_i replace the tag "movie" at the root of the tree so that for all $i \in [1, 10]$, the documents in G_i are rooted by tag t_i. The new data set, called XMLMovieHtr, is much more heterogeneous: all documents are on the same topic and share common tags; however, the small difference in the roots avoids homogeneity and gives more importance to the closed frequent trees not using this root.

The performances for the processing of XMLMovieHtr are shown in Fig. 26a, with the average branching factor of the closed frequent trees shown in Fig. 26b.

This time, both algorithms could correctly process the data for all support values. The computation time difference was important between DRYADEPARENT and CMTreeMiner, so we had to use a logarithmic scale for the time in Fig. 26a. For all support values, even if the closed frequent trees to find were complex and numerous (more than 20,000 at $\text{support} = 2$ percent), DRYADEPARENT could achieve a several-order-of-magnitude improvement over CMTreeMiner, processing the data in 45 seconds for a support of 2 percent, whereas CMTreeMiner needed 2,846 seconds. Therefore, as expected, DRYADEPARENT is far better adapted for heterogeneous data than CMTreeMiner.

Closed frequent trees analysis. We now show how the closed frequent trees found could be useful in an XML data mining application. Our first interest, in such homogeneous data, is to find a schema common to all of the documents analyzed, which could stand for a very basic DTD (the *Document Type Definition*, or DTD, is the "grammar" of an XML document. More resources about XML can be found in [27]), especially in cases like XMLMovie where the documents are homogeneous, but no DTD was formally defined. This is close to grammatical inference [28], but the goal of grammatical inference on XML data is to find the complete DTD of all the XML documents [29], which is beyond the scope of frequent tree mining. Therefore, we ran DRYADEPARENT on XMLMovie with a support of 100 percent; the closed frequent tree found is shown in Fig. 27. We are assured that all the documents will contain this closed frequent tree, which can, for example, be useful for designing queries on these documents.

With lower support values, the closed frequent trees also allow extracting precise information from the data. The next closed frequent trees come from the mining of XMLMovieHtr with a support value of 5 percent.

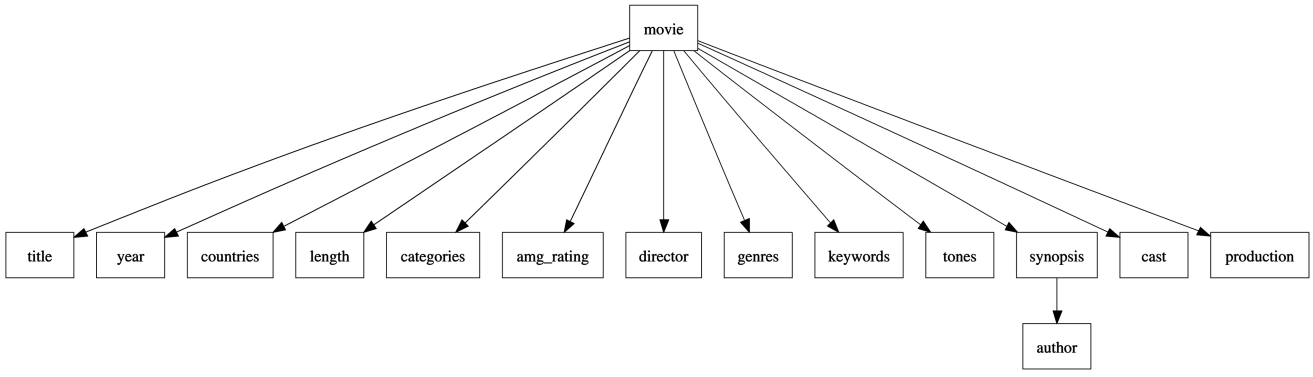

Fig. 27. Common schema to all of the 100 documents of XMLMovie.

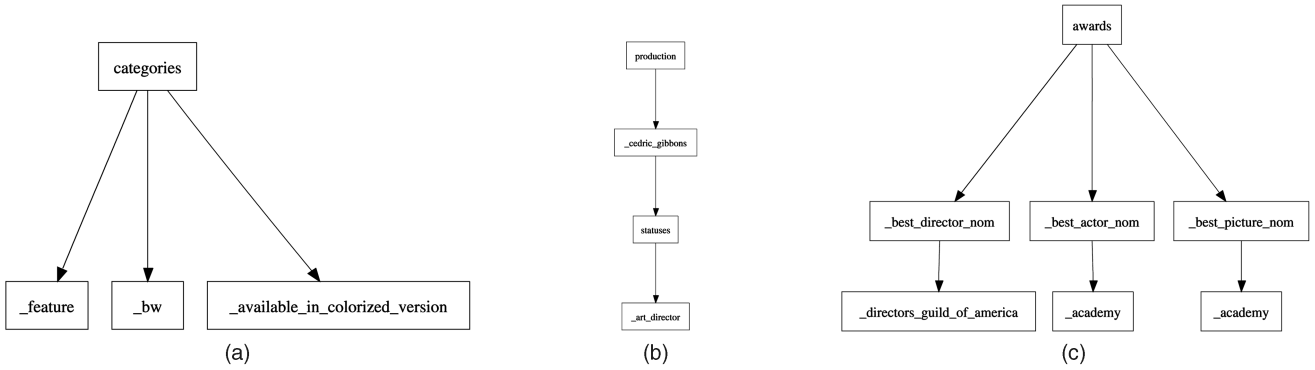

Fig. 28. Some closed frequent trees extracted from XMLMovieHtr with a support value of 5 percent. (a) Horizontal closed frequent tree. (b) Vertical closed frequent tree. (c) Hybrid closed frequent tree.

Following the node nesting, we have the following:

- The information can have a horizontal organization, like in Fig. 28a. Here, the closed frequent tree could have been found by a simple frequent item set mining algorithm, it represents the frequent children of the "categories" node. We learn that "features" that are in black and white ("bw") often are available in a colorized version.
- The information can have a vertical organization, like in Fig. 28b. Here, through the nesting of nodes expressing data and of nodes expressing what this data represents, we can learn that the individual "Cedric Gibbons" was a member of the production team of at least five movies and that his job was to be the art director.
- Finally, most closed frequent trees, like in Fig. 28c, combine vertically and horizontally organized information, which is the major advantage of tree mining. Here, we learn that at least 5 percent of the movies are nominated ("_nom" suffix after an award name) for the following awards: best director for the directors' guild of America and best actor and best picture for the Academy awards. This closed frequent tree can lead to many interpretations, for instance, that good movies are associated with the combination of a good director and a good main actor.

Some closed frequent trees can allow an even finer analysis of the data. Consider the closed frequent tree in Fig. 29, extracted from XMLMovieHtr with a support of 2 percent.

This big closed frequent tree is shared by two movies "Rebel without a Cause" (1955) and "The Graduate" (1962). Both of these films are American, were big successes, won awards, and have a five-star AMG rating. However, more importantly, the closed frequent tree was able to capture well what is common to these movies: they are movies about "coming of age" and "generation gap" (from the keywords), words like "rebellion" and "parents" appear in the synopsis, and words like "generation," "parents," "alienation," and "prosperity" appear in the review. Such words seem to characterize well both films made in a prosperous America, but where the young people were less and less attracted by their parents' model and tried to find another way of life. Searching Google with these two film names together confirmed that these two films are grouped together by sociologists when analyzing the America of the 1950s and 1960s (see, for example, [30], available at http://www.lib.berkeley.edu/MRC/nickray.html). The shopping site Amazon.com also rates the two movies as similar. Therefore, what is very interesting with that closed frequent tree is that it could group together two similar movies and even provide elements to describe what make them similar. Such kind of closed frequent trees are particularly useful for conceptual clustering, by grouping together similar elements and characterizing the cluster. The particular advantage here, due to the structural analysis of XML data, is that it provides very fine grained information, which could be particularly useful to people doing a detailed analysis of the data.

6 Conclusion and Perspectives

In this paper, we have presented the DRYADEPARENT algorithm, based on the computation of tiles (closed frequent attribute trees of depth 1) in the data and on an efficient hooking strategy that reconstructs the closed frequent trees from these tiles. This hooking strategy is radically different from current tree mining approaches like CMTreeMiner. Whereas CMTreeMiner uses a classical generate-and-test strategy to build candidate trees edge by edge, the hooking strategy of DRYADEPARENT finds a complete depth level at each iteration and does not need expensive tree mapping tests.

Thorough experiments have shown that DRYADEPARENT is faster than CMTreeMiner in most settings. Moreover, the performances of DRYADEPARENT are robust with respect to the structure of the closed frequent trees to find, whereas the performances of CMTreeMiner are biased toward trees having most of their edges on their rightmost branch.

We also have shown that in the analysis of XML data, as long as the data is heterogeneous, DRYADEPARENT can provide excellent performances, allowing a near-real-time analysis. We also have shown that the closed frequent trees found could capture very interesting information from the data.

We have proposed new benchmarks, taking into account the structure of the closed frequent trees, to test the behavior of tree mining algorithms. As far as we know, such kind of tests is new in the tree mining community.

Improving these benchmarks and making more detailed analyses are some of our future research directions. We think that our experiments proved that such tools are valuable for the tree mining community. We also plan to extend DRYADEPARENT to structures more general than attribute trees.

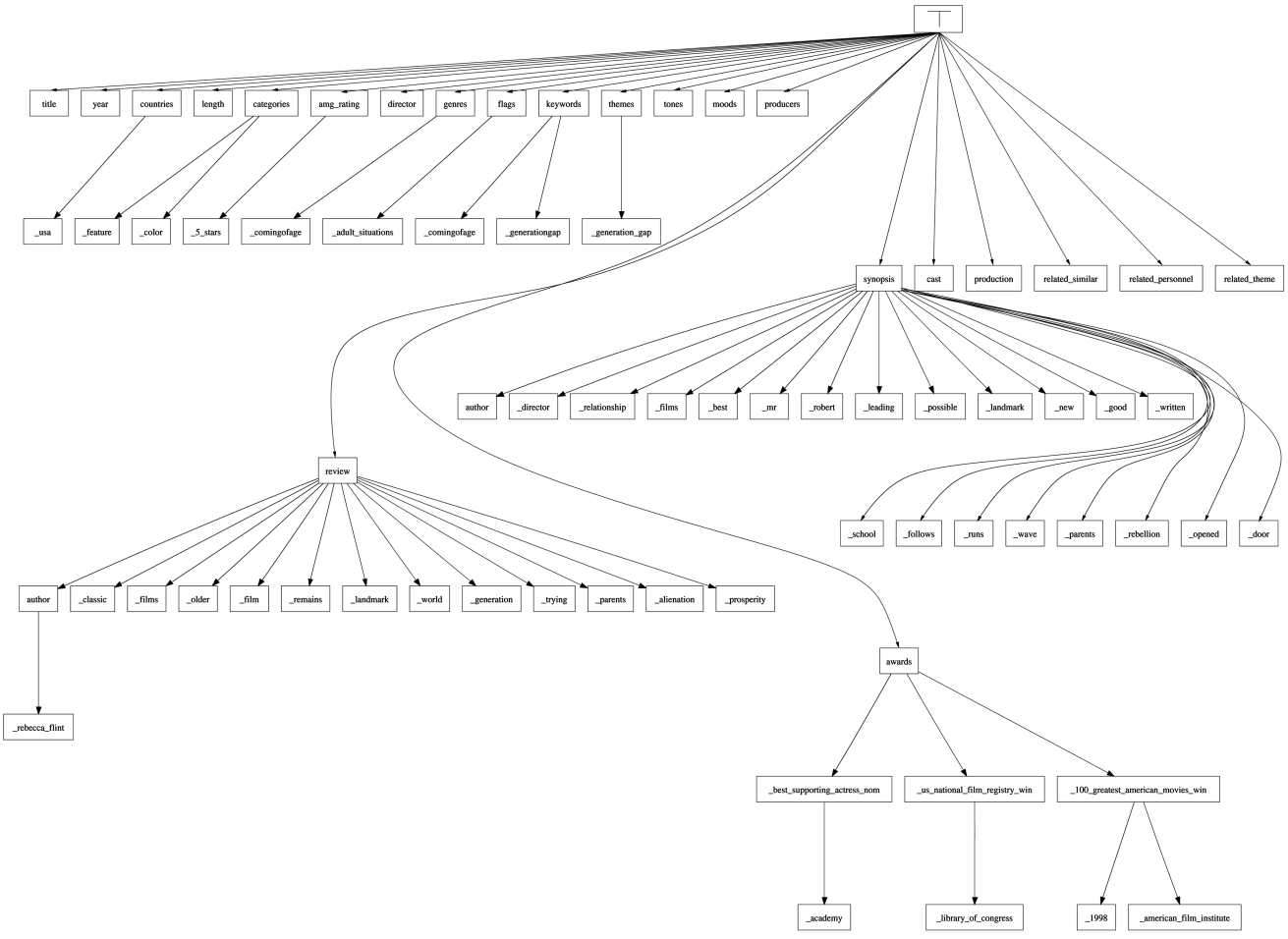

Fig. 29. Biggest closed frequent tree from XMLMovieHtr with a support of 2 percent.

ACKNOWLEDGMENTS

The authors wish to thank especially Takeaki Uno for the LCM2 implementation and Yun Chi for making available the CMTreeMiner implementation and giving us the NASA data set. This work was partly supported by the grant-in-aid of scientific research No. 16-04734.

REFERENCES

[1] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. 11th Int'l Conf. Data Eng. (ICDE '95)*, P.S. Yu and A.S.P. Chen, eds., pp. 3-14, citeseer.ist.psu.edu/agrawal95mining.html, 1995.

[2] H. Mannila, H. Toivonen, and A.I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259-289, citeseer.ist.psu.edu/mannila97discovery.html, 1997.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, "Efficient Substructure Discovery from Large Semi-Structured Data," *Proc. Second SIAM Int'l Conf. Data Mining (SDM '02)*, pp. 158-174, Apr. 2002.

[4] A. Inokuchi, T. Washio, and H. Motoda, "Complete Mining of Frequent Patterns from Graphs: Mining Graph Data," *Machine Learning*, vol. 50, no. 3, pp. 321-354, 2003.

[5] M. Kuramochi and G. Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 9, pp. 1038-1051, Sept. 2004.

[6] J.-H. Cui, J. Kim, D. Maggiorini, K. Boussetta, and M. Gerla, "Aggregated Multicast—A Comparative Study," *Proc. Second Int'l IFIP-TC6 Networking Conf. (NETWORKING '02): Networking Technologies, Services, and Protocols; Performance of Computer and Comm. Networks; and Mobile and Wireless Comm.*, pp. 1032-1044, 2002.

[7] D. Shasha, J.T.L. Wang, and S. Zhang, "Unordered Tree Mining with Applications to Phylogeny," *Proc. 20th Int'l Conf. Data Eng. (ICDE '04)*, p. 708, 2004.

[8] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 8, pp. 1021-1035, Aug. 2005.

[9] L.H. Yang, M.L. Lee, W. Hsu, and S. Acharya, "Mining Frequent Query Patterns from XML Queries," *Proc. Eighth Int'l Conf. Database Systems for Advanced Applications (DASFAA '03)*, p. 355, 2003.

[10] M.J. Zaki and C.C. Aggarwal, "XRules: An Effective Structural Classifier for XML Data," *Proc. ACM SIGKDD '03*, citeseer.ist.psu.edu/zaki03xrules.html, 2003.

[11] A. Termier, M. Rousset, and M. Sebag, "Dryade: A New Approach for Discovering Closed Frequent Trees in Heterogeneous Tree Databases," *Proc. Fourth IEEE Int'l Conf. Data Mining (ICDM '04)*, pp. 543-546, 2004.

[12] H. Arimura and T. Uno, "An Output-Polynomial Time Algorithm for Mining Frequent Closed Attribute Trees," *Proc. 15th Int'l Conf. Inductive Logic Programming (ILP '05)*, 2005.

[13] Y. Chi, Y. Yang, Y. Xia, and R.R. Muntz, "CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees," *Proc. Eighth Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD '04)*, 2004.

[14] P. Kilpeläinen, "Tree Matching Problems with Applications to Structured Text Databases," PhD dissertation, Technical Report A-1992-6, Dept. of Computer Science, Univ. of Helsinki, Nov. 1992.

[15] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering Frequent Closed Itemsets for Association Rules," *Proc. Seventh Int'l Conf. Database Theory (ICDT '99)*, 1999.

[16] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB '94)*, 1994.

[17] M.J. Zaki and C.-J. Hsiao, "Charm: An Efficient Algorithm for Closed Itemset Mining," *Proc. Second SIAM Int'l Conf. Data Mining (SDM '02),* Apr. 2002.
[18] T. Uno, M. Kiyomi, and H. Arimura, "LCM v.2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets," *Proc. Second Workshop Frequent Itemset Mining Implementations (FIMI '04),* 2004.
[19] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest," *Proc. Eighth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '02),* July 2002.
[20] T. Asai, H. Arimura, T. Uno, and S. Nakano, "Discovering Frequent Substructures in Large Unordered Trees," *Proc. Sixth Int'l Conf. Discovery Science (DS '03),* pp. 47-61, 2003.
[21] S. Nijssen and J.N. Kok, "Efficient Discovery of Frequent Unordered Trees," *Proc. First Int'l Workshop Mining Graphs, Trees and Sequences (MGTS '03),* 2003.
[22] Y. Xiao, J.-F. Yao, Z. Li, and M.H. Dunham, "Efficient Data Mining for Maximal Frequent Subtrees," *Proc. Third IEEE Int'l Conf. Data Mining (ICDM '03),* p. 379, 2003.
[23] M.J. Zaki, "Efficiently Mining Frequent Embedded Unordered Trees," *Fundamenta Informaticae,* special issue on advances in mining graphs, trees and sequences, vol. 65, nos. 1-2, pp. 33-52, Mar./Apr. 2005.
[24] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Dynamic Programming," *Introduction to Algorithms,* second ed., pp. 323-369, MIT Press, 2001.
[25] R. Chalmers and K. Almeroth, "Modeling the Branching Characteristics and Efficiency Gains of Global Multicast Trees," *Proc. IEEE INFOCOM '01,* Apr. 2001.
[26] L. Denoyer, "XML Mining Challenge," http://xmlmining.lip6.fr/Corpus, 2006.
[27] W. Consortium, Extensible Markup Language (XML) 1.0, fourth ed., http://www.w3.org/TR/REC-xml/, 2006.
[28] E. Gold, "Language Identification in the Limit," *Information and Control,* vol. 10, pp. 447-474, 1967.
[29] Y. Papakonstantinou and V. Vianu, "DTD Inference for Views of XML Data," *Proc. ACM SIGMOD,* 2000.
[30] I.C. Jarvie, "America's Sociological Movies," *Arts in Soc.,* vol. 10, no. 2, pp. 171-181, Summer-Fall 1973.

Alexandre Termier received the PhD degree from the University of Paris-South in 2004. He is an assistant professor of computer science at the University of Grenoble. His research interests include data mining, parallelism, and peer-to-peer networks. For data mining, he is especially interested in mining trees and directed acyclic graphs.

Marie-Christine Rousset is a professor of computer science at the University of Grenoble. Her research interests include knowledge representation and information integration. In particular, she works on the following topics: logic-based mediation between distributed data sources, query rewriting using views, automatic classification and clustering of semistructured data (for example, XML documents), peer to peer data sharing, distributed reasoning. She has published more than 70 refereed international journal articles and conference papers and participated in several cooperative industry-university projects. She received a best paper award from AAAI in 1996 and has been nominated as an ECCAI Fellow in 2005. She has served in many program committees of international conferences and workshops and on the editorial boards of several journals.

Michèle Sebag received the degree in Maths from Ecole Normale Supérieure, Paris, and the PhD degree in computer science in 1990 and her Habilitation in 1997. She has been with the Centre National de la Recherche Scientifique (CNRS) since 1991, where she has been a senior researcher (Directeur de Recherche) since 2003. Primarily grounded in applications for numerical engineering, her research interests include relational learning and inductive logic programming, ensemble methods, evolutionary computation and genetic programming, and statistical learning. She is on the editorial boards of the *Machine Learning Journal* and *Genetic Programming and Evolvable Hardware*. She is an associate editor of *Knowledge and Information Systems* and was an associate editor of the *IEEE Transactions on Evolutionary Computation* from 1997 to 2003.

Kouzou Ohara received the ME degree in information and computer sciences and the PhD degree from Osaka University in 1995 and 2002, respectively. He is currently an assistant professor in the Institute of Scientific and Industrial Research, Osaka University. His research interests include machine learning, data mining, and personalization of intelligent systems. He is a member of the IEEE, the AAAI, the IEICE, the IPSJ, and the JSAI.

Takashi Washio received the PhD degree in nuclear engineering from Tohoku University, Japan, in 1983, on the topic of process plant diagnosis based on qualitative reasoning. He is a professor in the Institute of Scientific and Industrial Research (ISIR), Osaka University. At ISIR, he works on the study of scientific discovery, graph mining, and high-dimensional data mining. He received the best paper award from the Atomic Energy Society of Japan in 1996, the best paper award from the Japanese Society for Artificial Intelligence in 2001, and the Journal Award of Computer Aided Chemistry in 2002. He is a member of the IEEE Computer Society.

Hiroshi Motoda received the BS, MS, and PhD degrees in nuclear engineering from the University of Tokyo. He is a professor emeritus at Osaka University and a scientific advisor of the Asian Office of Aerospace Research and Development, Air Force Office of Scientific Research, US Air Force Research Laboratory (AFOSR/AOARD). His research interests include machine learning, knowledge acquisition, scientific knowledge discovery, and data mining. He is a member of the steering committee of PAKDD, PRICAI, DS, and ALT. He received the best paper award twice from the Atomic Energy Society of Japan (in 1977 and 1984) and three times from JSAI (in 1989, 1992, and 2001), the outstanding achievement award from JSAI in 2000, and the Okawa Publication Prize from the Okawa Foundation in 2007. He is a member of the IEEE Computer Society.