Computation-time efficient and robust attribute tree mining with DRYADEPARENT

Alexandre Termier[1], Marie-Christine Rousset[2], Michèle Sebag[2], Kouzou Ohara[1], Takashi Washio[1], and Hiroshi Motoda[1]

[1] I.S.I.R., Osaka University
8-1, Mihogaoka, Ibarakishi, Osaka, 567-0047, Japan termier@ar.sanken.osaka-u.ac.jp
[2] CNRS & Université Paris-Sud (LRI) & INRIA (Futurs)
Building 490, Université Paris-Sud, 91405 Orsay Cedex, France

Abstract. In this paper, we present a new tree mining algorithm, DRYADEPARENT, based on the hooking principle first introduced in DRYADE [1]. In the experiments, we demonstrate that the branching factor and depth of the frequent patterns to find are key factor 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 datasets where the frequent patterns have a high branching factor.

1 Introduction

In the last ten years, the frequent pattern discovery task of data mining has expanded from simple itemsets to more complex structures: for example sequences, episodes, trees or graphs. 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, bioinformatics, XML documents databases mining, 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 high depth and high branching factor. Especially the high branching factor case has received very little attention by the tree mining community. However, performances of existing algorithms are dramatically affected by the branching factor of the patterns to find, as shown in our experiments.

Starting from this observation, we have developped the DRYADEPARENT algorithm. This algorithm is an adaptation of our earlier algorithm DRYADE [1]. 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 [3, 2] to make possible performance comparison with other existing systems based on different principles. We will show in this paper that DRYADEPARENT outperforms the up-to-date CMTreeMiner algorithm [2], and conduct a thorough study on the influence of structural characteristics of the patterns to find, like depth and branching factor, on the computing time performance of both algorithms.

The outline of the paper is 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 datasets. In section 6, we conclude and give some directions for future work.

2 Formal Background

Let $L = \{l_1, ..., l_n\}$ be a set of labels. A *labelled 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 labelling function $\varphi : N \mapsto L$ assigning a label to each node of the tree. We assume without loss of generality that edges are unlabelled: 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 [3]).

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. 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.

Tree inclusion

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 an *induced subtree* of 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 $AT \sqsubseteq T$, and we will sometimes say that AT is included into T.

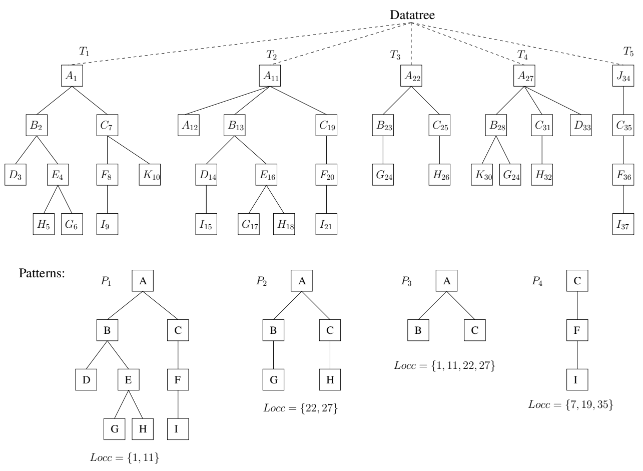


Fig. 1: Datatree example (node identifiers are subscripts of node labels), and patterns for $\varepsilon = 2$

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 $\mathcal{EM}(AT, T)$. The set of *occurrences* of AT in T, denoted $Locc(AT, T)$, is the set of nodes of T onto which the root of AT is mapped by a mapping of $\mathcal{EM}(AT, T)$.

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 $\mathcal{EM}(AT, T)$.

Frequent attribute trees

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

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 $AT \sqsubseteq T_i$, 0 otherwise. *(document support)*
- $support_o(AT) = \sum_{i=1}^{m} \sigma_o(AT, T_i)$ where $\sigma_o(AT, T_i) = |Locc(AT, T_i)|$ *(occurrences 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. But for sake of completeness, our algorithm needs to keep track of all frequent occurrences, and will use the occurrences support for processing.

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 abuse of notation we will only denote it as $\mathcal{F}$ in the rest of this paper.

In the example of Fig. 1, with a support threshold of $\varepsilon = 2$, the attribute trees P_1, P_2, P_3, P_4 are all frequent by document support in the datatree.

Closed trees

A frequent attribute tree is *closed* if it is maximal, according to inclusion, for its set of occurrences.

Definition 1. *A frequent attribute tree $AT \in \mathcal{F}$ is closed either if it is not included into any other frequent attribute trees, or if it is included into a frequent attribute tree $AT' \in \mathcal{F}$, there exists a mapping in $\mathcal{EM}(AT, D_{TD})$ which 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.

In our example P_1 and P_2 are closed because they are not included into any other frequent attribute tree, P_3 is closed because even if it is included into P_1 and P_2, neither the occurrences of P_1 nor the occurrences of P_2 can cover all the occurrences of P_3, and in the same way P_4 is also closed as even if it is included into P_1, its mapping starting at occurrence 35 is not contained in any mapping of P_1.

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 smaller than the number of all frequent attribute trees, but the amount of information is the same in both cases: all 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.

From now on, we will refer to the closed frequent attribute trees as *patterns*.

3 Related work

Most tree mining algorithms deal with finding *all* the frequent subtrees from a collection of trees. One pioneering work is Asai & al.'s Freqt algorithm [4], discovering all frequent induced subtrees with preservation of the order of the siblings. The other pioneering work is Zaki's *TreeMiner* [5], using 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 these algorithms extend the Apriori algorithm [9] principle to trees: they use efficient candidate tree generation procedures, that cover all the search space without generating twice the same tree, and for each candidate test its frequency against the data. The enumeration technique builds a new candidate by adding one edge to a previously found frequent tree, along its *rightmost branch.*

The second generation of tree mining algorithms has been designed to get rid of the order preservation constraint. This was realised by basing the enumeration procedures on canonical forms, one canonical form representing all trees that are isomorphic except for the order of siblings. Such work include the Unot algorithm by Asai & al. [6], the work of Nijssen & al. [7] and the recent Sleuth algorithm by Zaki [8].

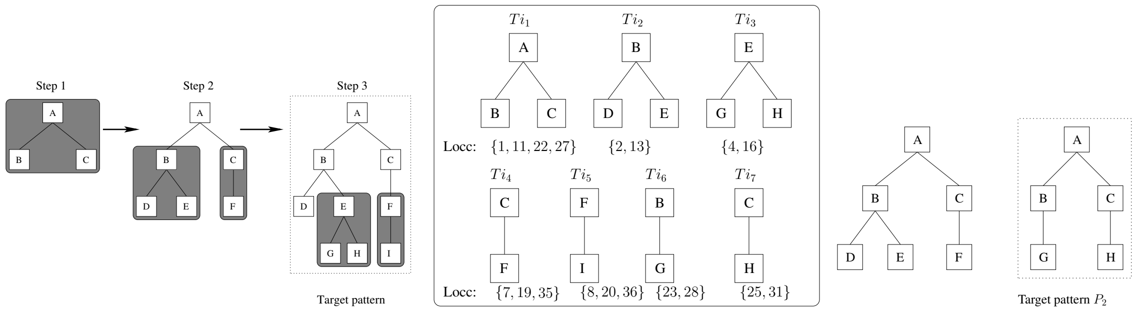
There are still very few algorithms mining closed frequent trees. We already mentioned our DRYADE algorithm [1], which relies on a very general tree inclusion definition and a new *hooking* principle. The only algorithm mining closed frequent induced subtrees is the CMTreeMiner algorihm of Chi & al. [2]. 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 & Uno proposed the CLOTT algorithm [3] for mining closed frequent attribute trees, in the same settings as those of 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 which depends heavily on the structure of the patterns to find. In case of big patterns with high depth and high branching factor, many edge-adding steps are needed to find these 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

We propose in this section an improved way of exploring the search space, namely the DRYADEPARENT algorithm. This method provides important perfomance gains for complex patterns.

Our goal is to find all the patterns in $\mathcal{C}$. Instead of discovering them edge by edge as done by most algorithms, we are interested in discovering the patterns depth level by depth level, starting with the root and finishing with the deepest leaves in a breadth-first fashion. An example of this discovery process is shown in Fig. 2a.



(a) Construction process (b) Tiles of our example (c) Hookings at iteration 1

Fig. 2: Tiles and hookings

We call *tiles* the attribute trees that must be added to discover a new depth level of a pattern. In Fig. 2a such tiles are enclosed in shaded boxes. A tile is a frequent attribute tree made from a node of a pattern of $\mathcal{C}$ and all its children. The set of all tiles is noted $\mathcal{T}$.

The essence of the DRYADEPARENT algorithm is to first discover all the tiles that are in the datatree, and then apply a fast levelwise strategy to *hook* these tiles together and compute the patterns of $\mathcal{C}$.

4.1 Discovering the tiles

Let us denote by $\mathcal{F}_1$ the subset of $\mathcal{F}$ made of the frequent attribute trees of depth 1. The set of closed frequent attribute trees of depth 1, denoted by $\mathcal{C}_1$, is the closure of $\mathcal{F}_1$, defined with Definition 1 where $\mathcal{F}$ is replaced by $\mathcal{F}_1$. DRYADEPARENT relies on the following property:

Property 1. The set of tiles is exactly the set of closed frequent attribute trees of depth 1, i.e.: $\mathcal{T} = \mathcal{C}_1$

Proof (sketch): ($\mathcal{T} \subseteq \mathcal{C}_1$) Consider a tile $T \in \mathcal{T}$, we have $T \in \mathcal{F}_1$, and by negation if there was a $T' \in \mathcal{F}_1$ preventing the closure of T, then the pattern of $\mathcal{C}$ which T comes from would not be closed as well, hence the negation. ($\mathcal{T} \supseteq \mathcal{C}_1$) Let $C \in \mathcal{C}_1$, as C is frequent it is included in at least one pattern of $\mathcal{C}$, so there exists a tile $T \in \mathcal{T}$ such that $C \sqsubseteq T$ and the occurrences of C are included in those of T, but by negation if C is different of T, then T contests the closure of C, hence the negation. □

Computing such closed frequent attribute trees of depth 1 can be efficiently done by constructing for each label l a matrix M_l where each line corresponds to a node of label l in the datatree, and with as many columns as labels in L, so that a 1 in cell (i, j) indicates that the node corresponding to line i has a child with label l_j. Applying a closed frequent itemset discovery algorithm like LCM2 [10] on matrix M_l will discover all the closed frequent attribute trees of depth 1 with a root of label l (to comply with document support, one only has to prune the occurence-frequent trees that do not meet the document frequency constraint). By repeating the process for all the labels of L, all the closed frequent attribute trees of depth 1, i.e. all the tiles, are discovered. The Fig. 2b shows the tiles for our example.

4.2 Hooking the tiles

Having found the tiles, the goal of DRYADEPARENT is to compute efficiently all the patterns through hookings of these tiles. We have chosen a levelwise strategy, where each iteration computes the next depth level for the patterns being constructed.

Root tiles To begin with, the tiles that correspond to the depth levels 0 and 1 of the patterns must be found in the set of tiles. Such tiles are called *root tiles*, for they are the starting point of pattern construction by our *hooking* principle. Some of these root tiles can be found in a straightforward manner: these are the tiles whose root cannot be mapped to the same node as the leaves of any other tiles. We call them *initial root tiles*. In our example Ti_1 is the only initial root tile because its occurrences 1, 11, 22 and 27 are never leaves in any other tile.

The other root tiles are not as easily found. As some of their root nodes can be mapped to the leave nodes of other tiles, these tiles are subtrees in some patterns, and root in some other patterns. For example, Ti_4 is as well a subtree in P_1, and the root tile of P_4. For sake of efficiency, it must be avoided as much as possible to identify incorrectly a tile as a root tile, this would lead to the construction of an attribute tree that would in fact only be a subtree of a pattern, i.e. having done redudant computation and getting an unclosed result. To avoid this, the starting points of DRYADEPARENT are the initial root tiles, and at the end of each iteration new root tiles are looked for. We will explain how in the "Preparing next iteration" subsection.

In the following, we will denote by $\mathcal{RP}_i$ the attribute trees actually constructed by the algorithm at iteration i, and by $\mathcal{C}_{\mathcal{RP}_i}$ the patterns that will be obtained by successive hookings on the attributes 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 $\mathcal{C}_{\mathcal{RP}_0} = \{P_1, P_2, P_3\}$ of Fig. 1.

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 w.r.t.

the patterns to get. This is done via the **hooking** operation: for an integer i, let T be an element of $\mathcal{RP}_i$, and $C \in \mathcal{C}_{\mathcal{RP}_i}$ such that the structures of T and C are isomorphic for all depth until i. The *hooking* operation consists in constructing a new attribute tree T' by hooking a set of *hooking tiles* $\{Ti_1, ..., Ti_k\}$ on the leaves of T such that the occurrences of T' include those of C, and the structures of T' and C are isomorphic for all depths until $i + 1$.

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 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 these potential hooking tiles, we want to find those which frequently appear together according to the occurrences of T. This is a closed frequent itemset discovery problem, and we can solve it by creating a matrix M whose each line k corresponds to an occurrence o_k of T, and each column j corresponds to a potential hooking tile Ti_j. $M[i, j] = 1$ iff. 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 itemset discovery algorithm like LCM2 on M enables discovering efficiently all the closed frequent hooking tile sets.

In our example, the frequent hooking tile sets on Ti_1 are $\{Ti_2, Ti_4\}$ and $\{Ti_6, Ti_7\}$. These hookings are illustrated in Fig. 2c. It can be seen that the pattern P_2 has been discovered.

The frequent attribute trees discovered that are not yet patterns are inserted into $\mathcal{RP}_{i+1}$ for further expansion in the next iteration.

Preparing next iteration *Next level root tiles:* Once all the uses of a tile as a hooking tile have been discovered, either all the occurrences of this tile have been involved in a hooking for the creation of a particular attribute tree, or some occurrences have never been used together in a hooking. In the latter case, this tile becomes a root tile at the next iteration, for we are sure that starting from all the occurrences of this tile will produce a new closed attribute tree. For example at the end of the first iteration, all the possible hookings of Ti_4 have been discovered. But the occurrence 35 of this tile has never been used, so tile Ti_4 becomes a root tile in iteration 2, enabling the discovery of pattern P_4.

This way all the patterns of $\mathcal{C}$ are discovered by DRYADEPARENT (this is proved in an even more general case in [11]).

Closure checking: Another important point is that in some cases hooking can lead to attribute trees that are not closed. Such cases can be detected quickly by analysing the hookings, for this purpose DRYADEPARENT keeps a database of all the hookings performed so far. When a new hooking is proposed, the algorithm checks that this new hooking satisfies the closure property w.r.t. the hookings of the database. Two non-closure cases can arise: 1) the new hooking is included into an existing hooking, then the new hooking is discarded; 2) the new hooking includes an existing hooking, then the existing hooking and the corresponding pattern are erased from the database, and a new pattern is created from the new hooking, which is registered into the hooking database.

The whole algorithm is summed up in Algorithm 1.

5 Experiments

This section reports on the experimental validation of DRYADEPARENT on real-world and artificial datasets. All runtimes are measured on 2.8 GHz Intel Xeon processor with 2GB memory (Rocks 3.3.0 Linux). DRYADEPARENT is written in C++, involving the closed frequent itemset algorithm LCM2 [10], kindly provided by Takeaki Uno. Reported results are wall-clock runtimes, including data loading and preprocessing.

Algorithm 1 The DRYADEPARENT algorithm

```
Input: A datatree $D_{TD}$ and an absolute frequency threshold $\varepsilon$
Output: The set $\mathcal{C}$ of all the patterns in $D_{TD}$ with frequency $\geq \varepsilon$
 1: $\mathcal{RP}_0 \leftarrow$ initial root tiles of $D_{TD}$
 2: $i \leftarrow 0$ ; $\mathcal{C} \leftarrow \emptyset$
 3: while $\mathcal{RP}_i \neq \emptyset$ do
 4:   $\mathcal{RP}_{i+1} \leftarrow \emptyset$
 5:   for all $RT \in \mathcal{RP}_i$ do
 6:     if no hooking is possible on $RT$ then
 7:       $\mathcal{C} \leftarrow \mathcal{C} \cup RT$
 8:     else
 9:       $\mathcal{RP}_{i+1} \leftarrow \mathcal{RP}_{i+1} \cup Hookings(RT)$ // Closure detection is performed in the Hookings procedure
10:     end if
11:   end for
12:   $\mathcal{RP}_{i+1} \leftarrow \mathcal{RP}_{i+1} \cup DetectNewRootTiles$
13:   $i \leftarrow i + 1$
14: end while
15: Return $\mathcal{C}$
```

5.1 Real datasets

In the tree mining litterature, two real-world datasets are widely used for testing: the NASA dataset sampled by Chi & al. from multicast communications during a shuttle launch event [12], and the CSLOGS dataset consisting of web logs collected over one month at the CS department of Rensselaer Institute [5].

The runtimes obtained for various frequency thresholds for both DRYADEPARENT and CMTreeMiner are displayed on Fig. 3.

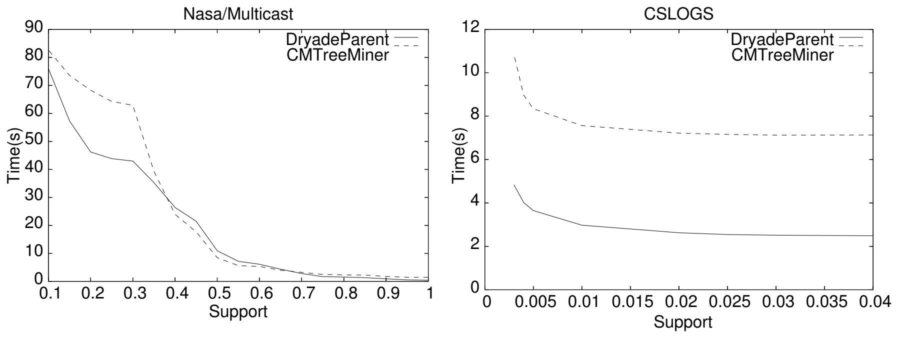
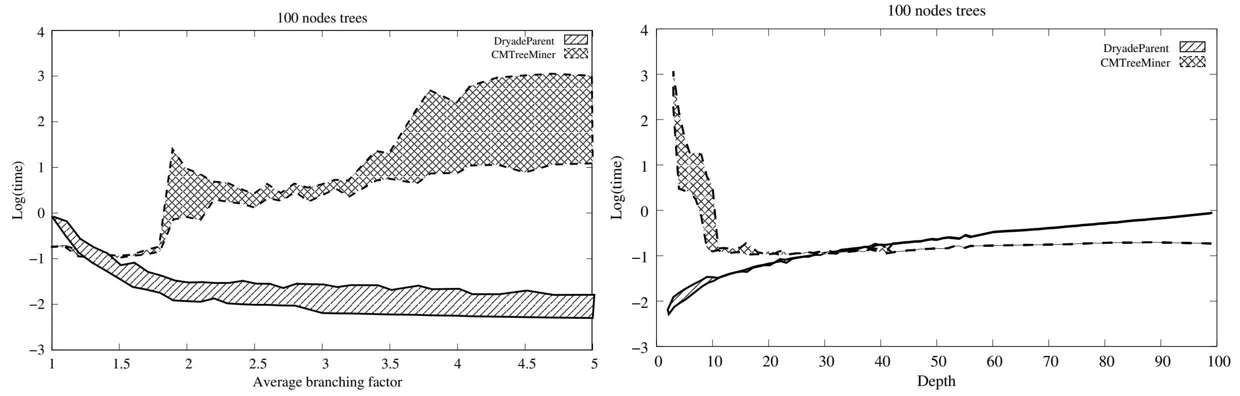


Fig. 3: Running time w.r.t. support for the Nasa/Multicast and CSLOGS datasets.

DRYADEPARENT is more than twice faster than CMTreeMiner on the CSLOGS dataset. For the NASA dataset 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 datasets consisting only of attribute trees. We were interested to know why DRYADEPARENT and CMTreeMiner have a bigger performance difference on the CSLOGS dataset than on the NASA dataset. Analysing the structure of the computed patterns in both cases, we found that in the CSLOGS dataset, for the support value 0.003 (lowest value tested), there are 924 patterns, with 3 nodes on average, and an average branching factor of 1.6. For the NASA dataset, the picture is different: at the support value 0.1, there are 737 patterns, with 42 nodes on average, an average depth of 12 and an average branching factor of 1.2. So patterns of NASA and patterns of CSLOGS have very different characteristics, and lead to different performance results for CMTreeMiner and DRYADEPARENT. Artificial datasets will be used to get a deeper understanding of the influence of the structure of the patterns on performance of these two algorithms.

5.2 Artificial datasets

In the usual tree mining algorithms studies, at most the length (i.e. the number of nodes) of the found patterns is reported, without any information about the structure of these patterns. However, branching factor and depth of the patterns intervene directly in the candidate generation process, so they are likely to play a major role w.r.t. 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 labelled with their pre-order 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 increment of 0.1. For each value of b, 10000 trees were generated. Let T be such a tree. For each T a dataset D_T was generated, consisting simply of 200 identical copies of T (we perform this 200-times 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 pattern 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 average branching factor b and depth d. Fig. 4a shows the logarithms of these averaged time values w.r.t. the average branching factor b, and Fig. 4b shows the logarithms of these averaged time values w.r.t. the depth d.



(a) Log(time)/average branching factor

(b) Log(time)/depth

Fig. 4: Random trees with 100 nodes

The Fig. 4a shows that DRYADEPARENT is orders of magnitudes faster than CMTreeMiner as long as the branching factor exceeds 1.3, that is the case in most of the experiments space. For lower branching factor values, CMTreeMiner has a small advantage. Patterns with such a low branching factor necessarily have a high depth, this is confirmed by Fig. 4b. This figure shows that DRYADEPARENT exhibits a linear dependency on the depth of the patterns. This is not surprising: each iteration of DRYADEPARENT computes one more depth level of the patterns, so very deep patterns 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 have a low depth with a high average branching factor will necessarily have some nodes with a very large branching factor. We plotted in Fig. 5 a new curve, showing the computation time with respect to the *maximal* branching factor.

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 branching factor of CMTreeMiner, and of the variability of its performances in general.

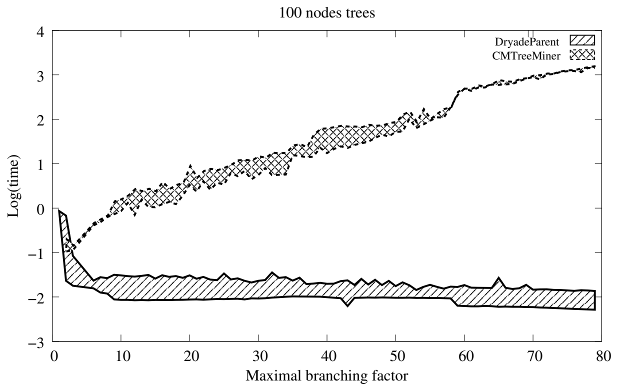


Fig. 5: Random trees with 100 nodes, log(time) w.r.t. maximal branching factor

We give a brief reminder of the candidate enumeration technique of CMTreeMiner, called rightmost branch expansion. To generate candidates with k nodes from a frequent tree with $k-1$ nodes, CMTreeMiner tries to add a new edge leading to a frequent node 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 high branching factor, finding correctly the set of its frequent children is a classical frequent itemset 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 chidren of a node, but then systematically needs to backtrack to check for frequent subsets of these children. In most cases, this leads to the generation of non-closed candidates. For example, compare the two patterns of Fig. 6. The linear pattern P_1 is found without generating any unclosed candidates. But the flat pattern P_2 is found after the generation of 3 unclosed candidates, so according to our experiments finding P_2 needs 7% more time than finding P_1 in this simple setting with 4 nodes, and 100% more time in a similar setting with 11 nodes.

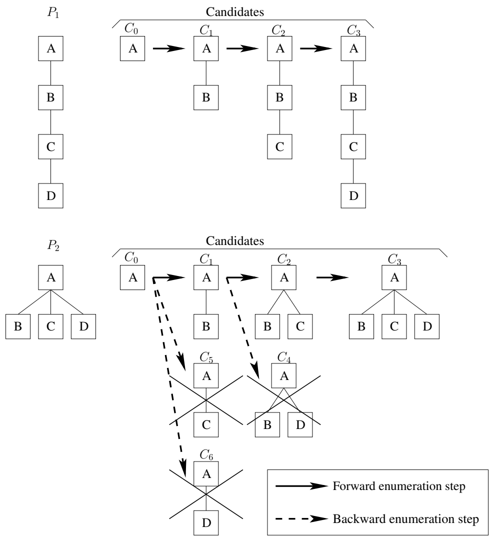


Fig. 6: CMTreeMiner candidate enumeration for a linear pattern and for a flat pattern

DRYADEPARENT also has to confront such a combinatorial problem in high branching factor cases, but it does so by using the LCM2 closed frequent itemset mining algorithm, which provides as of now the most efficient way to explore the search space of closed frequent itemsets. Furthemore, 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 patterns.

On 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 pattern of Fig. 7. As it can be seen, during candidate enumeration, unwanted can-

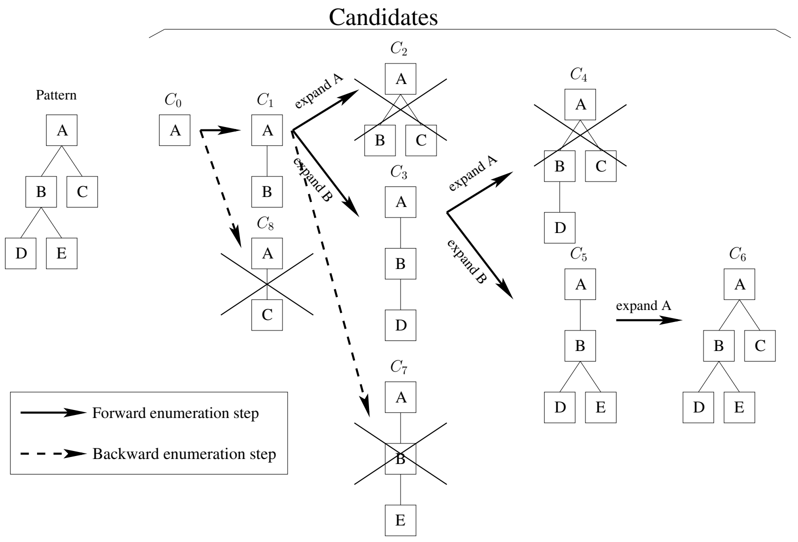


Fig. 7: CMTreeMiner enumeration for a left-balanced pattern

didates 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 pattern to find. But as some expansions must be made on the node labelled B, which is not on the rightmost branch of C_2, then 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 sub-optimal, it also undermines the robustness of CMTreeMiner. Consider the two patterns of figure 8.

Except for the names of labels, both these patterns exhibit the same tree structure, so it is expected that they are discovered in exactly the 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), pattern R, which is right-balanced, is an ideal case for enumeration by rightmost tree expansion. CMTreeMiner will check 43 candidates to discover it. Oppositely, the left-balanced pattern G is a worst case, and CMTreeMiner will require to check 79 candidates for its discovery. The computation times reflect this difference in candidates checking: time for finding G is 50 % higher than time for finding R, as shown in Tab. 1.

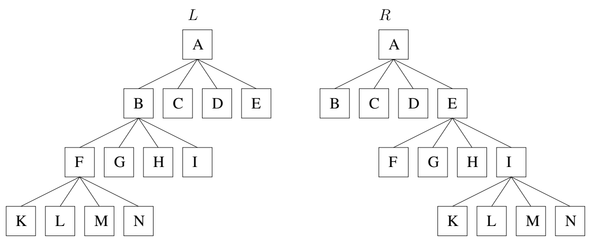


Fig. 8: L: left-balanced pattern, R: right-balanced pattern

Pattern	R	L
CMTreeMiner	0.0010 s	0.0015 s
DRYADEPARENT	0.0013 s	0.0013 s

Table 1: Computation time for finding patterns R and G

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

Last, we compared the scalability of DRYADEPARENT and CMTreeMiner both on time and space in Fig. 9. The dataset consists of 1000 to 10000 copies of a unique perfect binary tree of depth 5. We can see

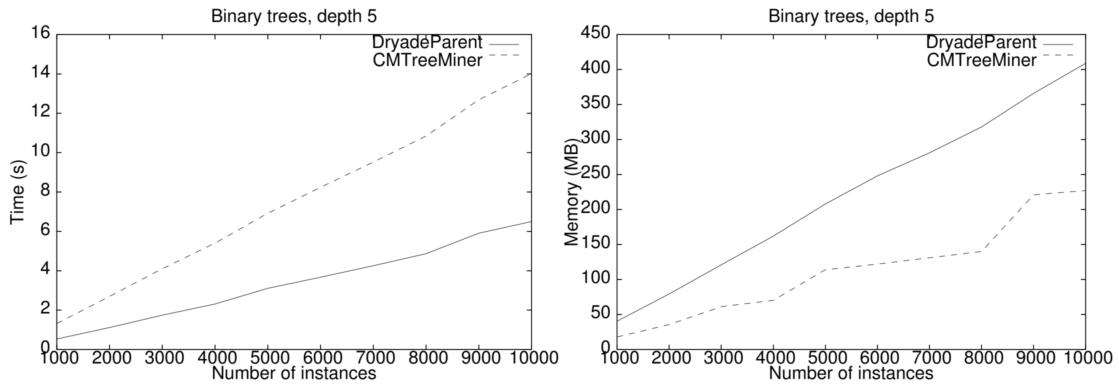


Fig. 9: Scalability tests, binary trees (time - memory)

that both on time and space, DRYADEPARENT scales linearly. The memory usage is higher for DRYADEPARENT, but here the reason is mostly implementation specific: for example DRYADEPARENT integer type is "integer" whereas CMTreeMiner's one is "short", which is 4 times smaller on our 64 bit machine. And DRYADEPARENT 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.

Discussion: Our artificial experiments have shown that the structure of the patterns to find, and 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 which vary considerably with the branching factor of the patterns, and even with their balance. The fact that CMTreeMiner and DRYADEPARENT have similar performances on the NASA dataset, with patterns having quite low branching factor, and that CMTreeMiner is slower than DRYADEPARENT on the CSLOGS dataset, with patterns having a higher branch factor, is consistent with our experiment 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 w.r.t. tree structure, delivering good performances with most tree structure configurations. Such a robustness is a desirable feature for most applications, especially the applications which deal with trees having a great diversity of structure, which cannot predict what will be the typical structure of patterns.

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 patterns from these tiles.

Thorough experiments have shown that DRYADEPARENT is faster than CMTreeMiner in most settings, and that its performances are robust w.r.t. the structure of the patterns to find.

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

Improving these benchmarks and making more detailed analyses is one 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.

Acknowledgements: We wish to thank especially Takeaki Uno for the LCM2 implementation, and Yun Chi for making available the CMTreeMiner implementation and giving us the Nasa dataset. This work is partly supported by the grant-in-aid of scientific research No. 16-04734.

References

1. Termier, A., Rousset, M., Sebag, M.: Dryade : a new approach for discovering closed frequent trees in heterogeneous tree databases. In: International Conference on Data Mining ICDM'04, Brighton, England. (2004) 543–546
2. Chi, Y., Yang, Y., Xia, Y., Muntz, R.R.: Cmtreeminer: Mining both closed and maximal frequent subtrees. In: The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04). (2004)
3. Arimura, H., Uno, T.: An output-polynomial time algorithm for mining frequent closed attribute trees. In: 15th International Conference on Inductive Logic Programming (ILP'05). (2005)
4. Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA. (2002) 158–174
5. Zaki, M.J.: Efficiently mining frequent trees in a forest. In: In Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2002)
6. Asai, T., Arimura, H., Uno, T., ichi Nakano, S.: Discovering frequent substructures in large unordered trees. In: the Proc. of the 6th International Conference on Discovery Science (DS'03). (2003) 47–61
7. Nijssen, S., Kok, J.N.: Efficient discovery of frequent unordered trees. In: First International Workshop on Mining Graphs, Trees and Sequences, 2003. (2003)
8. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae, special issue on Advances in Mining Graphs, Trees and Sequences **65** (2005) 33–52
9. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB Conference, Santiago, Chile (1994)
10. Uno, T., Kiyomi, M., Arimura, H.: Lcm v.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: 2nd Workshop on Frequent Itemset Mining Implementations (FIMI'04). (2004)
11. Termier, A.: Extraction of frequent trees in an heterogeneous corpus of semi-structured data: application to xml documents mining. Technical Report 1388, LRI (2004) http://www.lri.fr/∼termier/publis/phdTermierEN.ps.gz.
12. Chalmers, R., Almeroth, K.: Modeling the branching characteristics and efficiency gains of global multicast trees. In: Proceedings of the IEEE INFOCOM'2001. (2001)