Efficient mining of high branching factor attribute trees

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

[1] : I.S.I.R., Osaka University
8-1, Mihogaoka, Ibarakishi, Osaka, 567-0047, Japan
{termier, ohara, washio, motoda}@ar.sanken.osaka-u.ac.jp

[2] : CNRS & Université Paris-Sud (LRI) & INRIA (Futurs)
Building 490, Université Paris-Sud, 91405 Orsay Cedex, France.
{mcr, sebag}@lri.fr

Abstract

In this paper, we present a new tree mining algorithm, DRYADEPARENT, *based on the hooking principle first introduced in* DRYADE *[9]. 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. 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 recent tree mining research, most algorithms use the same generate-and-test principle that made the success of frequent itemset mining algorithms. They usually deal with finding *all* the frequent subtrees from a collection of trees. Pioneering algorithms by Asai & al. [3] and Zaki [12] extend the Apriori algorithm [1] principle to trees, using tree inclusion definitions imposing the preservation of the order of the siblings. The second generation of tree mining algorithms used canonical forms to get rid of the order preservation constraint [4, 7, 13]. Newest algorithms tend to search only closed frequent subtrees, for the computation gains that can be achieved without quality loss: DRYADE [9], relying on a very general tree inclusion definition, CMTreeMiner [6], mining closed frequent induced subtrees and the recent CLOTT [2] for mining closed frequent attribute trees (not yet implemented).

We present DRYADEPARENT, a new closed tree mining algorithm, which replaces the costly generate-and-test approach followed by most existing algorithms by an elaborate hooking of subtrees of depth 1 to leaves of frequent trees of depth k.

Experiments show that our approach has faster computation times, and is nearly unaffected by the structures of the frequent patterns to find, whereas the state of the art algorithm exhibits a severe dependency on branching factor.

The outline of the paper is as follows. Section 2 introduces notations and definitions. Section 3 gives an overview of the DRYADEPARENT algorithm. Section 4 reports detailed comparative experiments. In section 5, we conclude and give some directions for future work.

2. Formal Background

Let $L = \{l_1, ..., l_n\}$ 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. We assume that the reader is familiar with the notions of *child*, *parent*, *ancestor* and *descendant* for the nodes of a tree.

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 [2]).

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: $\forall u \in N_1 \quad \varphi_1(u) = \varphi_2(\mu(u))$ and 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.

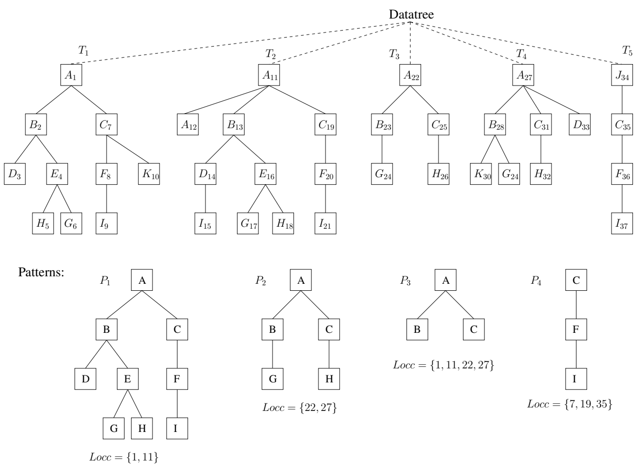


Figure 1: Datatree example (node identifiers are subscripts of node labels), and patterns for $\varepsilon = 2$

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

Frequent attribute tree: Let $TD = \{T_1, ..., T_m\}$ be a tree database. The *datatree* D_{TD} is the tree whose root is an unlabeled node, having the trees $\{T_1, ..., T_m\}$ as its direct subtrees.

Let ε be an absolute frequency threshold. AT is a frequent attribute tree of D_{TD} if $support_d(AT) \geq \varepsilon$, where $support_d(AT) = \sum_{i=1}^{m} \sigma_d(AT, T_i)$ with $\sigma_d(AT, T_i) = 1$ if $AT \sqsubseteq T_i$, 0 otherwise. The set of all frequent attribute trees is denoted by $\mathcal{F}(D_{TD}, \varepsilon)$, abbreviated as $\mathcal{F}$ in this paper.

Closed trees: A frequent attribute tree is *closed* if it is maximal, according to inclusion, for its set of occurrences, i.e.: a frequent attribute tree $AT \in \mathcal{F}$ is closed either if it is not included into any other frequent attribute tree, 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 $\mathcal{C}$.

In the example of Figure 1, the frequent attribute trees P_1 and P_2 are closed because they are not included into any other frequent attribute tree, P_3 is closed because though 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.

The tree mining problem we are interested in is to find all the closed frequent attribute trees for a given datatree and support threshold. From now on, we will refer to the closed frequent attribute trees as *patterns*.

3. The DRYADEPARENT algorithm

The goal of DRYADEPARENT is to find all the patterns in $\mathcal{C}$, depth level by depth level, starting with the root and finishing with the deepest leaves in a breadth-first fashion.

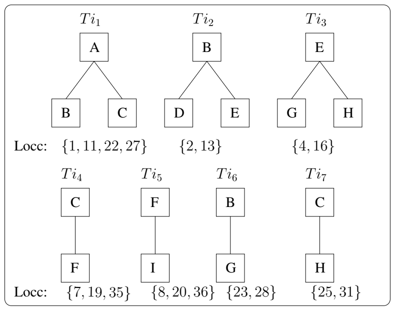

Locc: $\{1, 11, 22, 27\}$ $\{2, 13\}$ $\{4, 16\}$
Locc: $\{7, 19, 35\}$ $\{8, 20, 36\}$ $\{23, 28\}$ $\{25, 31\}$

Figure 2: Tiles of our example

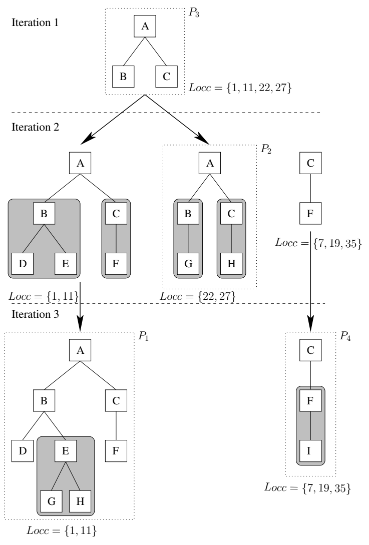


Figure 3: DRYADEPARENT discovery process

We only give in the following section the intuition behind the DRYADEPARENT algorithm, the interested reader is referred to [10] for more details.

3.1 Tiles

The essence of DRYADEPARENT is to build the patterns depth level by level through proper *hookings* (defined later) of the closed frequent attribute trees of depth 1, which we call *tiles*.

Finding such tiles can be reformulated as a propositional frequent itemset mining problem as follows: for each label, use a closed frequent itemset discovery algorithm like LCM2 [11] to compute all the closed frequent sets of children labels for this label. The Figure 2 shows the tiles for our example.

3.2 Hooking the tiles

The previously computed tiles can then be *hooked* together, i.e. a tile whose root has label l becomes a subtree of another tile having a leaf of label l, to build more complex trees. A proper strategy is needed to avoid as much as possible to construct attributes trees that would be found unclosed in a later iteration. Our strategy consists in constructing attributes trees which are isomorphic to the k first depth levels of the patterns, each iteration adding one depth level to the isomorphism. For this purpose, the first task of DRYADEPARENT is to discover in the tiles those corresponding to the depth levels 0 and 1 of the patterns, the *root tiles*. Some of these tiles can be found immediately for they cannot be hooked on any other tile, they will be the starting point for the first iteration of DRYADEPARENT. This is the case for Ti_1 in our example, as show in Figure 3. For the rest of the root tiles, they can also be used as building blocks for other patterns: they will be used as root of a pattern only when it will become clear that they are not only a building block, to avoid generating unclosed attribute trees. This is the case for Ti_4 in our example, which can be hooked on Ti_1. Only in iteration 2 will this tile be used as a root tile to construct pattern P_4. The computation of the set of tiles that must be hooked to the root tiles to make closed frequent attribute trees with one more depth level is delegated to a closed frequent item set mining algorithm. The attribute trees created by hooking become starting points for the next iteration.

The whole process is shown for our example in Figure 3. On the root tile Ti_1, one can either hook the tiles $\{Ti_2, Ti_4\}$ or the tiles $\{Ti_6, Ti_7\}$, the latter leading to the pattern P_2. Note the different occurrences of the two constructed attribute trees. From the hooking of $\{Ti_2, Ti_4\}$ on Ti_1, one can then hook the tile Ti_3, leading to the pattern P_1. The tile Ti_4 is not only a building block of P_1, it also have an occurrence which does not appear in P_1 (35): it is used as a root tile, and the only possible hooking on it is Ti_5, leading to the pattern P_4.

The soundness and completeness of this hooking mechanism has been proved in [8].

4. 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 [11], kindly provided by Takeaki Uno. Reported results are wall-clock runtimes, including data loading and preprocessing.

Real datasets: The runtimes obtained for various frequency thresholds for both DRYADEPARENT and CMTreeMiner are displayed on Figure 4, for the widely-known CSLOGS [12] and NASA [5, 6] tree datasets.

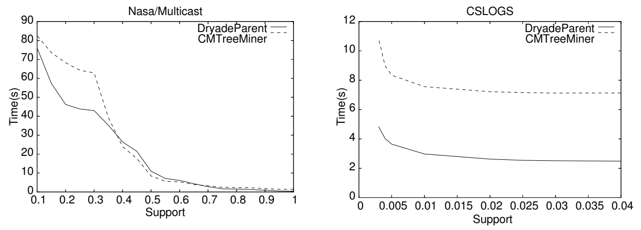
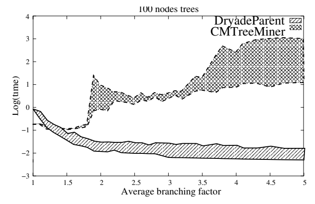


Figure 4: Running time w.r.t. support for the NASA and CSLOGS datasets.

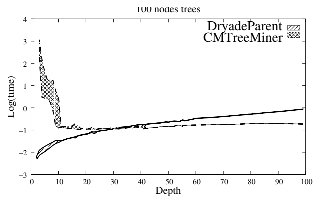
DRYADEPARENT is more than twice faster than CMTreeMiner on the CSLOGS dataset. For the NASA dataset the performances are similar, DRYADEPARENT having an advantage for the lowest support values. We discovered that the patterns found in both datasets were very different: NASA contains deep patterns with low branching factor, whereas CSLOGS contains shallow patterns with an higher branching factor. Artificial datasets will be used to get a deeper understanding of the influence of the structure of the patterns on compute-time performance of the two algorithms.

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 labeled 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 we generated 10,000 random trees and regrouped them 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. Figure 5a shows the logarithms of these averaged time values w.r.t. the average branching factor b, and Figure 5b shows the logarithms of these averaged time values w.r.t. the depth d.

The Figure 5a 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 Figure 5b. This figure shows that DRYADEPARENT exhibits a linear dependency on the depth of the patterns. This is



(a) Log(time)/average branching factor



(b) Log(time)/depth

Figure 5: Random trees with 100 nodes

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 Figure 6 a new curve, showing the computation time with respect to the *maximal* branching factor.

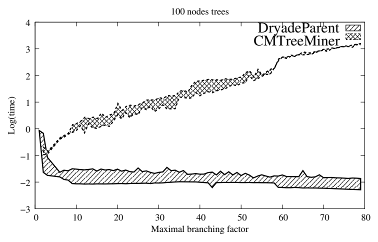


Figure 6: Random trees with 100 nodes, log(time) w.r.t. maximal branching factor

DRYADEPARENT is nearly unaffected by the maximal branching factor, but the computation time of CMTreeMiner depends strongly on this parameter.

5. Conclusion and perspectives

In this paper, we have presented the DRYADEPARENT algorithm, based on the computation of tiles 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 analyzes is one of our future research directions. We also plan to extend DRYADEPARENT to structures more general than attribute trees.

Acknowledgments: 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] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[2] H. Arimura and T. Uno. An output-polynomial time algorithm for mining frequent closed attribute trees. In *15th International Conference on Inductive Logic Programming (ILP'05)*, 2005.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA*, pages 158–174, Avril 2002.

[4] T. Asai, H. Arimura, T. Uno, and S. ichi Nakano. Discovering frequent substructures in large unordered trees. In *the Proc. of the 6th International Conference on Discovery Science (DS'03)*, pages 47–61, 2003.

[5] R. Chalmers and K. Almeroth. Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proceedings of the IEEE INFOCOM'2001*, April 2001.

[6] Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, 2004.

[7] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences, 2003*, 2003.

[8] A. Termier. Extraction of frequent trees in an heterogeneous corpus of semi-structured data: application to xml documents mining. Technical Report 1388, LRI, May 2004. http://www.lri.fr/∼termier/publis/phdTermierEN.ps.gz.

[9] A. Termier, M. Rousset, and M. Sebag. Dryade : a new approach for discovering closed frequent trees in heterogeneous tree databases. In *International Conference on Data Mining ICDM'04, Brighton, England*, pages 543–546, 2004.

[10] A. Termier, M. Rousset, M. Sebag, K. Ohara, T. Washio, and H. Motoda. Computation-time efficient and robust attribute tree mining with DRYADEPARENT. In *Third International Workshop on Mining Graphs, Trees and Sequences (MGTS)*, 2005.

[11] T. Uno, M. Kiyomi, and H. Arimura. Lcm v.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *2nd Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, 2004.

[12] M. J. Zaki. Efficiently mining frequent trees in a forest. In *In Proc. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2002.

[13] M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae, special issue on Advances in Mining Graphs, Trees and Sequences*, 65(1-2):33–52, March/April 2005.