HAL
open science

FuzBT: a Binary Approach for Fuzzy Tree Mining

Stephane Sanchez, Anne Laurent, Pascal Poncelet, Maguelonne Teisseire

To cite this version:

Stephane Sanchez, Anne Laurent, Pascal Poncelet, Maguelonne Teisseire. FuzBT: a Binary Approach for Fuzzy Tree Mining. IPMU: Information Processing and Management of Uncertainty in Knowledge-Based Systems, Jul 2006, Paris, France. lirmm-00135015

HAL Id: lirmm-00135015

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00135015v1

Submitted on 6 Mar 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FuzBT: a Binary Approach for Fuzzy Tree Mining

S. Sanchez
LIRMM - FRANCE
sanchez@lirmm.fr

A. Laurent
LIRMM - FRANCE
laurent@lirmm.fr

P. Poncelet
LGI2P - EMA - FRANCE
pascal.poncelet@ema.fr

M. Teisseire
LIRMM - FRANCE
teisseire@lirmm.fr

Abstract

The use of XML in data exchanges has extensively grown for the last few years. Furthermore, huge volumes of data are sent through the Internet. Today data mining tools are required in order to provide users with automatic tools to deal with the heterogeneity of the data in order to query it. Some approaches have been proposed in order to mine sub-structures (subtrees) from such XML databases. However, they are usually based on crisp methods. Here we propose to soften these approaches to extract more understandable and relevant knowledge.

Keywords : Fuzzy Data Mining, Semi-structured Data, XML, Schema Mining.

1 Introduction

As they play an increasing role in data exchanges, the volume of XML resources is extensively growing. In order to deal with both large amount and heterogeneity of data, users must be provided with automatic tools. These tools are particularly important when the user wants to query several heterogeneous databases without knowing their associated structures. Recent advances in distributed and heterogeneous databases provide the end-user with mediator schemas. These mediator schemas allow users to access *transparently* documents in digital form residing in one or more possibly independent repositories.

Usually mediator schemas are manually defined but it is now irrelevant to manage them in such a way. There is a need for tools focusing on the following problem : how to automatically define a mediator schema ? This problem has been recognized as one of the main problems in the Semantic Web framework, and one of the main difficult ones [2, 3, 6]. In such a context, schema mining approaches have been recently proposed to extract in an efficient way the commonly occurring schemas from a collection [14, 2]. Nevertheless, such approaches suffer from different drawbacks since they only explore crisp methods, which are too poor according to the semantic point of view compared to the user needs. We thus argue that fuzzy methods must be considered in this framework [4, 7].

In this paper, we show that methods must be softened in order to mine relevant knowledge, which will be understandable and usefull for the user. We propose a method based on a binary representation of XML data and we show the good behaviour of our method through experiments.

The rest of the paper is organized as follows : Section 2 proposes basic definitions and Section 3 goes deeper into presenting the problem. Section 4 introduces our proposition. We first introduce our data representation and then the fuzzy algorithm we defined. Section 5 presents performed experiments. Finally, in Section 6 we conclude the paper with future avenues.

2 Basic Concepts

Definition 1 *A rooted labelled tree* $T = (V, E)$ *is a direct, acyclic, connected graph with* $V = \{0, 1, \ldots, n\}$ *as the set of vertices (nodes) and* $E = \{(x, y) | x, y \in V\}$ *stands for the set of edges. We assume that there is a special vertex* $r \in V$ *designated as a root and for all* $x \in V$*, there is a unique path from* r *to* x*. Then if* $x, y \in V$ *and if there is a path from* x *to* y *then* x *is called an* ancestor *of* y *(*i.e. y *is a* descendant *of* x*). If the length of the path from two vertices* x, y *is reduced to one, then the ancestor relationship is considered as a* parent *relationship. For an internal node* $x \in V$*, we assume that its children* $x_1, x_2, \ldots, x_n$ *(*$n \geq 0$*) are ordered from left to right (*i.e. *there is a sibling relationship between children).*

Considering a tree T, the set of nodes from T is denoted by N_T, and the root is denoted by r_T.

Several kinds of tree inclusion can be defined [5], depending on the way ancestors and siblings are considered. In this paper, we consider that a tree is embedded in another one if the nodes and the links can be retrieved. However, we do not consider that the tree being included must be found *exactly* in the database tree since the ancestor-descendant relationship is soft (there must exist a path from the ancestor to the descendant but it is not necessary that this path contains only one vertice). Considering the example from fig. 1, the 4 sub-trees S_1, S_2, S_3, S_4 are embedded in T.

FIG. 1 – Example of tree inclusion

Definition 2 *A tree* S *is* embedded *into a tree* T *if there exists an injective and total function* $\phi : N_S \rightarrow N_T$ *such as :*

- ϕ *keeps the labels :* $L_S(n_x) = L_T(\phi(n_x))$ *;*
- ϕ *keeps the relations* ancestor-descendant *:* $(n_x, n_y) \overset{i}{\Longleftrightarrow} (\phi(n_x), \phi(n_y))$ *;*
- ϕ *keeps the* indirect *order relations :* $(n_x \preceq_S n_y) \overset{i}{\Longleftrightarrow} (\phi(n_x) \preceq_T \phi(n_y))$.

Given a database D, the *support* of a tree S is the proportion of trees from the database where S is embedded :

$$Support(S) = \frac{\#\ of\ trees\ where\ S\ is\ embedded}{\#\ of\ trees\ in\ D}$$

S is said to be frequent if $Support(S) \geq \sigma$ where σ is a user-defined minimal support threshold.

The tree mining problem consists in mining all frequent sub-trees from a tree database. For this purpose, levelwise approaches are mostly proposed. In these approaches, algorithms run as follows : in a first step, 1-node trees are built and for each of them, we check in how many trees from the database are they embedded. The ones appearing enough times in the database trees are said to be frequent. Those 1-node frequent trees are then combined together in order to build 2-node candidates which are thus checked over the database. From these 2-node frequent trees, 3-node trees are built and it is checked if they are (or not) frequent. This process goes on until no more frequent subtree is found. Note that the trees being built are called *candidates* before they are checked to be (or not) frequent.

3 Fuzzy Data Mining

As highlighted in [10], fuzzy data mining can help when mining frequent subtrees from a tree database. In this article, four ways to soften classical approaches are proposed :

- *ancestor-descendant degree* : in classical approaches, a node *is* or *is not* an ancestor of another one. In our approach, we propose to indicate by a degree between 0 and 1 to which extent a node is an ancestor of another one, meaning that if there are too many nodes between them, then this degree will decrease, as shown on Fig. 3.
- *sibling ordering degree* : in classical approaches, nodes *are* or *are not* searched in the initial order. In our approach, we pro-

pose to indicate by a degree the sibling disorder.

- partial inclusion : in classical approaches, all the nodes from the candidate must be in the tree. In our approach, we propose to soften this rule by considering the degree to which the nodes are embedded in the tree.
- *Node similarity* : in classical methods, a node label *is* or *is not* the same as another one. In our approach, we propose to soften this by indicating by a degree to which extent two nodes are similar (*e.g.* based on a taxonomy).

Let us now examine how the classical binary inclusion (is/is not included) can be transformed into a gradual inclusion when considering the ancestor-descendant relation. As shown on Fig. 2, some nodes may be positioned between an ancestor and a descendant.

Fig. 2 – Inter-Node Interval

In existing approaches, if some nodes are positioned between a node and another one, either these nodes are considered as being ancestor-descendant, or they are not. However, we argue that if the number of nodes between an ancestor and a descendant is too important, then the relation between these two nodes may not be considered. In order to convey the idea of *important number of nodes*, we thus consider a fuzzy membership function. Such a function is shown in Figure 3.

Fig. 3 – Ancestor-Descendant : A Fuzzy Definition

4 Our proposal

In this section, we first present the FuzBT structure we use to deal with the data. Second, we describe the associated mining algorithm.

4.1 Representing Data

In order to manage trees as efficiently as possible, each tree T is transformed into a binary representation denoted by T_B where each node cannot have more than two children [9]. For this purpose, we propose the following transformation : the first child of a node is put as the left-hand child while the other childs are put in the right-hand path, as illustrated in Fig. 4.

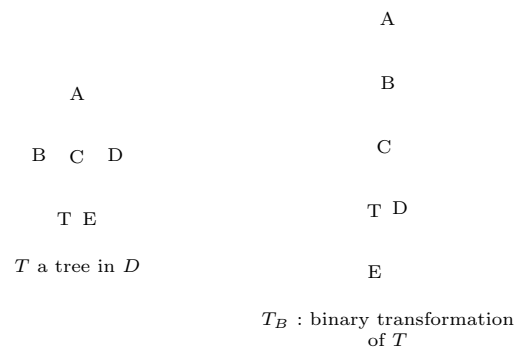

Fig. 4 – Example of a Binary Tree Transformation

Encoding Binary Trees

Once the tree has been transformed into a binary tree, nodes must be encoded in order to be retrieved. The encoding is then used first in order to identify each node and second in order to determine whether a node is a child or a brother. In order to do so, we consider the Huffman algorithm [8] which we slightly modify in order to fit our needs. The root has address 1. The other node addresses are computed by concatenating the father address with : 1 if it is a child (left-hand path) and 0 otherwise (right-hand path), as shown on Fig. 5.

Data Structure

The data structure being considered here is used in order to represent the trees and the

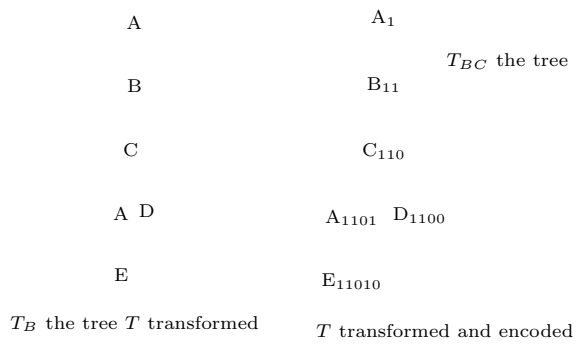

FIG. 5 – Node Addressing

candidates. In order to deal with huge amount of data, we aim at developping methods that are not memory-consuming. For this reason, considering a tree having $|2T|$ nodes, T must not be stored in more than $|2T|$ places.We consider here the tree representation presented in [11]. A tree is stored as two vectors [13]. The first vector, Adr, stores the unique address of each node in the tree. Note that the nodes are numbered in a depth-first numbering. The root T corresponds to the position 0 with $s[0] = -1$ since the root has no ancestor. The values $Adr[i], i = 0, 2, ..., k-1$ correspond to the positions considered for each node from the tree as proposed in [11].

Tab. 4.1 shows how the binary tree T_{BC} is managed in data structure ST.

Adr	1	11	110	1101	1100	11010
Lb	A	B	C	A	D	E

TAB. 1 – Data structure ST

This structure allows us to compute very quickly if a node n is within the scope[1] of another one. Considering the address of a node, the binary address of each child node is obtained in the following way : the first child is encoded by the concatenation of the father address and 1. All the brother nodes are encoded using the binary code of the father to which a 0 is concatenated.

As in [14], each node is associated with the interval of the positions from its descendants (the subtree having this node as a root). However, contrary to the Zaki's approach which keep this interval in memory, we do not need to keep it anymore. We rather retrieve it by using our data represention structure since, by using binary operations, its computation is very efficient. In order to decide whether a node is a descendant of another one, we consider the binary code of the potential father and the binary code of the second node. In a first step, the address of the second node is substracted from the first digits of the father address. Then we consider the following digit from the father address. If this digit is 0 then the second node is not a descendant. If this digit is 1 then the second node is a descendant. Note that if the digits of the second node are not the first digits of the potential ancestor, then these nodes cannot be related as an ancestor and a descendant.

As illustration, let us consider the potential ancestor node 1101. When considering the node 11011001, we retrieve the digits of the first node in the first digits of the second one. Then the next digit is a 1 so the second node is one of the descendant, as shown in Tab. 4.1.

A	1	1	0	1				
B	1	1	0	1	***1***	0	0	1
Indice	0	1	2	3	4	5	6	7

TAB. 2 – Evaluating two nodes uzing FuzBT

Moreover, our method is also very efficient to compute the number of nodes between an ancestor and one of its descendants. Indeed, this number of nodes is given by the number of 1 digits in the descendant node starting from the ancestor node. Let us consider the example from Tab. 4.1, there are two 1 digits in B starting from the end of the code of A (indice=4) which are found for indices 4 and 7, meaning that two nodes appear between A and B. Fig. 6 illustrates the ancestor-descendant relation and its binary representation.

[1]The scope of a node is constituted by the nodes for which n is an ancestor of another one.

Fig. 6 – Ancestor-Descendant Relation

4.2 Mining Subtrees

Like traditional approaches, we also adopt a levelwise algorithm [1]. As our generation phase is performed in a classical way, i.e. subtree candidates are built from existing subtrees, we rather focus on the process of checking whether a candidate is embedded or not within a tree from the database.

Let T be a tree from the database DB, C be a candidate to be tested and G be the solution graph. The algorithm is given in Algorithm 1. In a first step, all the possible nodes which can be a starting node to anchor the candidate C in the tree T are generated. In fact, these nodes correspond to the nodes from T which label is the label of the root of C. These nodes are put in the graph solution as the head of each solution being considered.

Algorithm 1: FuzBT, Anchoring

```
Algorithm: Fuzzy Anchoring
Data: T,C
Result: G
foreach node ∈ T do
    if node.lb == C.root.lb then
        G.add(node);
return G;
```

The second step consists in finding all the solutions for each of these anchoring nodes, as described in Algorithm 2.

Algorithm 2: FuzBT, Fuzzy Path

```
Algorithm: Fuzzy Path
Data: T,C,G,NodeC
Result: G
FatherID ← NodeC.ID;
NodeId ← NodeC.ID;
foreach ChildC ∈ NodeC do
    foreach Solution ∈ G do
        Continue ← true;
        if Solution is open then
            TempSol ← Copy(Solution);
            //Looking for the next position;
            Pos ← NextPos(T, ChildC.label,
            Solution[NodeID], Solution[FatherID], False);
            while Pos != Null do
                if Continue then
                    Solution.add(T[Pos]);
                    Continue ← False;
                else
                    G.add(TempSol);
                    G.back.add(T[Pos]);
                //Looking for the next position;
                Pos ← NextPos(T, ChildC.label, Pos, Solution[FatherID], True);
            if Continue then
                Solution.Close;
        PruneAllSolutionClosed(G);
        NodeID++;
        Poursuite(T,C,G,ChildC);
return G;
```

In this algorithm, the NextPos function aims at finding the next position from the tree T that has the same label as the node being considered in C.

Note that we are looking for the *best* way to embed C in T which leads to consider each possibility. This is the main difference between our approach and the propositions from the literature, since we aim at computing to which extent C is embedded in T.

At the end of this process, the solution graph contains all the ways a subtree is included wi-

thin a tree from the database. For each of these inclusion ways, a degree ranging from 0 to 1 has been computed by calculating the mean value of all the membership degrees of the ancestor-descendant degrees. The degree to which a subtree is included within a tree from the database is then computed as the maximal solution degree.

In order to compute the support of a candidate, we consider a thresholded Σ-count by suming all the maximal solution degrees if they are greater than a user-defined threshold.

5 Experiments

Experiments have been performed on synthetical data from the Termier data generator [12]. These experiments aim at highlighting the scalability of our approach.

The problem being considered is indeed hard. Although most of the approaches propose to simplify the problem by considering less complex inclusion definitions [14], we here consider a complete definition of inclusion. Moreover, we validate our candidates more precisely by considering the degree to which they are embedded.

We compare here our method to the vTreeminer one in terms of runtime. Fig. 7 (resp. Fig. 8) displays the results for a database containing $10,000$ (resp. $20,000$) trees in function of the minimum support being considered (ranging from 1% to 80%).

Fig. 7 – Runtime over $10,000$ trees

Fig. 9 (resp. Fig. 10) reports the memory consumed for a $10,000$ (resp. $20,000$) tree database, highlighting how low-memory-consuming is our approach.

Fig. 8 – Runtime over $20,000$ trees

Fig. 9 – Memory over $10,000$ trees

Fig. 11 (resp. Fig. 12) reports the number of frequent subtrees discovered compared to the number of solutions having been processed in function of the minimal support for a minimal ancestor-descendant distance equal to 1 and a maximal distance equal to 12 (resp. minimal ancestor-descendant distance = 3, maximal ancestor-descendant distance = 3).

Fig. 10 – Memory over 20,000 trees

Fig. 11 – Number of Frequent Subtrees with a 1 to 12 ancestor-descendant distance

Fig. 12 – Number of Frequent Subtrees with a 1 to 3 ancestor-descendant distance

6 Conclusion and perspectives

In the Semantic Web framework, many works deal with the querying of heteregoneous databases. In order to do so, the user (or the system) must be provided with an idea of a common schema of the data. For several years, this common schema has been manually defined. However, it is now impossible to manage this manually, it is thus necessay to consider algorithms for automatically extract these common structures. However, proposed approaches [14] do not allow to soften the rules making that a tree is embedded into another one and are thus less appropriate for Web Semantic. Here, we propose to consider gradual inclusion of a tree in another one by considering a soft ancestor-descendant relation.

Many perspectives are associated with this work. First, we would like to experiment our approach on real datasets instead of generated ones. Second we plan to optimize our algorithm by automatically stop the database scans as soon as possible. Finally, we plan to consider all the other ways fuzzy logic can be applied in the framework of data integration and schema matching.

Références

[1] R. Agrawal and R. Srikant. Mining sequential pattern : Generalizations and performance improvements. Technical report, IBM, IBM reserch division San Jose, 1999.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, and H. Sakamoto. *Efficient Substructure Discovery from Large Semi-structure Data.* In *2nd Annual SIAM Symposium on Data Mining, SDM2002*, Arlington, VA, USA, 2002. Springer-Verlag.

[3] V. Carchiolo, A. Longheu, and M. Malgeri. Hidden schema extraction in web documents. In *Databases in Networked Information Systems, Third International Workshop, DNIS 2003*, volume 2822 of *Lecture Notes in Computer Science*, page 4252, 2003.

[4] P. Ceravolo, M. Viviani, and M. C. Nocerino. Knowledge extraction from semi-structured data based on fuzzy techniques. *International Journal of Knowledge-Based Intelligent Engineering Systems*, 2004.

[5] Y. Chi, R. R.Muntz, S. Nijssen, and J. N. Kok. Frequent subtree mining - an overview. *Fundamenta Informaticae XXI*, pages 1001–1038, 2005.

[6] S. Decker, editor. *Proc. of the first International Workshop on Semantic Web and Databases (SWDB), co-located with VLDB'2003*, 2003.

[7] B. O. E. Damiani, N. Lavarini and L. Tanca. An approximate querying environment for xml data. In L. A. Zadeh, V. Loia, and M. Nikravesh, editors, *Fuzzy Logic and the Internet*, volume 137 of *Series on Studies in Fuzziness and Soft Computing*. Springer-Verlag, 2004.

[8] D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, 1952.

[9] D. Knuth. *The Art of Computer Programming, Volume 1 : Fundamental Algorithms*. Addison-Wesley, 1973.

[10] A. Laurent, M. Teisseire, and P. Poncelet. *Fuzzy Data Mining for the Semantic Web : Building XML Mediator Schemas*, chapter 12. Elsevier, E. Sanchez(ed.), To appear, 2006.

[11] F. D. R. Lopez, A. Laurent, P. Poncelet, and M. Teisseire. Rsf - a new tree mining approach with an efficient data structure. In *Proceedings of the joint Conference : 4th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2005)*, pages 1088–1093, 2005.

[12] A. Termier, M.-C. Rousset, and M. Sebag. TreeFinder, a first step towards xml data mining. In *Proceedings of the IEEE Conference on Data Mining (ICDM 02)*, pages 450–457, 2002.

[13] M. A. Weiss. *Data Structures And Algorithm Analysis In C*. Addison Wesley, 1998.

[14] M. J. Zaki. *Efficiently Mining Frequent Trees in a Forest*. In *In KDD'02*, Edmonton, Alberta, Canada, 2002. ACM.