Data Structures for Efficient Tree Mining with Soft Embedding Constraints

Federico Del Razo Lopez*, **Stéphane Sanchez***,
Anne Laurent*, **Pascal Poncelet**** and **Maguelonne Teisseire***
* LIRMM - CNRS - 161 rue Ada - 34392 Montpellier - FRANCE
{delrazo,sanchez,laurent,teisseire}@lirmm.fr
** EMA - Site EERIE - 69 rue G. Besse - 30035 Nîmes - FRANCE
pascal.poncelet@ema.fr

Abstract

XML is playing an increasing role in data exchanges and the volume of available resources is thus growing dramatically. As they are heterogeneous, these resources must be translated into a *mediator* schema to be queried. For this purpose, automatic tools are required. These tools must allow the extraction of common data structures from the tree-like XML data. In this paper, we present a novel approach based on a low memory-consuming representation which can be improved by considering a binary representation. We show that these representations have many properties to enhance subtree mining algorithms, especially when considering soft tree embedding constraints. Experiments highlight the interest of our proposition.

Keywords: Data Mining, XML, Frequent Sub-Tree Mining, Fuzzy Inclusion.

1 Introduction

The volume of data available from the Internet is growing dramatically. Although it provides rich information, it raises many problems when querying these huge volumes of data in order to retrieve relevant information. Since users can indeed hardly be aware of the way the data are organized, it is necessary to provide them with mediator schemas that are built automatically. In this framework, a mediator schema is meant as an interface between the user and the data. This interface provides the user with a schema to query heterogeneous and distributed data in a very simple way, as shown on Figure 1.

As highlighted by the World Wide Web Consortium, XML has been proposed to deal with huge volumes of electronic documents and is playing an increasing

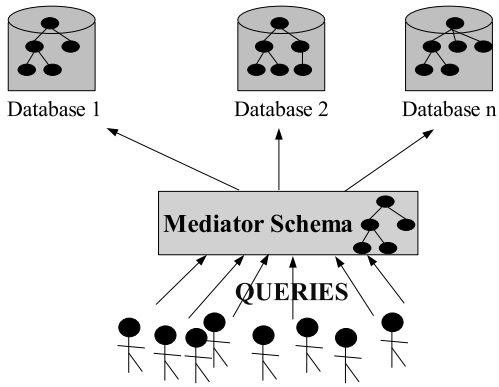

Figure 1: Querying Distributed Databases through a Mediator Schema

important role in the exchange of a wide variety of data on the Web. It is thus necessary to be provided with tools to integrate XML schemas in order to query XML data.

Even if recent works have been proposed to access data when a query schema is known [15], it is still a challenge to mine automatically such mediator schemas. Existing approaches from the literature are incomplete [12]. In this paper, we focus on frequent pattern mining from an XML database. In this framework, a frequent pattern is a subtree that occurs in *most* XML schemas from the database. This proportion is examined regarding a *support*, which corresponds to a minimal number of trees from the database that must contain the subtree in order to consider this subtree as being *frequent.* The mining process work as a repetitive two-step one: first *candidate subtrees* are built. Then these candidates are tested over the whole database in order to compute if they are frequent or not. If so, two frequent subtrees are combined to build a new candidate which will be tested, and so on up to the point where no more frequent subtrees can be mined. This mining process is complex since all the XML schemas from the database must be represented using a representation structure that can support data mining algorithms. In many approaches, this transformation leads to memory-consuming representations requiring twice or three times the number of nodes. As far as we know, there is no work in the literature that provides both a compact representation and useful properties for data mining algorithms. In this paper, we thus propose a new structure being a compact representation as well as a relevant support for data mining, as some properties can be used when generating candidates and pruning non frequent candidates. This approach is compared to existing works [2, 11, 19] and we show its interest through experiments.

Moreover, we address the problematic of soft embedding constraints. This kind of constraints are indeed important to mine subtrees that are *globally* embedded whithout forgetting a tree for which only a very little part does not fit.

Among the possible ways to consider fuzzy tree mining [9], we focus here on the ancestor-descendant relation, meaning that we consider that when looking for a relation between the node *author* and the node *name*, we may have some other nodes between (e.g. *civil status*). For this purpose, we show that representing trees using a binary representation is very efficient.

This paper is organized as follows. Section 2 presents related work. Section 3 presents the core of our proposition defining a new method to represent tree data and the associated algorithms for frequent subtree mining. Section 4 presents the problematic of fuzzy tree inclusion when considering the ancestor-descendant relation. Section 5 introduces a binary representation in order to efficiently manage fuzzy inclusion. Finally, Section 6 concludes and presents some further work associated to our proposition.

2 Related Work

Mining frequent subtree has recently received a great deal of attention [2, 8, 11, 16, 19]. In the following, we first introduce preliminary definitions. Second we focus on principal approaches.

2.1 Preliminary Definitions

A *tree* is a connected graph containing no cycle. A tree is composed by nodes, which are linked by edges such that their exists a particular node called *root* and such that all the nodes but the root are composed by sub-trees. A tree is said to be an *ordered tree* if the children from a node are ordered. A tree is said to be an *unordered tree* otherwise.

Let $\mathcal{L} = \{a, b, c, ...\}$ be a set of labels. A *labeled ordered tree* is a tree $T = \{r, N, B, L, \prec_F\}$ where: r is the root, N is the set of nodes, B is the set of edges such that $B \subseteq V^2$, $(L : N \rightarrow \mathcal{L})$ is a mapping from the set of labels $\mathcal{L}$ to the set of nodes N, and $\prec_F$ is an ordered relation between brother nodes.

Several nodes can have the same label, which raises the polysemy problem, since a label can be associated to several nodes. The *size* of T is the number of nodes in T: $|T|$. The *order* of T is the number of edges in T.

Each node n is associated with a unique number i, corresponding to its position in a breadth-first traversal of the tree. n_i denotes the ith node in such a traversal ($i = 0, \ldots, |T| - 1$).

For each pair $(x, y) \in N \times N$, it is said that there is a *direct* relation between x and y if there exists an edge from x to y. x is said to be the *predecessor* and y is said to be the *successor*. It is said that there is a *indirect* relation between x and y if there exists a path between x and y, meaning that there exists a non-empty set of nodes $\nu_1, \ldots, \nu_k$ such that there exists edges from x to ν_1, from ν_1 to ν_2, ... and from ν_k to y. In this case, x is said to be the *ancestor* and y is said to be the *descendant*.

Figure 2 shows a direct relation in T between the nodes 0 and 3, and an indirect relation between the nodes 3 and 6.

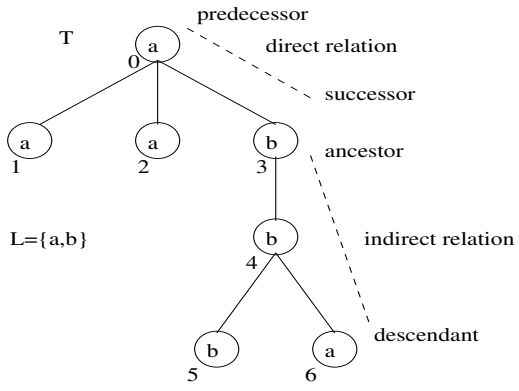

Figure 2: A Tree

$S \preceq T$ denotes that S is a subtree of T (*cf* Figure 3). Considering an indirect link *ancestor-descendant*, $S \preceq T$ holds under the following conditions:

1. $N_S \subseteq N_T$
2. for each edge $b_S = (x, y) \in B_S$, x is an ancestor of y in T
3. for each edge $b_T = (x, y) \in B_T$, x is an ancestor of y in S
4. for each $b^1_S = (x, y_1) \in B_S$ and $b^2{}_S = (x, y_2) \in B_S$, $y_1 \prec_F y_2 \Rightarrow y_1 \prec_F y_2$ in T

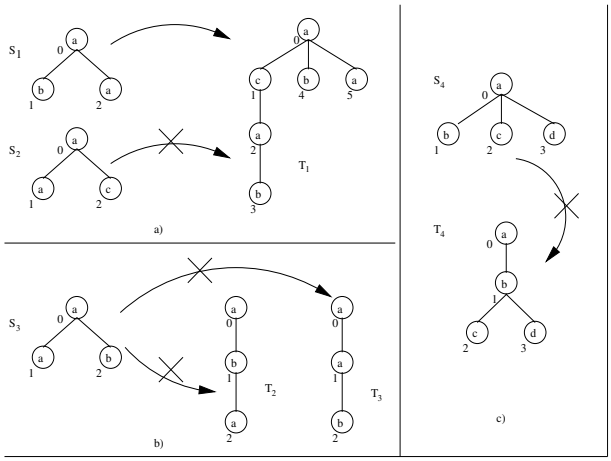

Figure 3: Inclusion and non inclusion of S in T

Note that a tree may be included in another one several times, as shown on Figure 4 when considering indirect ancestor-descandant links.

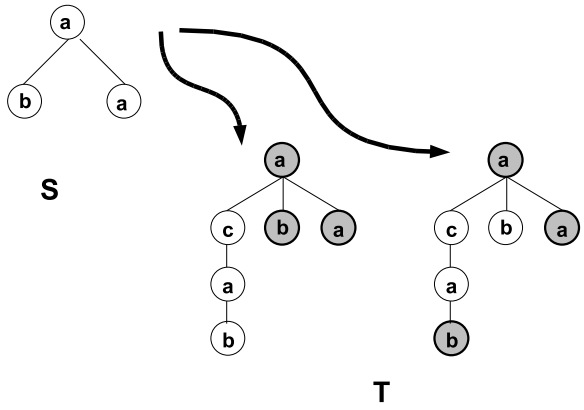

Figure 4: Several Ways to Include S in T. Indirect Links.

2.2 Mining Frequent Sub-trees

The approaches proposed in the literature for mining frequent subtrees are mostly based on the data representation they use in order to speed up the data mining algorithms.

The $TreeMiner$ algorithm [19] proposes a method for frequent subtree discovery. As we propose here, this work is based on an original representation of the trees, which facilitates the management of the candidates. This approach needs a storage space of three times the size of each tree ($3|T|$), as shown by Figure 5.

The $FreqT$ approach proposed in [2] is devoted to ordered trees. The data structure that has been chosen leads to interesting results regarding runtime. However, this proposal is as memory-consuming as [19], *i.e.* $3|T|$, as shown by Figure 6.

Although some other data structures have been proposed recently ($Chopper$ [13], $FreeTreeMiner$ [5] and $CMTreeMiner$ [4]), they do not offer useful properties for the management of the candidates that could be as interesting as $TreeMiner$ [19] and $FreqT$ [2]. These representations are shown on Figures 7 and 8.

Our aim is thus to have both a data structure that is not memory-consuming and that has some good properties regarding data mining.

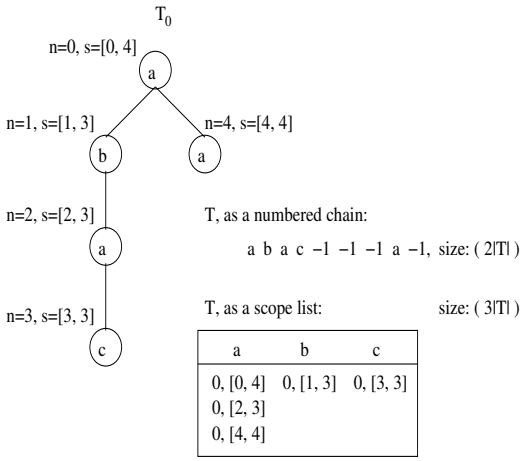

Figure 5: Data Structure proposed in [19]. $TreeMiner$

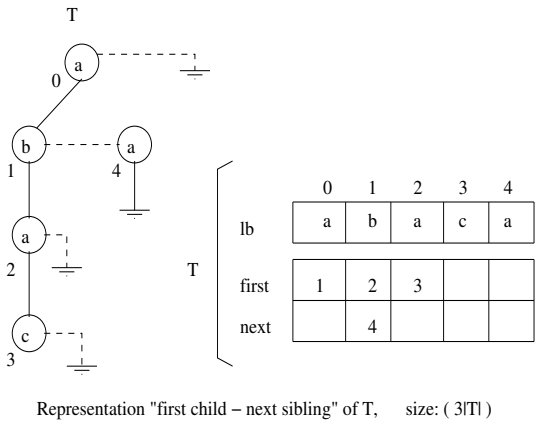

Figure 6: Data Structure proposed in [2]. $FreqT$

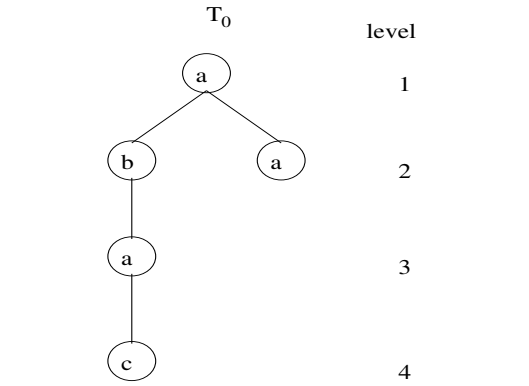

Figure 7: Data Structure proposed in [13]. *Chopper*

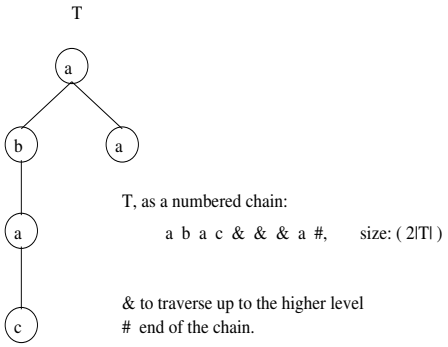

Figure 8: Data Structure proposed in [4, 5]. *FreeTreeMiner* and *CMTreeMiner*.

3 RSF: A Novel Efficient Representation of Trees

In this paper, we propose an original representation of the trees which allows the efficient generation and pruning of candidate subtrees.

3.1 RSF: Representating a Tree in Two Vectors

When representing a tree T, we keep in mind the following property: all the nodes but the root have one and only one predecessor. We propose thus to use two vectors to represent a tree, as proposed in [14]. The first vector is denoted by st. It stores the position of each node predecessor. Nodes are numbered considering a depth-first traversal. The root is numbered as being at position 0, with $st[0] = -1$ since it has no predecessor. The values $st[i], i = 1, 2, ..., k-1$ correspond to all other predecessor positions, as shown on Figure 9.

This representation provides a constant-time method to retrieve the predecessor of a node. Moreover, it allows us to find directly the most right leaf when considering an index k since it is the node being stored in the last position of the vector (Fig. 9). Finally, when visiting the tree, it is possible to build all direct links from predecessors to descendants.

The second vector is denoted by lb. It is used to store all the tree labels. $lb[i], i = 0, 1, ..., k-1$ are the labels of each node $n_i \in T$.

The data structure we have chosen needs very low memory since it is reduced to the size of $2|T|$. Moreover, it has good properties when mining frequent subtrees (see section 3.2).

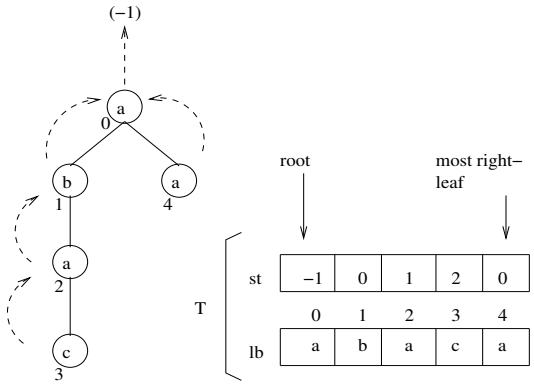

Figure 9: Representation of a Tree

3.2 Generating and Pruning Candidates

Candidates of size 1 (single nodes) are obtained by visiting all the nodes from the trees of the database. Each node is mapped to a support which is incremented during the traversal. Only the nodes having a support value greater than the minimum user-defined threshold are kept. The database is then transformed in order to delete all non frequent nodes, as shown on Figure 10.

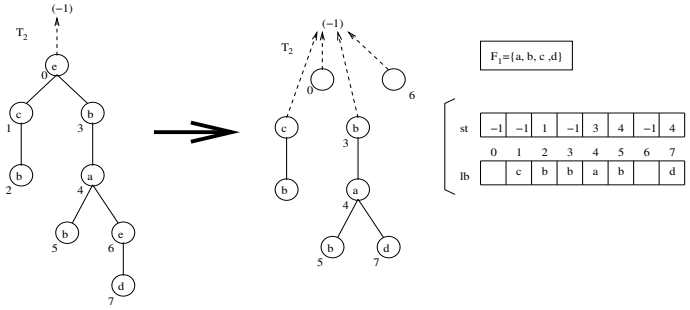

Figure 10: Transformation of the database after generation of F_1

Candidates of size 2 are generated by combining all the pairs of candidates of size 1.

The database is then updated by modifying the trees from the root, the nodes, and the leaves so that only frequent links are kept.

Figures 11 and 12 illustrate this process, with $\sigma = 7$ and $F_2 = \{a-d, a-b\}$.

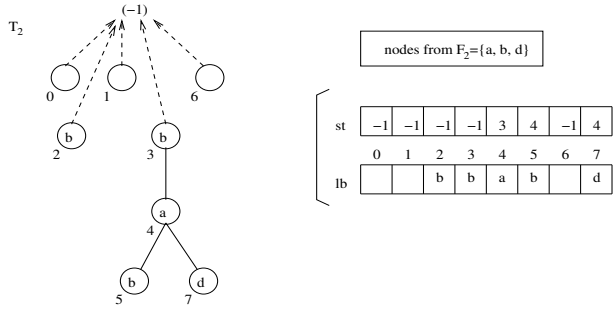

Figure 11: Transformation of the database - root

The generation of candidates of size $k \geq 3$ is performed by a levelwise APriori-like algorithm [1], by combining frequent subtrees of size $k-1$.

The originality of our approach lies in the use of our representation for candidate pruning when deleting non frequent candidates. Computing the support of each candidate is performed by counting the number of trees which contain the candidate being considered. For this purpose, for each tree of the database,

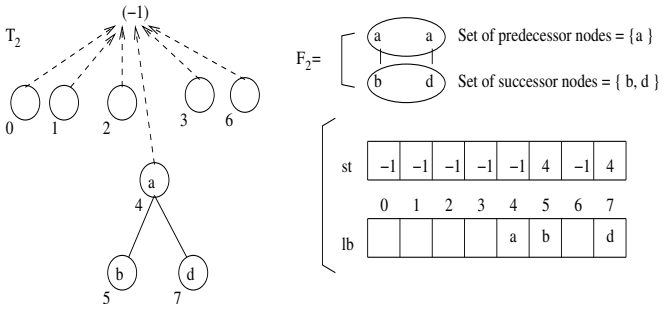

Figure 12: Transformation of the database - nodes

we aim at finding an *anchor node* on which the root of the subtree to be tested can be deployed. For each anchor point that can be found out, our method checks whether it is possible or not to deploy all the nodes. Please note that a *perfect* deployment is looked for. If all the nodes of the candidate tree can be found in a part of the tree from the database being considered, then this tree is counted and the support is incremented.

3.3 Experiments

Experiments are led on synthetic data obtained by the XML data generator developped by A. Termier [11]. Results, shown on Figures 13, 14 and 15, highlight the interest of our approach when considering scalability (runtime and memory) over the number of trees in the database. These results compare our approach (RSF) to *FreqT* (implementation provided by the authors) [2].

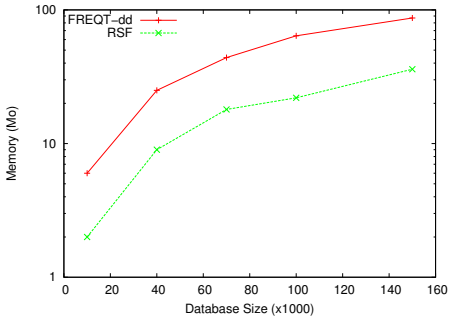

Figure 13: Memory over number of trees

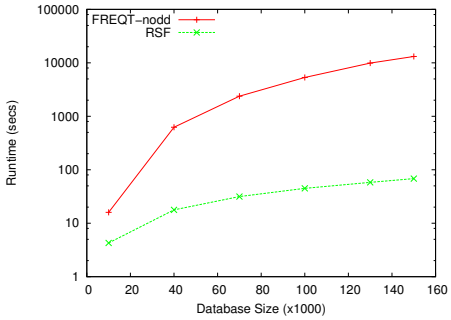

Figure 14: Runtime over database size, FreqT-nodd and RSF. $min\_sup = 0.05$

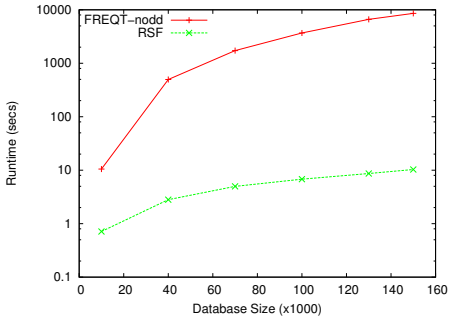

Figure 15: Runtime over database size, FreqT-nodd and RSF. $min\_sup = 0.5$

4 Soft Embedding Constraints

4.1 Problematic

As highlighted in [9], fuzzy data mining can help when mining frequent subtrees from a tree database. In this section, we show 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 Figure 16, some nodes may be positioned between an ancestor and a descendant.

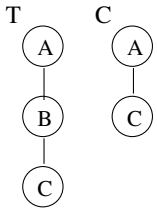

Figure 16: Inter-Node Interval

In existing approaches, if some nodes are positioned between a node and another one, either these nodes are considered as being ancestor-descendant [19], or they are not [3]. 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 17.

When taken into account, embedded trees are counted within the final support whatever the ancestor-descendant relationship may be. There is no consideration of the number of nodes separating the ancestor node from the descendant one. In our approach, we propose thus to give a scope to this ancestor-descendant relationship. This scope is defined considering the number of nodes between ancestor and descendant nodes. Since it does not make sense to consider crisp boundaries, we propose to consider a fuzzy scope for the ancestor-descendant relationship.

4.2 Handling Soft Constraints

We consider fuzzy membership functions describing the ancestor-descendant relationship depending on the number of nodes separating the two nodes being considered, as shown on Figure 17 when considering *no more than five nodes* in a fuzzy way. For this purpose, we use fuzzy quantifiers [17, 18].

When considering tree inclusion, a tree may be included in another one in different manners, as shown on Figure 4. When mining crisp frequent subtrees, the aim is to discover subtrees that are found frequently within the trees from the database, whatever the number of possibilities. When considering soft embedding inclusion, each possibility is associated to a degree between 0 and 1

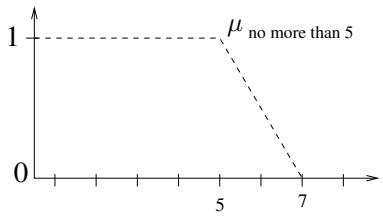

Figure 17: Ancestor-Descendant: A Fuzzy Definition

indicating to which extent the tree being considered is embedded (for instance this degree may indicate to which extent the constraint *no more than 5 nodes* holds).

This raises several questions:

- how to compute efficiently the indirection between two nodes (number of intermediate nodes) as a degree between 0 and 1,
- how to merge the degrees for each indirection from the tree as a degree between 0 and 1,
- how to efficiently compute the *best way* to include a tree in another one.

There may be several ways to include a tree in another one, as it has been shown by Figure 4. In order to compute the *best way* to include a tree within another one, It is necessary to consider all the possibilities while remaining scalable.

In our approach, we consider the following method to compute the fuzzy support of a tree S within a tree database DB:

- For each possibility to include the tree S within a tree $T \in DB$, this solution is associated with a degree ranging from 0 to 1 which is computed by considering the average of all membership degrees of the fuzzy inter-nodes distance being considered.
- For each tree $T \in DB$, we compute the inclusion degree as the *maximum* of all the degrees described above.
- All the degrees are merged in order to compute the *support* of S using a thresholded sigma-count.

Figure 18 sums up this process. In this example, we consider that *no more than 2 nodes* must be between an ancestor and a descendant. For each possible inclusion of the candidate subtree within a tree from the database, each edge is labeled with a degree ranging from 0 to 1. This degree indicates to which extent the indirection holds, depending on the membership degree computed regarding the number of intermediate nodes. In this example, we know that when there is no intermediate node, then the membership degree to the fuzzy set *no more*

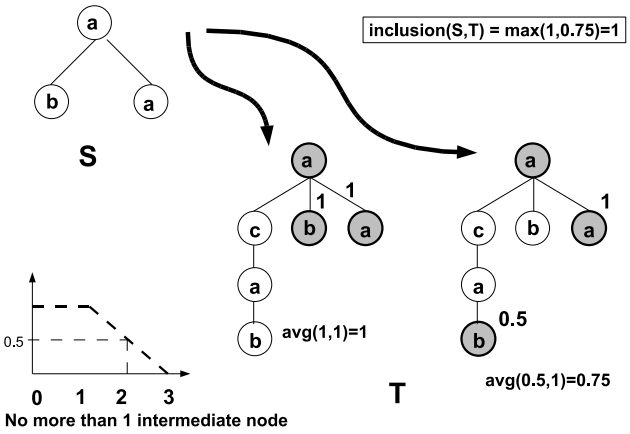

Figure 18: Computing the Fuzzy Inclusion Degree.

than 2 nodes is 1 while it is 0.5 when there are 2 intermediate nodes. For each possible inclusion, these degrees are merged using the average, leading to 1 and 0.75 in this example. Finally, the best inclusion is considered by computing the maximum. In this example, it turns up that the candidate tree is fully included in the database tree (support=1).

Note that these are choices based on the idea that we must compute the best way to consider the inclusion (which leads to the use of the *maximum* to merge different inclusion solutions) and that an inclusion may not be disturbed by extrema (which leads to discard *maximum* and *minimum* within a solution). However, our algorithms and implementations are very general and may be used with other operators.

As shown previously, RSF is an efficient data structure when dealing with induced tree mining (looking for direct relations between nodes). However, RSF is not well adapted to soft constraints since it is necessary to traverse a lot of nodes from a tree when computing the number of intermediate nodes between an ancestor and a descendant. We thus propose a new data structure based on a binary representation of a tree.

5 FuzBT: An Efficient Binary Representation for Soft Embedding Constraints

5.1 Binary Representation

In order to manage trees as efficiently as possible (as described in next sections), each tree T is transformed into a binary representation denoted by T_B where each node cannot have more than two children [7]. For this purpose, we propose

the following transformation: the first child of a node is put as the left-hand child while the other children are put in the right-hand path, as illustrated in Fig. 19.

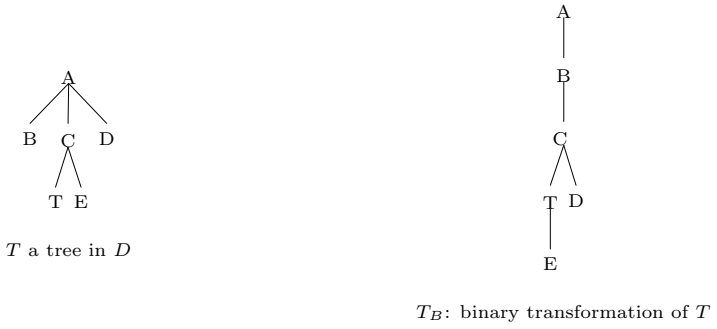

Figure 19: Example of a Binary Tree Transformation

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 [6] 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. 20.

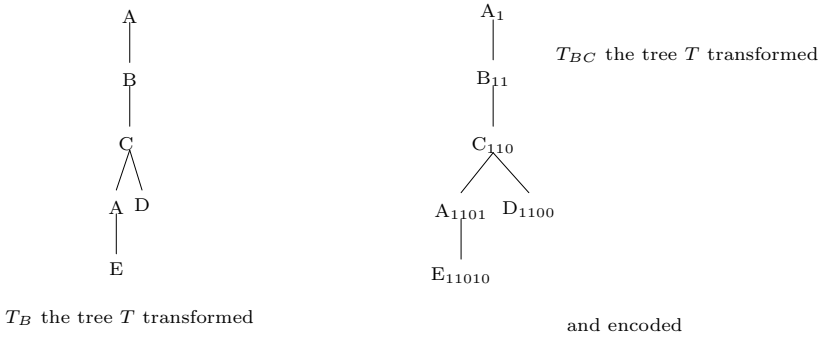

Figure 20: Node Addressing

Data Structure

The data structure being considered here is used in order to represent the trees and the candidates. In order to deal with huge amounts of data, we aim at developping methods that are not memory-consuming. For this reason, considering a tree having $2|T|$ nodes, T must not be stored in more than $2|T|$ places, as proposed in [10].

Tab. 5.1 shows how the binary tree T_{BC} is managed in data structure ST.

st	1	11	110	1101	1100	11010
lb	A	B	C	A	D	E

Table 1: Data structure ST

5.2 Managing Soft Embedding Constraints with the Binary Representation

In this section, we show how our data representation can manage soft embedding constraints, especially when considering the binary tree representation.

As highlighted previously, the structure we propose here stores data in a 2 $|T|$ data structure, following a depth-first traversal of the tree. 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 [19], each node is associated with the interval of the positions from its descendants (the subtree having this node as a root). However, contrary to Zaki who keeps this interval in memory, we do not need to represent it [19]. We rather retrieve it by using our data represention structure since this computation is very efficient by using binary operations. 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.

For instance, 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 table 2.

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

Table 2: Evaluating two nodes uzing FuzBT.

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

Moreover, we show that our method is 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 Table 2, 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.

Figure 21 illustrates the ancestor-descendant relation and its binary representation.

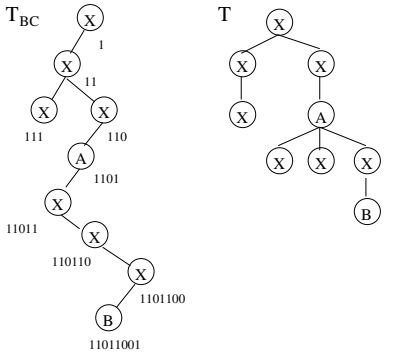

Figure 21: Ancestor-Descendant Relation

When mining frequent subtrees, the *best* way to embed a candidate tree in tree from the database is computed. This leads to consider each possibility. This is the main difference between our approach and the propositions from the literature.

For this purpose, a graph is built, containing all the ways a subtree is included within 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.

5.3 Experiments

We compare here the two data structures we have proposed when dealing with soft constraints. We show that the binary representation leads to better results. Figures 22 and 23 report the memory and runtime depending on the size of the database while Figures 24 and 25 report the memory and runtime depending on the support, showing that FuzBT behaves better than RSF.

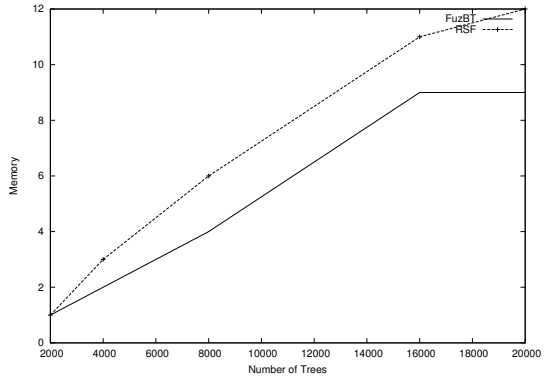

Figure 22: Memory over Number of Trees. Support = 0.2

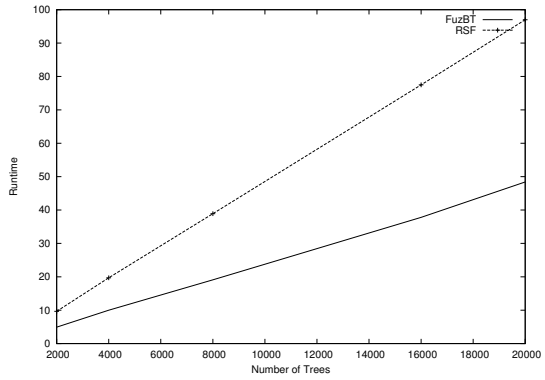

Figure 23: Runtime over Number of Trees. Support = 0.2

6 Conclusion and Perspective

When addressing the problem of frequent subtree mining, it is interesting to consider soft embedding constraints in order to define gradual inclusion of a tree within another one. In this work, we propose data structures to represent trees in an efficient manner to ease fuzzy subtree mining. We focus on the

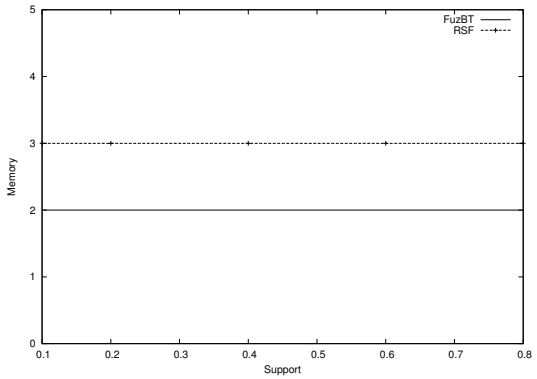

Figure 24: Memory over Support. 20,000 Trees.

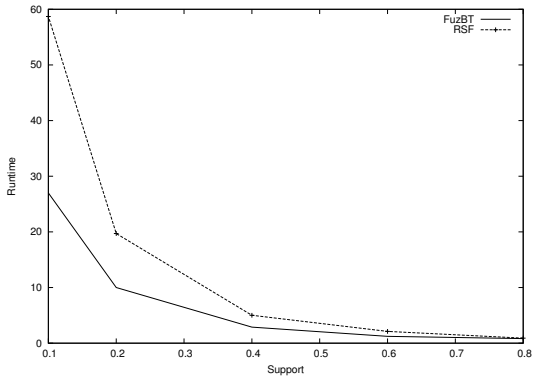

Figure 25: Runtime over Support. 20,000 Trees.

problem of the ancestor-descendant relation by considering a degree between 0 and 1 to indicate to which extent a node is an ancestor of another one.

Our proposition can be applied in many domains, especially for data mediation. Frequent subtrees that are mined by our method can indeed help building a mediator schema. Such a solution can also be taken in the framework of online data mining for data streams. This perspective is very promising since it provides efficient and fast methods to deal with high volumes of XML data on the Internet.

We also aim at building candidates more efficiently in order to enhance our proposition.

Finaly, we aim at using the data structures we introduce here to manage all the ways to handle fuzziness proposed in [9].

References

[1] R. Agrawal et R. Srikant, *Fast algorithms for mining association rules in large databases*, *in* Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura et 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] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto et S. Arikawa, *Efficient substructure discovery from large semistructured data*, *in* SIAM Int. Conf. on Data Mining, 2002.

[4] Y. Chi, Y. Yang et 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.

[5] Y. Chi, Y. Yang et R. R. Muntz, *Indexing and mining free trees.*, *in* International Conference on Data Mining 2003 (ICDM2003), 2003.

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

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

[8] M. Kuramochi et G. Karypis, *Frequent subgraph discovery*, *in* IEEE International Conference on Data Mining (ICDM), 2001.

[9] A. Laurent, P. Poncelet et M. Teisseire, *Fuzzy data mining for the semantic web: Building XML mediator schemas*, *in* Fuzzy Logic and the Semantic Web, E. Sanchez, éd., Capturing Intelligence, ELSEVIER, to appear, 2006.

[10] F. D. R. Lopez, A. Laurent, P. Poncelet et 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), 2005, p. 1088–1093.

[11] A. Termier, M.-C. Rousset et M. Sebag, *Treefinder, a first step towards XML data mining*, *in* IEEE Conference on Data Mining (ICDM), 2002, p. 450–457.

[12] J. Tranier, R. Baraer, Z. Bellahsene et M. Teisseire, *Where's Charlie: Family based heuristics for peer-to-peer schema integration*, *in* Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04), Coimbra, Portugal, July, 7th - 9th 2004.

[13] C. Wang, Q. Yuan, H. Zhou, W. Wang et B. Shi, *Chopper: An efficient algorithm for tree mining*, Journal of Computer Science and Technology, 19 (May 2004), p. 309–319.

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

[15] L. Xyleme, *A dynamic warehouse for xml data of the web*, *in* IEEE Data Engineering Bulletin, 2001.

[16] X. Yan et J. Han, *gspan: Graph-based substructure pattern mining*, *in* IEEE Conference on Data Mining (ICDM), 2002.

[17] M. Ying et B. Bouchon, *Quantifiers, modifiers and qualifiers in fuzzy logic*, Journal of Applied Non-classical Logics, 7 (1997), p. 335–342.

[18] L. A. Zadeh, *A computational approach to fuzzy quantifiers in natural languages*, Computing and Mathematics with Applications, 9 (1983), p. 149–184.

[19] M. Zaki, *Efficiently mining frequent trees in a forest*, *in* ACM-SIGKDD'02, 2002.