Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2007)

Clustering Rooted Ordered Trees

Mostafa Haghir Chehreghani[1], Masoud Rahgozar[2], Caro Lucas[3] and Morteza Haghir Chehreghani[4]

***Abstract*— Recently, tree structures have gained popularity for storing data from different domains such as XML documents, bioinformatics and so on. Clustering these data can facilitate different operations. In this paper, we propose *TreeCluster*, a novel and heuristic algorithm for clustering tree structured data. This algorithm considers a representative tree for each cluster. For each input tree *T*, *TreeCluster* computes the composition of the tree *T* and each of the clusters. Tree *T* belongs to the cluster which its composed tree gains the best score. After adding a tree to a cluster the representative tree of that cluster is updated. We evaluate the accuracy of the *TreeCluster* algorithm in comparison to the previous works.**

I. Introduction

Recently, tree structures have gained popularity as a means for storing and manipulating data from different domains. The reason is that the flexible structure of trees allows the modeling of a wide variety of databases such as XML documents, bioinformatics, etc in an efficient and compact way. With the continuous growth in the amount of these data, discovering knowledge from them becomes increasingly important. One of the important tasks in mining tree-structured data is to group them into clusters. Clustering tree-structured data has a lot of practical applications in different domains.

One possible solution for clustering tree-structured data can be to borrow the traditional information retrieval techniques. In these techniques, each tree is treated as a bag of words and as a result a significant amount of information hidden inside the structure of tree-structured data is ignored. Zaki et al. in [9] presented the *XRULE* algorithm, a structural classifier based on frequent tree patterns, and showed its high performance compared to information retrieval based classifiers. However, *XRULE* needs a training step. Moreover, in this method the trained classification rules during the training step are not revised during the testing step and therefore it is highly probable that the classification rules over-fit for the training data. Another problem of these pattern based approaches is the high complexity of the algorithms that are used for finding frequent tree patterns.

Some other works on clustering tree-structured data are based on *tree edit distance*. The *edit distance* between two *rooted ordered labeled trees* with vertex labels is the minimum cost of transforming one tree into the other by a sequence of elementary operations consisting of deleting and relabeling existing nodes as well as inserting new nodes [4].

Shasha et al. in [2] proposed a tree edit distance metric that permits the insertion and deletion of single nodes anywhere in the tree, not just at the leaves. Chawathe in [15] utilized the inserting and deleting and relabeling operations and tried to optimize the situations when external memory is needed. In addition to these traditional operations, Chawathe et al. in [16] defined a move operator that can move a subtree in a single edit operation. In the subsequent work [17], the copying of subtrees was added to the set of single operations.

In [7] Garofalakis et al. proposed the *XTRACT* algorithm to extract the DTD automatically. In [1] Nierman et al. defined a new method for computing the distance between any two XML documents by introducing *allowable edit operations*.

In [21] Zhao et al. evaluated different partitional and agglomerative approaches for hierarchical clustering. In [3] the authors provided a matching algorithm for measuring the structural similarity between an XML document and a DTD. In [5] a new sequential pattern mining scheme for XML document similarity computation is proposed.

[6] Presented an algorithm for computing differences between old and new versions of an XML document based on hybridization of bottom-up and top-down methods. The next work on detecting the changes between the different versions of an XML documents is [12].

In [19] Lian et al. proposed a hierarchical algorithm for clustering XML documents based on structural information in the data. Flesca et al. in [18] proposed representing the structure of an XML document as a time series, in which each occurrence of a tag in a given context corresponds to an impulse.

Generally, the time complexity for each of these algorithms is at least $O(n^2)$ (n is the number of nodes of each of two trees) and this complexity make these algorithms too heavy for large amount of data.

The more recent work on clustering tree-structured data is the *XCLS* algorithm [14]. This algorithm does not use the

[1] Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email m.haghir@ece.ut.ac.ir.

[2] Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email rahgozar@ut.ac.ir.

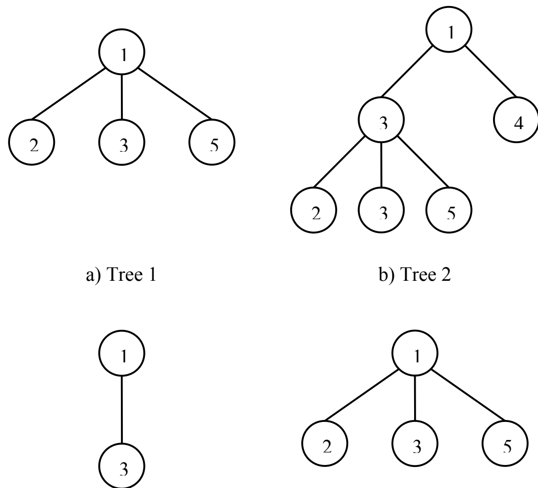
[3] Database Research Group, Control and Intelligent Processing Center Of Excellence, Faculty of ECE, School of Engineering, University of Tehran, Tehran, Iran, email lucas@ipm.ir.

[4] Faculty of CE, Sharif University, Tehran, Iran, email haghir@ce.sharif.edu.

1-4244-0705-2/07/$20.00 ©2007 IEEE

edit distance between two trees. It clusters heterogeneous XML documents based on the global criterion function *LevelSim*. In this algorithm, the hierarchical relationships of elements are considered by counting common elements sharing common *ancestors*.

However, the process of structure matching between two objects in *XLCS*, ignores some embedded similarities. For example consider figure 1. The *XCLS* algorithm generates the tree shown in figure 1.(c) as the similarity between 'tree 1' and 'tree 2'. It seems that the real similarity between these two trees is tree 1.(d). Moreover, when comparing two ordered trees, *XCLS* do not consider the order of children of trees.



a) Tree 1 b) Tree 2

c) The similarity between 'tree 1' and 'tree 2' returned by *XCLS* d) The proper similarity between 'tree 1' and 'tree 2'

Figure 1. A similarity ignored by *XCLS*

To overcome these problems we propose *TreeCluster*, a novel and heuristic algorithm for clustering ordered rooted trees. This algorithm considers a representative tree for each cluster. For each input tree *T*, *TreeCluster* computes the composition of *T* and each of the tree clusters. The tree *T* belongs to the cluster whose composed tree gains the best score. After adding a tree to a cluster the representative tree of that cluster is updated.

The rest of this paper is organized as follows. In section 2, the problem of clustering tree-structured data and required definitions are given. Section 3 describes the process of composition of two trees. In section 4, we define a scoring method. Section 5 describes our proposed algorithm, *TreeCluster*. We empirically evaluate the effectiveness and accuracy of the algorithm in section 6 and the paper is concluded in section 7.

II. DEFINITIONS

Rooted labeled tree [8]: A *rooted labeled tree* $T=(V,E)$, is a *directed*, *acyclic* and *connected* graph with $V=\{0,1,...,n\}$ as the set of vertices and $E=\{(x,y)\mid x,y\in V\}$ as the set of edges. One distinguished vertex $r\in V$ is selected as the root, and for all $x\in V$, there is a unique path from r to x. Further, $l:V\rightarrow L$ is a labeling function mapping vertices to a set of labels $L=\{l_1,l_2,...\}$.

Embedded subtree [20]: For a rooted *unordered tree T* with vertex set *V*, edge set *E*, and no labels on the edges, a tree T' with vertex set V', edge set E', and no labels on the edges, is an embedded subtree of *T* if and only if (1) $V'\subseteq V$, (2) the labeling of the nodes of V' in *T* is preserved in T' and (3) $(v_1,v_2)\in E'$, where v_1 is the parent of v_2 in T', only if v_1 is an ancestor of v_2 in *T*. If *T* and T' are rooted ordered trees, then for T' to be an embedded subtree of *T*, a fourth condition must hold: (4) for $(v_1,v_2)\in V'$, $preorder(v_1)<preorder(v_2)$ in T' if and only if $preorder(v_1)<preorder(v_2)$ in *T*, where the preorder of a node is its index in the tree according to the preorder traversal.

Ascendant-descendant relation: Consider node *i* and node *j* in the rooted ordered tree *T* with *r* as its root and assume that node *j* is deeper than node *i*. There is *ascendant-descendant* relation between *i* and *j* if *i* has placed on the unique path from *j* to *r*. j is called *ascendant* and *i* is called *descendant*.

Sibling relation: Consider node *i* in the rooted ordered tree *T*. There is sibling relation between *i* and any node *j* if these two nodes do not have *ascendant-descendant* relation.

III. COMPOSITION OF TREES

The composition process takes two trees as input and returns their composed tree. The tree generated from the composition of two trees is called *composed tree* and each of two input trees are called *composing trees*.

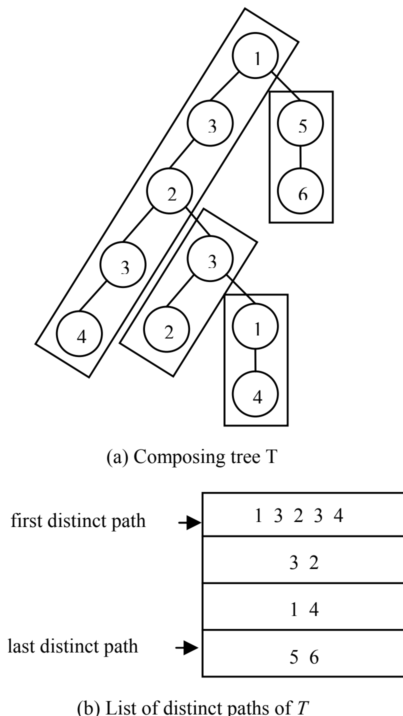
When composing two trees, at first each input tree is divided into some distinct paths. For each leaf node, there is a distinct path and there is no node in tree belonging to more than one distinct path. Moreover, all nodes of a distinct path have *ascendant-descendant* relation. For example, consider the tree of figure 2. In this figure, each rectangle shows a distinct path. We define the parent of distinct path *p* as the parent of the first node of *p* and the depth of *p* as the depth of its parent. The *rightmost distinct path* is the distinct path that contains the rightmost node of the tree in preorder traverse.

Distinct paths of tree *T* are constructed by traversing it in preorder. While *T* is traversed in preorder, when a backtrack (moving from a node to its *ascendant*) occurs a new distinct path is constructed and added to the end of the list of distinct paths. When a node is reached, it is added to the end of the last constructed distinct path. For each of two composing trees a list of distinct paths are formed. These lists are called composing lists.

Lemma 1. Construction of distinct paths of a composing tree is done based on the increasingly sorted preorder number of the first node of each distinct path.

Proof. Directly from the definition of distinct path. ■

By scanning a tree (only once), it is fragmented into its distinct paths and also the parent and the depth of each distinct path is determined.



(a) Composing tree T

(b) List of distinct paths of *T*

Figure 2. Distinct paths of a composing trees

The next step is to construct the distinct paths of the composed tree from the distinct paths of two composing trees. For this purpose, we start from the head of two composing lists and compare their distinct paths. When comparing two distinct paths, one of the following states may occur:

- If both of two distinct paths have parents with the same label, and one of them is the *'embedded subpath'* of another, the *superpath* is added to the composed tree and two next distinct paths are selected from two lists.
- If both of two distinct paths have parents with the same label, but none of them is the *'embedded subpath*' of another, the *'longest common subpath'* of two distinct paths are added to the composed tree and two next distinct paths are selected from two lists.
- Otherwise two current distinct paths are ignored and two next distinct paths from two lists are selected for the comparison.

Lemma 2. The order of construction of distinct paths of a composed tree is according to the preorder number of the first node of each distinct path.

Proof. Directly from the definition of composed tree and lemma 1. ■

A. *Embedded subpath*

Path *a* is the *embedded subpath* of path *b* if: i) all nodes of path *a* do exist in path *b*, ii) for any two nodes $n_1, n_2 \in a$, node n_1 is the *ascendant* of node n_2 if n_1 is the *ascendant* of node n_2 in path *b*. If path *a* is the embedded subpath of path *b*, path *b* would be the embedded superpath of path *a*. For example, consider the path '1—2—4—4—2'. Two paths '1—2—4' and '1—4—2' are the embedded subpaths of this path.

Assume that subpath *b* to node *j*-1 is the embedded subpath of path *a* and node *i-1* of path *a* is equivalent to node *j-1* of path *b*. We want to know if adding node *j* to path *b* keeps it the embedded subpath of path *a*. For this purpose, we start from node *i* of path *a*, and scan *a* until it is terminated or a node with the label of node *j* of path *b* is found. In the first state, there is no node in path *a* equivalent to node *j* of path *b*, and therefore path *b* is not the embedded subpath of path *a*. In the second state, assume that node *k* of path *a* is equivalent to node *j* of path *b*. in this state, path *b* is the embedded subpath of path *a* to node *j*, thus we set *i* to *k*+1 and *j* to *j*+1 and continue the process.

B. *Longest common subpath of two paths*

Path *a* is the *longest common subpath* of two paths *b* and *c* if: i) the set of nodes of *a* is the subset of both the set of nodes of *b* and the set of nodes of *c*, ii) node *x* is the parent of node *y* in *a* if node *x* is the *ascendant* of node *y* in both of *b* and *c*. iii) for each path *d* which satisfies properties i and ii, the length of *a* is greater than or equal to the length of *d*.

Consider two paths *a* and *b*. Assume that their longest common subpath to the node (*i*-1) of path *a* and to node (*j*-1) of path *b* has been found. We want to find the next element of the longest common subpath of these two paths. For this purpose, we read node *i* of path *a*, then we scan path *b* beginning from node *j* until path *b* is terminated or a node with the label of node *i* is found. In the first state, there is no node in path *b* equivalent to node *i* of path *a*, therefore in both paths *a* and *b* the next node is read (*i*++ and *j*++). In the second state, assume that node *k* of path *b* is equivalent to node *i* of path *a*. Node *i* is added to the set of common elements and *i* is increased by one and *j* is set to *k*+1.

The above process for determining the longest common subpath of two paths is not commutative. For example, consider two paths '1—4—2—3' and '1—3—4—2'. The longest common subpath of '1—4—2—3' and '1—3—4—2' is '1—4—2', but the longest common subpath of '1—3—4—2' and '1—4—2—3' is '1—3'. To solve this problem; we compute the longest common subpath of two paths in both of these two states, and return the longer *longest common subpath*. In the above example the path '1—4—2' is considered as the *longest common subpath* of two paths '1—3—4—2' and '1—4—2—3'.

C. *Merging Distinct Paths*

We explained above, how the distinct paths of the composed tree are constructed from the distinct paths of two composing trees. This is not sufficient by itself. The tree generated from the composition of two trees must also be consistent. This means that two nodes in the composed tree have *ascendant-descendant* relation, if this relation does exist in both of composing trees. And two nodes in the composed tree have sibling relation, if this relation does exist in both of composing trees. To hold this property in the composed tree, we must determine the parent of each distinct path in the composed tree correctly. Determining the parent of distinct paths in composed tree is not straight forward. Especially, when the distinct path containing the parent of the next distinct path is the *longest common subpath* of two distinct paths in composing trees; or, when there are some nodes with the same label and one of these labels is the parent of the next distinct path.

Lemma 3. If in tree T the preorder number of node a is less than the preorder number of node b, and a and b belong to two different paths (neither a is the *ascendant* of b nor b is the *ascendant* of node a), the preorder number of all *ascendants* and *descendants* of node a would be less than the preorder number of all *ascendants* (except the root of T) and *descendants* of node b.

Proof. Directly from the definition of preorder traverse. ■

In our approach for determining the parent of distinct path, when a distinct path is added to the composed tree, the *number of possible children* of any of its nodes is determined. The number of possible children of each node is defined as the minimum number of the children of that node in two composing trees. In order to determine the parent of a distinct path that is to be added to the composed tree we take advantage of lemma 4.

Lemma 4. The parent of the distinct path constructed from the top distinct paths of two composing lists is the last node of the composed tree whose number of possible children is greater than zero.

Proof. Omitted due to lack of space. ■

Lemma 5. The nodes of the composed tree whose numbers of possible children are greater than zero belong to the rightmost path of the composed tree.

Proof. Omitted due to lack of space. ■

Lemma 6: When traversing tree T in preorder, whenever a direct backtrack (moving from a node to its direct parent) occurs the stack is popped and whenever a node is reached it is pushed on the stack, after traversing the part of T that is placed before the node i: 1) the top element of the stack would show the parent of i, 2) the content of the stack would show the set of *ascendants* of i, and 3) the content of the stack would show the rightmost path of the scanned part of T.

Proof. Directly from the definition of preorder traverse. ■

Since the parents of the distinct paths of the composed tree belong to the rightmost path of the constructed part of composing tree, as lemma 5 says, we can use a stack-based approach in order to determine the nodes of the composed tree to whom a distinct path can be added. In this approach, for the composed tree a stack is kept and when a node is added to the composed tree and the number of its possible children is greater than zero, this node is pushed to the stack. The parent of next distinct path would be the node at the top of the stack. When a distinct path is added to node a, after decreasing the number of its possible children, if this number is zero, node a will be popped.

IV. SCORE METHOD

There are two states which in some nodes from both of two composing trees are transferred to the composed tree. The first state occurs when two distinct paths from two composing trees have the embedded subpath relation and the second state occurs when the *longest common subpath* of two distinct paths is transferred to the composed tree. Regarding these two states, we define the following score method:

$$Score(\text{tree } T, \text{cluster } C) = \sum_{\text{for each embedded relation}} \frac{N_1}{N_2} + \sum_{\text{for each largest common relation}} \frac{N_3}{N_4}$$

Where N_1 is the number of nodes of the embedded subpath, N_2 is the number of nodes of the embedded superpath, N_3 is the number of nodes of the longest common part of two distinct paths and N_4 is the number of nodes of the larger distinct path.

V. *TREECLUSTER* ALGORITHM

In this algorithm, for each cluster one tree is considered. This tree represents all the trees that are the members of this cluster. The tree related to each cluster is called its *cluster tree*. This algorithm composes two trees, one of them is a *cluster tree*, and another is the tree for which we want to determine the cluster. A score is allocated to each *composed tree*. Tree T belongs to the cluster C if the composition of T and C gains the highest score among all clusters, and then their *composed tree* is considered as the new *cluster tree* of cluster C. The high level pseudo code of *TreeCluster* is presented in figure 3.

VI. EXPERIMENTAL RESULTS

We performed extensive experiments to evaluate the performance and correctness of the *TreeCluster* algorithm using both synthetic data and data from real applications.

Due to the space limitation, here we only report the results for one real dataset. We did our experiments on a 3GHz Intel Pentium IV PC with a 1GB main memory, running windows XP operating system. All algorithms were implemented in C++.

```
TreeCluster (a collection of tree-structured data, number of clusters)
1.  Initiate the cluster trees
2.  For each tree T
3.  Begin
4.          For each cluster C
5.          Begin
6.                  ComposedTree = Composed (T, C);
7.                  If (Score (CompositeTree) > MaxScore)
8.                  Begin
9.                          MaxScore = Score (CompositeTree);
10.                         MaxCluster = CompositeTree;
11.                  End // of if
12.          End // of for each cluster
13.          Dedicate T to cluster i which has gained the highest score;
14.          Set the cluster tree of clusters i equal to MaxCluster;
15.          MaxScore = 0;
16. End // of for each tree
```

Figure 3. The high level pseudo-code of *TreeCluster* algorithm

A. Dataset

We compared the performance of *TreeCluster* using a dataset of IP multicast trees. This dataset consists of MBONE multicast data that was measured during the NASA shuttle launch between the 14th and 21th of February, 1999. It has 333 distinct vertices and each vertex takes the IP address as its label. The data was sampled from this NASA data set with 10 minutes sampling interval and has 1,000 transactions [10], [11].

B. Evaluation Criteria

We use two commonly used evaluation measures: the *intra-cluster* measure and the *inter-cluster* measure. The *intra-cluster* similarity measures the cohesion within a cluster. It determines the amount of similarity among the members of a cluster. The *intra-cluster* similarity of a clustering algorithm is the weighted sum of the *intra-cluster* similarities of all clusters. The *inter-cluster* similarity measures the amount of separation among different clusters. The *inter-cluster* similarity of a clustering algorithm is the weighted sum of *inter-cluster* similarities between any two clusters.

For measuring the amount of similarity between two trees, we use the tree edit distance. Here, we use the Klein's method [13] to compute the edit distance between two trees.

For measuring the *intra-cluster* similarity of each cluster C we propose the following relation:

$$Intra-Cluster\ (cluster\ C) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{TES(T_i, T_j)}$$

Where T_i and T_j are two members of cluster *C*, method *TES* returns the tree edit distance between two trees based on Klein's algorithm [13] and *n* is the number of members of cluster *C*. The *intra-cluster* similarity of a clustering algorithm is the weighted sum of *intra-cluster* of each cluster:

$$Intra-Cluster\ (a\ clustering\ algorithm) = \sum_{i=1}^{k} Intra-Cluster\ (C_i) \times N_i$$

Where k is number of clusters and N_i is number of members of cluster C_i. As long as this measure increases, the accuracy of clustering method increases, too.

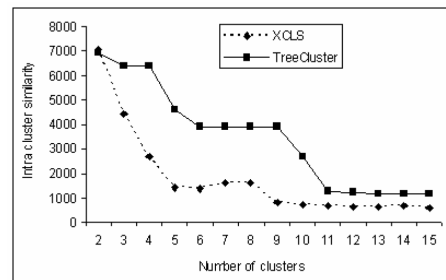
To measure the *inter-cluster* similarity, we present the following relation:

$$Inter-Cluster\ (a\ clustering\ algorithm) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{2TES(C_i, C_j)}{k(k-1)}$$

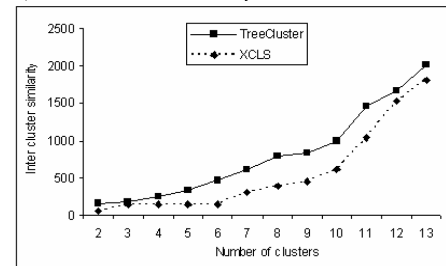
Where C_i and C_j are two clusters of solution, and k is the number of clusters. Similar to *intra-cluster* similarity, by increasing the amount of *inter-cluster* similarity of a clustering solution, the accuracy of clustering increases, too.

C. Empirical Evaluation

Figure 4 displays the *intra-cluster* similarity and *inter-cluster* similarity of *TreeCluster* and *XCLS* on the NASA dataset for different numbers of clusters. For this dataset, when the number of clusters is 2, the values of *intra-cluster* similarity and *inter-cluster* similarity of two algorithms are near, but for any other number of clusters the *TreeCluster* shows better values in comparison to the *XCLS* algorithm.



a) The *intra-cluster* similarity of *TreeCluster* vs *XCLS*



b) The *inter-cluster* similarity of *TreeCluster* vs *XCLS*

Figure 4. *TreeCluster* vs *XCLS* on NASA multicast dataset

Our experiments show that the difference between the *intra-cluster* similarity and *inter-cluster* similarity of our algorithm and those of the *XCLS* algorithm increases when the data are strongly correlated and there are lots of embedded tree patterns between input data. The reason of this behavior is that the *TreeCluster* algorithm is based on finding the embedding or longest common relations between

different parts of trees and as long as number of these relations between two trees increases, this algorithm can rank the cluster trees more precisely. Dense data is the situation in which the other algorithms such as *XCLS* have more problems to cluster correctly.

TreeCluster and *XCLS* are sensitive with respect to the initial values of cluster trees. I.e. when the tree clusters are initialized improperly, the correctness of the clustering degrades. In our experiments, for both algorithms, we initialized each cluster by randomly selecting a tree. The algorithm is run for 10 times and in any of next rounds, the trees obtained from the previous round are considered as the initial values of clusters.

Note that *TreeCluster* holds the commutative property when comparing two trees and do not need to compute the similarity between two trees for two times. However, the *XCLS* algorithm does not hold this property.

VII. Conclusion

In this paper we tried to address the problem of clustering rooted ordered trees by proposing the *TreeCluster* algorithm. This algorithm considers a representative tree for each cluster. For each input tree T, *TreeCluster* computes the composition of T and each of the cluster trees, and T belongs to the cluster c which its composed tree gains the highest score. Then the composed tree is considered as the new tree cluster of c.

As our experimental results show, the *TreeCluster* algorithm considers more embedded similarities between input trees and as a result the correctness and the accuracy of this algorithm is higher than the *XCLS* algorithm. The difference between the *intra-cluster* similarity and *inter-cluster* similarity of our algorithm and those of the *XCLS* algorithm increases when the data are strongly correlated and there are lots of embedded patterns between data.

References

[1] A. Nierman and H. V. Jagadish. Evaluating Structural Similarities in XML Documents. Proceedings of the Fifth International Workshop on the Web and Databases (WebDB), 2002.

[2] D. Shasha and K. Zhang, Approximate tree pattern matching. In Pattern Matching in Strings, Trees and Arrays, chapter 14. Oxford University Press, 1995.

[3] E. Bertino, G. Guerrini, and M. Mesiti, A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its applications. Information Systems, 29(1): 23-46, 2004.

[4] E. D. Demaine, S. Mozes, B. Rossman and O. Weimann. An O(n3)-Time Algorithm for Tree Edit Distance, 2006.

[5] H. P. Leung, F. L. Chung and S. C. F. Chan, On the use of hierarchical information in sequential mining-based XML document similarity computation. Knowledge and Information Systems, 7(4), pp 476-498, 2005.

[6] K. H. Lee, Y. C. Choy and S. B. Cho, An Efficient Algorithm to Compute Differences between Structured Documents, IEEE Transactions on Knowledge and Data Engineering, v.16 n.8, p.965-979, August 2004.

[7] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri and K. Shim, Xtract: A system for extracting document type descriptors from XML documents. In Proc. ACM SIGMOD, pages 165-176, 2000.

[8] M. J. Zaki, Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications, in IEEE Transaction on Knowledge and Data Engineering, vol. 17, no. 8, pp. 1021-1035, 2005.

[9] M. J. Zaki and C. Aggarwal, XRules: an effective structural classifier for XML data. KDD 2003, 316-325, 2003.

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

[11] R. Chalmers and K. Almerothx, On the topology of multicast trees, Technical Report, UCSB, March 2002.

[12] R. Al-Ekram, A. Adma and O. Baysal, diffX: an algorithm to detect changes in multi-version XML documents, Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research, p.1-11, October 17-20, 2005, Toranto, Ontario, Canada.

[13] P. N. Klein, Computing the edit-distance between unrooted ordered trees. In Proceedings of the 6th annual European Symposium on Algorithms (ESA), pages 91–102, 1998.

[14] R. Nayak and S. Xu, XCLS: A Fast and Effective Clustering Algorithm for Heterogenous XML Documents, Accepted by the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Singapore, April 2006.

[15] S. Chawathe, Comparing hierarchical data in extended memory. In Proc. of VLDB, pages 90–101, 1999.

[16] S. Chawathe, A. Rajaraman, H. G. Molina and J. Widom, Change detection in hierarchically structured information. In Proc. of ACM SIGMOD, pages 493–504, 1996.

[17] S. Chawathe and H. G. Molina, Meaningful change detection in structured data. In Proc. Of ACM SIGMOD, pages 26–37, 1997.

[18] S. Flesca, G. Manco, E. Masciari, L. Pontieri and A. Pugliese, Fast Detection of XML Structural Similarities. IEEE Transaction on Knowledge and Data Engineering, Vol 7 (2), pp 160-175, 2005.

[19] W. Lian, D. W. L. Cheung, N. Mamoulis and S. M. Yiu, An Efficient and Scalable Algorithm for Clustering XML Documents by Structure, IEEE Transactions on Knowledge and Data Engineering, v.16 n.1, p.82-96, January 2004.

[20] Y. Chi, R. Muntz, S. Nijssen and J. Kok, Frequent Subtree Mining - An Overview, Fundam. Inform. 66(1-2): 161-198, 2005.

[21] Y. Zhao and G. Karypis, Evaluation of Hierarchical Clustering Algorithms for Document Datasets. The 2002 ACM CIKM, USA.