# Clustering XML Documents using Structural Summaries*

Theodore Dalamagas[1], Tao Cheng[2], Klaas-Jan Winkel[3], and Timos Sellis[1]

[1] School of Electr. and Comp. Engineering
National Technical University of Athens
Zographou, 15773, Athens, Greece
{dalamag,timos}@dblab.ece.ntua.gr
[2] Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
taocheng@cs.ucsb.edu
[3] Faculty of Computer Science
University of Twente
7500, AE Enschede, the Netherlands
winkel@cs.utwente.nl

**Abstract.** This work presents a methodology for grouping structurally similar XML documents using clustering algorithms. Modeling XML documents with tree-like structures, we face the 'clustering XML documents by structure' problem as a 'tree clustering' problem, exploiting distances that estimate the similarity between those trees in terms of the hierarchical relationships of their nodes. We suggest the usage of tree structural summaries to improve the performance of the distance calculation and at the same time to maintain or even improve its quality. Experimental results are provided using a prototype testbed.

## 1 Introduction

Grouping together structurally similar XML documents refers to the application of clustering methods using distances that estimate the similarity between tree structures in terms of the hierarchical relationships of their nodes.

Clustering by structure can be a useful task for many applications. Since the XML language can encode hierarchical data, clustering XML documents by structure can be exploited in any application domain that needs management of hierarchical structures, for example: (a) the discovery of structurally similar web navigational pathways, or tree-like patterns, (b) the discovery of structurally similar macromolecular tree patterns in bioinformatics [10, 5]. Moreover, many XML documents are constructed from data sources without DTDs. XTRACT [7] and DDbE[4] are systems that automatically extract DTDs from XML documents. Identifying groups of XML documents with similar

* Work supported in part by DELOS Network of Excellence on Digital Libraries, IST programme of the EC FP6, no G038-507618, and by PYTHAGORAS EPEAEK II programme, EU and Greek Ministry of Education.

[4] http://www.alphaworks.ibm.com/tech/DDbE

structure can be useful for such systems, where a collection of XML documents should be first grouped into sets of structurally similar documents and then a DTD can be assigned to each set individually.

The main contribution of this work is a methodology for grouping structurally similar XML documents. Modeling XML documents as rooted ordered labeled trees, we face the 'clustering XML documents by structure' problem as a 'tree clustering' problem. We propose the usage of tree structural summaries that have minimal processing requirements instead of the original trees representing the XML documents. We present a new algorithm to calculate tree edit distances and define a structural distance metric to estimate the structural similarity between XML documents. Using this distance, we perform clustering of XML datasets. Experimental results indicate high quality clustering and performance. Using structural summaries instead of the original trees, improves further the performance of the distance calculation without affecting its quality.

We start discussing background information on tree editing issues. Section 3 introduces a metric of structural distance and Section 4 suggests the tree structural summaries. Section 5 presents a new algorithm to calculate tree edit distances. Section 6 discusses evaluation results, and, finally, Section 7 concludes our work.

## 2 Tree editing

The *XML data model* is a graph representation of a collection of atomic and complex objects that without the IDREFS mechanism becomes a *rooted ordered labeled tree* [1]. Since we use such rooted ordered labeled trees to represent XML data, we exploit the notions of *tree edit sequence* and *tree edit distance* originating from editing problems for rooted ordered labeled trees [10]. A *tree edit sequence* is a sequence of tree edit operations (insert node, delete node, etc) to transform $T_1$ to $T_2$. Assuming a cost model to assign costs for every tree edit operation, the *tree edit distance* between $T_1$ and $T_2$ is the minimum cost among the costs of all possible tree edit sequences that transform $T_1$ to $T_2$.

All of the algorithms for calculating the tree edit distance for two ordered labeled trees (tree edit algorithms) [12, 11, 2, 3, 16] are based on dynamic programming techniques related to the string-to-string correction problem [14]. In this work, we consider Chawathe's algorithm [2] as the basic point of reference for tree edit distance algorithms, since it is the fastest available ($O(MN)$, $M$, $N$ the number of nodes in trees) and permits insertion and deletion only at leaves. We believe that using insertion and deletion only at leaves fits better in the context of XML data. For example, it avoids deleting a node and moving its children up one level. The latter destroys the membership restrictions of the hierarchy and thus is not a 'natural' operation for XML data. We next discuss briefly the Chawathe's algorithm.

Chawathe in [2] suggests a recursive algorithm to calculate the tree edit distance between two rooted ordered labeled trees, using a shortest path detection technique on an *edit graph*. The edit graph of trees $T_1$ and $T_2$ is an $(M+1) \times (N+1)$ grid of nodes, having a node at each $(x, y)$ location, $x \in [0 \dots (M+1)]$ and $y \in [0 \dots (N+1)]$. Edit scripts on such trees can be represented using directed lines connecting the nodes in the edit graph. A horizontal line $((x-1, y), (x, y))$ denotes deletion of $T_1[x]$, where $T_1[x]$

refers to the $x$th node of $T_1$ in its preorder sequence. A vertical line $((x, y-1), (x, y))$ denotes insertion of $T_2[y]$. Finally, a diagonal line $((x-1, y-1), (x, y))$ denotes update of $T_1[x]$ by $T_2[y]$.

Figure 1 shows an example of an edit graph which represents an edit script to transform tree $T_1$ to tree $T_2$. Notice that $T_1$ becomes $T_2$ by $(Rep(T_1[2], c)$, $Rep(T_1[3], d)$, $Ins(T_2[4], T_1[1], 3)$. Every edit script that transforms $T_1$ to $T_2$ can be mapped to a path

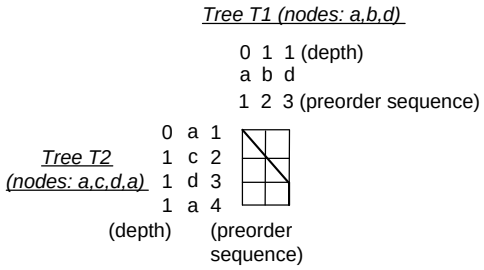


**Fig. 1.** An example of an edit graph.

in an edit graph. The tree edit distance between two rooted ordered labeled trees $T_1$ and $T_2$ is the shortest of all paths to which edit scripts are mapped in an edit graph.

## 3 Structural distance

We next define a structural distance metric to estimate the structural similarity between rooted ordered labeled trees that represent XML documents. This distance can be calculated using tree edit algorithms which determine node operations to transform a tree to another one (like the ones presented in the previous section).

**Definition 1.** *Let $T_1$ and $T_2$ be two trees that represent two XML documents, $\mathcal{D}(T_1, T_2)$ be their tree edit distance and $\mathcal{D}'(T_1, T_2)$ be the cost to delete all nodes from $T_1$ and insert all nodes from $T_2$. The structural distance $\mathcal{S}$ between $T_1$ to $T_2$ is defined as $\mathcal{S}(T_1, T_2) = \frac{\mathcal{D}(T_1,T_2)}{\mathcal{D}'(T_1,T_2)}$.*

The $\mathcal{S}(T_1, T_2)$ is low (high) when the trees have similar (different) structure and high (low) percentage of matching nodes ($0$ ($1$) is the min (max) value). In the example illustrated in Figure 2, $\mathcal{D}'(T_1, T_2) = 12$, since $5$ nodes must be deleted from $T_1$ and $7$ nodes must be inserted from $T_2$, thus $\mathcal{S}(T_1, T_2) = 0.4166$, since tree distance is $5$.

We emphasize on the efficient computation of the structural distance metric (a) showing how to maintain the structural information present in XML documents using compact trees, called structural summaries and (b) proposing a new algorithm to calculate tree edit distances to use it for the structural distance calculation.

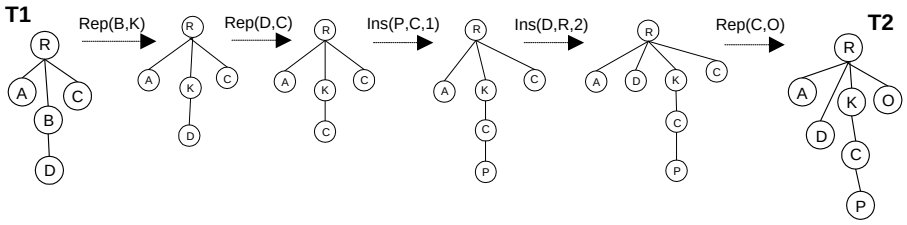


**Fig. 2.** The sequence of tree edit operations to transform $T_1$ to $T_2$ with minimum cost: total cost=5 (assuming unit cost for each operation).

## 4 Tree structural summaries

Nesting and repetition of elements is the main reason for XML documents to differ in structure although they come from a data source which uses one DTD. A *nested-repeated node* is a non-leaf node whose label is the same with the one of its ancestor. Following a pre-order tree traversal, a *repeated node* is a node whose path (starting from the root down to the node itself) has already been traversed before. Figure 5 has an example of redundancy: trees $T_1$ and $T_3$ differ because of nodes $A$ (nested-repeated) and $B$ (repeated).

We perform *nesting reduction* and *repetition reduction* to extract structural summaries for rooted ordered labeled trees which represent XML documents. For nesting reduction, we traverse the tree using pre-order traversal to detect nodes which have an ancestor with the same label in order to move up their subtrees. For repetition reduction, we traverse the tree using pre-order traversal, too, ignoring already existed paths and keeping new ones, using a hash table. The two algorithms are presented in Figures 3 and 4.

```
void reduceNesting(TreeNode node) {
    TreeNode pos = FindAncestor(node);
    if (pos == null) {
      for (int i=0; i<node.numOfChildren(); i++)
          reduceNesting(node.getChild(i));
    }
    else {
      for (int i=0; i<node.numOfChildren(); i++)
      {
          node.getChild(i).setParentNode(pos);
          pos.addChild(node.getChild(i));
          node.getChildNodes().remove(i);
          i=i-1;
      }
    }
}
```

**Fig. 3.** Nesting reduction.

```
void reduceRepeat(TreeNode node, String currentPath) {
    String path = currentPath + "/" + node.getNodeName();
    if (!hash.containsKey(path)) {
      hash.put(path, node);
      for (int i=0; i<node.numOfChildren(); i++)
          reduceRepeat(node.getChild(i), path);
    }
    else {
      TreeNode destination = (TreeNode)hash.get(path);
      int numOfOldChildren = destination.numOfChildren();
      for (int i=0; i<node.numOfChildren(); i++)
          destination.addChild(node.getChild(i));
      node.DeleteNode();
    }
    for (int i = numOfOldChildren;
          i<destination.numOfChildren(); i++)
          reduceRepeat(destination.getChild(i), path);
}
```

**Fig. 4.** Repetition reduction.

Figure 5 presents an example of structural summary extraction from $T_1$. Applying the nesting reduction phase on $T_1$ we get $T_2$, where there are no nested-repeated nodes. Applying the repetition reduction on $T_2$ we get $T_3$ which is the structural summary of $T_1$.

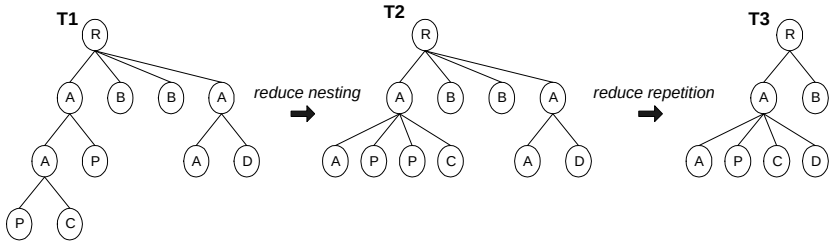


**Fig. 5.** Structural summary extraction.

## 5 Calculating tree edit distances

Our approach for calculating tree edit distances uses a dynamic programming algorithm which is close to Chawathe's algorithm [2] in terms of the tree edit operations that are used. However, the recurrence that we use does not need the costly edit graph calculation of the latter (see Section 2 as well as the timing analysis in Section 6.1). We next present our tree edit distance algorithm.

Permitted tree edit operations are *insertion*, *deletion* and *replacement* of nodes (insertion and deletion restricted to leaf nodes). Method $CalculateDistance(r_1, r_2)$ calculates the tree edit distance of $T_1$ and $T_2$, with roots $r_1$ and $r_2$, respectively:

```
int CalculateDistance(TreeNode s, TreeNode t) {
   int[][] D=new int[numOfChildren(s)+1][numOfChildren(t)+1];
   D[0][0]=UpdateCost(LabelOf(s), LabelOf(t));
   for (int i=1; i≤numOfChildren(s); i++) D[i][0]= [i-1][0]+numOfNodes(s_i);
   for(int j=1; j≤numOfChildren(t); j++) D[0][j]=D[0][j-1]+numOfNodes(t_j);
   for (int i=1; i≤numOfChildren(s); i++)
      for (int j=1; j≤numOfChildren(t); j++)
         D[i][j]=Min(D[i-1][j-1]+CalculateDistance(s_i,t_j), D[i][j-1]+numOfNodes(t_j),
                     D[i-1][j]+numOfNodes(s_i));
   Return D[numOfChildren(s)][numOfChildren(t)]; }
```

where: $s_i$ ($t_j$) is the $i_{th}$ ($j_{th}$) subtree of node $s$ ($t$), $numOfChildren(s)$ returns the number of child nodes of node $s$, $numOfNodes(s)$ returns the number of nodes of the subtree rooted at $s$, $UpdateCost(LabelOf(s), LabelOf(t))$ returns the cost to make the label of node $s$ the same as the label of node $t$ ( 0 if $LabelOf(s) = LabelOf(t)$ or 1 otherwise).

In the algorithm, $D[i][j]$ keeps the tree edit distance between tree $T_1$ with only its first $i$ subtrees and tree $T_2$ with only its first $j$ subtrees. Assuming unit cost (1) for

an insert or delete operation, we use $numOfNode(s_i)$ to represent the cost to delete the $i$th subtree of node $s$ and $numOfNodes(t_j)$ to represent the cost to insert the $j$th subtree of node $t$. We next describe in detail how the algorithm computes the minimum distance between trees:

1. Having the value $D[i][j-1]$ and the number of nodes in the subtree rooted at $t_j$, we spend $d_1 = D[i][j-1] + numOfNodes(t_j)$ to transform the subtree rooted at $s$ to the subtree rooted at $t$. Since the cost of an *insert node* operation is 1, we use $numOfNodes(t_j)$ to represent the cost to insert the $j_{th}$ subtree of node $t$ in the subtree rooted at $s$.
2. Similarly, having the value $D[i-1][j]$ and the number of nodes in the subtree rooted at $s_i$, we spend $d_2 = D[i-1][j] + numOfNodes(s_i)$ to transform the subtree rooted at $s$ to the subtree rooted at $t$. Since the cost of a *delete node* operation is 1, we use $numOfNodes(s_i)$ to represent the cost to delete the $i_{th}$ subtree of $s$.
3. Having the value $D[i-1][j-1]$, we spend
   $d_3 = D[i-1][j-1] + CalculateDistance(s_i, t_j)$
   to transform the subtree rooted at $s$ to the subtree rooted at $t$. $CalculateDistance$ is recursively called for the $i_{th}$ and $j_{th}$ children of nodes $s$ and $t$, respectively.

$D[i][j]$ keeps the minimum from $d_1, d_2$ and $d_3$ values. Figure 6 shows an example of $D[][]$ calculation. $D$[2][3] is the distance between $T_1$ with only its first 2 subtrees and $T_2$ with only its first 3 subtrees.

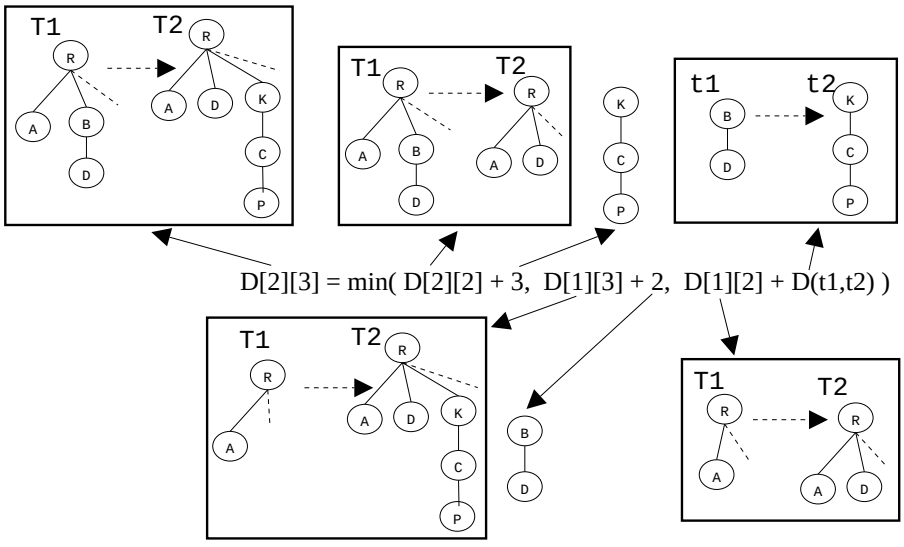


**Fig. 6.** Calculating $D$[2][3] for $T_1$ and $T_2$.

We call the function $CalculateDistance$ once for each pair of nodes at the same depth in the 2 structural summary trees, so the complexity is $O(MN)$, where $M$ is the

number of nodes in the tree rooted at $s$, and $N$ is the number of nodes in the tree rooted at $t$.

## 6 Evaluation

We implemented a testbed to perform clustering on synthetic and real data, using structural distances[5]. Two sets of 1000 synthetic XML documents were generated from 10 real-case DTDs[6], varying the parameter $MaxRepeats$ to determine the number of times a node will appear as a child of its parent node. For real data set we used 150 documents from the ACM SIGMOD Record and ADC/NASA[7].

We chose single link hierarchical clustering method to be the basic clustering algorithm since it has been shown to be theoretically sound, under a certain number of reasonable conditions [13]. However, the structural distance metric can be exploited by any other clustering algorithm to discover groups of structurally similar XML documents. To determine the most appropriate clustering level for the single link hierarchies, we adopted $C-$index [8], by calculating its values, varying the clustering level in different steps.

While checking time performance is straightforward, checking clustering quality involves the calculation of metrics based on priori knowledge of which documents should be members of the appropriate cluster. Thus, the evaluation procedure raises the following issues:

1. The number of clusters discovered should ideally match the number of DTDs where the XML documents are based on. To estimate the number of clusters, we adopt the $C$-index in the single-link clustering method.
2. The clusters discovered should be mapped to the original DTDs where the XML documents are based on. For this reason, we performed the following tasks:
   (a) We derived DTDs $D_1^c, D_2^c, \ldots, D_k^c$ for every cluster $C_1, C_2, \ldots, C_k$, using the XML documents assigned to that cluster[8].
   (b) We parsed the derived DTDs $D_1^c, D_2^c, \ldots, D_k^c$ and the original DTDs $D_1, D_2, \ldots, D_m$, creating derived trees $t_1^c, t_2^c, \ldots, t_k^c$ trees and original trees $t_1, t_2, \ldots, t_m$, respectively[9].
   (c) For every original tree $t_i$, $1 \leq i \leq m$, we calculated the structural distances $\mathcal{S}(t_i, t_1^c)$, $\mathcal{S}(t_i, t_2^c)$, $\ldots$, $\mathcal{S}(t_i, t_k^c)$. The lowest of these values $\mathcal{S}_{min}(t_i, t_p^c)$, $1 \leq p \leq k$, indicates that the original DTD $D_i$ corresponds to cluster $C_p$. After that, we had a mapping between the original DTDs and the clusters produced.

   We note that the $C$-index method might give a number of clusters which is different than the number of DTDs where the XML documents are based on ($m \neq k$), that is there might be clusters not mapped to any of the original DTD. In such case, clustering quality metrics will be affected.

[5] All the experiments were performed on a Pentium III 800MHz, 192MB RAM.
[6] IBM's Alphaworks generator (DTDs: www.xmlfi les.com and www.w3schools.com)
[7] www.acm.org/sigmod/record/xml and xml.gsfc.nasa.gov
[8] using AlphaWorks Data Descriptors by Example: www.alphaworks.ibm.com/tech/DDbE
[9] DTD parser: www.wutka.com/dtdparser.html

To evaluate the clustering results, we used two popular metrics: *precision* $PR$ and *recall* $R$ [13]. Let (a) $a_i$ be the number of the XML documents in the extracted cluster $C_i$ that were indeed members of that cluster (correctly clustered), (b) $b_i$ be the number of XML documents in $C_i$ that were not members of that cluster (misclustered) and (c) $c_i$ be the number of XML documents not in $C_i$, although they should be $C_i$'s members. Then, $PR = \sum_i a_i / (\sum_i a_i + \sum_i b_i)$ and $R = \sum_i a_i / (\sum_i a_i + \sum_i c_i)$. High precision and recall indicate excellent clustering quality.

### 6.1 Timing analysis

We compared the time to derive the two structural summaries from two trees representing two XML documents, and calculate their structural distance vs the time to calculate the structural distance between the two original trees. We tested both Chawathe's algorithm and our algorithm using randomly generated XML documents, with their number of nodes ranging from 0 to 2000. The time decrease (%) for calculating the structural distance between 2 XML documents using their summaries instead of using the original trees lays around $80\%$ on average for Chawathe's and around $50\%$ on average for our algorithm. Figure 7 presents the time decrease (%) for calculating the structural distance between 2 XML documents, using our algorithm instead of Chawathe's algorithm (52% on average). Chawathe's algorithm is slower due to the pre-calculation of the edit graph (it spends more than $50\%$ of the time needed for the overall distance calculation). To give a sense about the scaling of the calculations, Figure 8 presents the timing performance for our algorithm, with or without summaries.

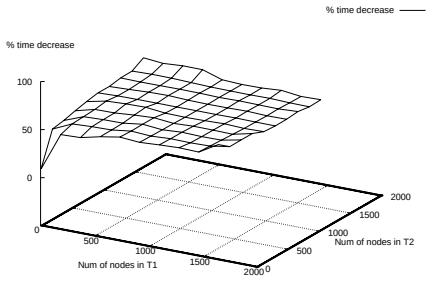


**Fig. 7.** Time decrease (%) for structural distance calculation using our algorithm instead of Chawathe's.

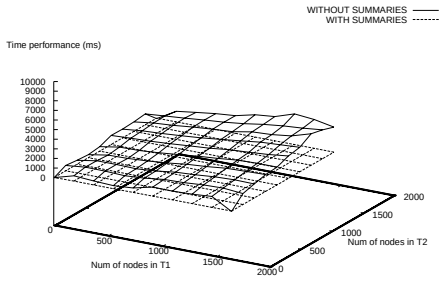


**Fig. 8.** Calculating the structural distance between 2 trees using our algorithm: time performance with or without summaries (ms).

### 6.2 Clustering evaluation

We performed single link clustering on synthetic and real data, using structural distances returned from Chawathe's algorithm and our algorithm, with or without structural summaries, and calculated $PR$ and $R$ values.

Table 1 presents the $PR$ and $R$ values using the two algorithms on synthetic and real data. For both algorithms, we note that for small trees ($maxRepeats = 3$) with only a few repeated elements and, thus, with the structural summaries being actually the original trees, the clustering results are the same with or without summaries. On the other hand, for larger trees ($maxRepeats = 6$) with many repeated elements, clustering quality is improved in Chawathe's and maintained in high levels in our algorithm. For real data, summary usage maintains the already high quality clustering results obtained without using summaries, for both algorithms. We should note that the differences in the clusters obtained by the two algorithms in identical datasets, although both calculate the minimum cost to transform a tree to another one, are due to the cost models used for the tree edit operations. This does not affect the evaluation procedure, since our concern is to show the effect of summaries on clustering quality in both algorithms.

The evaluation results indicate that structural summaries maintain the clustering quality, that is they do not hurt clustering. Thus, using structural summaries we can clearly improve the performance of the whole clustering procedure, since summaries have lower processing requirements than the original trees. A further decrease can be achieved exploiting our algorithm since it shows improved performance compared to Chawathe's.

## 7 Conclusions

This work studied methods to cluster XML documents by structure, exploiting structural distances and structural summaries. We defined a structural distance metric to estimate the structural similarity between two XML documents. We proposed structural summaries that have minimal processing requirements, maintaining the structural relationships of the elements in an XML document. Also, we presented a new algorithm to calculate tree edit distances. Finally, we implemented a testbed to perform clustering on synthetic and real data, using structural distances. Our results showed that structural summaries clearly improved the performance of the clustering procedure, while maintaining the clustering quality. Moreover, our structural distance algorithm showed improved performance compared to Chawathe's.

Methods for file change detection (see for example [4]) are related to our work, but most of them do not compute the minimal tree edit sequence. Other works, like [15],

| **Synthetic data** | |
|---|---|
| **CH-n-3**: 11 clusters, $PR = 0.71$, $R = 0.90$ | **CH-y-3**: 11 clusters, $PR = 0.71$, $R = 0.90$ |
| **CH-n-6**: 11 clusters, $PR = 0.58$, $R = 0.89$ | **CH-y-6**: 12 clusters, $PR = 0.83$, $R = 0.96$ |
| **D-n-3**: 11 clusters, $PR = 1.00$, $R = 0.98$ | **D-y-3**: 11 clusters, $PR = 1.00$, $R = 0.98$ |
| **D-n-6**: 12 clusters, $PR = 1.00$, $R = 0.97$ | **D-y-6**: 11 clusters, $PR = 1.00$, $R = 0.98$ |
| **Real data** | |
| **CH-n**: 4 clusters, $PR = 1.00$, $R = 0.98$ | **CH-y**: 3 clusters, $PR = 1.00$, $R = 1.00$ |
| **D-n**: 4 clusters, $PR = 1.00$, $R = 0.98$ | **D-y**: 3 clusters, $PR = 1.00$, $R = 1.00$ |

**Table 1.** $PR, R$ values for clustering synthetic and real data (**CH**: Chawathe's algorithm, **D**: our algorithm, **n** (**y**): without (with) summary, **3** (**6**): maxRepeat=3 (6)).

concentrate on unorder trees. To the best of our knowledge, the only work directly compared with ours is [9]. Their set of tree edit operations include two new ones which refer to whole trees. They preprocess the trees to detect whether a subtree is contained in another tree. There are no results about $PR$ and $R$ values. In our work, we diminish the possibility of having repeated subtrees using structural summaries instead of expanding the tree edit operations. Summaries are used as an index structure to speed up the tree distance calculation. Such an approach can reduce the performance cost in every algorithm that estimates the structural distance between trees.

As a future work, certain properties of the structural distance should be explored and confirmed. Positivity, symmetry and triangular inequality are such properties. The experimental results show that these properties hold (see for example Figures 7, 8 for the symmetry) but formal study is needed to confirm it. Also, we will study how to employ vector-based representation of tree structures (like in [6]) to further explore the problem of clustering by structure.

## References


1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kaufmann, 2000.
2. S. S. Chawathe. Comparing hierarchical data in external memory. In *Proc. of the VLDB Conference, Edinburgh, Scotland, UK*, 1999.
3. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. of the ACM SIGMOD Conference*, USA, 1996.
4. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proc. of the ICDE Conference*, San Jose, USA, 2002.
5. H. G. Direen and M. S. Jones. Knowledge management in bioinformatics. In A. B. Chaudhri, A. Rashid, and R. Zicari, editors, *XML Data Management*. 2003. Addison Wesley.
6. S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Detecting similarities between XML documents. In *Proc. of WebDB'02*, 2002.
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. of the ACM SIGMOD Conference,Texas, USA*, 2000.
8. G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50, 1985.
9. A. Nierman and H. V. Jagadish. Evaluating structural similarity in xml documents. In *Proc. of the WebDB Workshop*, Madison, Wisconsin, USA, June 2002.
10. D. Sankoff and J. Kruskal. *Time Warps, String Edits and Macromolecules, The Theory and Practice of Sequence Comparison.* CSLI Publications, 1999.
11. S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6:184–186, 1977.
12. K. C. Tai. The tree-to-tree correction problem. *Journal of ACM*, 26, 1979.
13. C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.
14. R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, 1974.
15. Y. Wang, D. DeWitt, and Jin-Yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proc. of the ICDE Conference*, Bangalore, India, 2003.
16. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.