Clustering XML documents by structure

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

[1]School of Electr. and Comp. Engineering
National Techn. University of Athens
15773, Zographou, Athens, Greece
{dalamag,timos}@dblab.ece.ntua.gr

[2]Dept. of Computer Science
Zhejiang University
310027, Hangzhou, Zhejiang, China
chengtao@zju.edu.cn

[3]Faculty of Computer Science
University of Twente,
7500, AE Enschede, the Netherlands
winkel@cs.utwente.nl

Abstract

While the processing and management of XML data are popular research issues, operations based on the structure of XML data have not yet received strong attention. These operations involve, among others, the grouping of structurally similar XML documents. Such grouping refers to the application of clustering methods using distances that estimate the similarity between tree structures. This paper presents a framework for clustering XML documents by structure. Modeling the XML documents as rooted ordered labeled trees, we explore the application of clustering algorithms using distances that estimate the similarity between those trees in terms of the hierarchical relationships of their nodes. We suggest the usage of structural summaries for trees to improve the performance of the distance calculation and at the same time to maintain or even improve its quality. Our approach is tested using a prototype testbed.

Keywords: XML, structural similarity, tree edit distance, structural summary, clustering

1 Introduction

The *eXtensible Markup Language (XML)*[1] is becoming the standard data exchange format among Web applications, providing interoperability and enabling automatic processing of Web resources. An XML document is a hierarchically structured and self-describing piece of information consisting of atomic elements or complex elements, that is elements with nested subelements.

[1]http://www.w3c.org/XML

An XML document incorporates structure and data in one entity. To this extend, XML data are semistructured data [1].

While the processing and management of XML data are popular research issues (see [2] as a reference), operations based on the structure of XML data have not yet received strong attention. The structure of an XML document can be specified using a set of regular expression patterns that element sequences should conform to, called *Document Type Descriptor (DTD).* Richer constraints, not only on the structure but also on the type of data in an XML document, can be set using the XML Schema[2].

Applying structural transformations and grouping together structurally similar XML documents are operations based on the structure of XML data. Structural transformations are the basis for using XML as a common data exchange format: $XSLT$[3] uses templates rules to define transformations for XML documents, while [3] presents a high level language to specify structural document transformations in a descriptive way. 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.

1.1 Motivating examples

There are many cases where clustering by structure can assist application tasks:

- **Automatic extraction of DTDs:** Many XML documents are constructed from data sources like RDBMSs, flat files, etc, without DTDs. XTRACT [4] and IBM AlphaWorks DDbE[4] are systems that automatically extract DTDs from XML documents. Identifying groups of XML documents that share a similar 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.

- **General grouping by structure:** Since the XML language can encode and represent various kinds of hierarchical data, clustering XML documents by structure can be exploited in any application domain that needs management and processing of hierarchical data. Some related examples follow:

  – **Bioinformatics:** The discovery of structurally similar macromolecular tree patterns, encoded as XML documents, is a quite useful task in bioinformatics. The detection of homologous protein structures encoded as XML documents (i.e. sets of protein structures sharing a similar structure) is such an example (see [5]). Other XML encodings for life sciences are presented in [6].

[2]www.w3.org/XML/Schema
[3]http://www.w3c.org/TR/XSLT
[4]http://www.alphaworks.ibm.com/tech/DDbE

– **Spatial data management:** Spatial data are often organized in data model catalogs expressed in XML's hierarchical format. For example, areas that include forests with lakes and farms can be represented as tree-like structures using XML documents. Clustering by structure can identify spatial entities with similar structure, e.g. entities with areas that include forests with lakes. Examples on using XML representation for geographical data are presented in [7].

1.2 Contribution

The main contribution of this work is a methodology for clustering XML documents by structure, exploiting algorithms to calculate the minimum cost (known as tree edit distance) to transform a rooted ordered labeled tree to another one, using operations on nodes. Specifically:

1. We provide an overview of algorithms that calculate the tree edit distance for two rooted ordered labeled trees.
2. Modeling XML documents as rooted ordered labeled trees, we suggest the usage of tree structural summaries. These summaries maintain the structural relationships between the elements of an XML document and at the same time have minimal processing requirements instead of the original trees representing the XML documents.
3. We propose a new algorithm to calculate tree edit distances and we define a structural distance metric to estimate the structural similarity between two rooted ordered labeled trees.
4. We present a prototype testbed to perform clustering of large XML datasets using the structural distance metric. Experimental results indicate that
   (a) our algorithm for calculating the structural distance between two rooted ordered labeled trees, representing XML documents, provides high quality clustering and improved performance compared to others,
   (b) using structural summaries to represent XML documents instead of the original trees, improves further the performance of the structural distance calculation without affecting its quality.

1.3 Outline

Section 2 presents background information for the representation of XML data as rooted ordered labeled trees or graphs and analyzes various algorithms related to the tree editing problem and tree editing distance. Section 3 suggests the structural summaries for rooted ordered labeled trees. Section 4 presents a new algorithm to calculate the tree edit distance between two rooted ordered labeled trees and introduces a metric of structural distance. Section 5 analyzes the clustering methodology. Section 6 describes the architecture of our testbed used for the evaluation procedure and presents the evaluation results, and finally Section 7 concludes our work.

2 Background

XML data are semistructured data, that is a hierarchically structured and self-describing piece of information, consisting of atomic and complex objects. We next present background information related to (a) popular models used for representing semistructured data and (b) editing problems for rooted ordered labeled trees produced from such kind of models.

2.1 Modeling semistructured data

Models for semistructured data are mainly graph-based or tree-based. They are simple and flexible models which capture schemaless, self-describing and irregular data.

The *object exchange model (OEM)* is a graph representation of a collection of objects. OEM was introduced in the TSIMMIS project [8, 9]. Every OEM object has an identifier (oid) and a value, atomic or complex. An atomic value is an integer, real, string or any other data, while a complex value is a set of oids, each linked to the parent node using a textual label. Objects with atomic values are called atomic objects and objects with complex values are called complex objects. Figure 1 presents an example of an OEM database. In this example there are four complex objects (the root, two 'SLR camera' objects and one 'Point & Shoot camera' object) and seven atomic objects (three 'brand', three 'price' and one 'color').

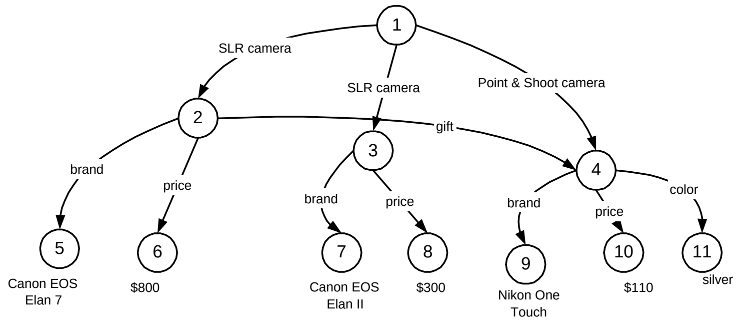


Figure 1: OEM example

The *XML data model* is another graph representation of a collection of atomic and complex objects. However, while the OEM model denotes graphs with labels on edges the XML data model denotes graphs with labels on nodes. See how the example of Figure 1 can be expressed using the XML data model in Figure 2. The XML data model provides a mechanism (IDREFS) to define references, that is unique identifiers for elements. Using references, one can refer to an element using its identifier. See for example the dotted edge in Figure 2, which is a reference from element 7 to element 4.

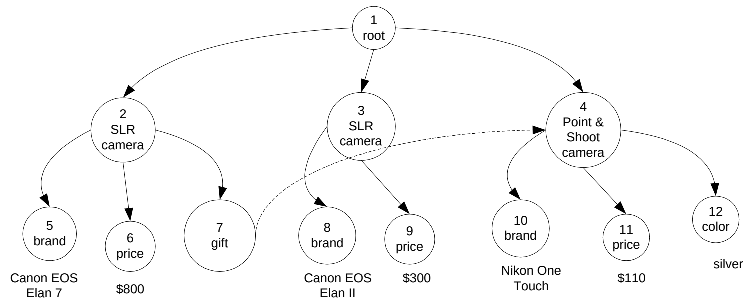


Figure 2: XML data model example

Without IDREFs, the XML data model becomes a *rooted ordered labeled tree*. Since we use such rooted ordered labeled trees to represent XML data, we exploit ideas originating from editing problems for rooted ordered labeled trees.

2.2 Tree editing

A *rooted ordered labeled tree* T is a set of $(n+1)$ nodes $\{r, n_i\}$ with $i = 1 \dots n$. The children of each node are ordered. A label is associated with every node. r is the root of T and the remaining nodes $n_1 \dots n_n$ are partitioned into m sets $T_1 \dots T_m$, each of which is a tree. These trees $(T_1 \dots T_m)$ are called *subtrees* of the root of T. The root of T_i, $i = 1 \dots m$, is the ith *child* of the root of T and the root of T is the *parent* of the root of T_i. Generally, if $t_1 \dots t_k$ are subtrees of the root of a tree t, with t be a subtree of T, then the root of t is a *parent* for the roots of $t_1 \dots t_k$ and the roots of $t_1 \dots t_k$ are *children* of the root of t. Node x is an *ancestor* of y and y is a *descendant* of x if there is a path of nodes $n_0, n_1, \dots, n_k$ such that $x = n_0$, $y = n_k$ and $n_i = parent(n_{i+1})$ for $i = 0 \dots k$. A *leaf* is a node with no descendants.

An *atomic tree edit operation* on a rooted ordered labeled tree is either the deletion of a node, or the insertion of a node, or the replacement of a node by another one. A *complex tree edit operation* is a set of atomic tree edit operations, treated as one single operation. An example of a complex tree edit operation is the insertion of a whole tree as a subtree in another tree, which is actually a sequence of atomic node insertion operations.

The *tree edit sequence* and the *tree edit distance* between two rooted ordered labeled trees that represent two XML documents are defined as follows:

Definition 1 *Let T_1 and T_2 be rooted ordered labeled trees. A* tree edit sequence *is a sequence of tree edit operations that transforms T_1 to T_2.*

Definition 2 *Let T_1 and T_2 be rooted ordered labeled trees. 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 between the costs of all possible tree edit sequences that transform T_1 to T_2.*

Figure 3 illustrates an example of a tree edit sequence of node insertions, deletions and replacements to transform a tree T_1 to a tree T_2. Assuming unit costs for all operations, that sequence does not have the minimum cost (see Figure 10 for the sequence of operations with the minimum cost).

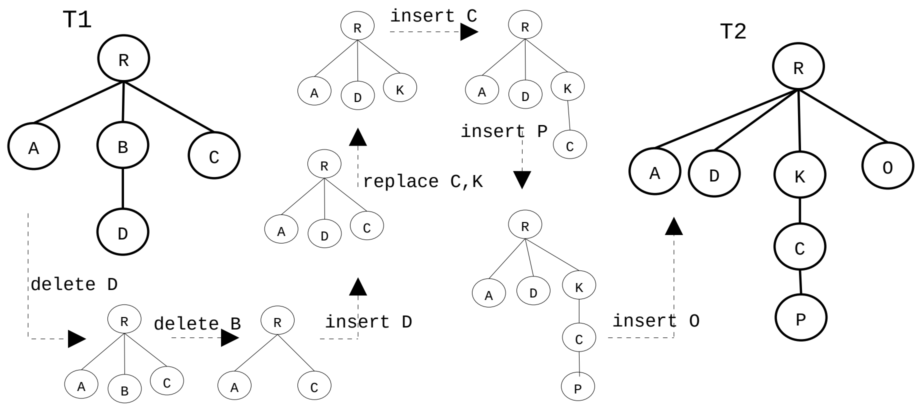


Figure 3: An example of an edit sequence to transform T_1 to T_2.

There are different approaches [10, 11, 12, 13] to determine tree edit sequences and tree edit distances. All utilize similar tree edit operations with minor variations. Before we discuss each algorithm in detail, we present a general form of those tree edit operations with the variations that the aforementioned algorithms use.

1. **insert node**:

   (a) *Variation I* ($Ins^l(x, y, i)$): In this variation every new node is inserted only as a leaf. Let x be a node to be inserted as the i_{th} child of node y in tree t_1 and $y_1 \dots y_n$ be the children of y. In the new tree t_2 produced after inserting the node x, node y will have $y_1 \dots y_{i-1}, x, y_i, y_{i+1}, \dots y_n$ as children.

   (b) *Variation II* ($Ins(x, y, i)$): In this variation, the restriction that a new node can be inserted only as a leaf is relaxed. Let x be a node to be inserted as the i_{th} child of node y in tree t_1 and $y_1 \dots y_n$ be the children of y. In the new tree t_2 produced after inserting node x, x takes a subsequence of the children of y as its own children. Thus, given p, node y will have $y_1 \dots y_j, x, y_{p+1}, \dots y_n$ as children and x will have $y_{j+1}, y_{j+2}, \dots y_p$ as children.

We assign a unit cost $c_i(x)$ to *insert node* operation for a node x.

2. **delete node**:

   (a) *Variation I* ($Del(y)$): In this variation, the children of the deleted node become the children of its parent. Let x be a node in tree t_1 and $x_1 \ldots x_n$ be the children of x. Let $y = x_i$ be one of $x_1 \ldots x_n$ nodes with $y_1 \ldots y_m$ as children. In the new tree t_2 produced after deleting the node y, node x will have $x_1 \ldots x_{i-1}, y_1 \ldots y_m, x_{i+1} \ldots x_n$ as children.

   (b) *Variation II* ($Del^l(y)$): In this variation, only leaf nodes can be deleted. Let x be a node in tree t_1 and $x_1 \ldots x_n$ be the children of x. Let $y = x_i$ be one of $x_1 \ldots x_n$ nodes (y is a leaf). In the new tree t_2 produced after deleting the node y, node x will have $x_1 \ldots x_{i-1}, x_{i+1} \ldots x_n$ as children.

   We assign a unit cost $c_d(y)$ to *delete node* operation for a node y.

3. **replace node** *($Rep(x, y)$)*: Let y be a new node and x a node in tree t_1 to be replaced by y. In the new tree t_2 produced after replacing node x with y, node y will have the same father and the same children as x in t_1. We assign a cost $c_r(x, y)$ to *replace node* operation for a node x replaced by y. This cost may be variable (for example 1 if the node to be replaced has different label and 0 otherwise) or a constant unit.

4. **move subtree** *($Mov(x, y, k)$)*: Let x be the root of a subtree in tree t_1. $Mov(x, y, k)$ moves the entire subtree rooted at x, along with x, to be the k_{th} child of y in t_2. We assign a unit cost $c_m(x)$ to *move subtree* operation.

2.3 Review of tree edit algorithms

We next discuss each algorithm [10, 11, 12, 13] in detail. All algorithms permit tree edit operations from the set of operations presented in the previous section.

2.3.1 Selkow's algorithm

Selkow in [10] suggests a recursive algorithm to calculate the tree edit distance between two rooted ordered labeled trees. An *insert node* operation is permitted only if the new node becomes a leaf. A *delete node* operation is permitted only at leaf nodes. Any node can be updated using the *replace node* operation. So, the set of permitted tree edit operations is $\{Ins^l(x, y, i)$, $Del^l(y)$, $Rep(x, y)\}$, with costs $c_i(x)$, $c_d(y)$, and $c_r(x, y)$ ($c_r(x, y) = 1$ if the node to be replaced has different label, $c_r(x, y) = 0$ otherwise), respectively (see Section 2.2). The cost $W_i(x)$ to insert a whole subtree t_2, rooted at node x, anywhere in tree t_1 follows:

$$W_i(x) = \sum_{j=0}^{k} c_i(x_j) \tag{1}$$

where $x_0 = x$ and $x_1 \ldots x_k$ are all descendants of x. The cost $W_d(y)$ to delete a whole subtree t_2, rooted at node y, anywhere in tree t_1 follows:

$$W_d(y) = \sum_{j=0}^{k} c_d(y_j) \tag{2}$$

where $y_0 = y$ and $y_1 \ldots y_k$ are all descendants of y.

A tree T is denoted as $T(1, n_k)$, where 1 is the label of its root, k is the number of subtrees connected to the root, and n_k is the last node of the k_{th} subtree in T. All nodes are labeled according to the preorder sequence. Figure 4 presents such an arrangement for trees $T_1(1, i_m)$ and $T_2(1, j_n)$. The algorithm to compute the distance $\mathcal{D}$ between the two trees proceeds recursively

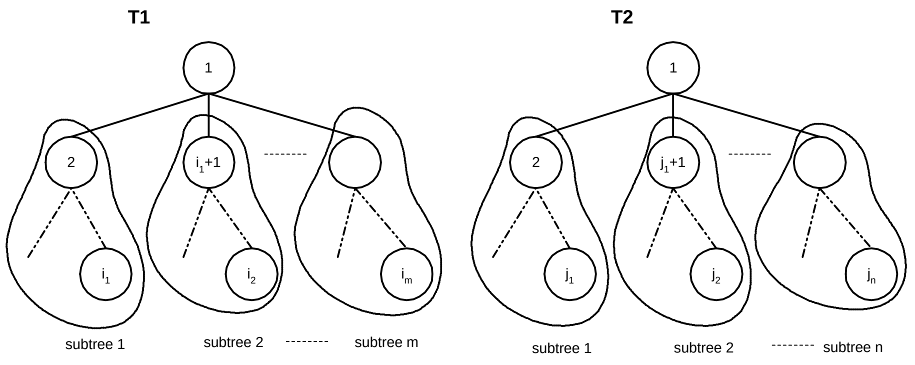


Figure 4: Example trees.

by calculating the distance between their subtrees. The idea of the main recursion is that the calculation of the distance between two (sub)trees t_1 and t_2 requires the calculation of 4 distances:

1. distance between t_1 without its last subtree and t_2,
2. distance between t_1 and t_2 without its last subtree,
3. distance between t_1 without its last subtree and t_2 without its last subtree, and
4. distance between the last subtree of t_1 and the last subtree of t_2.

Let r be the root of current subtree t_1 of T_1, k the number of subtrees in r, and i the last node of last subtree of t_1 ($i = i_k$). Similarly, let s be the root of current subtree t_2 of T_2, l the number of subtrees in s, and j the last node of last subtree of t_2 ($j = j_l$). $D(r, i : s, j)$ denotes the structural distance between t_1 and t_2. Analytically, the algorithm proceeds as follows:

1. if $(r == i)$ and $(s == j)$ then $\mathcal{D} = 0$:
   If t_1 and t_2 consist only of one node each (their roots), then the cost to transform t_1 to t_2 is equal to 0 (roots are the same).

2. if $(s == j)$ then $\mathcal{D} = W_d(i_{k-1}+1) + D(r, i_{k-1} : s, j)$:
   If t_2 consists only of one node then the cost to transform t_1 to t_2 is equal to the cost to delete the k_{th} subtree of t_1 (which is the last subtree of the root of t_1), plus the cost to transform t_1' (which is t_1 without its k_{th} subtree) to t_2.

3. if $(r == i)$ then $\mathcal{D} = W_i(j_{l-1}+1) + D(r, i : s, j_{l-1})$:
   If t_1 consists only of one node then the cost to transform t_1 to t_2 is equal to the cost to insert the l_{th} subtree of t_2 (which is the last subtree of the root of t_2) in t_1 plus the cost to transform t_1 to t_2' (which is t_2 without its l_{th} subtree).

4. In any other case find the minimum between the following three costs, $\mathcal{D} = min(d_1, d_2, d_3)$:

   (a) $d_1 = W_d(i_{k-1}+1) + D(r, i_{k-1} : s, j)$:
   the cost to delete the k_{th} subtree of t_1 (which is the last subtree of the root of t_1) plus the cost to transform t_1' (which is t_1 without its k_{th} subtree) to t_2.

   (b) $d_2 = W_i(j_{l-1}+1) + D(r, i : s, j_{l-1})$:
   the cost to insert the l_{th} subtree of t_2 (which is the last subtree of the root of t_2) in t_1 plus the cost to transform t_1 to t_2' (which is t_2 without its l_{th} subtree).

   (c) $d_3 = D(r, i_{k-1} : s, j_{l-1}) + c_r(i_{k-1}+1, j_{l-1}+1) +$
   $D(i_{k-1}+1, i_k : j_{l-1}+1, j_l)$:
   the cost to transform t_1' (which is t_1 without its k_{th} subtree) to t_2' (which is t_2 without its l_{th} subtree) plus the cost to replace the root of the k_{th} subtree of t_1 with the root of the l_{th} subtree of t_2 plus the cost to transform the k_{th} subtree of t_1 to the l_{th} subtree of t_2.

The complete algorithm follows:

```
$D(r, i : s, j)$
begin
if  $((r == i)$  and  $(s == j))$  then  $\mathcal{D} = 0$  else
  if  $(s == j)$  then  $\mathcal{D} = W_d(i_{k-1}+1) + D(r, i_{k-1} : s, j)$  else
    if  $(r == i)$  then  $\mathcal{D} = W_i(j_{l-1}+1) + D(r, i : s, j_{l-1})$  else
```

```
$\mathcal{D} = Min\Big\{\{W_d(i_{k-1}+1) + D(r, i_{k-1} : s, j)\},$
                  $\{W_i(j_{l-1}+1) + D(r, i : s, j_{l-1})\},$
                  $\{D(r, i_{k-1} : s, j_{l-1}) + c_r(i_{k-1}+1, j_{l-1}+1) +$
                   $D(i_{k-1}+1, i_k : j_{l-1}+1, j_l)\}\Big\}$
return $\mathcal{D}$
end
```

The method should be called as $D(i_0, i_k : j_0, j_l)$, where i_0 the root of T_1, i_k the last node of the k_{th} subtree (the last one) of T_1, j_0 the root of T_2, j_l the last node of the l_{th} subtree (the last one) of T_2. T_1 and T_2 must have the same root. If not, one can create a new node and make it the root for both.

2.3.2 Zhang's algorithm

Zhang in [11] suggests a recursive algorithm to calculate the tree edit distance between two rooted ordered labeled trees, permitting tree edit operations anywhere in the trees. So, the set of permitted tree edit operations is $\{Ins(x, y, i), Del(y), Rep(x, y)\}$, with costs $c_i(x)$, $c_d(y)$, and $c_r(x, y)$, respectively (see Section 2.2).

A tree T is denoted as $T[i : j]$, where i is the label of its root and j is the label of its rightmost leaf. All nodes are labeled according to the postorder sequence. The subtree of T rooted at node i is denoted as $T[i]$. Finally, $t[i]$ refers to the node i of T, and $l[i]$ refers to the postorder number of the leftmost leaf of the subtree rooted at $t[i]$.

A *tree mapping* M on two trees T_1 and T_2 is an one-to-one relationship between nodes of T_1 and nodes of T_2. A mapping M includes a set of pairs (i, j). For any two pairs (i_1, j_1) and (i_2, j_2) in M the following statements hold:

1. $i_1 = i_2$ iff $j_1 = j_2$,

2. $t_1[i_1]$ is to the left of $t_1[i_2]$ iff $t_2[j_1]$ is to the left of $t_2[j_2]$,

3. $t_1[i_1]$ is an ancestor of $t_1[i_2]$ iff $t_2[j_1]$ is an ancestor of $t_2[j_2]$.

Every mapping M corresponds to a sequence of edit operations. Nodes in T_1 which are untouched by M correspond to $Del(y)$ operations in T_1. Nodes in T_2 which are untouched by M correspond to $Ins(x, y, i)$ operations in T_1. Nodes in T_1 related to nodes in T_2 by M correspond to $Rep(x, y)$ operations.

The algorithm calculates the minimum cost between the costs of the sequences of edit operations that transform a tree T_1 to the tree T_2, produced by all possible valid mappings on T_1 and T_2. Let $fd(T_1[i' : i], T_2[j' : j])$ be the distance between trees $T_1[i' : i]$ and $T_2[j' : j]$. Then:

1. $fd(0, 0) = 0$ (one-node trees)

2. $fd(T_1[l(i_1):i],0) = fd(T_1[l(i_1):i-1],0) + c_d(t_1[i])$

3. $fd(0,T_2[l(j_1):j]) = fd(0,T_2[l(j_1):j-1]) + c_i(t_2[j])$

4. $fd(T_1[l(i_1):i],T_2[l(j_1):j]) = min(d_1,d_2,d_3)$, where

   (a) $d_1 = fd(T_1[l(i_1):i-1],T_2[l(j_1):j]) + c_d(t_1[i])$

   (b) $d_2 = fd(T_1[l(i_1):i],T_2[l(j_1):j-1]) + c_i(t_2[j])$

   (c) $d_3 = fd(T_1[l(i_1):l(i)-1],T_2[l(j_1):l(j)-1]) + fd(T_1[l(i):i-1],T_2[l(j):j-1]) + c_r(t_1[i],t_2[j])$

where i and j are descendants of $t_1[i_1]$ and $t_2[j_1]$ respectively. Roots are labeled as 0. We note that the recursion is similar to the one in Selkow's algorithm presented in the previous section. However, deletions and insertions are permitted anywhere in the tree. The complete algorithm follows:

```
int CalculateDistance(TreeNode $i$, TreeNode $j$) {
    $fd(0,0) = 0$;
    for $i_1 = l(i)$ to $i$ do
        $fd(T_1[l(i):i_1],0) = fd(T_1[l(i):i_1-l],0) + c_d(t_1[i_1])$;
    for $j_1 = l(j)$ to $j$ do
        $fd(0,T_2[l(j):j_1]) = fd(0,T_2[l(j):j_1-1]) + c_i(t_2[j_1])$;
    for $i_1 = l(i)$ to $i$ do
        for $j_1 = l(j)$ to $j$ do
            if $l(i_1) = l(i)$ and $l(j_1) = l(j)$ then
                $fd(T_1[l(i):i_1],[T_2[l(j):j_1]) = min\{fd(T_1[l(i_1):i-1],T_2[l(j_1):j]) + c_d(t_1[i])$,
                                                    $fd(T_1[l(i_1):i],T_2[l(j_1):j-1]) + c_i(t_2[j])$,
                                                    $fd(T_1[l(i_1):i-1],T_2[l(j_1):j-1])+$
                                                    $c_r(t_1[i],t_2[j])\}$;
                $D[i_1][j_1] = fd(T_1[l(i):i_1],[T_2[l(j):j_1])$;
            else
                $fd(T_1[l(i):i_1],T_2[l(j):j_1]) = min\{fd(T_1[l(i):i_1-1],T_2[l(j):j_1]) + c_d(t_1[i_1])$,
                                                    $fd(T_1[l(i):i_1],T_2[l(j):j_1-1]) + c_i(t_2[j_1])$,
                                                    $fd(T_1[l(i):l(i_1)-1],T_2[l(j):l(j_1)-1])+$
                                                    CalculateDistance($i_1$,$j_1$)$\}$;
    Return $D[M][N]$;
}
```

At the end, the algorithm returns $D[M][N]$ as the tree edit distance for T_1 and T_2.

2.3.3 Chawathe's algorithm (I)

Chawathe in [12] suggests a recursive algorithm to calculate the tree edit distance between two rooted ordered labeled trees, using a predefined set of matching nodes between the trees. An *insert node* operation is permitted only if the new node becomes a leaf. A *delete node* operation is permitted only at leaf nodes. Any node can be updated using the *replace node* operation. A *move subtree* operation is also available. So, the set of permitted tree edit operations is $\{Ins^l(x,y,i), Del^l(y), Rep(x,y), Mov(x,y,k)\}$, with costs $c_i(x)$, $c_d(y)$, $c_r(x,y)$, and $c_m(x)$, respectively (see Section 2.2).

Let T_1 and T_2 be rooted ordered labeled-valued trees. A *partial matching* is a correspondence between nodes that have identical or similar values. The algorithm finds the edit script that transforms T_1 to T_2 with the minimum number of tree edit operations, and calculates the minimum cost to transform T_1 to T_2 using the unit costs for the operations. The algorithm proceeds in five phases:

1. *insert*: let r_1 and r_2 be the roots of T_1 and T_2 respectively. If $(r_1, r_2) \notin M$, then create new roots r'_1 and r'_2 for both and assume that $(r'_1, r'_2) \in M$. Then, insert all unmatched nodes z (i.e. nodes which do not take part in a partial matching) of T_2 which have their parent matched (does take part in a partial matching) in T_1.

2. *replace*: look for node pairs $(T_1.x, T_2.y) \in M$ such that their labels differ and replace every x with the corresponding y.

3. *move*: look for node pairs $(T_1.x, T_2.y) \in M$ such that their parents $(T_1.p(x), T_2.p(y)) \notin M$. In that case move the subtree rooted at x in T_1 to node u in T_1, where u is the matching node of $T_2.p(y)$.

4. *align*: The children u, v of node $T_1.x$ and u', v' of node $T_2.y$ are *misaligned* if $(u, u') \in M$, $(v, v') \in M$ and while u is to the left of v in T_1, u' is to the right of v' in T_2. Move operations are necessary to align the children.

5. *delete*: look for unmatched nodes in T_1 and delete them.

The complete algorithm that finds the minimum number of tree edit operations to transform T_1 to T_2 follows (M: initial partial matching):

```
$E \leftarrow \epsilon, M' \leftarrow M$
while traversing the nodes of $T_2$ in breadth-first order do
{
    $x$ is the current node in $T_2$, $y = parent(x)$ in $T_2$
    find $z$ in $T_1$ where $z$ matches with $y$
    if $x$ does not have a matching node in $T_1$:
    {
        $k \leftarrow FindPosition(x)$
        $k \leftarrow$ apply $Ins(w, z, k)$ to $T_1$
        /* $w$:  a new node */
    }
    else if $x$ does have a matching node $w$ in $T_1$:
        {
            if $label(w) \neq label(x)$ then apply $Rep(w,x)$ to $T_1$
            $v = parent(w)$ in $T_1$, $y = parent(x)$ in $T_2$
            if $(y,v) \notin M'$ then:
                find $z$ in $T_1$ where $(z,y) \in M'$
                $k \leftarrow FindPosition(x)$
                apply $Mov(w,z,k)$ to $T_1$
        }
        align children of $w$ and $x$ /* alignment problem */
    }
Delete all nodes in $T_1$ which do not have a matching node in $T_2$


$FindPosition(x)$
{
    If $x$ is the leftmost child of $y$ then return 1
    else return $i+1$, where $i$ is the number assigned
        to node $u$ in $T_1$ (1 for the leftmost),
        the matching node of $v$ in $T_2$ which is
        the rightmost sibling of $x$ that is to the left of $x$
}
```

The author treats the alignment problem as the *longest common subsequence (LCS) problem*: having two sequences S_1 and S_2, the LCS of S_1 and S_2 is a sequence S of pairs $(x_1, y_1), \ldots (x_k, y_k)$ such that:

1. $x_1 \ldots x_k$ and $y_1 \ldots y_k$ are subsequences of S_1 and S_2 respectively,
2. $equal(x_i, y_i)$ for some predefined equality function $equal$, $1 \leq i \leq k$.
3. S is the longest possible sequence that satisfies the above conditions.

Myers' algorithm [14] is used to compute the LCS S of the matched children of nodes x and y using the function $equal(u,v)$ that is true if $(u,v) \in M$. Then, having the children of x in S fixed, the matched children of y are moved in order to be aligned.

Having the minimum number of tree edit operations to transform T_1 to T_2, one can calculate the minimum cost $\mathcal{D}$ to transform T_1 to T_2 using the unit costs for the operations.

2.3.4 Chawathe's algorithm (*II*)

Chawathe in [13] 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. An *insert node* operation is permitted only if the new node becomes a leaf. A *delete node* operation is permitted only at leaf nodes. Any node can be updated using the *replace node* operation. So, the set of permitted tree edit operations is $\{Ins^l(x, y, i),\ Del^l(y),\ Rep(x, y)\}$, with costs $c_i(x)$, $c_d(y)$ and $c_r(x, y)$, respectively (see Section 2.2).

Let T_1 and T_2 be two rooted ordered labeled trees with M and N nodes respectively. Edit scripts on such trees can be represented using *edit graphs.* The edit graph of T_1 and T_2 is an $(M + 1) \times (N + 1)$ grid of nodes, having a node at each (x, y) location, $x \in [0 \ldots (M + 1)]$ and $y \in [0 \ldots (N+1)]$. Directed lines connect the nodes. A horizontal line $((x-1, y), (x, y))$ denotes deletion of $T_1[x]$, where $T_1[x]$ refers to the xth node of T_1 in its preorder sequence. Horizontal lines can be drawn only if node $T_2[y]$ is deeper than node $T_1[x]$. A vertical line $((x, y-1), (x, y))$ denotes insertion of $T_2[y]$, where $T_2[x]$ refers to the xth node of T_2 in its preorder sequence. Vertical lines can be drawn only if node $T_1[x]$ is deeper than node $T_2[y]$. Finally, a diagonal line $((x - 1, y - 1), (x, y))$ denotes update of $T_1[x]$ by $T_2[y]$. Diagonal lines can be drawn only if nodes $T_1[x]$ and $T_2[y]$ have the same depth in trees T_1 and T_2 respectively.

Every line has a weight equal to the cost of the corresponding edit operation. Line drawing follows certain constraints Drawing a horizontal line to denote deletion of a node M leads to drawing more lines to denote the deletion of all nodes in M's subtree. Drawing a vertical line to denote insertion of a node N leads to drawing more lines to denote the insertion of all nodes in N's subtree. Figure 5 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 in an

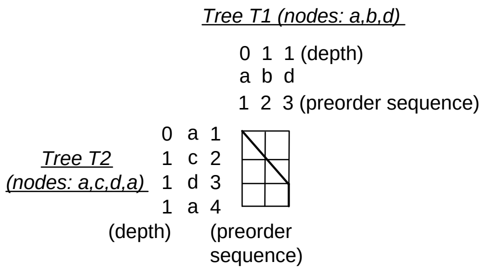


Figure 5: An example of an edit graph.

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

An edit graph G is constructed as a $(M + 1) \times (N + 1) \times 3$ matrix, whose cells contain the cost of the corresponding edit operation. The third dimension is used to determine the direction of the line drawing, that is the type of the operation, for example [0] for horizontal lines, i.e. *delete node*, [1] for vertical lines, i.e. *insert node node*, and [2] for diagonal lines, i.e. *replace node.* If a line is missing from the edit graph, the corresponding cell contains the infinite value. For example $G[4][6][0] = \infty$ means that there is no horizontal line from node 4 to node 6 in the edit graph.

Consider any path that connects the node $(0,0)$ to node n (x,y) in an edit graph. Node n (x,y) is the last node in the path. The distance D of n from $(0,0)$ cannot be greater than that distance of its left node plus the cost of the line connecting that node to n. Similarly, D can neither be greater than that distance of n's top node plus the cost of the line connecting that node to n, nor greater than that distance of n's diagonal node plus the cost of the line connecting that node to n. Based on the above remarks, the following recurrence calculates the shortest path $D[x,y]$ from $(0,0)$ to (x,y) in an edit graph G:

$$D(x,y) = min(m_1, m_2, m_3)$$

where

- $m_1 = D[x-1,y-1] + c_r(T_1[x], T_2[y])$, if $((x-1,y-1),(x,y)) \in G$ (the distance of (x,y)'s diagonal node in G plus the cost to replace $T_1[x]$ with $T_2[y]$), or ∞ otherwise,
- $m_2 = D[x-1,y] + c_d(T_1[x])$, if $((x-1,y),(x,y)) \in G$ (the distance of (x,y)'s left node in G plus the cost to delete $T_1[x]$), or ∞ otherwise,
- $m_3 = D[x,y-1] + c_i(T_2[y])$, if $((x,y-1),(x,y)) \in G$ (the distance of (x,y)'s top node in G plus the cost to insert $T_2[y]$), or ∞ otherwise.

The complete algorithm follows:

```
$D[0,0] = 0$;
for (i=1; i<=M; i++) do
  $D[i,0] = D[i-1,0] + c_d(T_1[i])$;
for (j=1; j<=N; j++) do
  $D[0,j] = D[0,j-1] + c_i(T_2[j])$;
for (i=1; i<=M; i++) do
  for (j=1; j<=N; j++) do
  {
    $m_1 = m_2 = m_3 = \infty$;
    if ($T_1[i].depth = T_2[j].depth$)
        then $m_1 = D[i-1,j-1] + c_r(T_1[i], T_2[j])$;
    if (($T_1[i].depth \geq T_2[j+1].depth$) or ($j = N$))
        then $m_2 = D[i-1,j] + c_d(T_1[i])$;
    if (($T_2[j].depth \geq T_1[i+1].depth$) or ($i = M$))
        then $m_3 = D[i,j-1] + c_i(T_2[j])$;
    $D[i,j]$ =minimum($m_1, m_2, m_3$);
  }
```

In the algorithm, $D[i,j]$ keeps the tree edit distance between tree T_1 with only its i nodes, assuming pre-order traversal, and tree T_2 with only its its j nodes assuming pre-order traversal. For example $D[3,0]$ keeps the tree edit distance between tree T_1 with only its first 3 nodes (pre-order) and tree T_2 with only its root and $D[0,4]$ keeps the distance between T_1 with only its root and T_2 with only its first 4 nodes (pre-order). $D[0,0]$ keeps the distance between T_1 and T_2, having only their roots (initially 0, since the examined trees are assume to have same roots). The costs c_i, c_d and c_r are taken from the corresponding edit graph matrix. The tree edit distance $\mathcal{D}$ for T_1 and T_2 is $D[M,N]$.

2.3.5 Discussion

All of the algorithms for calculating the edit distance for two ordered labeled trees are based on dynamic programming techniques related to the string-to-string correction problem [15]. The first work that defined the tree edit distance and provided algorithms to compute it, permitting operations anywhere in the tree, was [16]. Selkow's [10] and Chawathe's (II) [13] algorithms allow insertion and deletion only at leaf nodes, and relabel at every node. The latter algorithm is based on the model of edit graphs which reduces the number of recurrences needed, compared to the former. However, this work (Chawathe's (II)) is focused on the utilization of external memory to perform such calculations. Chawathe's (I) algorithm [12] starts using a pre-defined set of matching nodes between the trees, and is based on a different set of tree edit operations than Chawathe's (II). It allows insertion and deletion only at leaf nodes. Zhang's algorithm [11] permits operations anywhere in the tree and uses the a similar recurrence with Selkow's [10] algorithm. 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. Table 1 summarizes the results. In this work, we consider Chawathe's (II) algorithm as the basic point of reference for tree edit distance algorithms, since it permits insertion and deletion only at leaves and is the fastest available.

Algorithm	**Operations**	**Restricted to leaves**	**Complexity**
Selkow's	insert node, delete node, replace node	insert node, delete node	exponential: $4^{min(NM)}$, M and N are the numbers of nodes for each tree
Zhang's	insert node, delete node, replace node		$O(MNbd)$, M and N are the numbers of nodes for each tree, and b and d are the depths of the two trees, respectively
Chawathe's (I)	insert node, delete node, replace node, move subtree	insert node, delete node	$O(ND)$, N the number of nodes in both trees and D the number of misaligned nodes
Chawathe's (II)	insert node, delete node, replace node	insert node, delete node	$O(MN)$, M and N are the dimensions of the matrix that represents the edit graph

Table 1: Tree edit distance algorithms

In the following sections, we analyze our framework for clustering XML documents by structure. We start discussing how to maintain the structural information present in XML documents using compact trees, called structural summaries, instead of the original trees representing the XML documents. Structural summaries have minimal processing requirements compared to original trees. Then, we propose a new algorithm to calculate tree edit distances and we define a structural distance metric to estimate the structural similarity between structural summaries of two rooted ordered labeled trees. The suggested distance is used in a clustering task to identify groups of XML documents that share a similar structure.

3 Tree structural summaries

Real XML documents tend to have many repeated elements. As a result, the trees representing XML documents (see Section 2.1) can be large and deeply nested, and may have quite different size and structure even if they are based on the same DTD. This affects the performance of the tree edit algorithms and makes them slow and sometimes inaccurate: a tree edit algorithm will output a large distance between two XML documents which are based on the same DTD, with one of the two being quite long due to many repeated elements. We detect such kind of redundancy looking for nested-repeated and repeated nodes in XML documents.

- 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 6 presents examples of redundancy. Trees A_1 and A_2 differ because of the nesting of node R (nested-repeated node), but they share `DTD-1`. Trees B_1 and B_2 differ because of the repeated node C, but they share `DTD-2`.

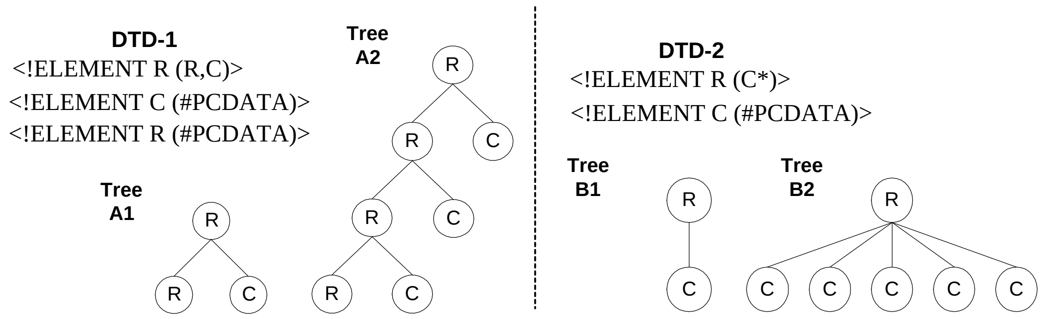


Figure 6: Element repetition and nesting

We perform (a) nesting reduction and (b) repetition reduction to extract *structural summaries* for rooted ordered labeled trees which represent XML documents. Structural summaries maintain the structural relationships between the elements of an XML document and at the same time have minimal processing requirements to extract and use instead of the original XML documents in the clustering procedure. Structural summaries resemble the *dataguide* summaries [17]. However, a dataguide is a summary of the structure of semistructured data described by the OEM model, while structural summaries are based on the XML data model (see Section 2.1). The next sections show how we exploit nesting reduction and repetition reduction to construct structural summaries.

3.1 Nesting reduction

The aim of this phase is to reduce the nesting in the original tree so that there will be no nested-repeated nodes. We traverse the tree using pre-order traversal. For the current node, we check if there is an ancestor with the same label. If there is no such ancestor, we go on to the next node. If there is such ancestor, then we move all current node's subtrees to that ancestor. We add the subtrees at the end of the ancestor's child list so that we will traverse these nodes later. Nothing will be moved if the current node is a leaf. This process may cause non-repeated nodes to become repeated ones. This is why we deal first with the nesting reduction and then with the repetition reduction. Nesting reduction requests only a pre-order traversal on the original tree. The algorithm follows:

```
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--;
        }
     }
}
```

3.2 Repetition reduction

The aim of this phase is to reduce the repeated nodes in the original tree. We traverse the tree using pre-order traversal. At each node, we check whether the path from the root to the node already exists or not by looking up in a hash table keeping the paths. If there is no such a path, we store this node in the hash table, with its path being the index. If there is already one such path in the hash table, then this node is a repeated node, and in that case:

1. we move all its subtrees to the destination node we find in the hash table by using the path as index,
2. we add the subtrees at the end of the destination node's child list so that we will also traverse these subtrees later, and
3. we delete the current node and start to traverse the subtrees which have been moved to the destination node.

After traversing all the nodes that have been moved, we go on to traverse the right sibling of the node which is deleted. If there is no such node the traversal ends. Repetition reduction requests only a pre-order traversal on the original tree. The algorithm follows:

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

Figure 7 illustrates an example of structural summary extraction. 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 tree without nested/repeated and repeated nodes.

Once trees have been compacted using structural summaries, so that nesting and repetition are reduced, structural distances can be computed. We next describe our method for computing such distances.

4 Tree structural distance

Our algorithm for calculating the tree edit distance between structural summaries of rooted ordered labeled trees that represent XML documents uses a dynamic programming algorithm

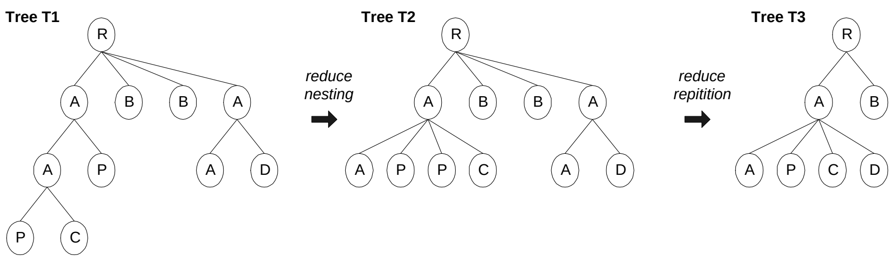


Figure 7: Structural summary extraction

which is close to Chawathe's algorithm (*II*) [13] 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 the timing analysis in Section 6.4). A similar recurrence but for a different set of tree edit operations has been used in [18] (see Section 7).

An *insert node* operation is permitted only if the new node becomes a leaf. A *delete node* operation is permitted only at leaf nodes. Any node can be updated using the *replace node* operation. So, the set of permitted tree edit operations for our approach is $\{Ins^l(x,y,i)$, $Del^l(y)$, $Rep(x,y)\}$, with costs $c_i(x) = 1$, $c_d(y) = 1$, and $c_r(x,y) = 1$ if the node to be replaced has different label ($c_r(x,y) = 0$ otherwise), respectively (see Section 2.2). The cost $W_i(x)$ to insert a whole subtree t_2, rooted at node x, anywhere in a tree t_1, is actually the number of nodes in t_2:

$$W_i(x) = \sum_{j=0}^{k} c_i(x_j) = k+1 \tag{3}$$

where $x_0 = x$ and $x_1 \dots x_k$ are all descendants of x. The cost $W_d(y)$ to delete a whole subtree t_2, rooted at node y, anywhere in a tree t_1, is actually the number of nodes in t_2:

$$W_d(y) = \sum_{j=0}^{k} c_d(y_j) = k+1 \tag{4}$$

where $y_0 = y$ and $y_1 \dots y_k$ are all descendants of y.

Given T_1 and T_2 with roots r_1 and r_2 respectively, the following method calculates their tree edit distance ($CalculateDistance(r_1, r_2)$):

```
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] = D[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][j-1] + numOfNodes($t_j$),
                          D[i-1][j] + numOfNodes($s_i$),
                          D[i-1][j-1] + CalculateDistance($s_i$,$t_j$));
    Return D[numOfChildren(s)][numOfChildren(t)];
}
```

where:

1. s_i is the i_{th} child of node s and t_j is the j_{th} child of node t.

2. $numOfChildren(s)$ returns the number of child nodes of node s.

3. $numOfNodes(s)$ returns the number of nodes of the subtree rooted at s (including s).

4. $LabelOf(s)$ returns the label of node s.

5. $UpdateCost(LabelOf(s), LabelOf(t))$ returns the cost c_r to make the label of node s the same as the label of node t: 1 if $LabelOf(s) = LabelOf(t)$ or 0 otherwise.

We call the function `CalculateDistance` once for each pair of nodes s and t at the same depth in the two structural summary trees. $D[i][j]$ keeps the tree edit distance between tree rooted at s with only its first i subtrees and tree rooted at t with only its first j subtrees. $D[0][0]$ keeps the distance between tree rooted at s and tree rooted at t, both having only their roots. The main `for` nested loop first calculates the tree edit distance between tree rooted at s with only its first subtree and tree rooted at t with only its first subtree, and then proceeds by adding more subtrees to the explored trees. At the end, the algorithm returns the distance between tree rooted at $s = r_1$ (the root of T_1) with all its subtrees and tree rooted at $t = r_2$ (the root of T_2) with all its subtrees. Since `CalculateDistance` is called once for each pair of nodes at the same depth in the two structural summary trees, 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.

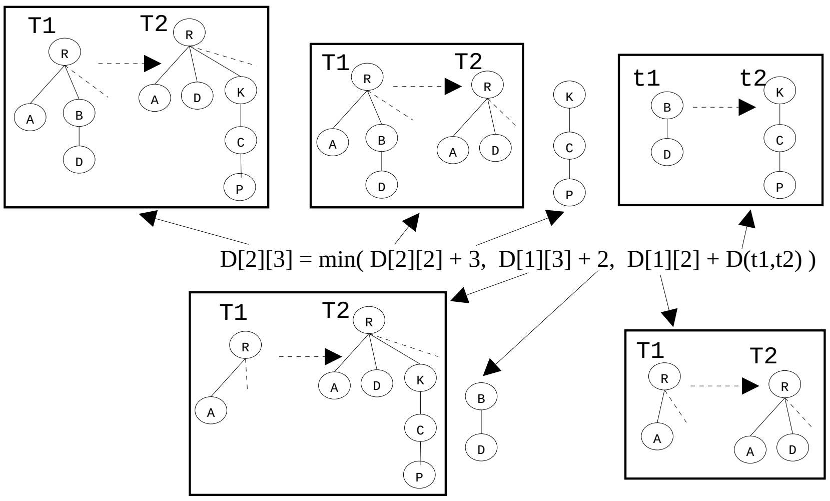
We next describe in detail how the algorithm computes the minimum distance between s and t:

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 8 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.



T1
T2
t1
t2
R
A
B
D
K
C
P
D[2][3] = min(D[2][2] + 3, D[1][3] + 2, D[1][2] + D(t1,t2))

Figure 8: Calculating $D[2][3]$ for T_1 and T_2.

A trace of the algorithm using trees T_1 and T_2 in Figure 9 is presented in Table 2. In the step where subtrees t_1 and t_2, rooted at B of T_1 and K of T_2, respectively, are explored (example 1 in Figure 9), we note the following calculations (all operations are applied in t_1):

1. $D[0][0] = 1$: the roots of t_1 and t_2 are different, so the algorithm spends $c_r = 1$ to replace B in t_1 with K.

2. $D[0][1] = 3$: $D[0][1]$ keeps the distance between t_1 with only its root B and t_2 with only its first subtree (the path $K/C/P$). Having only the root node B from t_1, the algorithm spends $c_r = 1$ to replace B with K, $c_i = 1$ to insert node C under K and $c_i = 1$ to insert node P under C, getting $K/C/P$: a cost of 3 units.

3. $D[1][0] = 2$: $D[1][0]$ keeps the distance between t_1 with only its first subtree (the path B/D) and t_2 with only its root K. The algorithm spends $c_d = 1$ to delete D and $c_r = 1$ to replace B with K, getting K: a cost of 2 units.

4. $D[1][1] = 3$: $D[1][1]$ keeps the distance between t_1 with only its first subtree (the path B/D) and t_2 with only its first subtree (the path $K/C/P$). The algorithm spends $c_r = 1$ to replace B with K, $c_r = 1$ to replace D with C and $c_i = 1$ to insert P under C, getting $K/C/P$: a cost of 3 units.

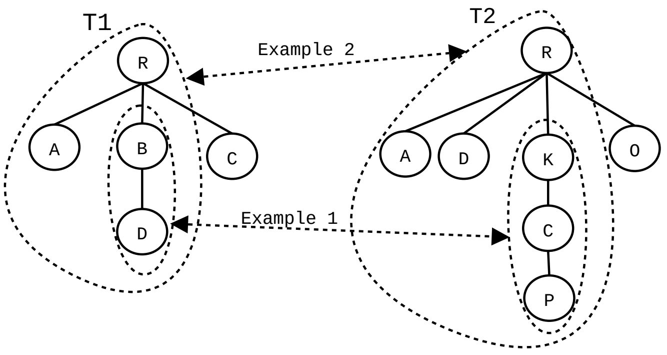


Figure 9: An example of tree distance calculation (see also Table 2).

subtrees t_1, t_2	**D[i,j]**
Root of $t_1 = C$, Root of $t_2 = K$	D[0][0]=1, D[0][1]=3
Root of $t_1 = B$, Root of $t_2 = K$	D[0][0]=1, D[0][1]=3, D[1][0]=2, D[1][1]=3
Root of $t_1 = R$, Root of $t_2 = R$	D[0][0]=0, D[0][1]=1, D[0][2]=2, D[0][3]=5, D[0][4]=6, D[1][0]=1, D[1][1]=0, D[1][2]=1, D[1][3]=4, D[1][4]=5, D[2][0]=3, D[2][1]=2, D[2][2]=2, D[2][3]=4, D[2][4]=5, D[3][0]=4, D[3][1]=3, D[3][2]=3, D[3][3]=5, D[3][4]=5 ($Distance = 5$, $Total\ cost = 0.417$)

Table 2: A trace of the algorithm running for trees T_1 and T_2 in Figure 9.

We now look into the step where subtrees t_1 and t_2 are rooted at R of T_1 and R of T_2 (example 2 in Figure 9), that is $t_1 = T_1$ and $t_2 = T_2$ (all operations are applied in t_1):

1. $D[2][3] = 4$: $D[2][3]$ keeps the distance between t_1 with only its first 2 subtrees and t_2 with only its first 3 subtrees. The algorithm spends $c_r = 1$ to replace B with K, $c_r = 1$ to replace D with C, $c_i = 1$ to insert P under C and $c_i = 1$ to insert D under R: a cost of 4 units.

2. $D[3][4] = 5$: $D[3][4]$ keeps the distance between t_1 with its first 3 subtrees and t_2 with its first 4 subtrees. Actually, this is the distance between T_1 and T_2. The algorithm spends $c_r = 1$ to replace B with K, $c_r = 1$ to replace D with C, $c_i = 1$ to insert P under C, $c_i = 1$ to insert D under R and $c_r = 1$ to replace C with O: a cost of 5 units.

Figure 10 presents the sequence of tree edit operations to transform T_1 to T_2 with minimum cost (see also Figure 9).

We can now define the structural distance $\mathcal{S}$ between two structural summaries for rooted ordered labeled trees which represent XML documents.

Definition 3 *Let T_1 and T_2 be two structural summaries for rooted ordered labeled trees that represent two XML documents, $\mathcal{D}(T_1, T_2)$ be their tree edit distance and $\mathcal{D}_{max}(T_1, T_2)$ be the*

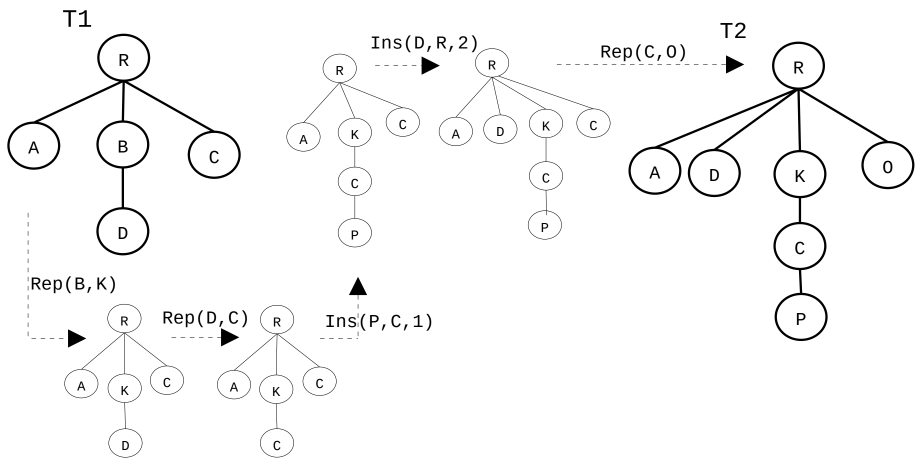


Figure 10: The sequence of tree edit operations to transform T_1 to T_2 with minimum cost (see also Figure 9.

maximum cost between the costs of all possible sequences of tree edit operations that transform T_1 to T_2. The structural distance $\mathcal{S}$ between T_1 to T_2 is defined as $\mathcal{S}(T_1, T_2) = \frac{\mathcal{D}(T_1,T_2)}{\mathcal{D}_{max}(T_1,T_2)}$.

To calculate $\mathcal{D}_{max}(T_1, T_2)$, we calculate the cost to delete all nodes from T_1 and insert all nodes from T_2. The $\mathcal{S}(T_1, T_2)$ value is

1. 0 when the trees have exactly the same structure and the same labels in their matching nodes,
2. 1 when the trees have totally different structure and not even two pairs of matching nodes with the same ancestor/descendant relationship,
3. low when the trees have similar structure and high percentage of matching nodes and
4. high when the trees have different structure and low percentage of matching nodes.

In the example illustrated in Figure 9 and Table 2, $\mathcal{D}_{max}(T_1, T_2) = 12$, 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.

5 Clustering XML documents

We deal with the problem of clustering XML documents using

1. structural summaries of their representative rooted ordered labeled trees,
2. tree edit distances between these summaries,
3. structural distances calculated from these tree edit distances, and

4. clustering algorithms, well-known from text information retrieval, that use pairwise structural distances to detect groups of data.

Figure 11 presents our framework.

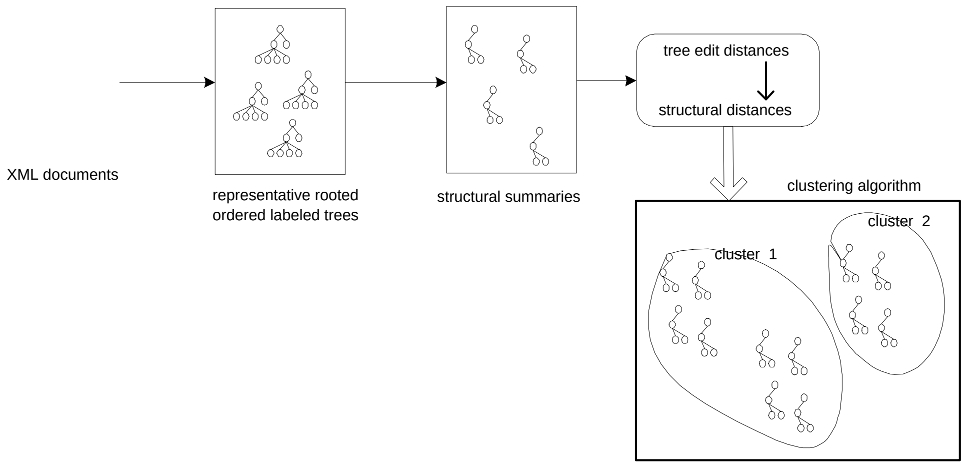


Figure 11: Clustering XML documents

5.1 Clustering algorithms

Clustering methods are usually divided into two broad categories. *Non-hierarchical methods* group a data set into a number of clusters. *Hierarchical methods* produce nested sets of data (hierarchies), in which pairs of elements or clusters are successively linked until every element in the data set becomes connected. Non-hierarchical methods have low computational requirements, ($O(kn)$, if for example n documents need to be grouped into k clusters), but certain parameters like the number of formed clusters must be known a priori. Hierarchical methods are computationally expensive, with time requirements of $O(n^2)$, if n documents need to be clustered. However, hierarchical methods have been used extensively as a means of increasing the effectiveness and efficiency of retrieval [19, 20, 21]. For a wide ranging overview of clustering methods one can refer to [22, 23]. *Single link*, *complete link* and *group average link* are known as hierarchical clustering methods. All these methods are based on a similar idea:

1. Each element of the data set to be clustered is considered to be a single cluster.
2. The clusters with the minimum distance (i.e. maximum similarity) are merged and the distance between the remaining clusters and the new, merged one is recalculated.
3. While there are more than one clusters, go again to step 2.

In single link (complete link), the distance between two non-single clusters is defined as the minimum (maximum) of the distances between all pairs of elements so that one element is in the one cluster and the other element is in the other cluster. In group average link, the distance between two non-single clusters is defined as the mean of the distances between all pairs of elements so that one element is in the one cluster and the other element is in the other cluster. We chose single link to be the basic clustering algorithm for the core part of the experiments for our work since it has been shown to be theoretically sound, under a certain number of reasonable conditions [24].

5.1.1 Single link

We implemented a single link clustering algorithm using Prim's algorithm [25] for computing the *minimum spanning tree (MST)* of a graph. Given a graph G with a set of weighted edges E and a set of vertices V, a MST is an acyclic subset $T \subseteq E$ that links all the vertices and whose total weight $W(T)$ (the sum of the weights for the edges in T) is minimized. It has been shown [26] that a MST contains all the information needed in order to perform single link clustering.

Given n structural summaries of rooted labeled trees that represent XML documents, we form a fully connected graph G with n vertices $\in V$ and $n(n-1)/2$ weighted edges $\in E$. The weight of an edge corresponds to the structural distance between the vertices (trees) that this edge connects. The single link clusters for a *clustering level* l_1 can be identified by deleting all the edges with weight $w \geq l_1$ from the MST of G. The connected components of the remaining graph are the single link clusters. Figure 12(a) shows a graph with 7 nodes that correspond to 7 structural summaries, and 10 edges. The weight of an edge is the structural distance between the involved structural summaries. For example the structural distance between summaries 1 and 2 is 0.2. The missing edges, that is the extra edges that make the graph fully connected, are those that have weight 1. Figure 12(b) shows the minimum spanning tree of (a). Figure 12(c) presents the graph remaining after deleting all edges with weight ≥ 0.4. There are 2 connected components that include nodes $(1, 2, 3, 6)$ and nodes $(7, 5)$, respectively. This indicates the presence of 2 clusters: cluster 1 with $(1, 2, 3, 6)$ as members and cluster 2 with $(7, 5)$ as members. Nodes which are not connected to other nodes will be considered as single-node clusters.

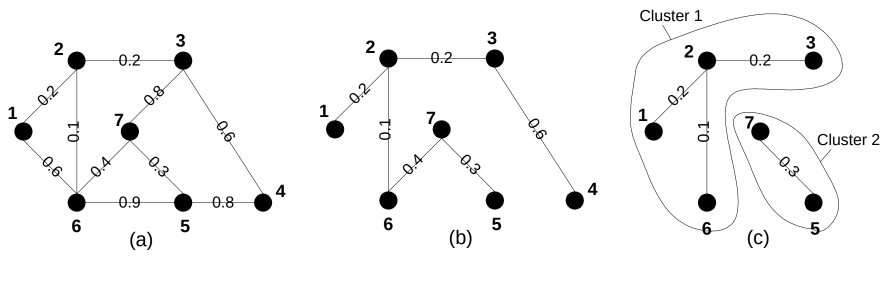


Figure 12: Minimum Spanning Tree (MST) detection and single link clustering at level 0.6

A stopping rule is necessary to determine the most appropriate clustering level for the single link hierarchies. Milligan et al. present 30 such rules [27]. Among these rules, $C-$index [28] exhibits excellent performance (found in the top 3 stopping rules). We next present the way we adopt the $C-$index in a hierarchical clustering procedure.

5.1.2 $C-$index for hierarchical clustering

$C-$index is a vector of pairs $((i_1, n_1), (i_2, n_2), \ldots, (i_p, n_p)...$, where $i_1, i_2, \ldots, i_p$ are the values of the index and $n_1, n_2, \ldots, n_p$ the number of clusters in each clustering arrangement produced by varying the clustering level of a hierarchical clustering procedure in p different steps. Let l_1 be the first selected clustering level, which produces an arrangement of N_1 clusters (that is $n_1 = N_1$): C_1 with c_1 elements, C_2 with c_2 elements, $\ldots$, C_{N_1} with c_{N_1} elements. We can calculate i_1 in order to have the first pair (i_1, n_1) of C-index vector:

$$i_1 = (d_w - min(d_w))/(max(d_w) - min(d_w))$$

where:

1. $d_w = Sum(d_{w1}) + Sum(d_{w2}) + \ldots + Sum(d_{w_{N_1}})$, with $Sum(d_{w_i})$ to be the sum of pairwise distances of all members of cluster $C_i, 1 \leq i \leq n_1$,

2. $max(d_w)$: the sum of the n_d highest pairwise distances in the whole set of data (that is, sort distances, highest first, and take the Top-n_d sum),

3. $min(d_w)$: the sum of the n_d lowest pairwise distances in the whole set of data (that is, sort distances, highest first, and take the Bottom-n_d sum),

with $n_d = c_1 * (c_1 - 1)/2 + c_2 * (c_2 - 1)/2 + \ldots + c_{N_1} * (c_{N_1} - 1)/2$ (that is the number of all within cluster pairwise distances). Similarly we calculate all values of $C-$index for all different p clustering levels, getting the vector $((i_1, n_1), (i_2, n_2), \ldots, (i_p, n_p))$. We point out that:

- Although all pairwise structural distances are needed to compute the C-Index, this doesn't require any additional computation because these distances need to be computed anyway for the hierarchical clustering procedure itself.

- Since multiple successive clustering levels can generate the same number of clusters, we compute the $C-$Index not for each level but for each number of clusters generated by different levels.

- The number of clusters with the lowest $C-$Index is chosen as the correct clustering, as [27] suggests.

6 Experimental evaluation

We have developed a prototype and performed extended evaluation of our framework for clustering XML documents. We tested the performance as well as the quality of the clustering results using synthetic and real data. All the experiments were performed on a PC, Pentium III 800MHz, 192MB RAM.

6.1 Architecture

The prototype testbed is a java-based software that can

- generate synthetic XML documents or use existing ones
- extract structural summaries from XML documents,
- calculate pairwise structural distances between these summaries,
- perform single-link clustering as well as utilize clustering algorithms provided by other software packages,
- perform k-NN classification, using already discovered clusters,
- calculate evaluation metrics to judge the performance and the quality of the clustering results.

Figure 13 presents its architecture.

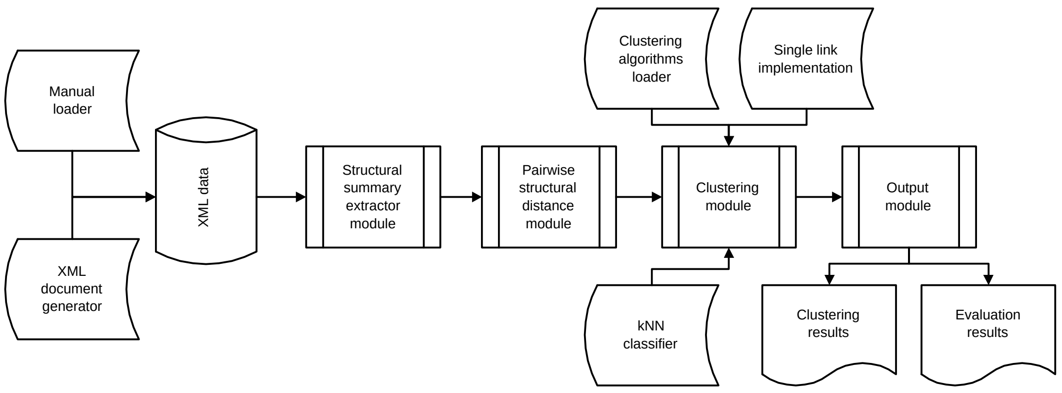

Manual loader
XML document generator
XML data
Structural summary extractor module
Pairwise structural distance module
Clustering algorithms loader
Single link implementation
Clustering module
kNN classifier
Output module
Clustering results
Evaluation results

Figure 13: Evaluation testbed

6.2 Data sets and clustering algorithms

Experiments were performed on both synthetic and real data, where appropriate. For real data set we used documents from the ACM SIGMOD Record and ADC/NASA[5]:

[5]www.acm.org/sigmod/record/xml and xml.gsfc.nasa.gov respectively

- **ACM SIGMOD Record:** 70 XML documents were sampled from IndexTermsPage.dtd and OrdinaryIssuePage.dtd that are used for ACM SIGMOD Record.
- **ADC/NASA:** 70 documents from both catalogs and journals, all conforming to the Astronomical Dataset Markup Language DTD.

Synthetic XML documents were generated in our prototype using IBM's AlphaWorks XML generator[6]. We used 10 real-case DTDs[7] and 2 sets of 1000 XML documents, generated from these DTDs. Both datasets were generated by varying the parameter *MaxRepeats* that determines the number of times a node will appear as a child of its parent node (when + or * is used in the DTD). The actual number of repeats generated is a random value between 0 and *MaxRepeats*. The first set of synthetic XML documents was generated with that parameter set to 3 and the second one was generated with that parameter set to 6. Parameter *numLevels* that determines the maximum number of tree levels was set to 7.

We chose single link to be the basic clustering algorithm for the core part of the experiments, providing our own implementation. However, preliminary results with other clustering algorithms are also presented, using libraries of CLUTO[8], a research tool for clustering datasets and analyzing the characteristics of the various clusters.

6.3 Evaluation procedure

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 method in the single-link clustering method (see Section 5.1.2).
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[9].
   (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[10].

[6]www.alphaworks.ibm.com/tech/xmlgenerator
[7]from www.xmlfiles.com and http://www.w3schools.com
[8]www-users.cs.umn.edu/˜karypis/cluto/
[9]using AlphaWorks Data Descriptors by Example: www.alphaworks.ibm.com/tech/DDbE
[10]DTD parser: www.wutka.com/dtdparser.html

(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 (see next paragraphs).

To evaluate the clustering results, we used two metrics quite popular in information retrieval: *precision* PR and *recall* R [24, 29, 30]. For an extracted cluster C_i that corresponds to a DTD D_i let:

1. a_i be the number of the XML documents in C_i that were indeed members of that cluster (correctly clustered),

2. b_i be the number XML documents in C_i that were not members of that cluster (misclustered),

3. c_i be the number of XML documents not in C_i, although they should be C_i's members.

Then:

$$PR = \frac{\sum_i a_i}{\sum_i a_i + \sum_i b_i}, \quad R = \frac{\sum_i a_i}{\sum_i a_i + \sum_i c_i} \tag{5}$$

High precision means high accuracy of the clustering task for each cluster while low recall means that there are many XML documents that were not in the appropriate cluster although they should have been. High precision and high recall indicate excellent clustering quality. In the case where there are clusters not mapped to any of the original DTD, PR and P will be affected, since all XML documents in such clusters will be treated as misclustered documents.

Based on the above, we present the timing analysis for calculating structural distances and then we evaluate the clustering results.

6.4 Efficiency of structural distance algorithms

We compared

1. the time to derive the 2 structural summaries from 2 rooted ordered labeled trees representing 2 XML documents $\underline{\text{plus}}$

2. the time to calculate the structural distance between those 2 summaries,

vs the time to calculate the structural distance between 2 rooted ordered labeled trees of 2 XML documents (without using structural summaries).

We compared Chawathe's algorithm and our algorithm using randomly generated XML documents, with their nodes ranging from 0 to 2000. This timing analysis gives an indication of how fast a file for storing pairwise structural distances is constructed. Such a file can then be used as an input in any clustering algorithm to discover clusters. Recall that a clustering algorithm needs to calculate $N * (N-1)/2$ pairwise structural distances, where N the number of documents to be clustered.

Figure 14 shows the % time decrease for calculating the structural distance between 2 XML documents using their summaries instead of using the original trees, for Chawathe's algorithm. Using summaries, the decrease lays around 76% on average, compared to the time needed without using summaries. At the point where we have around 50% time increase using summaries, the 2 trees have 1120 and 3 nodes respectively, and the time needed for calculating the structural distance is $38ms$ and $57ms$, without and with summaries, respectively. We note that for 2 trees, having around 1000 nodes each, the time needed on average for calculating the structural distance is $8000ms$ and $120ms$, without and with summaries, respectively.

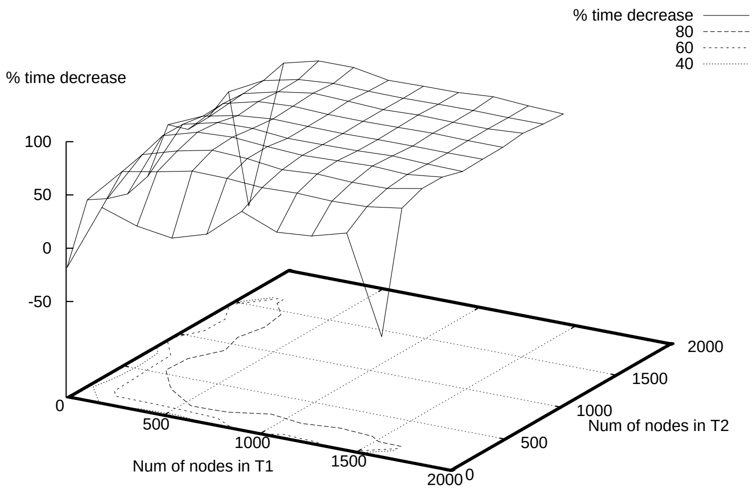


Figure 14: Calculating the structural distance between 2 XML documents using Chawathe's algorithm: % time decrease using their summaries instead of using the original trees.

Figure 15 shows similar results concerning the % time decrease for calculating the structural distance between 2 XML documents using their summaries instead of using the original trees, for our algorithm. Using summaries, the decrease lays around 51% on average, compared to the time needed without using summaries. At the point where we have around 280% time increase using summaries, the two trees have 1551 and 3 nodes respectively, and the time needed for calculating

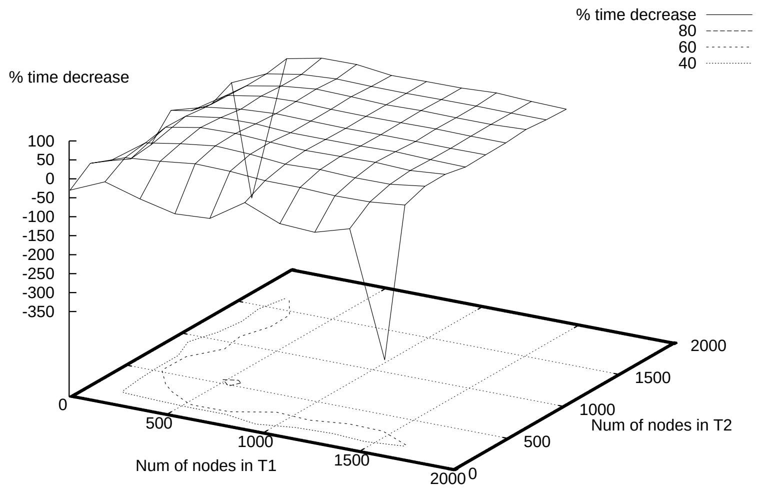


Figure 15: Calculating the structural distance between 2 XML docs using our algorithm: % time decrease using their summaries instead of using the original trees.

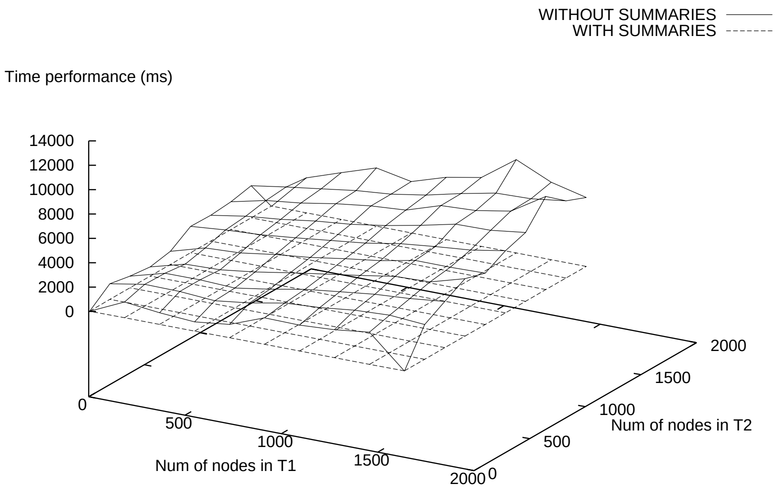


Figure 16: Calculating the structural distance between 2 trees using Chawathe's algorithm: time performance with or without summaries (ms).

the structural distance is $28ms$ and $108ms$, without and with summaries, respectively. We note that for 2 trees, having around 1000 nodes each, the time needed on average for calculating the structural distance is $1200ms$ and $100ms$, without and with summaries, respectively.

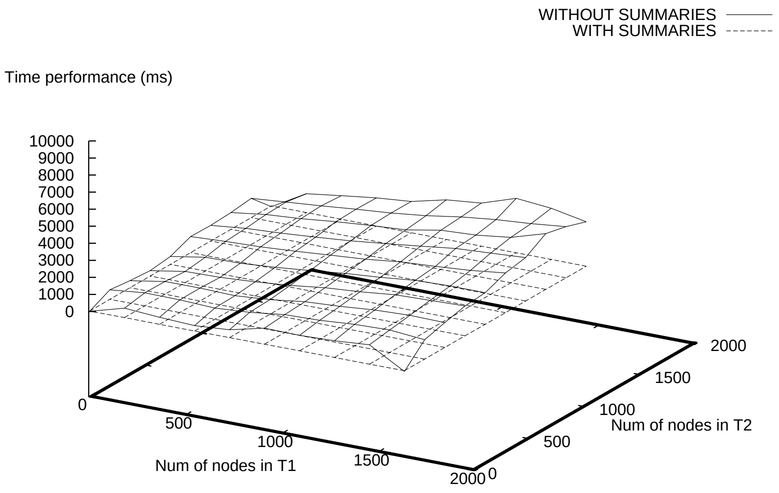


Figure 17: Calculating the structural distance between 2 trees using our algorithm: time performance with or without summaries (ms).

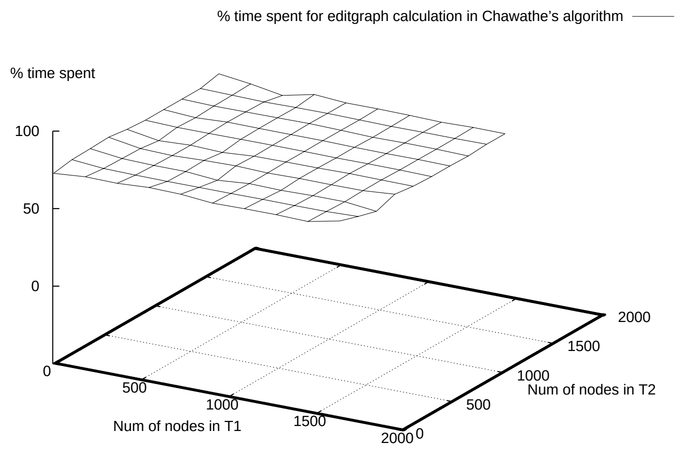


Figure 18: Calculating the editgraph for Chawathe's algorithm: % time spent out of the whole time needed for structural distance calculation.

To give a sense about the scaling of the calculations, Figures 16 and 17 present detailed analysis of the timing performance for both algorithms, with or without summaries. Notice that Chawathe's algorithm is significantly slower than our algorithm due to the pre-calculation of the editgraph, as Figure 18 shows. Editgraph calculation spends around 73% on average of the time needed for the overall distance calculation. Figure 19 presents the % time decrease for calculating the structural distance between 2 XML documents, using our algorithm instead of

Chawathe's algorithm. The decrease lays around 55% on average.

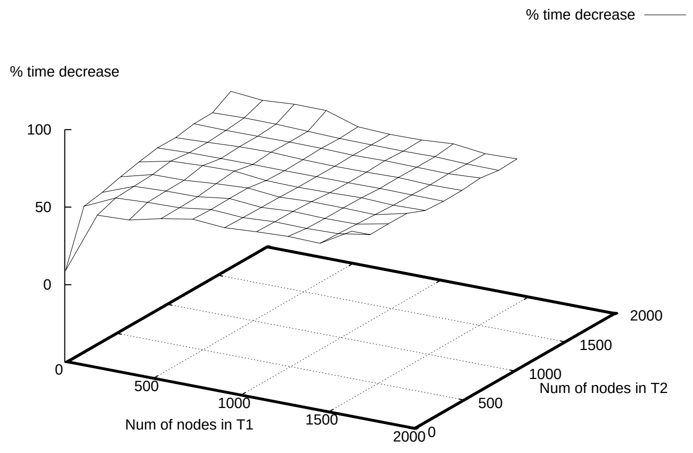


Figure 19: Calculating the structural distance between 2 trees using Chawathe's and our algorithm: % time decrease using our algorithm.

6.5 Clustering results

We performed single link clustering using Chawathe's algorithm and our algorithm on synthetic and real data, with or without structural summaries, and calculated PR and R values. In all presented Tables (3, 4, 5, 6) that follow, $NumOfClusters$ is the number of clusters discovered from the single link clustering task, using C-index. $Cluster.level$ is the level where the single link task was stopped, that is the level with low value of C-index (see Section 5.1). After the mapping of the discovered clusters to the original DTDs (see Section 6.3), clusters remain unmapped. For example, single link clustering discovered 11 clusters in the test case that Table 3 presents. The documents of the cluster which was not mapped to any of the 10 original DTDs were treated as misclustered documents, increasing the b value. See for example the b value for clusters 6 and 9 in Table 3.

6.5.1 Working on synthetic data

Tables 3 and 4 present the (a, b, c) values for PR and R calculation as well as the PR and P values themselves, using Chawathe's algorithm on synthetic data with $maxRepeats = 3$ and $maxRepeats = 6$. Notice 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 there is a clear improvement using summaries,

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1 (DTD 1)	60	0	40	1 (DTD 1)	60	0	40
2 (DTD 2)	62	0	38	2 (DTD 2)	62	0	38
3 (DTD 3)	80	0	20	3 (DTD 3)	80	0	20
4-5 (DTDs 4-5)	100	0	0	4-5 (DTDs 4-5)	100	0	0
6 (DTD 6)	100	185	0	6 (DTD 6)	100	185	0
7-8 (DTDs 7-8)	100	0	0	7-8 (DTDs 7-8)	100	0	0
9 (DTD 9)	100	185	0	9 (DTD 9)	100	185	0
10 (DTD 10)	100	0	0	10 (DTD 10)	100	0	0
$NumOfClusters = 11$, $Cluster.\ level = 0.37$				$NumOfClusters = 11$, $Cluster.\ level = 0.37$			
$PR = 0.71$, $R = 0.90$				$PR = 0.71$, $R = 0.90$			

Table 3: Chawathe's algorithm on synthetic data with $MaxRepeats = 3$.

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1 (DTD 1)	28	0	72	1-2 (DTDs 1-2)	87	0	13
2 (DTD 2)	87	0	13	3 (DTD 3)	84	0	16
3 (DTD 3)	84	0	16	4 (DTD 4)	100	100	0
4 (DTD 4)	100	200	0	5-7 (DTDs 5-7)	100	0	0
5 (DTD 5)	100	0	0	8 (DTD 8)	100	100	0
6 (DTD 6)	100	42	0	9 (DTD 9)	100	4	0
7 (DTD 7)	100	0	0	10 (DTD 10)	100	0	0
8-9 (DTDs 8-9)	100	200	0				
10 (DTD 10)	100	0	0				
$NumOfClusters = 11$, $Cluster.\ level = 0.51$				$NumOfClusters = 12$, $Cluster.\ level = 0.50$			
$PR = 0.58$, $R = 0.89$				$PR = 0.83$, $R = 0.96$			

Table 4: Chawathe's algorithm on synthetic data with $MaxRepeats = 6$.

especially in the precision value (PR).

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1-2 (DTDs 1-2)	100	0	0	1-2 (DTDs 1-2)	100	0	0
3 (DTD 3)	80	0	20	3 (DTD 3)	80	0	20
4-10 (DTDs 4-10)	100	0	0	4-10 (DTDs 4-10)	100	0	0
$NumOfClusters = 11$, $Cluster.\ level = 0.51$				$NumOfClusters = 11$, $Cluster.\ level = 0.51$			
$PR = 1.00$, $R = 0.98$				$PR = 1.00$, $R = 0.98$			

Table 5: Our algorithm on synthetic data with $MaxRepeats = 3$.

Tables 5 and 6 present the (a, b, c) values for PR and R calculation as well as the PR and P values themselves, using our algorithm on synthetic data with $maxRepeats = 3$ and $maxRepeats = 6$. Summary usage keeps the already high quality clustering results obtained by clustering without using summaries. In any case, with or without summaries, our algorithm shows better clustering quality either with small trees and only a few repeated elements or with larger trees and many repeated elements. Notice that PR and R reach excellent values ($PR = 1.00, R = 0.97, 0.98$).

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1 (DTD 1)	100	0	0	1-2 (DTDs 1-2)	100	0	0
2 (DTD 2)	87	0	13	3 (DTD 3)	84	0	16
3 (DTD 3)	84	0	16	4-10 (DTDs 4-10)	100	0	0
4-10 (DTDs 4-10)	100	0	0				
$NumOfClusters = 12$, $Cluster.\ level = 0.61$				$NumOfClusters = 11$, $Cluster.\ level = 0.56$			
$PR = 1.00$, $R = 0.97$				$PR = 1.00$, $R = 0.98$			

Table 6: Our algorithm on synthetic data with $MaxRepeats = 6$.

6.5.2 Working on real data

Tables 7 and 8 present the (a, b, c) values for PR and R calculation as well as the PR and P values themselves, using Chawathe's algorithm and our algorithm on real data. The summary usage maintains the already high quality clustering results obtained by clustering without using summaries. PR and R reach excellent values ($PR = 1.00, R = 0.98, 1.00$).

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1 (DTD 1)	70	0	0	1 (DTD 1)	70	0	0
2 (DTD 2)	70	0	0	2 (DTD 2)	70	0	0
3 (DTD 3)	66	0	4	3 (DTD 3)	70	0	0
$NumOfClusters = 4$, $Cluster.\ level = 0.63$				$NumOfClusters = 3$, $Cluster.\ level = 0.63$			
$PR = 1.00$, $R = 0.98$				$PR = 1.00$, $R = 1.00$			

Table 7: Chawathe's algorithm on real data.

Without structural summaries				**With structural summaries**			
Cluster No	a	b	c	Cluster No	a	b	c
1 (DTD 1)	70	0	0	1 (DTD 1)	70	0	0
2 (DTD 2)	70	0	0	2 (DTD 2)	70	0	0
3 (DTD 3)	66	0	4	3 (DTD 3)	70	0	0
$NumOfClusters = 4$, $Cluster.\ level = 0.63$				$NumOfClusters = 3$, $Cluster.\ level = 0.63$			
$PR = 1.00$, $R = 0.98$				$PR = 1.00$, $R = 1.00$			

Table 8: Our algorithm on real data.

6.5.3 Remarks

The evaluation results indicate the following:

- Structural summaries maintain the clustering quality, that is they do not hurt clustering. Also, using structural summaries we can clearly improve the performance of the whole clustering procedure, since the decrease on the time needed to calculate the tree distances is more the 50% in any case.
- With or without summaries, our algorithm shows excellent clustering quality, and improved performance compared to Chawathe's (55% on average).

6.5.4 Further discussion

We confirmed our results using hierarchical clustering methods from CLUTO. Since CLUTO expects the desired amount of clusters as an input, we experimented using 10 and x clusters, where x is the $NumOfClusters$ returned by C-index in every test case:

1. to check if the algorithms have the potential of 100% correct clustering, having the right number of clusters, which is 10, and
2. to check the performance of the algorithms using what C-index gave as an estimation of the number of clusters in each test case.

Notice that a 100% correct clustering means that exactly 10 clusters with 100 files originating from the same DTD were generated ($PR = R = 1$). CLUTO made similar cluster configurations using its single link algorithm. Having 10 clusters as an input, both single link and complete link performed 100% correctly. Having $NumOfClusters$ as an input, the results were similar to ours. Non hierarchical methods, like repeated bisections algorithms [31], showed similar results.

We also performed the single link clustering task using IBM's TreeDiff[11], a set of Java beans that enable efficient differentiation and updating of DOM trees, providing its own tree distance. We used the synthetic dataset used in our main experiments. The results gave PR and R values lower than 0.7.

Having DTDs which are different from each other makes the clustering procedure successful. It is interesting to see how clustering groups together XML data from similar DTDs related to the same domain. For this reason, we performed single link clustering on 300 synthetic XML documents generated using 3 DTDs: bookstore1.dtd, bookstore2.dtd and bookstore3.dtd (see Figure 20). These DTDs were quite similar to each other, making the clustering task quite hard. Preliminary results showed that we were unable to identify groups of XML documents

bookstore1.dtd

```
<!ELEMENT entry (book* )>
<!ELEMENT book (title, author+,
publisher, price )>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

bookstore2.dtd

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, (author+ | editor+ ),
publisher, price )>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first )>
<!ELEMENT editor (last, first, affiliation )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT affiliation (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

bookstore3.dtd

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, author+,
publisher, cost )>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT cost (#PCDATA )>
```

Figure 20: 3 similar DTDs.

using clustering without tree summaries. Calculated PR values were lower than 0.3. On the other hand, we got good quality results using our algorithm and tree summaries. Table 9 presents

[11] http://www.alphaworks.ibm.com/tech/xmltreediff

the (a, b, c) values as well as PR, R values for synthetic data with $maxRepeats = 6$, using our algorithm and tree summaries.

Cluster No	a	b	c
1 (DTD 1)	70	0	0
2 (DTD 2)	70	0	0
3 (DTD 3)	66	0	4
$NumOfClusters = 3$, $Cluster.\ level = 0.20$			
$PR = 0.78$, $R = 0.78$			

Table 9: Homogeneous synthetic data, $MaxRepeats = 6$.

Methods were discussed to cluster a set of existing XML documents by structure at once. However, sometimes there is a need to assign new incoming XML documents to already discovered clusters, instead of applying a clustering method again to the whole set of documents, including the new ones. The latter costs time since all pairwise distances should be calculated again. Classification algorithms can assign new data to clusters already present. k-NN classification is a simple yet quite effective method [32]. A set of M training XML documents is randomly selected from each cluster. Having a new, incoming XML document, we rank the training documents according to their structural distance with the incoming one (the training document with the lowest distance will be on the top). Recall that the structural distance is calculated between the structural summaries of these trees. Then the k top-ranked documents are used to decide the winning cluster(s) by adding the distances for the training documents which represent the same cluster [33, 32]:

$$y(\mathbf{x}, c_j) = \sum_{\mathbf{d}_i \epsilon kNN} S(\mathbf{x}, \mathbf{d}_i) \times y(\mathbf{d}_i, c_j) \tag{6}$$

where:

1. $\mathbf{x}$ is an incoming document, $\mathbf{d}_i$ is a training document, c_j is a category,
2. $y(\mathbf{d}_i, c_j) = 1$ if $\mathbf{d}_i$ belongs to c_j or 0 otherwise,
3. $S(\mathbf{x}, \mathbf{d}_i)$ is the structural distance between the incoming document $\mathbf{x}$ and the training document $\mathbf{d}_i$,

Using thresholds on these scores we obtain binary cluster assignments and we allow the method to assign a document to more than one cluster. Instead, we can just use the cluster with the lowest score as the right one for the incoming document. In our work we followed the second approach. Preliminary results showed excellent classification performance. Having a number of discovered clusters, we tested the kNN classification method for 5 new data sets of 1000 XML synthetic documents each. The method proved quite reliable, since it gave the right decision for 99.7% of the documents without using structural summaries and 100% using structural summaries.

7 Conclusions

This work presented a framework for clustering XML documents by structure. Structural clustering refers to the task of grouping together structurally similar data. In case of XML documents, the application of clustering methods needs distances that estimate the similarity between tree structures in terms of the hierarchical relationship of their nodes.

Modeling XML documents as rooted ordered labeled trees, we faced the 'clustering XML documents by structure' problem as a 'tree clustering' problem. We proposed the usage of tree structural summaries that have minimal processing requirements instead of the original trees representing the XML documents. Those summaries maintain the structural relationships between the elements of an XML document, reducing repetition and nesting of elements and making its structure closer to the structure of its unknown DTD. Also, we presented a new algorithm to calculate tree edit distances and defined a structural distance metric to estimate the structural similarity between the structural summaries of two rooted ordered labeled trees.

In order to experimentally validate our proposals, we implemented a testbed using clustering methods and data sets. We adapted the $C-$index stopping rule in the hierarchical clustering procedure to determine the most appropriate clustering level for the cluster hierarchies hierarchies in order to discover the clusters. We performed extensive evaluation using synthetic and real data sets, providing timing analysis as well as precision PR and recall R values for each test case. Our results showed that:

1. Structural summaries clearly improved the performance of the whole clustering procedure, since the decrease on the time needed to calculate the tree distances using summaries is high. On the other hand, summaries maintained the clustering quality.

2. Our structural distance algorithm showed excellent clustering quality, and improved performance compared to Chawathe's.

3. Excellent results were also obtained when assigning new incoming XML documents to already discovered clusters using the kNN classification method with structural summaries, instead of applying a clustering method again to the whole set of documents, including the new ones. Re-clustering is expensive since all pairwise distances should be calculated again.

4. Preliminary results showed also that structural summaries can clearly help even at clustering XML data coming from similar DTDs, while clustering without summaries failed even to identify groups using such data.

To the best of our knowledge, the only work directly compared with ours is [18]. Their set of tree edit operations include two new ones which refer to whole trees (*insert_tree* and *delete_tree* operations) rather than nodes. They pre-process the trees to detect whether a subtree is contained in another tree. Such a pre-process is needed to precalculate costs for sequences of single *insert_tree* operations, or combinations of *insert_tree* operations and *insert* (node) operations. According to their approach, any tree distance that permits change to only one node at a time will find a large distance between a pair of XML documents coming from the same DTD but having different sizes, so they expand the permitted set of tree edit operations to deal with this problem. Their approach requires the same amount of computation with Chawathe's algorithm. There are no detailed evaluation results, showing PR and R values. Instead, only the number of misclustered documents is presented. In our work, we diminish the possibility of having repeated subtrees using structural summaries instead of expanding the tree edit operations. Structural summaries are used as an index structure to speed up the tree distance calculation. Such an approach has the advantage of being useful to reduce the performance cost in every algorithm that estimates the structural distance between rooted ordered labeled trees.

To conclude, this work successfully applied clustering methodologies for grouping XML documents which have similar structure, by modeling them as rooted ordered labeled trees, and utilizing their structural summaries to reduce time cost while maintaining the quality of the clustering results. As a future work, we will work on two directions. We will explore properties that tree distances present. Also, we will test how our approach scales using larger data sets of XML documents.

References

[1] S. Abiteboul, Querying semi-structured data, in: Proceedings of the ICDT Conference, Delphi, Greece, 1997.

[2] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web., Morgan Kaufmann Publishers, 2000.

[3] X. Tang, F. W. Tompa, Specifying transformations for structured documents, in: Proceedings of the WebDB Workshop, Santa Barbara, CA, USA, 2001, pp. 67–72.

[4] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim, XTRACT: A system for extracting document type descriptors from XML documents, in: Proceedings of the ACM SIGMOD Conference,Texas, USA, 2000.

[5] D. Sankoff, J. Kruskal, Time Warps, String Edits and Macromolecules, The Theory and Practice of Sequence Comparison, CSLI Publications, 1999.

[6] H. G. Direen, M. S. Jones, Knowledge management in bioinformatics, in: A. B. Chaudhri, A. Rashid, R. Zicari (Eds.), XML Data Management, 2003, Addison Wesley.

[7] R. Wilson, M. Cobb, F. McCreedy, R. Ladner, D. Olivier, T. Lovitt, K. Shaw, F. Petry, M. Abdelguerfi, Geographical data interchange using xml-enabled technology within the GIDB system, in: A. B. Chaudhri, A. Rashid, R. Zicari (Eds.), XML Data Management, 2003, Addison Wesley.

[8] Y. Papakonstantinou, H. Garcia-Molina, J. Widom, Object exchange across heterogenous information sources, in: Proceedings of IEEE International Conference on Data Engineering, 1995, pp. 251–260.

[9] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, J. Widom, The TSIMMIS approach to mediation: Data models and languages, Journal of Intelligent Information Systems 8 (2) (1997) 117–132.

[10] S. M. Selkow, The tree-to-tree editing problem, Information Processing Letters 6 (1977) 184–186.

[11] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM Journal of Computing 18 (1989) 1245–1262.

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, Change Detection in Hierarchically Structured Information, in: Proceedings of the ACM SIGMOD Conference, USA, 1996.

[13] S. S. Chawathe, Comparing hierarchical data in external memory, in: Proceedings of the VLDB Conference, Edinburgh, Scotland, UK, 1999, pp. 90–101.

[14] E. W. Myers, An $O(ND)$ difference algorithm and its variations, Algorithmica 1 (1986) 251–256.

[15] R. Wagner, M. Fisher, The string-to-string correction problem, Journal of ACM 21 (1) (1974) 168–173.

[16] K. C. Tai, The tree-to-tree correction problem, Journal of ACM 26 (1979) 422–433.

[17] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: Proceedings of the VLDB Conference, Athens, Greece, 1997.

[18] A. Nierman, H. V. Jagadish, Evaluating structural similarity in xml documents, in: Proceedings of the WebDB Workshop, Madison, Wisconsin, USA, 2002.

[19] N. Jardine, C. J. van Rijsbergen, The use of hierarchical clustering in information retrieval, Information storage and retrieval 7 (1971) 217–240.

[20] E. Voorhees, The effectiveness and efficiency of agglomerative hierarchic clustering in document retrieval, Ph.D. thesis, Cornell University, Ithaca, New York (Oct. 1985).

[21] M. Hearst, J. O. Pedersen, Reexamining the cluster hypothesis: Scatter/gather on retrieval results, in: Proceedings of the ACM SIGIR Conference, Zurich, Switzerland, 1996, pp. 76–84.

[22] E. Rasmussen, Clustering algorithms, in: W. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures and Algorithms, Prentice Hall, 1992.

[23] M. Halkidi, Y. Batistakis, M. Vazirgiannis, Clustering algorithms and validity measures, in: SSDBM Conference, Virginia, USA, 2001.

[24] C. J. van Rijsbergen, Information Retrieval, Butterworths, London, 1979.

[25] T. Cormen, C. Leiserson, R. Rivest, Introduction to algorithms, MIT Press, 1990.

[26] J. C. Gower, G. J. S. Ross, Minimum spanning trees and single linkage cluster analysis, Applied Statistics 18 (1969) 54–64.

[27] G. W. Milligan, M. C. Cooper, An examination of procedures for determining the number of clusters in a data set, Psychometrika 50 (1985) 159–179.

[28] L. J. Hubert, J. R. Levin, A general statistical framework for accessing categorical clustering in free recall, Psychological Bulletin 83 (1976) 1072–1082.

[29] D. Lewie, Evaluating text categorization, in: Proceedings of the Speech and Natural Language Workshop, 1991.

[30] Y. Yang, An evaluation of statistical approaches to text categorization, Information Retrieval 1 (1).

[31] Y. Zhao, G. Karypis, Evaluation of hierarchical clustering algorithms for document datasets, Tr-02–022, Department of Computer Science, University of Minnesota, Minneapolis (2002).

[32] Y. Yang, X. Liu, A re-examination of text categorization methods, in: Proceedings of the 22nd Annual International Conference on Research and Development in Information Retrieval, (ACM-SIGIR'99), 1999, pp. 42–49.

[33] Y. Yang, Expert network: Effective and efficient learning from human decisions in text categorization and retrieval, in: Proceedings of ACM SIGIR Conference, London, UK, 1994, pp. 13–22.