Evaluating Structural Similarity in XML Documents[*]

Andrew Nierman and H. V. Jagadish
University of Michigan
{andrewdn, jag}@eecs.umich.edu

Abstract

XML documents on the web are often found without DTDs, particularly when these documents have been created from legacy HTML. Yet having knowledge of the DTD can be valuable in querying and manipulating such documents. Recent work (cf. [10]) has given us a means to (re-)construct a DTD to describe the structure common to a given set of document instances. However, given a collection of documents with unknown DTDs, it may not be appropriate to construct a single DTD to describe every document in the collection. Instead, we would wish to partition the collection into smaller sets of "similar" documents, and then induce a separate DTD for each such set. It is this partitioning problem that we address in this paper.

Given two XML documents, how can one measure structural (DTD) similarity between the two? We define a tree edit distance based measure suited to this task, taking into account XML issues such as optional and repeated sub-elements. We develop a dynamic programming algorithm to find this distance for any pair of documents. We validate our proposed distance measure experimentally. Given a collection of documents derived from multiple DTDs, we can compute pair-wise distances between documents in the collection, and then use these distances to cluster the documents. We find that the resulting clusters match the original DTDs almost perfectly, and demonstrate performance superior to alternatives based on previous proposals for measuring similarity of trees. The overall algorithm runs in time that is quadratic in document collection size, and quadratic in the combined size of the two documents involved in a given pair-wise distance calculation.

1 Introduction

The Extensible Mark-up Language (XML) is seeing increased use, and promises to fuel even more applications in the future. But many of these XML documents, especially those beginning to appear on the web, are without Document Type Descriptors (DTDs). In [10] the authors provide a method to automatically extract a DTD for a set of XML documents. They provide several benefits for the existence of DTDs. Given that more repositories of XML documents will exist in the future, methods will be needed to access these documents and perform queries over them, much as we do today with traditional database systems. Just as schemas are necessary in a DBMS for the provision of efficient storage mechanisms, as well as the formulation and optimization of queries, the same is true for XML repositories and DTDs (which provide the schema). For instance, a DTD could allow a search to only access the relevant portions of the data, resulting in greater efficiency.

The algorithm in [10] is useful only when we apply it to a repository of XML documents where the repository is a homogeneous collection. If the collection includes structurally unrelated documents, then the DTD inferencing procedure will result in DTDs that are of necessity far too general and therefore not of much value. Ideally, the repository would be divided into groups of structurally similar documents first, and then the DTD inferencing mechanism could be applied individually to each of these groups.

In this paper we define a new method for computing the distance between any two XML documents in terms of their structure. The lower this distance, the more similar the two documents are in terms of structure, and the more likely they are to have been created from the same DTD. Crafting a good distance metric for this setting is somewhat difficult since two documents created from the same DTD can have radically different structures (due to repeating and optional elements), but we would still want to compute a small distance between these documents. We account for this by introducing edit operations that allow for the cutting and pasting of whole sections of a document. Using our resulting pair-wise distance measure, we show that standard clustering algorithms do very well at pulling together documents derived from the same DTD.

2 Background

2.1 XML Data Model

An XML document can be modeled as an ordered labeled tree [9]. Each node in this tree corresponds to an element in the document and is labeled with the element tag name. Each edge in this tree represents inclusion of the element corresponding to the child node under the element corresponding to the parent node in the XML file.

XML documents may also have hyper-links to other documents. Including such links in the model gives rise to a graph rather than a tree. Such links can be important in actual use of the XML data. However, they are not important as far as the structure of the document at hand, and hence we will not consider them further in this paper.

A DTD provides rules that define the elements, attributes associated with elements, and relationships among

[*]This work was supported in part by NSF grant number IIS-0002356.

elements, that *may* occur in an XML document. DTDs have the expressive power of regular languages: elements may be required, optional, or may be repeated an arbitrary number of times. Attributes may also be required or optional.

2.2 Attributes in the Data Model

Elements in XML can have attributes, and these attributes can play an important role in the DTD determination problem we are attempting to tackle. The traditional DOM labeled ordered tree has one node for every element in the document: attributes adorn the node corresponding to the element of which they are attributes. To incorporate attributes into our distance calculation, we create an additional node in the tree for each attribute, and label it with the name of the attribute. These attribute nodes appear as "children" of the node that they adorned in the DOM representation, sorted by attribute name, and appearing before all sub-element "siblings".

In short, we represent each XML document as a labeled ordered tree with a node corresponding to each element and to each attribute. We do not represent the actual values of the elements or attributes in the tree – we are only interested in the structural properties of the XML file.

2.3 Related Work

There is considerable previous work on finding edit distances between trees [5–8, 13–17]. Most algorithms in this category are direct descendants of the dynamic programming techniques for finding the edit distance between strings [12]. The basic idea in all of these tree edit distance algorithms is to find the cheapest sequence of edit operations that can transform one tree into another.

A key differentiator between the various tree-distance algorithms is the set of edit operations allowed. An early work in this area is by Selkow [13], and allows inserting and deleting of single nodes at the leaves, and relabeling of nodes anywhere in the tree. The work by Chawathe in [5] utilizes these same edit operations and restrictions, but is targeted for situations when external memory is needed to calculate the edit distance. There are several other approaches that allow insertion and deletion of single nodes anywhere within a tree [14–17].

Expanding upon these more basic operators, Chawathe, et. al. [7] define a move operator that can move a subtree as a single edit operation, and in subsequent work [6] copying (and its inverse, gluing) of subtrees is allowed. These two operations bear some resemblance to the insert subtree and delete subtree operations that are used in this paper, but the approaches in [6, 7] are heuristic approaches and the algorithm in [6] operates on unordered trees, making it unsuitable for computing distances between XML documents.

3 Tree Edit Distance

Two XML documents produced from the same DTD can have very different sizes on account of optional and repeating elements. Any edit distance metric that permits change to only one node at a time will necessarily find a large distance between such a pair of documents, and consequently will not recognize that these documents should be clustered together as being derived from the same DTD. In this section, we develop an edit distance metric that is more indicative of this notion of structural similarity. First we present a few supporting definitions.

3.1 Basic Definitions

Definition 3.1 [Ordered Tree] An ordered tree is a rooted tree in which the children of each node are ordered. If a node x has k children then these children are uniquely identified, left to right, as $x_1, x_2, \ldots, x_k$.

Definition 3.2 [First-Level Subtree] Given an ordered tree T with a root r of degree k, the first-level subtrees, $T_1, T_2, \ldots, T_k$ of T are the subtrees rooted at $r_1, r_2, \ldots, r_k$.

Definition 3.3 [Labeled Tree] A labeled tree T is a tree that associates a label, $\lambda(x)$, with each node $x \in T$. We let $\lambda(T)$ denote the label of the root of T.

Definition 3.4 [Tree Equality] Given two ordered labeled trees A and B, and their first-level subtrees $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$, $A = B$ if: $\lambda(A) = \lambda(B)$, $m = n$, and $i = j \Rightarrow A_i = B_j$, for $0 \leq i \leq m$, $0 \leq j \leq n$.

3.2 Tree Transformation Operations

We utilize five different edit operations in the construction of our algorithm. Given a tree T with $\lambda(T) = l$ and first-level subtrees $T_1, ..., T_m$, the tree transformation operations are defined as follows:

Definition 3.5 [Relabel] $Relabel_T(l_{new})$ is a relabel operation applied to the root of T that yields the tree $T\prime$ with $\lambda(T\prime) = l_{new}$ and first-level subtrees $T_1, ..., T_m$.

Definition 3.6 [Insert] Given a node x with degree 0, $Insert_T(x, i)$ is a node insertion operation applied to T at i that yields the tree $T\prime$ with $\lambda(T\prime) = l$ and first-level subtrees $T_1, \ldots, T_i, x, T_{i+1}, \ldots, T_m$.

Definition 3.7 [Delete] If the first-level subtree T_i is a leaf node, $Delete_T(T_i)$ is a delete node operation applied to T at i that yields the tree $T\prime$ with $\lambda(T\prime) = l$ and first-level subtrees $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$.

Definition 3.8 [Insert Tree] Given a tree A, $InsertTree_T(A, i)$ is an insert tree operation applied to T at i that yields the tree $T\prime$ with $\lambda(T\prime) = l$ and first-level subtrees $T_1, \ldots, T_i, A, T_{i+1}, \ldots, T_m$.

Definition 3.9 [Delete Tree] $DeleteTree_T(T_i)$ is a delete tree operation applied to T at i that yields the tree $T\prime$ with $\lambda(T\prime) = l$ and first-level subtrees $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$.

Associated with each of these edit operations is a non-negative cost. Our algorithms work with general costs, but in this paper we restrict our presentation and experimentation to constant (unit) costs.

3.3 Allowable Edit Sequences

The usual way in which the edit distance is found between two objects is to consider alternative sequences of edit operations that can transform one object into the other. The cost of the operations in each sequence is considered, and the lowest cost sequence among these defines the edit distance between the two objects. In our case, rather than considering all possible sequences of edit operations, we restrict ourselves to all "allowable sequences" of edit operations. We do this both for computational reasons, as well as to improve our results in the XML domain.

Definition 3.10 [Minimum Edit Distance Cost (δ)] Given any trees A and B and the set Ξ of all allowable sequences of edit operations that when applied to A will yield a tree equal to B, we let $\delta(A, B)$ denote the minimum of the sums of the costs of each sequence in Ξ.

Definition 3.11 [Allowable] A sequence of edit operations is *allowable* if it satisfies the following two conditions:

1. A tree P may be inserted only if P already occurs in the source tree A. A tree P may be deleted only if P occurs in the destination tree B.
2. A tree that has been inserted via the $InsertTree$ operation may not subsequently have additional nodes inserted. A tree that has been deleted via the $DeleteTree$ operation may not previously have had (children) nodes deleted.

The first restriction limits the use of the insert tree and delete tree operations to when the subtree that is being inserted (or deleted) is shared between the source and destination tree. We can only insert (delete) subtrees that are already "contained in" the source (destination) tree. A pattern tree P is said to be $containedIn$ tree T, if all nodes of P occur in T, with the same parent/child edge relationships and same sibling order; additional siblings may occur in T, even between sibling nodes in the embedding of the pattern tree. This allows for matching of trees when optional elements are used in DTDs. See Figure 1 for some examples of the $containedIn$ relation, where a pattern tree P is potentially $containedIn$ various other trees. Without this first restriction on allowable sequences of edit operations, one could delete the entire source tree in one step and insert the entire destination tree in a second step – totally defeating the purpose of the insert tree and delete tree operations.

The second restriction provides us with an efficient means for computing the costs of inserting and deleting the subtrees found in the destination and source trees, respectively. This procedure is outlined in the next section.

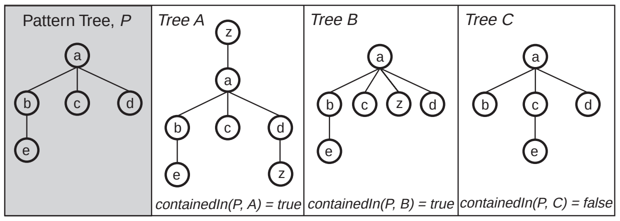

Figure 1: Examples of the $containedIn$ Procedure

4 Algorithm

4.1 Dynamic Programming Formulation

Dynamic programming is frequently used to solve minimum edit distance problems. In determining the distance between a source tree, A, and a destination tree, B, the key to formulating the problem using dynamic programming is to first determine the cost of inserting every subtree of B, and the cost of deleting every subtree of A.

When determining the cost of inserting a subtree T_i, this could possibly be done with a single $InsertTree$ operation (if it is allowable), or with some combination of $InsertTree$ and $Insert$ operations. There is a cost associated with each possible sequence of $InsertTree$ and $Insert$ operations that result in the construction of the subtree T_i. The minimum of these costs is denoted as the $graft$ cost of T_i, or $Cost_{Graft}(T_i)$. A $prune$ cost is defined similarly for the minimum cost sequence of $Delete$ and $DeleteTree$ operations needed to remove a subtree.

Due to the constraints specified in definition 3.11 for an allowable sequence, we have a simple and efficient bottom-up procedure for computing the graft cost, $Cost_{Graft}$. At each node $v \in B$ we calculate the cost of inserting the single node v and add the graft cost of each child of v, we call this sum d_0. We also check whether the pattern tree P, which is the subtree rooted at v, is $containedIn$ the source tree A. If $containedIn(P, A)$ is true, we compute the insert tree cost for P, we call this sum d_1. The graft cost for the subtree rooted at v is the minimum of d_0 and d_1. Prune costs are computed similarly for each node in A.

Given a source tree A and a destination tree B, we can determine the minimum cost of transforming A into B using the operators defined in section 3.2, and the notion of allowable sequences in section 3.3. This dynamic programming algorithm is shown in figure 2. Pre-computed costs for the graft and prune costs are used in lines 8 and 10, and in the nested loops at lines 16 and 17 of the algorithm.

4.2 Complexity

In this section we analyze the complexity of computing our edit distance measure between a source tree A and a destination tree B. There are two stages to the algorithm. In the first stage, all the graft and prune costs are pre-computed. In the second stage, we use these pre-computed values to compute the actual edit distance, as given in figure 2.

```
private int editDistance(Tree A, Tree B) {
    int M = Degree(A);
    int N = Degree(B);
    int[][] dist = new int[0..M][0..N];
    dist[0][0] = $Cost_{Relabel}(\lambda(A), \lambda(B))$;

    for (int j = 1; j $\leq$ N; j++)
        dist[0][j] = dist[0][j-1] + $Cost_{Graft}(B_j)$;
    for (int i = 1; i $\leq$ M; i++)
        dist[i][0] = dist[i-1][0] + $Cost_{Prune}(A_i)$;

    for (int i = 1; i $\leq$ M; i++)
        for (int j = 1; j $\leq$ N; j++)
            dist[i][j] = min{
                dist[i-1][j-1] + editDistance($A_i$, $B_j$),
                dist[i][j-1] + $Cost_{Graft}(B_j)$,
                dist[i-1][j] + $Cost_{Prune}(A_i)$
            };
    return dist[M][N];
} //editDistance
```

Figure 2: Edit Distance Algorithm

4.2.1 Stage One – Computing $Cost_{Graft}$ and $Cost_{Prune}$

Given that we perform the pre-computation of graft and prune costs in a naive manner, the complexity of the first stage would dominate the second stage. The central issue is that determining whether a subtree is contained in another tree is potentially an expensive operation, and this operation may have to be performed repeatedly. We present a few implementation details necessary to reduce the complexity for computing graft costs. A complementary method is used for prune costs.

First, for each leaf node $v \in B$, we determine its $containedInList$, that is, which nodes in A have the same label as v. Rather than do so repeatedly for each node individually, we do so by node label. We perform an in-order walk of A and append each node to a list corresponding to its label (these lists are kept in a hash, with the labels as keys). This costs $O(|A|)$. Then we walk through the leaves of B, and for each leaf node v, we set a pointer for that node to its corresponding list of nodes from A, based on the label. This costs $O(|B|)$, and the overall procedure for determining the leaf node $containedInLists$ is $O(|A| + |B|)$.

Now, we perform a post-order traversal of the nodes of B, and at each non-leaf node, we calculate the $containedInList$ based upon the $containedInLists$ of each of its children. The process is similar to a simple merge operation. All of the $containedIn$ lists of the children are already sorted based on position in A. In total, across the entire traversal, we will have $|B|$ lists to merge, with each of these $|B|$ lists being of length at most $|A|$. Given that the maximum degree of any node in B is assumed constant, independent of the sizes of the trees, the time complexity for this procedure is $O(|A||B|)$.

Having computed these $containedIn$ relations, the graft and prune costs can be calculated, as in section 4.1, by simply performing post-order traversals of B and A, respectively, so the complexity of these operations is simply $O(|B|)$ and $O(|A|)$, and the overall complexity of this stage is $O(|A||B|)$.

4.2.2 Stage Two – Dynamic Programming ($editDistance$)

The $editDistance$ procedure in figure 2 is called once for each pair of vertices at the same depth in the input trees A and B. This results in a complexity of $O(|A||B|)$ [13].

4.2.3 Overall Complexity

$O(|A||B|)$ is the time complexity for both the precomputation phase, and the dynamic programming phase, so $O(|A||B|)$ is the overall complexity of our algorithm to compute structural edit distance between two trees A and B. This linear dependence on the size of each tree (and quadratic dependence on the combined size of the two trees) is borne out in the experimental results shown in Section 5.3.

5 Experimental Evaluation

The goal of our work is to find documents with structural similarity, that is, documents generated from a common DTD. We apply a standard clustering algorithm based on the distance measures computed for a given collection of documents with known DTDs. For any choice of distance metric, we can evaluate how closely the reported clusters correspond to the actual DTDs.

5.1 Setup

5.1.1 Algorithms Used

In addition to our edit distance measure, we evaluate two measures proposed previously in the literature for tree edit distance – which we refer to as Chawathe [5] and Shasha [14] respectively, and a third non-structural baseline metric. We report results in this section for these three measures in addition to our own.

The Chawathe measure – In [5] an algorithm is presented for computing differences between hierarchically structured data such as XML. Disregarding the work's contribution towards efficient use of secondary storage, our algorithm can be seen as a strict generalization of this approach. Specifically, if we disallow tree insertions and deletions in our measure, we would obtain exactly the Chawathe measure. The complexity of this approach is $O(|A||B|)$, when finding the minimum edit distance between the trees A and B.

The Shasha Measure – Dennis Shasha and Jason Wang propose a tree edit distance metric in [14] that permits the addition and deletion of single nodes anywhere in the tree, not just at the leaves. However, entire subtrees cannot be inserted or deleted in one step. The complexity of this approach is $O(|A||B|\, depth(A)\, depth(B))$.

Tag Frequency (δ_{freq}) – A good question to ask is whether all this complex tree structure based difference is a good thing to do in the first place. How about a simple measure that looks at the count of each type of label in the two documents, and adds up the absolute values of the differences? By utilizing a simple hash data structure for the element names and the frequencies, we can compute the tag frequency distance, δ_{freq}, between two trees A and B in $O(|A| + |B|)$.

5.1.2 Data Sets Used

We performed experiments on both real and synthetic data sets. For a real data set, we used XML data obtained from the online XML version of the ACM SIGMOD Record [1]. Specifically, we sampled documents from each of the following DTDs: ProceedingsPage.dtd, IndexTermsPage.dtd, and OrdinaryIssuePage.dtd.

We also utilize synthetic data generated in an automated fashion from real DTDs. Real-world DTDs were obtained online from [2, 3][1] and an XML document generator [4] that accepts the DTDs as input was used to generate the XML documents. We varied the following two key parameters to generate repositories:

MaxRepeats The maximum number of times a child element node will appear as a child of its parent node (when the * or + option is used in the DTD). A value between 0 and MaxRepeats is chosen randomly for each repeating node (rather than once for the entire document). The greater this number, the greater the fanout and also the greater the variability in fanout.

Attribute Occurrence Probabilities There are both required and optional attributes specified in a DTD. We let $Prob_{Attribute}$ equal the probability that an optional attribute will occur.

We experimented with values for $MaxRepeats$ in the range [2,12]. Also, we tested the following values for the attribute occurrence probabilities: $Prob_{Attribute} \in \{.1, .25, .5, .75, .9, 1\}$[2]. In this paper, we present a representative sampling of these tests with the following synthetic data sets:
Data Set 1: $MaxRepeats$ = 4, $Prob_{Attribute} = .75$;
Data Set 2: $MaxRepeats$ = 4, $Prob_{Attribute} = 1$ (attributes always appear);
Data Set 3: $MaxRepeats$ = 8, $Prob_{Attribute} = .75$;
Data Set 4: $MaxRepeats$ = 8, $Prob_{Attribute} = 1$.

All operator costs were set equal to 1 for all of the experiments that we present in this paper[3].

5.1.3 Computing Environment

These tests were done on an IBM RS6000 with dual processor 604e PowerPC Processors, running at 332 MHz. All approaches, except the Shasha measure, were implemented by us in Java. The Shasha measure is implemented in C and thus the timing results cannot be directly compared with the other methods.

5.2 Clustering

Due to lack of space we do not present the distances obtained from comparing all pairs of documents to each other, rather we simply present the clustering results that were obtained using these distances. We utilize well-known techniques in hierarchical agglomerative clustering [11] (although any form of clustering could be used).

The end result can be represented visually as a tree of clusters called a *dendrogram*. The dendrogram shows the clusters that were merged together, and the distance between these merged clusters (the horizontal length of the branches is proportional to the distance between the merged clusters). Two example dendrograms can be seen in figure 3.

Clustering algorithms require knowledge of the distance between any pair of clusters, including single document "clusters". For this purpose, we use the Unweighted Pair-Group Averaging Method (or UPGMA). The distance between clusters C_i and C_j is computed as follows:

$$Distance(C_i, C_j) = \frac{\sum_{k=1}^{|C_i|} \sum_{l=1}^{|C_j|} \delta(doc_k^{C_i}, doc_l^{C_j})}{|C_i||C_j|}$$

Where $|C_i|$ is the number of XML documents contained in cluster C_i and $doc_k^{C_i}$ is the k^{th} XML document in the cluster C_i.

In order to compare the hierarchical clustering results, we introduce our notion of a "mis-clustering". Given a dendrogram, the number of mis-clusterings is equal to the minimum number of documents in the dendrogram that would have to be moved, so that all documents from the same DTD are grouped together. A small sample clustering is shown in figure 3: in this example our approach has no mis-clusterings, while the Chawathe approach has three mis-clusterings.

A summary of the number of mis-clusterings, for each of the data sets, is found in table 1. Our approach performs better than the competing approaches for each of the data sets (and in fact, in the underlying data, there is no average intra-DTD distance that is lower than our approach, and no average inter-DTD distance that is higher). Our approach does better when all attributes are forced to appear, since there would be more subtrees that "look" the same in documents generated from the same DTD, and the $containedIn$ procedure would return true more often.

[1]The DTDs that were used were: HealthProduct.dtd, blastxml.dtd, dri.dtd, flights.dtd, flixml.dtd, roamops-phonebook.dtd, vcard.dtd, and dsml.dtd.

[2]For all values of $Prob_{Attribute}$ not equal to 1, there were no appreciable differences in the results, and we simply show the results for $Prob_{Attribute} = .75$.

[3]Small changes in operator costs did little to affect overall clustering accuracy. These results are not presented due to lack of space.

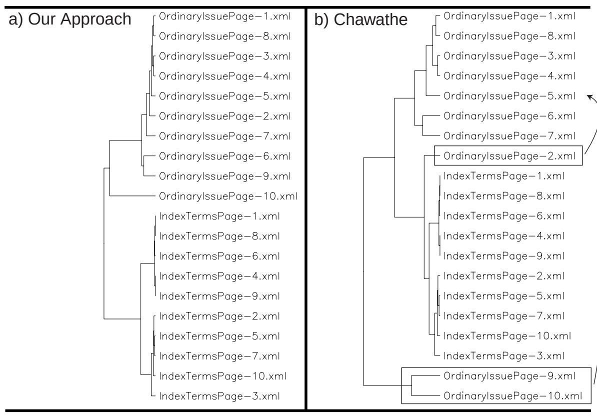

Figure 3: Sample Clustering Results for SIGMOD Record.

	Data Set 1	Data Set 2	Data Set 3	Data Set 4	SIGMOD Record
Our Approach	10	2	11	9	0
Chawathe	16	8	30	25	3
Shasha	16	9	32	39	3
Tag Frequency	22	21	35	40	3

Table 1: Number of Mis-Clusterings for Each Approach

5.3 Timing Analysis

Our algorithm appears more complex conceptually, however, its asymptotic time complexity ($O(|A||B|)$) is the same as the Chawathe algorithm, and slightly better than Shasha's algorithm ($O(|A||B|\, depth(A)\, depth(B))$). We are asymptotically worse than the δ_{freq} approach, which is $O(|A|+|B|)$, but this approach performs poorly in terms of clustering the documents. The formulae are verified experimentally, and we show the timing results for our approach in figure 4. The time to find the edit distance between pairs of trees of various sizes grows in an almost perfect linear fashion with tree size (of each tree). The corresponding times for the Chawathe technique were smaller by (only) a factor of 1.6 on average.

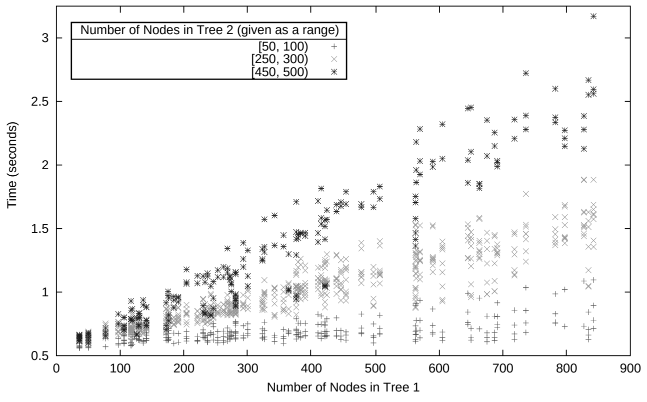

Figure 4: Our Approach - Timing Results (to compute pairwise distance) for Various Tree Sizes

6 Conclusion

XML is becoming all-pervasive, and effective management of XML data is a high priority. The applicability of many database techniques to XML data depends on the existence of DTDs (or schema) for this data. In the laissez-faire world of the Internet, though, we frequently have to deal with XML documents for which we do not know the schema. While there has been previous work on deducing the DTD for a collection of XML documents, such algorithms depend critically on being given a relatively homogeneous collection of documents in order to determine a meaningful DTD.

In this paper we have developed a structural similarity metric for XML documents based on an "XML aware" edit distance between ordered labeled trees. Using this metric, we have demonstrated the ability to accurately cluster documents by DTD. In contrast, we have shown that several other measures of similarity do not perform as well, while requiring approximately the same amount of computation.

References

[1] Available at `http://www.acm.org/sigmod/record/xml`.

[2] Available at `http://www.schema.net`.

[3] Available at `http://www.xml.org`.

[4] Available at `http://www.alphaworks.ibm.com`.

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

[6] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proc. of ACM SIGMOD*, pages 26–37, 1997.

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

[8] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *Proc. of ICDE*, 2002.

[9] World Wide Web Consortium. The document object model. `http://www.w3.org/DOM/`.

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

[11] N. Jardine and R. Sibson. *Mathematical Taxonomy*. John Wiley and Sons, New York, 1971.

[12] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.*, 6:707–710, 1966.

[13] S. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

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

[15] K. C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26:422–433, 1979.

[16] J. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE TKDE*, 6(4):559–571, 1994.

[17] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, December 1989.