Similarity of DTDs Based on Edit Distance and Semantics[⋆]

Aleš Wojnar, Irena Mlýnková and Jiří Dokulil

Charles University in Prague, Czech Republic
ales.wojnar@gmail.com, irena.mlynkova@mff.cuni.cz,
jiri.dokulil@mff.cuni.cz

Summary. In this paper we propose a technique for evaluating similarity of XML schema fragments. Contrary to existing works we focus on structural level in combination with semantic similarity of the data. For this purpose we exploit the idea of edit distance utilized to constructs of DTDs which enables to express the structural differences of the given data more precisely. In addition, in combination with the semantic similarity it provides more realistic results. Using various experiments we show the behavior and advantages of the proposed approach.

1 Introduction

The eXtensible Markup Language (XML) [3] has already become a standard for data representation and manipulation and, thus, it appears in most areas of information technologies. A possible optimization of XML-based methods can be found in exploitation of similarity of XML data. The most common areas of exploitation of data similarity are clustering, dissemination-based applications (e.g. [1]), schema integration systems (e.g. [7]), data warehousing, e-commerce, semantic query processing etc. But despite the amount of existing similarity-based approaches is significant, there is still a space for both improvements and new ways of similarity exploitation.

In this paper we focus on similarity of XML schema fragments expressed in DTD language [3] and, in particular, on persisting disadvantages of the existing approaches. The key emphasis is currently put on the semantic similarity of schema fragments reflecting the requirements of corresponding applications. And if the approaches consider DTD structure, they usually analyze only simple aspects such as, e.g., leaf nodes or child nodes of roots of the fragments. Therefore, we focus on more precise analysis of the structure, but, on the other hand, we still preserve the exploitation of semantic similarity. For this purpose we combine and adapt to DTD constructs two verified approaches – edit distance and semantics of element/attribute names.

[⋆] This work was supported in part by the National Programme of Research (Information Society Project 1ET100300419).

The paper is structured as follows: Section 2 overviews the related works. Section 3 describes the proposed approach and Section 4 results of related experiments. Section 5 provides conclusions and outlines future work.

2 Related Work

The number of existing works in the area of XML data similarity evaluation is nontrivial. We can search for similarity among XML documents, XML schemes, or between the two groups. We can distinguish several levels of similarity, such as, e.g., structural level, semantic level or constraint level. Or we can require different precision of the similarity.

In case of document similarity we distinguish techniques expressing similarity of two documents D_A and D_B using edit distance, i.e. by measuring how difficult is to transform D_A into D_B (e.g. [10]) and techniques which specify a simple and reasonable representation of D_A and D_B, such as, e.g., using a set of paths, that enables their efficient comparison and similarity evaluation (e.g. [12]). In case of similarity of a document D and a schema S there are also two types of strategies – techniques which measure the number of elements which appear in D but not in S and vice versa (e.g. [2]) and techniques which measure the closest distance between D and "all" documents valid against S (e.g. [9]). And finally, methods for measuring similarity of two XML schemes S_A and S_B combine various supplemental information and similarity measures such as, e.g., predefined similarity rules, similarity of element/attribute names, equality of data types, similarity of schema instances or previous results (e.g. [4, 5]). But, in general, the approaches focus mostly on semantic aspects of the schema fragments, whereas structural ones are of marginal importance.

3 Proposed Algorithm

The proposed algorithm is based mainly on the work presented in [10] which focuses on expressing similarity of XML documents D_A and D_B using tree edit distance, i.e. the amount of operations necessary to transform D_A to D_B. The main contribution of the algorithm is in introducing two new edit operations *InsertTree* and *DeleteTree* which allow manipulating more complex structures than a single node. And repeated structures can be found in a DTD as well if it contains shared or recursive elements. But, contrary to XML documents that can be modeled as trees, DTDs can, in general, form general cyclic graphs. Hence, procedures for computing edit distance of trees need to be utilized to DTD graphs. In addition, not only the structural, but also the semantic aspect of elements is very important. Therefore, we will also concern semantic similarity of element/attribute names.

The method can be divided into three parts depicted in Algorithm 1, where the input DTDs are firstly parsed (line 1 and 2) and their tree representations are constructed. Next, costs for tree inserting (line 3) and tree

Algorithm 1 Main body of the algorithm

```
Input: $XSD_A$, $XSD_B$
Output: Edit distance between $XSD_A$ and $XSD_B$
1: $T_A$ = ParseXSD($XSD_A$);
2: $T_B$ = ParseXSD($XSD_B$);
3: $Cost_{Graft}$ = ComputeCost($T_B$);
4: $Cost_{Prune}$ = ComputeCost($T_A$);
5: return EditDistance($T_A$, $T_B$, $Cost_{Graft}$, $Cost_{Prune}$);
```

deleting (line 4) are computed. And in the final step (line 5) we compute the resulting edit distance, i.e. similarity, using dynamic programming.

3.1 DTD Tree Construction

The key operation of our approach is tree representation of the given DTDs. Nevertheless, the structure of a DTD can be quite complex – the specified content models can contain arbitrary combinations of operators (i.e. "`|`" or "`,`") and cardinality constraints (i.e. "`?`", "`*`" or "`+`"). Therefore, we firstly simplify the complex regular expressions using a set of transformation rules.

Simplification of DTDs. For the purpose of simplification of DTD content models we can use various transformation rules. Probably the biggest set was defined in [11], but these simplifications are for our purpose too strong. Hence, we use only a subset of them as depicted in Figures 1 and 2.

I-a) $(e_1\|e_2)^* \rightarrow e_1^*, e_2^*$
I-b) $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
I-c) $(e_1, e_2)? \rightarrow e_1?, e_2?$
I-d) $(e_1, e_2)^+ \rightarrow e_1^+, e_2^+$
I-e) $(e_1\|e_2) \rightarrow e_1?, e_2?$

Fig. 1. Flattening rules

II-a) $e_1^{++} \rightarrow e_1^+$	II-b) $e_1^{**} \rightarrow e_1^*$
II-c) $e_1^*? \rightarrow e_1^*$	II-d) $e_1?^* \rightarrow e_1^*$
II-e) $e_1^{+*} \rightarrow e_1^*$	II-f) $e_1^{*+} \rightarrow e_1^*$
II-g) $e_1?^+ \rightarrow e_1^*$	II-h) $e_1^+? \rightarrow e_1^*$
II-i) $e_1?? \rightarrow e_1^?$	

Fig. 2. Simplification rules

The rules ensure that each cardinality constraint operator is connected to a single element and avoid usage of "`|`" operator, though at the cost of a slight information loss.

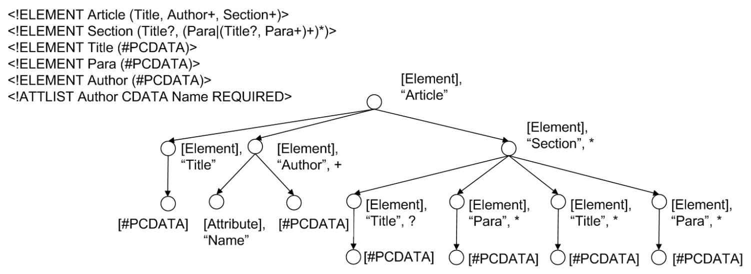
DTD Tree. Having a simplified DTD, its tree representation is defined as:

Definition 1. *A* DTD Tree *is an ordered rooted tree $T = (V, E)$, where*

1. *V is a finite set of nodes, s.t. for $\forall v \in V$, $v = (v_{Type}, v_{Name}, v_{Cardinality})$, where v_{Type} is the type of a node (i.e. attribute, element or #PCDATA), v_{Name} is the name of an element/attribute, and $v_{Cardinality}$ is the cardinality constraint operator of an element/attribute,*

2. $E \subseteq V \times V$ *is a set of edges representing relationships between elements and their attributes or subelements.*

An example of a DTD and its tree representation (after simplification) is depicted in Figure 3.



```
<!ELEMENT Article (Title, Author+, Section+)>
<!ELEMENT Section (Title?, (Para|(Title?, Para+)+)*)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Para (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ATTLIST Author CDATA Name REQUIRED>
```

Fig. 3. An example of a DTD and its tree representation

Shared and Repeatable Elements. The structure of a DTD does not have to be purely tree-like. There can occur both shared elements which invoke undirected cycles and recursive elements which invoke directed cycles. In case of a shared element we easily create its separate copy for each sharer. But, in case of recursive elements the same idea would invoke infinitely deep trees. However, we exploit the observation of a statistical analysis of real-world XML data [8] that the amount of repetitions is in general very low – less than 10. Actually, for our method it is not important exactly how many occurrences we use because each of them can be transformed using a single edit operation.

3.2 Tree Edit Operations

Having the above described tree representation of a DTD, we can now easily utilize the tree edit algorithm proposed in [10]. For a given tree T with a root node r of degree m and its first-level subtrees $T_1, T_2, ..., T_m$, the tree edit operations are defined as follows:

Definition 2. *Substitution*$_T(r_{new})$ *is a node substitution operation applied to* T *that yields the tree* T' *with root node* r_{new} *and first-level subtrees* $T_1, ..., T_m$.

Definition 3. *Given a node* x *with degree 0,* $Insert_T(x, i)$ *is a node insertion operation applied to* T *at* i *that yields the new tree* T' *with root node* r *and first-level subtrees* $T_1, ..., T_i, x, T_{i+1}, ..., T_m$.

Definition 4. *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' *with root node* r *and first-level subtrees* $T_1, ..., T_{i-1}, T_{i+1}, ..., T_m$.

Definition 5. *Given a tree T_j, $InsertTree_T(T_j, i)$ is an insert tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, ..., T_i, T_j, T_{i+1}, ..., T_m$.*

Definition 6. *$DeleteTree_T(T_i)$ is a delete tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, ..., T_{i-1}$, T_{i+1}, $..., T_m$.*

Transformation of a source tree T_A to a destination tree T_B can be done using various sequences of the operations. But, we can only deal with so-called *allowable* sequences, i.e. the relevant ones. For the purpose of our approach we only need to modify the original definition [10] as follows:

Definition 7. *A sequence of edit operations transforming a source tree T_A to a destination tree T_B is* allowable *if it satisfies the following two conditions:*

1. *A tree T may be inserted only if tree similar to T already occurs in the source tree T_A. A tree T may be deleted only if tree similar to T occurs in the destination tree T_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 nodes deleted.*

While the original definition requires exactly the same nodes and trees, we relax the requirement only to similar ones. The exact meaning of the similarity is explained in the following text and enables to combine the tree edit distance with other approaches. Also note that each of the edit operations is associated with a non-negative cost.

3.3 Costs of Inserting and Deleting Trees

Inserting (deleting) a subtree T_i can be done with a single operation *InsertTree* (*DeleteTree*) or with a combination of *InsertTree* (*DeleteTree*) and *Insert* (*Delete*) operations. To find the optimal variant the algorithm uses pre-computed cost for inserting T_i, $Cost_{Graft}(T_i)$, and deleting T_i, $Cost_{Prune}(T_i)$. The procedure can be divided into two parts: In the first part *ContainedIn* list is created for each subtree of T_i; in the second part $Cost_{Graft}$ and $Cost_{Prune}$ are computed for T_i. For our purpose we modify procedure defined in [10] to involve similarity.

Similarity of Elements/Attributes. Similarity of elements/attributes can be evaluated using various criteria. Since the structural similarity is solved via the edit distance, we focus on semantic, syntactic and cardinality-constraint similarity.

Semantic similarity is a score that reflects the semantic relation between the meanings of two words. We exploit procedure *SemanticSim* described in [5] which determines ontology similarity between two words w_1 and w_2 by iterative searching a thesaurus and comparing w_1 with synonyms of w_2.

Syntactic similarity of element/attribute names is determined by computing the edit distance between their labels. For our purpose the classical Levenshtein algorithm [6] is used that determines the edit distance of two strings using inserting, deleting or replacing single characters.

And finally, we consider *similarity of cardinality constraints* of elements specified by the cardinality compatibility table depicted in Table 1.

Table 1. Cardinality compatibility table

	*	+	?	none
*	1	0.9	0.7	0.7
+	0.9	1	0.7	0.7
?	0.7	0.7	1	0.8
none	0.7	0.7	0.8	1

The overall similarity of elements/attributes e_1 and e_2 is computed as $Sim(e_1, e_2) = Max(SemanticSim(e_1, e_2), SyntacticSim(e_1, e_2)) \times \alpha + CardinalitySim(e_1, e_2) \times \beta$, where $\alpha + \beta = 1$ and $\alpha, \beta \geqslant 0$.

***ContainedIn* Lists.** The procedure for determining element/attribute similarity is used for creating *ContainedIn* lists which are then used for computing $Cost_{Graft}$ and $Cost_{Prune}$. The list is created for each node of the destination tree and contains pointers to similar nodes in the source tree.

The procedure for creating *ContainedIn* lists is shown in Algorithm 2. Since creating of lists starts from leaves and continues to root, there is recursive calling of procedure at line 2. At line 4 we find all similar nodes of n in tree T_A and add them to a temporary list. If n is a leaf node, the *ContainedIn* list is created. For a non-leaf node we have to filter the list with lists of its descendants (line 6). At this step each descendant of n has to be found at corresponding position in descendants of nodes in the created *ContainedIn* list. More precisely, let $v_A \in n_{ContainedIn}$, $children_{v_A}$ is the set of v_A descendants, and c is a child of n. Then $c_{ContainedIn} \cap children_{v_A} \neq \emptyset$, otherwise v_A is removed from $n_{ContainedIn}$.

Costs of Inserting Trees. When the *ContainedIn* list with corresponding nodes is created for node r, the cost for inserting the tree rooted at r can be assigned. The procedure is shown in Algorithm 3. The *foreach* loop computes sum, sum_{d_0}, for inserting node r and all its subtrees. If *InsertTree* operation can be applied (*ContainedIn* list of r is not empty), sum_{d_1}, is computed for this operation at line 8. The minimum of these costs are finally denoted as $Cost_{Graft}$ for node r.

Costs of Deleting Trees. Since the rules for deleting a subtree from source are same as rules for inserting a subtree into destination tree, costs for deleting trees are obtained by the same procedures. We only switch tree T_A with T_B in procedures *CreateContainedInLists* and *ComputeCost*.

Algorithm 2 CreateContainedInLists(T_A, n)

```
Input: tree $T_A$, root $n$ of $T_B$
Output: $CointainedIn$ lists for all nodes in $T_B$
 1: for all $child$ of $n$ do
 2:     CreateContainedInLists($T_A$, $child$);
 3: end for
 4: $n_{ContainedIn}$ = FindSimilarNodes($T_A$, $n$);
 5: for all $child$ of $n$ do
 6:     $n_{ContainedIn}$ = FilterLists($n_{ContainedIn}, child_{ContainedIn}$);
 7: end for
 8: Sort($n_{ContainedIn}$);
```

Algorithm 3 ComputeCost(r)

```
Input: root $r$ of $T_B$
Output: $Cost_{Graft}$ for $T_B$
 1: $sum_{d_0} = 1$;
 2: for all $child$ of $r$ do
 3:     ComputeCost($child$);
 4:     $sum_{d_0}$ += $Cost_{Graft}$(child);
 5: end for
 6: $sum_{d_1} = \infty$;
 7: if $r_{ContainedIn}$ is not empty then
 8:     $sum_{d_1} = ComputeInsertTreeCost(r)$;
 9: end if
10: $Cost_{Graft}$(r) = Min($sum_{d_0}$,$sum_{d_1}$);
```

3.4 Computing Edit Distance

The last part of the algorithm, i.e. computing the edit distance, is based on dynamic programming. At this step the procedure decides which of the operations defined in Section 3.2 will be applied for each node to transforming source tree T_A to destination tree T_B. This part of algorithm does not have to be modified for DTDs so the original procedure presented in [10] is used. (We omit the formal algorithm for the paper length.)

3.5 Complexity

In [10] it was proven that the overall complexity of transforming tree T_A into tree T_B is $O(|T_A||T_B|)$. In our method we have to consider procedures for constructing DTD trees and for evaluating similarity. Constructing a DTD tree can be done in $O(|T_A|)$ for tree T_A. Complexity of finding similarity depends on procedures *SemanticSim*, *SyntacticSim* and *CardinalitySim*. *SyntacticSim* is computed for each pair of elements in trees T_A and T_B, so its complexity is $O(|T_A||T_B||\omega|)$, where ω is maximum length of an element/attribute label. *CardinalitySim* is also computed for each pair of elements, however, with constant complexity, i.e. in $O(|T_A||T_B|)$. Complexity of *SemanticSim* depends on the size of the thesaurus, so the overall complexity is $O(|T_A||T_B||\Sigma|)$, where Σ is the set of words in the thesaurus. And it also determines the complexity of the whole algorithm.

4 Experiments

To analyze the behavior of the proposal we have performed various experiments with both real-world and synthetic XML data.

Real-World XML Data. In the first test we have used seven different real-world DTDs. First five DTDs (**c1**, **c2**, ..., **c5**) represent an object `CUSTOMER`, but in more or less different ways. Next two DTDs represent different objects – `TVSCHEDULE` (**tv**) and `NEWSPAPER` (**np**). The parameters have been set to default values and both structural and semantic similarities have been exploited. The resulting similarities are depicted in Table 2.

Table 2. Structural and semantic similarity of real-world DTDs

	c1	**c2**	**c3**	**c4**	**c5**	**tv**	**np**
c1	1	0.57	0.43	0.19	0.71	0.08	0.42
c2	0.57	1	0.57	0.45	0.48	0.10	0.11
c3	0.43	0.57	1	0.39	0.36	0.01	0.13
c4	0.19	0.45	0.39	1	0.21	0.00	0.00
c5	0.71	0.48	0.36	0.21	1	0.00	0.11
tv	0.08	0.10	0.01	0.00	0.00	1	0.00
np	0.42	0.11	0.13	0.00	0.11	0.00	1

Table 3. Similarity of real-world DTDs without semantic similarity

	c1	**c2**	**c3**	**c4**	**c5**	**tv**	**np**
c1	1	0.45	0.23	0.09	0.57	0.00	0.13
c2	0.45	1	0.50	0.42	0.32	0.00	0.00
c3	0.23	0.50	1	0.30	0.15	0.00	0.00
c4	0.09	0.42	0.30	1	0.20	0.00	0.00
c5	0.57	0.32	0.15	0.20	1	0.00	0.00
tv	0.00	0.00	0.00	0.00	0.00	1	0.00
np	0.13	0.00	0.00	0.00	0.00	0.00	1

Expectably, DTDs representing the same object, i.e. `CUSTOMER`, have higher mutual similarities (the average similarity is 0.44) than similarities among DTDs representing different objects (the average for `NEWSPAPER` DTD is 0.13 and average for `TVSCHEDULE` DTD is only 0.03). The only one exception is between `CUSTOMER1` and `NEWSPAPER` due to their structural similarity.

In the second test we have used the same DTDs, but we have evaluated their similarities regardless semantic similarity. As we can see in Table 3, the resulting values are lower, however, the trend between same and different objects is same as in the first test.

Semantic Similarity. In the next set of tests we have focused on various parameters of the similarity measure using synthetic data. Firstly, we have dealt with semantic similarity. For this purpose, we defined three DTDs (see Figure 4) with exactly the same structure, but different element names. In addition, element names of the first and second DTD have similar meaning while the element names of the third DTD have no lexical meaning. The results are depicted in Table 4.

```
<!ELEMENT PERSON (DOMICILE, WORK)>
<!ELEMENT DOMICILE (STATE, TOWN)>
<!ELEMENT STATE (#PCDATA)>
<!ELEMENT TOWN (#PCDATA)>
<!ELEMENT WORK (#PCDATA)>
<!ATTLIST PERSON SURNAME CDATA
                #REQUIRED>
```

```
<!ELEMENT USER (RESIDENCE, JOB)>
<!ELEMENT RESIDENCE (COUNTRY, CITY)>
<!ELEMENT COUNTRY (#PCDATA)>
<!ELEMENT CITY (#PCDATA)>
<!ELEMENT JOB (#PCDATA)>
<!ATTLIST USER LASTNAME CDATA
                #REQUIRED>
```

```
<!ELEMENT AAA (BBB, DDD)>
<!ELEMENT BBB (EEE, FFF)>
<!ELEMENT DDD (#PCDATA)>
<!ELEMENT EEE (#PCDATA)>
<!ELEMENT FFF (#PCDATA)>
<!ATTLIST AAA CCC CDATA
                #REQUIRED>
```

Fig. 4. Synthetic DTDs for analysis of semantic similarity

As we can see, there is a significant difference in comparing the first two DTDs – they were correctly identified as almost similar when we used semantic similarity. Consequently, despite the semantic similarity is a time-consuming task due to necessary searching through thesaurus, it is enables to acquire more precise similarity values.

Table 4. Influence of semantic similarity

Semantic similarity	✓	×
`PERSON` x `USER`	0.92	0.40
`PERSON` x `AAA`	0.33	0.33

Table 5. Comparing different costs of edit operations

Cost	**1**	**5**	**10**	**100**
`USER1` x `USER2`	0.92	0.74	0.52	0.52

Edit Distance Operations. In the last test we have focused on two key edit operations used for transforming DTD trees, *InsertTree* and *DeleteTree*, proposed for transforming repeating structures of a tree. For the purpose of the test we have defined two similar DTDs depicted in Figure 5, whereas one of them involves shared elements. We have performed their similarity evaluation with different costs of edit operations *InsertTree* and *DeleteTree*.

```
<!ELEMENT USER (CUSTOMER, EMPLOYEE)>
<!ELEMENT CUSTOMER (USERDATA, ORDERS)>
<!ELEMENT EMPLOYEE (USERDATA, POSITION)>
<!ELEMENT USERDATA (ID, NAME, BIRTHDAY)>
<!ELEMENT ORDERS (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT BIRTHDAY (#PCDATA)>
```

```
<!ELEMENT USER (CUSTOMER)>
<!ELEMENT CUSTOMER (USERDATA, ORDERS)>
<!ELEMENT USERDATA (ID, NAME, BIRTHDAY)>
<!ELEMENT ORDERS (#PCDATA)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT BIRTHDAY (#PCDATA)>
```

Fig. 5. Synthetic DTDs for analysis of edit operations

As we can see in Table 5, in first two cases the operations were really used, but in the last two comparisons the costs for the operations were too high and the repeating tree structures were transformed using a sequence of single-node edit operations. Hence, the DTDs were correctly identified as similar only when the costs of the operations were set sufficiently low. This observation is similar to the observation made for edit distance algorithms used for XML documents.

5 Conclusion

The aim of this paper was a proposal of an algorithm for evaluating XML schema similarity on structural level. In particular, we have focused on DTDs which are still more popular than other languages for schema specification. We have combined two approaches and adapted them to DTD-specific structure – edit distance and semantic similarity. The exploitation of edit distance enables to analyze the structure of DTDs more precisely, whereas the semantic similarity enables to get more precise results, though at the cost of searching a thesaurus.

In our future work we will focus mainly on further improvements of our approach, such as other edit operations (e.g. moving a node or adding/deleting a non-leaf node) or XML Schema definitions that involve new constructs (e.g. unordered sequences of elements) as well as plenty of syntactic sugar.

References

1. M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB'00*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann.
2. E. Bertino, G. Guerrini, and M. Mesiti. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Inf. Syst.*, 29(1):23–46, 2004.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006.
4. H. H. Do and E. Rahm. COMA – A System for Flexible Combination of Schema Matching Approaches. In *VLDB'02*, pages 610–621, Hong Kong, China, 2002. Morgan Kaufmann.
5. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: Clustering XML Schemas for Effective Integration. In *CIKM'02*, pages 292–299, New York, NY, USA, 2002. ACM Press.
6. V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
7. T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *VLDB'98*, pages 122–133, San Francisco, CA, USA, 1998. Morgan Kaufmann.
8. I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing.
9. P. K.L. Ng and V. T.Y. Ng. Structural Similarity between XML Documents and DTDs. In *ICCS'03*, pages 412–421. Springer-Verlag, 2003.
10. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *WebDB'02*, pages 61–66, Madison, Wisconsin, USA, 2002.
11. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann.
12. Z. Zhang, R. Li, S. Cao, and Y. Zhu. Similarity Metric for XML Documents. In *FGWM'03*, Karlsruhe, Germany, 2003.