

# Computing Structural Similarity of Source XML Schemas against Domain XML Schema

Jianxin Li<sup>†</sup>  
Jixue Liu<sup>§</sup>

Chengfei Liu<sup>†</sup>  
Guoren Wang<sup>¶</sup>

Jeffrey Xu Yu<sup>‡</sup>  
Chi Yang<sup>†</sup>

<sup>†</sup> Swinburne University of Technology, Australia  
Email: {jili, cliu, cyang}@ict.swin.edu.au

<sup>‡</sup> Chinese University of Hong Kong, China  
Email: yu@se.cuhk.edu.hk

<sup>§</sup> University of South Australia, Australia  
Email: Jixue.liu@unisa.edu.au

<sup>¶</sup> Northeastern University, China  
Email: wanggr@mail.neu.edu.cn

## Abstract

In this paper, we study the problem of measuring structural similarities of large number of source schemas against a single domain schema, which is useful for enhancing the quality of searching and ranking big volume of source documents on the Web with the help of structural information. After analyzing the improperness of adopting existing edit-distance based methods, we propose a new similarity measure model that caters for the requirements of the problem. Given the asymmetric nature of the similarity comparisons of source schemas with a domain schema, similarity preserving rules and algorithm are designed to filter out uninteresting elements in source schemas for the purpose of optimizing the similarity computation. Based on the model, a basic algorithm and an improved algorithm are developed for structural similarity computation. The improved algorithm makes full use of a new coding scheme that is devised to reduce the number of comparisons. Complexities of both algorithms are analyzed and extensive experiments are conducted showing the significant performance gain achieved by the improved algorithm.

*Keywords:* Structural Similarity, XML Schema

## 1 Introduction

Since XML has become the standard for representing, exchanging and integrating data on the Web, more and more information or application data stored or exchanged on the Web is adhering to this format. Searching the Web for finding interesting services or information now becomes part of people's lives. The flexible structures of XML documents make this kind of search quite complex, and sometimes may impose relaxed conditions and return approximate results. Due to different perceptions of an application domain, different providers of a service may define different schemas which we call *source schemas* for their source data. Meanwhile, clients who require the service may issue queries based on the common understanding of

the domain which we call a *domain schema*. For example, Figure 1(a) shows a domain schema  $T_0$  for universities. The schema defines university, department, student, professor, library, campus name, and book as its interesting elements and their relationships. Figure 1(b) shows a source schema  $T$  for a particular university. Apart from all interesting elements in Figure 1(a), this schema has elements for faculty and campus which may not be interesting. There are also some structural differences between  $T_0$  and  $T$ , such as the relationship between departments and professors. Therefore, to be able to get results or approximate results directly from a source XML document, it is very important to work out the similarity of the source schema with regards to the domain schema. Recently, research in information retrieval on XML documents over the Web attracts a lot of attention. Instead of using a pure CO (content only) query, we may now use a so called CAS (content-and-structure) query (Gövert & Kazai 2002) to express the topic statement more precisely by adding explicit references to XML structure, by restricting either the context of interest or the context of certain search concepts. In this kind of search, the structural similarity of the source schemas compared with the domain schema against which a CAS query is issued is again a key point for ranking the set of source XML documents. The problem is how to efficiently compute the structural similarities of the potentially huge number of candidate source schemas against a domain schema.

In recent years, a great deal of attention has been put on computing the structural similarity between XML documents (Cobena et al. 2002, Flesca et al. 2002, Bertino et al. 2004). Work has also been presented on matching XML schemas for schema mapping and integration (Madhavan et al. 2001, Do & Rahm 2002). To the best of our knowledge, none of them is proposed to tackle the problem that we propose in this paper. The structural similarity problem of our interest is to compare source schemas with a domain schema for the purpose of searching and ranking those source documents that conform to their corresponding source schemas. Queries are issued against the domain schema and the returned source documents are ranked based on the similarities between their corresponding source schemas and the domain schema. Therefore, the problem is somehow asymmetric for the schemas to be compared. It takes as input two schemas,  $T$  and  $T_0$  representing a candidate source XML schema and the domain XML schema, re-

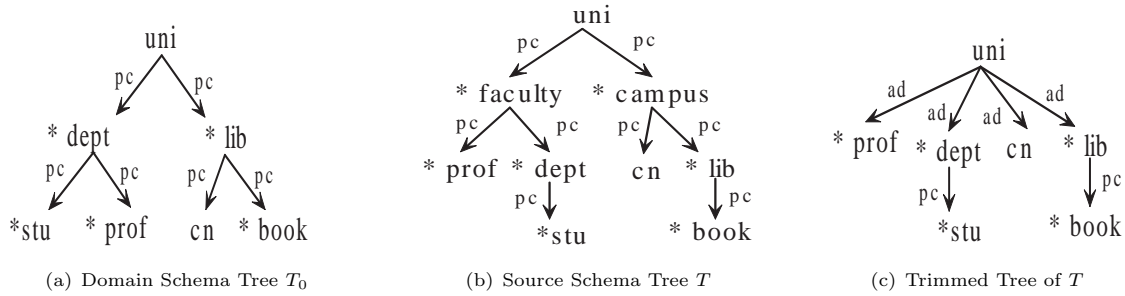


Figure 1: Extracting Interesting Structure from Source Schema based on Domain Schema

spectively. We are able to take  $T_0$  as a base to trim  $T$  first, and then compare their structures. For example, the source schema  $T$  in Figure 1(b) can be trimmed based on the domain schema  $T_0$  in Figure 1(a), yielding a trimmed schema as shown in Figure 1(c). In this paper, we present a framework and algorithms to measure the structural similarity of  $T$  with regards to  $T_0$ .

The contributions of this paper are as follows:

- We propose a new similarity measure model to compute the structural similarity degree of any candidate source XML schema against a given domain XML schema after analyzing the requirements of the problem and the imperpness of adopting existing edit-distance based methods to tackle the problem (Section 3).
- Given the asymmetric nature of the similarity comparisons of schemas to be compared, we design several similarity preserving rules and an algorithm to filter out uninteresting elements from source schemas for the purpose of optimizing the similarity computation (Section 4).
- We develop a basic algorithm and an improved algorithm to compute the structural similarity between a source schema and a domain schema with complexity analysis. An efficient coding scheme is devised to speed up the similarity computation and is fully used in the improved algorithm (Section 5). Experimental results show the significant performance gain of the improved algorithm (Section 6).

A discussion of related work and conclusions of this paper are provided in Section 2 and Section 7, respectively.

## 2 Related Work

At the instance level, a lot of studies measure the similarity between XML document trees using edit distance (Cobena et al. 2002), e.g. detecting the required changes from one XML document to another rather than directly comparing them on the basis of their structure. Usually, there are three kinds of basic edit operations on tree nodes: re-labeling, deleting, and inserting. As mentioned in (Zhang & Shasha 1989), a mapping between two trees can specify the edit operations applied to each node in the two trees graphically. Cobena et al. (Cobena et al. 2002) considered, besides the above operations, a move operation on sub-trees, which is essential in the context of XML, and used signatures to match sub-trees that were left unchanged between the old and new versions. Bertino et al. (Bertino et al. 2004) exploited a graph-matching algorithm to associate elements in the XML document with element definitions in the DTD. Flesca et al. (Flesca et al. 2002) described

the structure of an XML document into a time series where each occurrence of a tag corresponds to a given impulse. By analysing the frequencies of the corresponding Fourier transform (Rafiei & Mendelson 1998), they can state the degree of similarity between documents. Yang et al. in (Yang et al. 2005) transformed tree-structured data into corresponding binary trees, and then encoded the binary trees and generated the corresponding approximate numerical multidimensional vectors to compute similarity.

At the schema level, different methods computing XML schema similarity have been studied for the purpose of generating qualified schema matching. Cupid (Madhavan et al. 2001) proposed a hybrid match approach combining a name matcher with a structural match algorithm. It derived the similarity of elements based on the similarity of their components hereby emphasising the name and data type similarities presented at the leaf level. Its structural algorithm only considered the elements with the same number of leaves and immediate descendants. XClust (Lee et al. 2002) and Similarity Flooding (Melnik et al. 2002) took similar approaches. XClust also considered the cardinality of elements. Fixed-point computation is used in Similarity Flooding. Yi and Weng et. al. (Yi et al. 2005) represented XML schemas by constructing a universal semantic model and then compared the generated models to compute the similarity between XML schemas. The semantic model consisted of three components: ontological categories, properties and contextual constraints. In addition, the operation of relaxation labeling was devised to improve the quality of schema matching. Formica (Formica 2007) computed the similarity of XML schema elements by combining two parts: the maximum information content that is measured by the minimal common type to be shared and their own type declarations. COMA (Do & Rahm 2002) is a generic schema match system which provided an extensible library of simple and hybrid match algorithms and supported a framework for combining the match results obtained from different algorithms. In the system, the similarity between the elements was recursively computed from the similarity between their respective children with a leaf-level matcher. To support efficient computation of schema similarity, Duta (Duta et al. 2006) proposed seven reduction rules to transform XML schemas into minimum structures capable of storing the same information and preserved the cardinality constraints of leaf nodes. Rahm and Bernstein (Rahm & Bernstein 2001) did a comprehensive survey of some early work in schema matching.

In this work, we are motivated to compute the similarity of multiple source schemas against a given domain schema for the purpose of querying source documents conforming to the corresponding source schemas based on the domain schema. The similarity computation is somewhat asymmetric that is different from all the above work. As such, we are more con-

cerned with those elements appeared in the domain schemas and their relationships such as an ancestor-descendant relationship because they are important in formulating a query. In addition, any pair of elements in corresponding XML schemas will be compared in our similarity model and the index is also introduced to improve the efficiency of computation.

### 3 Measuring Structural Similarity

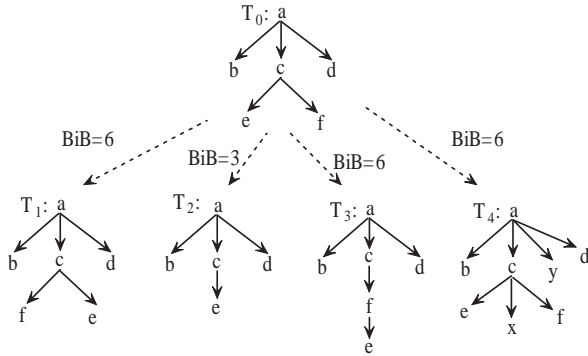


Figure 2: A Motivating Example

In this section, we present a framework for measuring the structural similarity of a candidate source schema against a domain schema. First we use a motivating example to illustrate what are required for the similarity problem we are targeting and why an existing edit-distance method is not suitable to the problem. Following the discussions, we propose a new similarity measure model. Finally we justify that the proposed model is effective for solving the similarity problem by reviewing the motivating example and comparing with the edit-distance method.

#### 3.1 A Motivating Example

To motivate our work, let us look at an example shown in Figure 2. In the example, a domain schema tree  $T_0$  and four source schema trees  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_4$  are given. If we use an edit-distance based method such as *BiBranch* proposed in (Yang et al. 2005) to compute the structural similarity, the similarity values of  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_4$  against  $T_0$  are 6, 3, 6, and 6, respectively. They are shown in Figure 2 denoted as *BiB*. In *BiBranch*, the smaller the *BiB* value is, the more similar its corresponding pair of trees are.

There are several problems in the results calculated using *BiBranch*.

- $BiB(T_0, T_1) = 6$  is not correct. Instead, we would expect  $BiB(T_0, T_1) = 0$ . In other words, when we query a source document conforming to  $T_1$  based on  $T_0$ , we do not care about the order of the sibling elements  $e$  and  $f$  and the results retrieved from those source documents conforming to  $T_1$  should be ranked among the highest.
- $BiB(T_0, T_2) = 3$  and  $BiB(T_0, T_3) = 6$  are not expected. Instead, we would expect  $BiB(T_0, T_2) > BiB(T_0, T_3)$ . In other words,  $T_3$  is more similar to  $T_0$  than  $T_2$ . When searching documents, it is most important that all interesting elements are not missed out. Missing an element will impact more on the similarity than placing the element in an inconsistent position. In  $T_2$  element  $f$  is missing while in  $T_3$  only the relationship between  $e$  and  $f$  is inconsistent with that in  $T_0$ . The relationship between  $c$  and  $e$  is slightly

different where a parent-child (pc) relationship (“/”) holds in  $T_0$  while an ancestor-descendant (ad) relationship (“//”) holds in  $T_3$ . When a query containing element  $f$  is issued against  $T_0$ , no result will be returned from those source documents conforming to  $T_2$ . However, results will be returned from those source documents conforming to  $T_3$  as long as the relationship between  $c$  and  $e$  is not strictly a pc relationship in the query.

- $BiB(T_0, T_4) = 6$  is not correct either. Instead, we would also expect  $BiB(T_0, T_4) = 0$ . In other words, when we query a source document conforming to  $T_4$  based on  $T_0$ , we do not care about whether the source document contains some “noise” elements that do not impact on the query results. Again, the results retrieved from those source documents conforming to  $T_4$  should be ranked among the highest.

From the above analysis, it is obvious that edit-distance based methods are not suitable for solving our schema similarity problem and a new similarity measure model is needed. To compute the similarity of a source schema against a domain schema, the new model is required to first prioritize the coverage in the source schema of every interesting element appearing in the domain schema and then to consider the consistency of the relationships between any pair of interesting elements in both schemas. From the example, we have pc, ad, and sibling relationships. We also know that the order of sibling elements do not matter and that “noise” elements in a source schema can be ignored if they do not affect the relationships of any pair of elements in the schema.

#### 3.2 Structural Similarity Model SSD

In this section, we present a new similarity model for measuring similarity of source schemas against a domain schema (*SSD*). The *SSD* model provides accurate similarity measures by taking into account two main factors that contribute to the structural similarity or difference: element coverage and consistency of relationships of element pairs. In addition, we also consider the difference of element cardinality in the model. We choose the similarity degree value in  $[0, 1]$ . Unlike *BiBranch*, the bigger the *SSD* value is, the more similar a source schema is with regards to the domain schema.

Both the source schemas and the domain schema are represented as schema trees. We first define a schema tree.

**Definition 1 XML Schema Tree:** An XML schema tree is defined as  $T = (V, E, v_r, Card)$  where

- $V$  is a finite set of nodes, representing elements and attributes of the schema.
- $E$  is a set of directed edges. Each edge  $e(v_1, v_2)$  represents the parent-child relationship between two nodes  $v_1, v_2 \in V$ , denoted by  $P(v_2)=v_1$  or  $v_2 \in Ch(v_1)$  where  $P : V \rightarrow V$ , for any  $v \in V$  and  $v \neq v_r$ ,  $P(v)$  is the parent node of  $v$ ;  $Ch : V \rightarrow 2^V$ , for any  $v \in V$ ,  $Ch(v)$  is a set of child nodes of  $v$ .
- $v_r \in V$  is the root node of tree  $T$ .
- $Card : V \rightarrow \{“1”, “*”\}$  where  $Card(v) = “1”$  represents that there is only one occurrence of  $v$  under  $P(v)$  in a document conforming to  $T$ ;  $Card(v) = “*”$  represents that there are more than one occurrences of  $v$  under  $P(v)$ .

Given a domain schema tree  $T_0 = (V_0, E_0, v_{r0}, Card)$  and a source schema tree  $T = (V, E, v_r, Card)$ , we need to compute  $SSD(T_0, T)$ . The first contributing factor is the element coverage. This can be calculated by the ratio of interesting objects (RIO) showing the proportion of interesting elements of  $T_0$  in  $T$ , which is calculated by the following formula.

**Ratio of Interesting Object ( $RIO(V_0, V)$ )**

$$RIO = \frac{|V'|}{|V_0|} \quad (1)$$

where  $V' = V \cap V_0$  is the set of interesting nodes in  $V$ .

Compared with the domain schema tree  $T_0$  in Figure 2, the source schema tree  $T_4$  contains all the interesting elements while  $T_2$  includes 5 interesting nodes. So we have  $RIO(V_0, V_4) = 6/6 = 1$  and  $RIO(V_0, V_2) = 5/6 = 0.833$ , respectively.

The second contributing factor is the consistency degree of all node pairs in  $T$  compared with the corresponding node pairs in  $T_0$ . Before we discuss the similarity of node pairs, we first discuss the cardinality that may affect the similarity of node pairs. In a schema tree, the cardinality of each element is recorded. Based on this cardinality, we can derive the relative cardinality of a pair of nodes in the schema tree.

**Definition 2 Relative Cardinality:** Given a schema tree  $T = (V, E, v_r, Card)$  and any two nodes  $v_1, v_2 \in V$  such that there exists a path from  $v_1$  to  $v_2$ , we define the relative cardinality between  $v_1$  and  $v_2$  as  $RCard(v_1, v_2)$  where  $RCard(v_1, v_2)$  is set to "1" if every node  $v$  on the path from  $v_1$  to  $v_2$  satisfies  $Card(v) = "1"$ ; otherwise,  $RCard(v_1, v_2)$  is set to "\*".

Given a pair of nodes  $(v_1, v_2)$  in  $V'$  and its counterpart  $(v_{01}, v_{02})$  in  $V_0$ , we can first define the cardinality similarity of node pairs (CSNP) indicating the cardinality difference between node pairs that satisfy the ad or pc relationships.

**Cardinality similarity of node pairs**  
( $CSNP(v_1, v_2, v_{01}, v_{02})$ )

$$CSNP = \begin{cases} \omega, & RCard(v_1, v_2) \neq RCard(v_{01}, v_{02}) \\ 1, & RCard(v_1, v_2) = RCard(v_{01}, v_{02}) \end{cases} \quad (2)$$

CSNP is set to 1 when the two node pairs have consistent relative cardinality; otherwise it is set to  $\omega$ . Here,  $\omega \in [0, 1]$  represents the degree that clients can tolerate with the cardinality difference. We permit clients to evaluate the effect of the cardinality on the similarity by adjusting the value of  $\omega$ . Normally this value is relatively high. We assume its default value is 0.8.

For any pair of nodes, they must have one of the three relationships: a pc relationship, an ad relationship, or an extended sibling relationship defined as follows.

**Definition 3 eSibling:** Given a pair of nodes  $(v_1, v_2)$ , if neither  $pc(v_1, v_2)$  or  $ad(v_1, v_2)$ , nor  $pc(v_2, v_1)$  or  $ad(v_2, v_1)$  hold,  $v_1$  and  $v_2$  are said to satisfy an extended sibling relationship and this relationship is denoted as  $eSibling(v_1, v_2)$ .

Now, we define the similarity of node pairs (SNP) that specifies the structural relationship between node pairs.

**Similarity of Node Pairs**

$$(SNP(v_1, v_2, v_{01}, v_{02}))$$

$$SNP = \begin{cases} CSNP(v_1, v_2, v_{01}, v_{02}), & \text{case 1} \\ \lambda \times CSNP(v_1, v_2, v_{01}, v_{02}), & \text{case 2} \\ 1, & \text{case 3} \\ 0, & \text{case 4} \end{cases} \quad (3)$$

where Case 1:  $((v_{01}/v_{02}) \wedge (v_1/v_2)) \vee ((v_{01}/v_{02}) \wedge (v_1//v_2))$  means if the node pairs satisfy the pc or ad relationships at the same time, we can directly compute the value of SNP according to the CSNP value of the node pairs. Case 2:  $((v_{01}/v_{02}) \wedge (v_1//v_2)) \vee ((v_{01}/v_{02}) \wedge (v_1/v_2))$  means that one node pair satisfies the pc relationship and the other satisfies the ad relationship, in this case, clients may choose to adjust the parameter  $\lambda \in [0, 1]$  which represents the degree that clients can tolerate the difference between "/" and "//". We assume its default value is also 0.8. Case 3:  $eSibling(v_1, v_2) \wedge eSibling(v_{01}, v_{02})$  means the node pairs  $(v_1, v_2)$  and  $(v_{01}, v_{02})$  are structurally consistent in that both of the node pairs have an extended sibling relationship and the value of SNP is set to 1. Case 4: if the above three cases do not hold, the value of SNP will be set to 0 representing that the node pairs are not matched, e.g., one pair satisfies pc/ad relationship while the other satisfies  $eSibling$  relationship.

Given a node pair  $(c, e)$  in  $T_0$  and its counterpart in  $T_3$  shown in Figure 2, it is easy to see  $(c, e)$  satisfies a pc relationship in  $T_0$  while its counterpart satisfies an ad relationship in  $T_3$ . And they have the same relative cardinality due to  $RCard(c, e) = 1$  in both sides. So we have  $SNP = \lambda \times 1 = 0.8$  where we use the default value of  $\lambda$ . If one of them changed its relative cardinality, i.e. their relative cardinality were not same, the value of SNP would be  $\lambda \times \omega = 0.8 \times 0.8 = 0.64$ .

Now we provide the SSD similarity value between  $T_0$  and  $T$ , which combines both contributing factors.

**Similarity of source schema w.r.t. domain schema ( $SSD(T_0, T)$ )**

$$SSD = RIO(V_0, V) \times \left( \frac{1}{C_{|V'|}^2} \sum SNP(v_1, v_2, v_{01}, v_{02}) \right) \quad (4)$$

In the above equation, the second contributing factor is calculated by taking into account similarities of all corresponding node pairs in  $V'$  and  $V_0$ . After we substitute Equation 1 and  $C_{|V'|}^2 = \frac{|V'| \times (|V'| - 1)}{2}$  into Equation 4, we can get the final similarity model:

$$SSD = \frac{2}{|V_0| \times (|V \cap V_0| - 1)} \sum SNP(v_1, v_2, v_{01}, v_{02}) \quad (5)$$

### 3.3 Effectiveness Analysis

Come back to the motivating example in Section 3.1, now we can compute the similarity values of  $T_1$ ,  $T_2$ ,  $T_3$  and  $T_4$  against  $T_0$  using Equation 5. For example, given the source schema tree  $T_3$  and the domain schema tree  $T_0$ , we have  $|V_0| = |V_3 \cap V_0| = 6$ , so  $RIO = 1$  because all the elements in  $T_0$  can be found in  $T_3$ . Then we check if the relationship between every two elements in  $T_0$  is consistent with its counterpart in  $T_3$ . From Figure 2, differences can be found for two node pairs  $(c, e)$  and  $(f, e)$ . For  $(c, e)$ , a pc relationship holds in  $T_0$  while an ad relationship holds in  $T_3$ , so it contributes to 0.8 (the default value of  $\lambda$ ). For  $(f, e)$ , an  $eSibling$  relationship holds in  $T_0$  while a pc relationship holds in  $T_3$ , so it contributes to 0 because

the inconsistency of the node pairs. It is easy to calculate  $\sum SNP(v_1, v_2, v_{01}, v_{02}) = 13 + 0.8 + 0 = 13.8$  out of the total of 15 node pairs. Now, we can get the final result  $SSD(T_0, T_3) = \frac{2}{6 \times (6-1)} \times 13.8 = 0.92$ .

According to the same procedure, we have  $SSD(T_0, T_1) = 1$ ,  $SSD(T_0, T_2) = 0.556$  and  $SSD(T_0, T_4) = 1$ , respectively.

Based on the above  $SSD$  results, we can see that the  $SSD$  similarity values of  $T_1$  and  $T_4$  against  $T_0$  are all 1. These two similarities are exactly what we expected. We can also see that the  $SSD$  similarity value of  $T_3$  against  $T_0$  (0.92) is much higher than that of  $T_2$  against  $T_0$  (0.556). This reflects that a missing element will affect more on the similarity than misplacing an element, which is what we expected. Compared with *BiBranch*,  $SSD$  is much more effective when computing the similarity of a source schema against its domain schema.

## 4 Similarity Preserving Trimming

In the  $SSD$  model, the “noise” elements of a source schema are not counted directly for the similarity computation because of the asymmetric nature of the similarity problem. To reduce the number of comparisons, it is desirable to filter out those “noise” elements while preserving the similarity of the source schema against the domain schema. In this section, we propose a set of trimming rules and an algorithm based on these rules to filter out all “noise” elements in a source schema based on the domain schema. We also prove that the order of applying rules is insignificant and the algorithm as well as each rule preserve the similarity property.

### 4.1 Basic Trimming Rules

Let  $T(V, E, v_r, Card)$  and  $T_0(V_0, E_0, v_{r0}, Card)$  be the XML schema trees for a source schema and a domain schema, respectively. As we discussed above, only those nodes in  $T_0$  and their relationships are interesting to clients. As such, we are able to take  $T_0$  as a base to trim  $T$  by deleting the uninteresting nodes  $v \in \{V - V_0\}$ . When a node is to be deleted, the edges linked to the node should be changed. According to the location of  $v$  in  $T$ , we have the following three updating rules in Figure 3:

- **Rule 1:** If  $v = v_r$  (node  $v$  is the root node), then all the edges  $\{(v, v_i) | v_i \in Ch(v)\}$  need to be deleted.
- **Rule 2:** If  $Ch(v) = \phi$  (node  $v$  is a leaf node), then the edge  $\{(P(v), v)\}$  needs to be deleted.
- **Rule 3:** If  $P(v) \neq \phi \wedge Ch(v) \neq \phi$  (node  $v$  is an internal node), then: (1) for all  $v_i \in Ch(v)$ , the cardinality of  $v_i$  is updated as  $\max\{Card(v), Card(v_i)\}$ ; (2) all edges relating to node  $v$ , i.e.  $\{(v, v_i) | v_i \in Ch(v)\} \cup \{(P(v), v)\}$  are deleted from  $E$ ; (3) new edges linking the parent node and all the child nodes, i.e.  $\{(P(v), v_i) | v_i \in Ch(v)\}$  are inserted into  $E$  and re-labeled to ad-edges.

The trimmed result remains as a schema tree if the remaining nodes are all connected, or becomes a schema forest containing multiple trees (e.g. we may employ a virtual root to connect the multiple trees in the schema forest). For example, we take  $T_0$  in Figure 1(a) as a base to trim  $T$  in Figure 1(b), the trimmed tree of  $T$  is shown in Figure 1(c).

**Definition 4 Trimming Schema Forest:** Given a domain XML schema tree  $T_0(V_0, E_0, v_{r0}, Card)$

and a candidate source XML Schema tree  $T(V_t, E_t, v_{rt}, Card)$ , the trimmed result is represented as a forest  $F(V, E, V_r, Card)$ , where

- $V = V_0 \cap V_t$ .
- $E$  is the set of edges after applying one of Rules 1-3 for each  $v \in \{V_t - V_0\}$ .  $E = E_c \cup E_d$  consists of two kinds of edges called pc-edges ( $E_c$ ) and ad-edges ( $E_d$ ), corresponding to the child and descendant axes of  $XPath$ .
- $V_r$  is the set of root nodes of the trees resulted from deleting  $v_{rt}$  and some top level nodes in  $\{V_t - V_0\}$ .  $|V_r|$  represents the number of trees in  $F$ .
- $Card$  is defined the same as that in Definition 1.

**Theorem 1** Let  $T_0(V_0, E_0, v_{r0}, Card)$  and  $F(V, E, V_r, Card)$  be a domain XML schema tree and a source schema forest (or tree). Each of the rule in Rule 1 to Rule 3 preserves similarity when applying to  $F$  based on  $T_0$  in terms of the  $SSD$  model.

**Proof 1** Suppose that the transformed source schema forest is  $F'(V', E', V'_r, Card)$  after applying a rule in Rule 1 to Rule 3, we prove that  $SSD(T_0, F') = SSD(T_0, F)$  in two parts. The first part is  $RIO$ . Obviously  $F'$  preserves all node in  $V \cap V_0$  because Rule 1 to Rule 3 only filters out a node in  $v \in \{V - V_0\}$ . So we have  $RIO(T_0, F') = RIO(T_0, F)$ . The second part is  $SNP$ . For any two nodes  $v_1, v_2 \in V \cap V_0$ , we show that the relationship between  $v_1$  and  $v_2$  remains unchanged. For Rule 1/Rule 2, when a root/leaf node  $v$  is removed, obviously there is no change to the relationship of any remaining node with the other nodes. For Rule 3, when an intermediate node  $v$  is removed,  $ad(v_1, v_2)$  and  $ad(v_1, v_3)$  remain unchanged because the new edges  $(v_1, v_2)$  and  $(v_1, v_3)$  are re-labeled as ad-edges. Furthermore,  $RCard(v_1, v_2)$  and  $RCard(v_1, v_3)$  are also adjusted accordingly. So  $F$  and  $F'$  agree on the  $SNP$  value for any node pair  $(v_1, v_2)$  where  $v_1, v_2 \in V \cap V_0$ .

### 4.2 Trimming Algorithm

Algorithm 1 gives the whole trimming process in a top-down manner. The queue *tempQueue* is used to hold elements waiting to be processed. At the beginning of each loop, we use the function *GetElement()* to get an element  $v$  from *tempQueue* and insert its child elements into *tempQueue*. And then we check if  $v$  is in  $V_0$  of the domain schema. If it does then if it is also a sub-root element we insert  $v$  into  $V_r$ . If  $v$  is not in  $V_0$ , there are three trimming cases: (1)  $v$  is a sub-root element - all the edges coming from  $v$  will be deleted (Lines 15-17, 30-31). (2)  $v$  is a leaf element - all the edges arriving to  $v$  will be deleted (Lines 18-20, 30-31). (3)  $v$  is an internal element - all the edges connecting to  $v$  will be deleted while a new set of ad-edges will be created and inserted into edge set  $E$  (Lines 21-29, 30-31).

**Corollary 1** Let  $T_0(V_0, E_0, v_{r0}, Card)$  be a domain schema tree,  $T(V_t, E_t, v_{rt}, Card)$  be a candidate source Schema tree. Let  $F(V, E, V_r, Card)$  the trimmed result source schema forest after applying a series of rules in Rule 1 to Rule 3. Then the order of applying these rules in Rule 1 to Rule 3 is not significant.

This can be inferred from Theorem1 because each application of a rule in Rule 1 to Rule 3 preserves similarity.

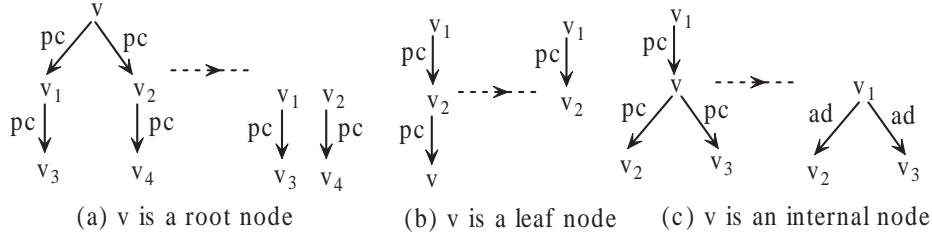


Figure 3: Trimming Rules

---

**Algorithm 1:** Trimming Tree Algorithm

---

**Input:** A domain schema tree  $T_0(V_0, E_0, v_r, Card)$  and a source schema tree  $T(V_t, E_t, v_{rt}, Card)$   
**Output:** A forest  $F(V, E, V_r, Card)$

- 1  $V = V_t;$
- 2  $E = E_t;$
- 3 push  $v_{rt}$  into a temporary queue  $tempQueue;$
- 4 **while**  $tempQueue \neq \phi$  **do**
- 5    $v = tempQueue.GetElement();$
- 6   **if**  $Ch(v) \neq \phi$  **then**
- 7     insert  $\{v_i \in Ch(v)\}$  into  $tempQueue;$
- 8   **end**
- 9   **if**  $v \in V_0$  **then**
- 10    **if**  $P(v) = \phi$  **then**
- 11     insert  $v$  into  $V_r;$
- 12    **end**
- 13   **end**
- 14   **else**
- 15    **if**  $P(v) = \phi$  **then**
- 16      $E = E - \{e(v, v_i) | v_i \in Ch(v)\};$
- 17    **end**
- 18    **else if**  $Ch(v) = \phi$  **then**
- 19      $E = E - \{e(P(v), v)\};$
- 20    **end**
- 21    **else**
- 22     **foreach**  $v_i \in Ch(v)$  **do**
- 23       $Card(v_i) =$   
 $\max\{Card(v), Card(v_i)\};$   
 $E = E - \{e(v, v_i)\};$   
 $E_d = E_d + \{e(P(v), v_i)\};$   
 $P(v_i) = P(v);$   
 $Ch(P(v)) = Ch(P(v)) + v_i;$
- 28     **end**
- 29    **end**
- 30     $E = E - \{e(P(v), v)\};$
- 31     $V = V - v;$
- 32   **end**
- 33 **end**

---

**Corollary 2** *Algorithm1 correctly trims a source schema tree  $T(V_t, E_t, v_{rt}, Card)$  based on the domain schema tree  $T_0(V_0, E_0, v_{r0}, Card)$ .*

First, Algorithm1 filters out all “noise” nodes that are in  $V_t$  but not in  $V_0$ . Second, from Theorem 1 and Corollary2, we know that the order of applying rules is not significant and the trimmed schema forest preserves the similarity of  $T$  against  $T_0$ .

## 5 Computing Similarity

Given a candidate source schema  $T$  and a domain schema  $T_0$ , we can take  $T_0$  as a base to trim  $T$  into a schema forest  $F$  by applying Algorithm 1. In this section, we compute the similarity of  $F$  against  $T_0$ . To

---

**Algorithm 2:** Coding A Forest

---

**Input:** A forest  $F(V, E, V_r, Card)$   
**Output:** A vector  $Vec$  of encoded nodes( $TreeID, pre, post, C, P, RD$ )

- 1  $TreeID = pre = post = 0;$
- 2 new stack  $S;$
- 3 **foreach** root node  $v$  in  $V_r$  **do**
- 4    $C = 0;$
- 5    $TreeID ++;$
- 6    $DFTraverse(TreeID, v);$
- 7 **end**
- 8 Return the vector  $Vec;$

---

speedup the computation, we develop a new coding scheme and algorithm to code  $F$  and  $T_0$  first. Based on the coding information, we develop two algorithms with complexity analysis.

### 5.1 Encoding XML Schema

Most of the previous coding schemes are used to quickly determine the structural relationship among any pair of tree nodes. To the best of our knowledge, it is Dietz’s numbering scheme that was firstly used to determine ad relationship between tree nodes (Dietz 1982) where each node is assigned with a pair of numbers (pre, post). His proposition is: given two nodes  $x$  and  $y$  of a tree  $T$ ,  $x$  is an ancestor of  $y$  iff  $x$  occurs before  $y$  in the preorder traversal of  $T$  and after  $y$  in the postorder traversal, i.e.  $x.pre < y.pre$  and  $x.post > y.post$ . Region coding scheme is another popular scheme adopted in many work (Chien et al. 2002, Jiang et al. 2003) with a pair of numbers ( $start, end$ ). Element  $x$  is the ancestor of element  $y$  iff  $x.start < y.start$  and  $y.end < x.end$ . In order to reduce update cost, Li et. al. (Li & Moon 2001) proposed a variant in the form of ( $order, size$ ) that reserves additional code space for elements.

However, most of the previous coding schemes were used to improve the query efficiency of individual XML documents, rather than to serve for comparison of a pair of schemas. To this end, we propose a coding scheme that extends Dietz’s numbering scheme for specifying more information for schema comparison.

**Definition 5 Coding Scheme:** *Any node  $v$  in a schema forest  $F$  can be represented with a tuple ( $pre, post, C, P, RD$ ), where*

- $pre$ : represents the position of  $v$  when it is traversed in pre-order.
- $post$ : represents the position of  $v$  when it is traversed in post-order.
- $C$ : records the cardinality information of  $v$ . If  $Card(v) = “*”$ ,  $v$ ’s  $pre$  is recorded; otherwise, if  $\exists v_a$  ( $v_a$  is  $v$ ’s nearest ancestor and  $Card(v_a) = “*”$ ),  $v_a$ ’s  $pre$  is recorded; otherwise, let  $C = 0$ .

---

**Algorithm 3:**  $DFTraverse(TreeID, v)$ 

---

```
1  $v.TreeID = TreeID;$ 
2  $pre = pre + 1;$ 
3  $v.pre = pre;$ 
4  $v.RD = pre;$ 
5 if  $Card(v) = "*" \text{ then}$ 
6    $v.C = v.pre;$ 
7    $C = pre;$ 
8 end
9 else
10   $v.C = C;$ 
11 end
12 if  $Ch(v) \neq \phi \text{ then}$ 
13    $Push(v, S);$ 
14   while  $Ch(v) \neq \phi \text{ do}$ 
15      $x = GetLeftNode(Ch(v));$ 
16      $Ch(v) = Ch(v) - \{x\};$ 
17      $DFTraverse(TreeID, x);$ 
18   end
19   if  $S \neq \phi \text{ then}$ 
20      $v = pop(S);$ 
21   end
22 end
23 if  $S \neq \phi \text{ then}$ 
24    $x = pop(S);$ 
25    $v.P = x.pre;$ 
26    $x.RD = v.RD;$ 
27    $C = x.C;$ 
28    $Push(x, S);$ 
29 end
30 else
31    $v.P = null;$ 
32 end
33  $post ++;$ 
34  $v.post = post;$ 
35  $Vec[v.pre] = v;$ 
```

---

- $P$ : records the parent's  $pre$  of  $v$ .
- $RD$ : records the rightmost descendant's  $pre$  of  $v$ .

We propose a coding scheme for determining the relationships between elements and locating the range of elements that need to be compared. It has many appealing features: (1) Given a forest of  $n$  nodes, the  $pre$  of all nodes is in the continuous range of  $[1, n]$ , therefore, the  $pre$  of a node can be served as the index for the node; (2) Useful information for schema comparison is recorded, including one index for parents ( $P - Index$ ) and another for the rightmost descendant ( $RD - Index$ ); (3) Cardinality information is also preserved.

The coding information can be used to optimize similarity optimization. For example, ad-relationship can be easily determined from the codes of a node pair. Definition 2 can be simplified as follows.

**Relative Cardinality:** Given any two nodes  $v_i, v_j \in V$ , if there exists a path from  $v_i$  to  $v_j$ , then

$$RCard(v_i, v_j) = \begin{cases} "1", & v_i.C = v_j.C \\ "*", & v_i.pre < v_j.C \leq v_j.pre \end{cases} \quad (6)$$

where  $v_i.C = v_j.C$  means that  $Card(v) = "1"$  holds for any node  $v$  on the path from  $v_i$  to  $v_j$ ;  $v_i.pre < v_j.C \leq v_j.pre$  means that either  $Card(v_j) = "*" \text{ holds or } Card(v) = "*" \text{ holds for any node } v \text{ on the path from } v_i \text{ to } v_j$ .

Algorithm 2 first initializes several global variables, including  $pre$  and  $post$  that are used to assign  $pre$  and  $post$  codes to nodes. Then it traverses

---

**Algorithm 4:** Basic Algorithm  $BA$ 

---

```
Input: node sets  $V$  and  $V_0$  with coding information
Output: Similarity  $simi$ 
1  $SimiOfPair = 0;$ 
2  $n = |V|;$ 
3  $n_0 = |V_0|;$ 
4 for  $(i = 1; i < n; i ++)$  do
5    $v_1 = V[i];$ 
6    $v_{01} = V_0[v_1.m];$ 
7   for  $(j = i + 1; j \leq n; j ++)$  do
8      $v_2 = V[j];$ 
9      $v_{02} = V_0[v_2.m];$ 
10     $SimiOfPair += SNP(v_1, v_2, v_{01}, v_{02});$ 
11  end
12 end
13 return  $simi = \frac{2 \times SimiOfPair}{n_0 \times (n-1)};$ 
```

---

each tree in the schema forest  $F$  in a depth-first manner and encodes each node according to Definition 5. For each tree, we get the root of the tree from  $V_r$  and assign it a number  $TreeID$  identifying the tree. Then we call the recursive function  $DFTraverse()$  with  $TreeID$  and the root as inputs to encode the nodes in the tree into a vector. Finally, the vector will be returned with all the encoded nodes.

In Algorithm 3, a recursive function  $DFTraverse()$ , with a tree id  $TreeID$  and the root node  $v$  as inputs, is used to traverse the tree (or sub-tree)  $T$  in a depth-first manner and to encode the nodes in  $T$ . Lines 2-4 increase the  $pre$  by 1 and then assign the value of  $pre$  as the  $pre$  code of  $v$  ( $v.pre$ ) and as the  $RD$  code of  $v$  ( $v.RD$ ) temporarily. Lines 5-11 are used to assign the  $C$  code of  $v$  ( $v.C$ ) as either the  $pre$  code of itself ( $v.pre$ ) if  $Card(v) = "*" \text{ or the } C \text{ value (0 as initial value) carry-forwarded from its parent. Notice, if } Card(v) = "*" \text{, } C \text{ will record the } pre \text{ code of } v \text{ and be passed down the tree as the } C \text{ code of its descendants possibly. Lines 12-22 check if } v \text{ has child nodes. If it does, } v \text{ is pushed to stack } S \text{, then recursively process its child nodes. After all its child nodes have been processed, pop up } v \text{. Line 23-29 are used to assign the } P \text{ code of } v \text{ and pass } RD \text{ code to its parent node. In case } v \text{ has no parent node, the } P \text{ code of } v \text{ is set to } null \text{ (Lines 30-32). Lines 33-35 are used to increase the global variable } post \text{ and then set the } post \text{ code of } v \text{ (} v.post \text{).}$

## 5.2 Basic Algorithm

Let  $V$  and  $V_0$  be sets of nodes of  $F$  and  $T_0$ , respectively. A node-to-node map  $m : V \rightarrow V_0$  is required for similarity computation. This map can be built by using an element-level matcher (Li & Clifton 2000), structure-level matcher (Bergamaschi et al. 1999), or linguistic matcher. These techniques compare the attributes, combinations or names of elements, and other textual descriptions respectively for finding the correspondent elements. We can treat  $m$  as another attribute of each  $v \in V$ . By using this attribute, we can conduct pair wise comparisons of the correspondent nodes. The detailed procedure of this basic algorithm (BA) is shown in Algorithm 4.

## 5.3 Improved Algorithm

BA uses coding information for computing the similarity of node pairs but fails to use it for reducing the number of similarity comparison. Given any

---

**Algorithm 5:** Improved Algorithm *IA*

---

**Input:** node sets  $V$  and  $V_0$  with coding information  
**Output:** Similarity *simi*

```
1 SimiOfPair = ProcessedV = 0;
2  $n = |V|$ ;
3  $n_0 = |V_0|$ ;
4 for ( $i = 1; i < n; i++$ ) do
5    $v_1 = V[i]$ ;
6    $v_{01} = V_0[v_1.m]$ ;
7    $V_0[v_{01}.pre].m = -1$ ;
8   for ( $j = i + 1; j \leq v_1.RD; j++$ ) do
9      $v_2 = V[j]$ ;
10     $v_{02} = V_0[v_2.m]$ ;
11     $v_{02}.m = i$ ;
12    SimiOfPair += SNP( $v_1, v_2, v_{01}, v_{02}$ );
13  end
14  UntouchedV_0;
15  for ( $j = v_{01}.pre + 1; j \leq v_{01}.RD; j++$ ) do
16    if ( $V_0[j].m \neq -1$ )  $\wedge$  ( $v_0[j].m \neq i$ ) then
17      UntouchedV_0 += 1;
18    end
19  end
20   $j = v_{01}.P$ ;
21  ParentV_0 = 0;
22  while ( $j \neq 0$ ) do
23    if ( $V_0[j].m \neq -1$ )  $\wedge$  ( $V_0[j].m \neq i$ ) then
24      ParentV_0 += 1;
25    end
26     $j = V_0[j].P$ ;
27  end
28  ProcessedV += 1;
29  SimiOfPair +=  $n - \textit{ParentV}_0 -$ 
    UntouchV_0 - ProcessedV - (v_1.RD - v_1.pre);
30 end
31 return simi =  $\frac{2 \times \textit{SimiOfPair}}{n_0 \times (n-1)}$ ;
```

---

node pair  $(v_1, v_2)$  in  $F$ , it is important to determine whether we have to check the comparison with its counterpart pair  $(v_{01}, v_{02})$  in  $T_0$ . From Equation 3, we may observe that if  $pc(v_1, v_2)$  or  $ad(v_1, v_2)$  holds then we have to check the similarity of node pairs (SNP) with  $(v_{01}, v_{02})$  because it is possible  $pc(v_{01}, v_{02})$  or  $ad(v_{01}, v_{02})$  also holds and hence necessary comparison is required by using Equation 2; otherwise, we do not have to check the details of the node pairs. With this in mind, we design the improved algorithm (IA) that makes full use of the coding information to improve the performance of BA. The detailed procedure is shown in Algorithm 5.

The algorithm proceeds in the *pre* order of  $V$  (Line 4). For  $v_1 \in V$ , its counterpart  $v_{01} \in V_0$  is selected and is marked as  $-1$  for processed (Lines 5-7). We use the attribute  $m$  of nodes in  $V_0$  for marking purpose. With the *RD - Index* of  $v_1$ , we only need to check pairs  $(v_1, v_2)$  that satisfies  $ad(v_1, v_2)$  and compute SNP with their counterpart pairs  $(v_{01}, v_{02})$  (Lines 8-13). We use  $v_1.pre$  which equals to  $i$  to temporarily mark  $v_{02}$  so that we do not need to reset this mark after use (Line 11). We can easily get the count (*UntouchedV\_0*) of those pairs  $(v_{01}, v_{02})$  in  $T_0$  that satisfy  $ad(v_{01}, v_{02})$  yet inconsistent with their counterparts in  $F$  (Lines 14-19). Similarly, we can get the count (*ParentV\_0*) of those pairs  $(v_{01}, v_{02})$  in  $T_0$  that satisfy  $ad(v_{02}, v_{01})$  yet inconsistent with their counterparts in  $F$  (Lines 19-24) using *P - Index* of  $v_{01}$ . The matching number of pairs in  $T_0$  with all those pairs  $(v_1, v_2)$  in  $F$  where  $ad(v_1, v_2)$  does not hold can be calculated in Line 25. The similarity of  $F$  w.r.t.

$T_0$  can be obtained by Equation 5.

## 5.4 Complexity Analysis

The space complexity in both algorithms is equal to the sum of the sizes of node sets  $V$  and  $V_0$ , i.e.  $O(n_1 + n_0)$ . For BA, the time complexity is  $O(n^2)$  because the algorithm is conducted by using pair wise comparison of the nodes in  $V$ . Let  $N_0$  and  $N$  be the maximum out-degree in  $T_0$  and  $F$ , respectively. Before we analyse the time complexity for IA, we need to use a property of tree: for an  $N$ -ary tree  $T$  with height  $H$ , the maximum number of nodes in  $T$  is  $n = \frac{N^{H+1}-1}{N-1}$ . IA mainly consists of three parts:

- Lines 3, 7-12: We know all nodes in  $V$  need to be processed. For any node  $v$  and its level  $i$ , the number of descendants of  $v$  is  $\frac{N^{i+1}-1}{N-1} - 1$ . And the number of nodes that are at the same level as  $v$  is  $N^{H-i}$ . If the maximum height of the tree is  $H$ , then the number of comparisons is  $\sum_{i=0}^H (\frac{N^{i+1}-1}{N-1} - 1) \times N^{H-i}$ . As  $H = \log_N^{(N-1)n-1}$ , the time complexity is  $O(N \times n \times \log_N^{(N-1)n+1})$ ;
- Lines 3, 14-17: Similar to the above, the time complexity is  $O(N_0 \times n_0 \times \log_{N_0}^{(N_0-1)n_0+1})$ ;
- Lines 3, 19-23: each time Lines 19-23 is executed, the number of ancestors is less than  $H_0$ . So the time complexity is  $O(n \times \log_{N_0}^{(N_0-1)n_0+1})$ .

So if the schema trees are relatively balanced, the overall time complexity is  $O(N \times n \times \log_N^{(N-1)n+1} + N_0 \times n_0 \times \log_{N_0}^{(N_0-1)n_0+1} + n \times \log_{N_0}^{(N_0-1)n_0+1})$ . However, when the schema trees are unbalanced, the performance of the algorithm will be degraded. The worst situation (i.e.  $N = 1$ ) is the same as that of BA.

## 6 Experimental Study

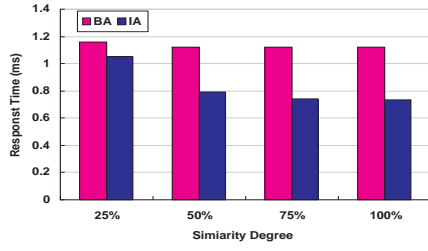
Experiments are carried out on a Pentium IV 3.00GHz PC with 512MB main memory. The algorithms are implemented in C++. We use both synthetic datasets and a number of publicly available schemas (Velegarakis et al. 2003) to compare the performance of our algorithms.

### 6.1 Sensitivity Test

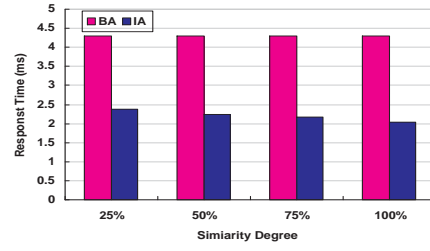
In the first set of the experiments, we carry out a series of sensitivity analysis for different features of the data.

**Sensitivity vs. Similarity Degree:** Figure 4(a-d) compares the performance between BA and IA for various similarity degrees when the schema size is 20, 40, 60, and 80, respectively. The domain schemas are created based on *genexml.xsd* with adaptation of size while the candidate source schemas are manually created with differences from adapted domain schemas. The results show that BA is nearly not affected by the changes of similarity. For IA, however, the more similar the two schemas are, the faster. Moreover, the performance of IA is getting better with the size increases.

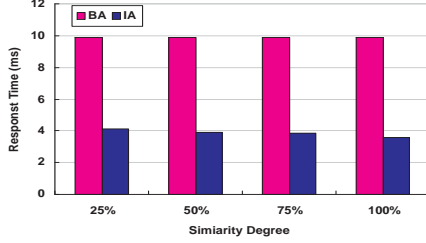
**Sensitivity vs. Nested Level:** We also use a set of synthetic datasets to evaluate the effect of nested level where every dataset includes 128 nodes and the level varies from 4 to 16. The results in Figure 5 illustrate that IA is much better than BA. The response time of IA is around 20% of that of BA. The performance of IA can be affected with the number



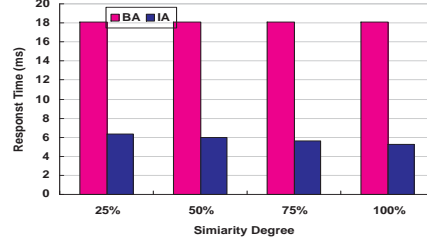
(a) 20 nodes



(b) 40 nodes



(c) 60 nodes



(d) 80 nodes

Figure 4: Response Time vs. Similarity Degree

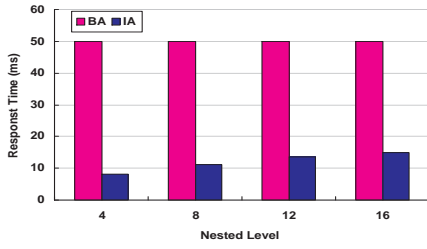


Figure 5: Response Time vs. Nested Level

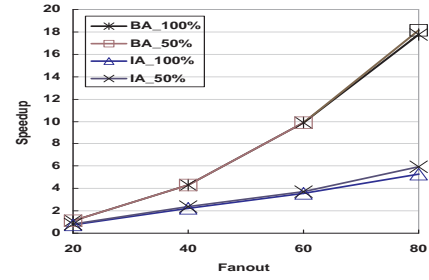


Figure 7: Response Time vs. the Size of Schema

of nested levels increases but still much better than that of BA. In real applications, it is seldom that the number of nested levels exceeds 20.

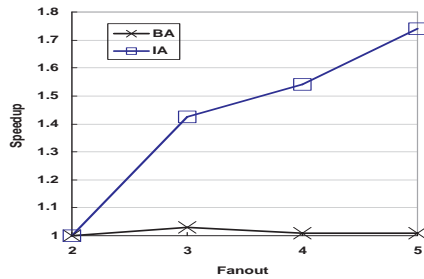


Figure 6: Response Time vs. Fanout

**Sensitivity vs. Fanout:** We finally use synthetic data to compare the performance of the two algorithms when the fanout varies. We generate four synthetic datasets that are of the same size of 128 nodes. The similarity degree is set to 100% because we only consider the impact of fanout. Figure 6 shows the speedup of the two algorithms when fanout changes from 2 to 5. The experimental results illustrate the performance of IA is much better than that of BA when the fanout is equal to or greater than 3. There are two factors: the nested level will become smaller

when the fanout is greater because we assume the size of schema is stable. On the other hand, the smaller nested structure is good to search its ancestors for every node when IA is carried out. But BA has to traverse every node that need to be compared, which cannot skip over any node.

## 6.2 Efficiency Test

We choose *genexml.xsd* as a base to evaluate our algorithms. The size varies from 20, 40, 60 and 80. Figure 7 shows IA is better than BA when the size is greater than 20. With the size increases, the processing time of BA increases greatly. However, the increasing trend is slow for IA. The experimental results in Figure 7 also show that IA needs less response time if the similarity degree is higher, e.g., when the schema size is less than 60, the response time is almost the same for schemas with similarity degree 100% and 50%, respectively. When the size adds up to 80, we can see slight difference. It also shows that the change in similarity degrees has little impact on the performance of BA.

Figure 8 provides the experimental results for the three public datasets: *TPC-H-nested.xsd* (17), *genexml.xsd* (85), and *mondial-3.0.xsd* (120). In this set of experiments, the domain and source schemas are the same. The results illustrate that IA performs much better than BA, especially when the schemas

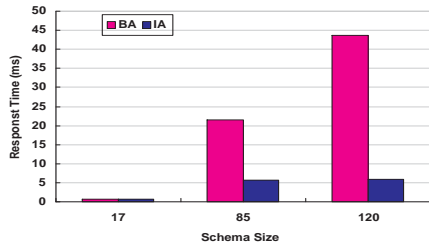


Figure 8: Response Time vs. the Size of Schema

contain more nodes. In real situation, potentially huge number of source schemas need to be compared with a domain schema. The performance gain of IA over BA will make a big difference.

## 7 Conclusions

We looked into the problem of efficiently computing structural similarity of potentially huge number of source schemas against a domain schema. This study is motivated from enhancing the quality of searching and ranking of huge number of XML source documents on the Web with the help of structural information, especially a domain schema against which queries are issued and big number of source schemas which source documents may conform to. In this paper, we proposed a new similarity model for measuring the structural similarity of a candidate source XML schema against a given domain schema and justified its effectiveness by comparing it with edit-distance based methods. To speed up similarity computation, we introduced a trimming process for filtering out uninteresting objects while preserving similarity. We also devised an efficient coding scheme. Two algorithms - the basic and the improved algorithms were developed with unnecessary comparisons removed in the improved algorithm. The experimental results showed that the improved algorithm outperforms significantly compared with the basic algorithm.

## 8 ACKNOWLEDGMENTS

The work described in this paper was supported by grants from the Australian Research Council Discovery Project (DP0559202) and the Research Grant Council of the Hong Kong Special Administrative Region, China (CUHK418205).

## References

Bergamaschi, S., Castano, S. & Vincini, M. (1999), 'Semantic integration of semistructured and structured data sources.', *SIGMOD Record* **28**(1), 54–59.

Bertino, E., Guerrini, G. & Mesiti, M. (2004), 'A matching algorithm for measuring the structural similarity between an xml document and a dtd and its applications.', *Inf. Syst.* **29**(1), 23–46.

Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V. J. & Zaniolo, C. (2002), Efficient structural joins on indexed xml documents., in 'VLDB', Morgan Kaufmann, pp. 263–274.

Cobena, G., Abiteboul, S. & Marian, A. (2002), Detecting changes in xml documents., in 'ICDE', pp. 41–52.

Dietz, P. F. (1982), Maintaining order in a linked list, in 'STOC', ACM, pp. 122–127.

Do, H. H. & Rahm, E. (2002), Coma - a system for flexible combination of schema matching approaches., in 'VLDB', pp. 610–621.

Duta, A. C., Barker, K. & Alhajj, R. (2006), Ra: An xml schema reduction algorithm, in 'ADBIS'.

Flesca, S., Manco, G., Masciari, E., Pontieri, L. & Pugliese, A. (2002), Detecting structural similarities between xml documents., in 'WebDB', pp. 55–60.

Formica, A. (2007), 'Similarity of xml-schema elements: A structural and information content approach', *Computer Journal*.

Gövert, N. & Kazai, G. (2002), Overview of the initiative for the evaluation of xml retrieval (inex) 2002., in 'INEX Workshop', pp. 1–17.

Jiang, H., Lu, H., Wang, W. & Ooi, B. C. (2003), Xr-tree: Indexing xml data for efficient structural joins., in U. Dayal, K. Ramamritham & T. M. Vijayaraman, eds, 'ICDE', IEEE Computer Society, pp. 253–263.

Lee, M.-L., Yang, L. H., Hsu, W. & Yang, X. (2002), Xclust: clustering xml schemas for effective integration, in 'CIKM', pp. 292–299.

Li, Q. & Moon, B. (2001), Indexing and querying xml data for regular path expressions., in P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao & R. T. Snodgrass, eds, 'VLDB', Morgan Kaufmann, pp. 361–370.

Li, W.-S. & Clifton, C. (2000), 'Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks.', *Data Knowl. Eng.* **33**(1), 49–84.

Madhavan, J., Bernstein, P. A. & Rahm, E. (2001), Generic schema matching with cupid., in 'VLDB', pp. 49–58.

Melnik, S., Garcia-Molina, H. & Rahm, E. (2002), Similarity flooding: A versatile graph matching algorithm and its application to schema matching, in 'ICDE', pp. 117–128.

Rafiei, D. & Mendelzon, A. O. (1998), Efficient retrieval of similar time sequences using dft., in 'FODO', pp. 249–257.

Rahm, E. & Bernstein, P. A. (2001), 'A survey of approaches to automatic schema matching', *VLDB J.* **10**(4), 334–350.

Velegrakis, Y., Miller, R. J. & Popa, L. (2003), Mapping adaptation under evolving schemas., in 'VLDB', pp. 584–595.

Yang, R., Kalnis, P. & Tung, A. K. H. (2005), Similarity evaluation on tree-structured data., in 'SIGMOD Conference', pp. 754–765.

Yi, S., Huang, B. & Chan, W. T. (2005), 'Xml application schema matching using similarity measure and relaxation labeling', *Inf. Sci.* **169**(1-2), 27–46.

Zhang, K. & Shasha, D. (1989), 'Simple fast algorithms for the editing distance between trees and related problems.', *SIAM J. Comput.* **18**(6), 1245–1262.