International Journal of Multimedia and Ubiquitous Engineering
Vol. 1, No. 3, September, 2006

Effective Similarity Discovery from Semi-structured Documents

Hyunjoo Moon[1], Kwanjoong Kim[2], Gilcheol Park[3], and Chae-woo Yoo[1]
[1]Department of Computer, Soongsil University
[2]Department of Computer and Information, Hanseo University
[3]Department of Multimedia, Hannam University
e-mail : hyunjoomoon@hotmail.com

Abstract The semi-structured data in XML format has been diffused through the widespread of the internet. To support the storage and retrieval of huge collections of such documents, reconciling similar DTDs within a cluster and using an effective similarity function are the keys of a successful data management process. XClust introduced WordNet ontology system to be widely extended the word compatibility performance. By using the ontology system, semantic compatibility can be stretched, but the velocity for the semantic similarity detection process is relatively increased in a great degree. This paper proposes a fast and effective method that can have ontological similarity flexibility same as XClust, but does not have big velocity delay. For practicality, we use a simple and very fast structural similarity detection method in the domain of frequencies, which can extremely elevate the performance of our similarity detection method. Our straightforward structural similarity detection method especially gets very fast and good results in such databases that have large number of similar documents.

Keyword: XML, DTD, Similarity Detection, WordNet, clustering, ontology

1. Introduction

The increasing use of XML as a standard document format for web data makes detection and management of XML document similarity to become an active area of research. XML has been widely adapted to store and exchange web data, and detection of similarity or changes between two XML documents is important function in many applications [4].

XML is a semi-structured language designed by W3C to have simple rules like HTML but strong functions like SGML. User-defined self-describing tags, one of the main characteristics of XML, make XML much stronger, but it also creates ambiguity when we interpret and group them into clusters. Similar documents having different tags but same meaning can be treated as low degrees of relationship, thus be clustered into different clusters. Similar documents having different hierarchical structures but same meaning also can be treated as non-similar documents.

XClust [1] suggested an ontological solution to solve the problems. XClust uses WordNet ontology system to achieve more flexible word compatibility, and traverses DTD trees to get hierarchical relationship information. The method provides more detail information of both structural and semantic similarity. However, XClust has a limitation in large-scale databases or similar document groups; it costs very expensive to get detail semantic information.

On the other hand, there are other approaches based on structural similarity, which can be an effective support for document storage retrieval. Time Warping Method [3] and DFT-based frequency similarity checking method [2,5] provide fast and practical solutions by attaching weight to structural characteristics, even though they don't matter the semantic ambiguity.

We propose a fast and efficient similarity detection method to address both semantic ambiguity and practicality. Using our method, both flexible word compatibility and practicality can be achieved. In heterogeneous and dynamic environment and huge collections of web data, this method can be easily applied for fast and effective clustering or retrieval of data.

This paper consists of 5 sections. Section 2 summaries related works including our main interest. Section 3 explains our similarity checking method in detail, and Section 4 shows how much different this method from other methods by experiments. Section 5 describes the conclusion.

2. Related Works

Many researches have studied the similarity-related problems to cluster similar documents, retrieve documents having user-requested data, or detect changes between documents in different versions.

A large number of initial similarity detection methods use Tree Edit Distance Algorithms (TDAs). TDAs are designed to detect changes in structured documents or semi-structured documents. TDAs such as Valiente's algorithm, Tai's tree-to-tree correction algorithm, Selkow's tree-to-tree correction algorithm, and Shasha and Zhang's algorithm are all based on atomic edit operations on nodes not considering subtree level operations, e.g. subtree deletes or moves, which are natural in XML documents [4].

Structural Document Differencing Algorithms (SDDAs) such as LaDiff and MH-Diff generating edit scripts are based on the weighted matching problem. SDDAs don't make good results in semi-structured documents or in documents having node duplications. On the other hand, Semi-Structured Document Differencing Algorithms (SSDDAs) such as XMLTreeDiff, XyDiff, or X-Diff detect changes in ordered trees using Bottom-Up Mapping (BUM) or Top-Down Mapping (TDM) mechanism. TDM mechanism has a drawback that can completely fail if all the leaves are modified or some structural changes isolate the top and bottom part of the old version in the new one, while TDM mechanism is much suitable for detecting changes between two versions of XMLized relational data sets, but it does not have any flexible semantic compatibility.

Time Sequence-based Algorithms (TSAs), such as Time Warping distance detection Method (TWM) or DFT(Discrete Fourier Transform)-based structural similarity detection Methods (DFTMs), totally differ from standard methods based on graph-matching algorithms that are generally computationally expensive, i.e., at least $O(N^2)$, where n is the number of elements in the two documents. TSAs focus on the fact that the computation of structural similarity is crucial to some clustering applications which can be of great value to the management of Web data, and allow a significant reduction of the required computation costs [2,3,5]. TSA does not have enough semantic similarity detection mechanism, but it has an attractive velocity feature than other algorithms.

Another interesting algorithm for similarity detection is XClust [1]. XClust introduced ontological word compatibility extension by using WordNet ontology system, and uses a tree traverse algorithm including PCC (Path Context Coefficient) concepts to capture the degree of similarity in the paths of two elements. XClust traverses DTD trees using both BUM and TDM to get hierarchical relationship information. XClust can provide much more flexible semantic compatibility, but it also causes a serious velocity limitation, especially in large-scale databases or dissimilar document groups. That is, it costs remarkably expensive to get flexible and detail semantic compatibility. Depending on the number of nodes and the similarity degree, the processing time in XClust can be thousands times longer than TSA's.

This paper introduces an effective and fast similarity detection method that covers both advantages in XClust and in TSAs. These advantages can be achieved by dividing the semantic similarity detection mechanism and the structural similarity detection mechanism, and by using a very simple structural difference detection mechanism without any tree traversal. It is slower than TSAs but has much stronger and flexible in semantic compatibility. It is much faster than XClust but there is no big difference on the structural information. If we consider further relationships between similarity detection algorithms and remarkably expensive additional ontological functions, this method can be one of the great solutions for the future.

3. Similarity Detection Scheme

In this section, we will describe our similarity detection scheme proposed in this paper. Our method is based on DTDs in this paper, but it can be easily changed for similarity detection methods between XML documents. Using DTD generators such as XML Spy, any XML document can be translated into a DTD. A DTD can be an isolated DTD or an abstract DTD representing a cluster including a number of DTDs. To reduce confusion between XML documents and DTD documents, we use the term a 'document' as a DTD document in Section 3 and 4. Only two documents can be compared in similarity detection process at one time, like other methods.

In typical similarity detection methods, a DTD parser translates DTDs into XML trees and similarity detection methods traverse the trees to get semantic and structural information. We use WordNet to extend the word compatibility and manipulate Name Matrix and Tag Sequences instead of the tree traversal. Name matrix is calculated at first, and then it is used for both semantic and structural similarity phases. By using the Tag Sequences, we can save a great portion of tree traversal time instead of time loss in semantic detection phase.

Our similarity detection system consists three modules; Semantic Similarity Detection Modules, Structural Similarity Detection Modules, and DTD Similarity Detection Modules. We will explain the modules in detail in following subsections.

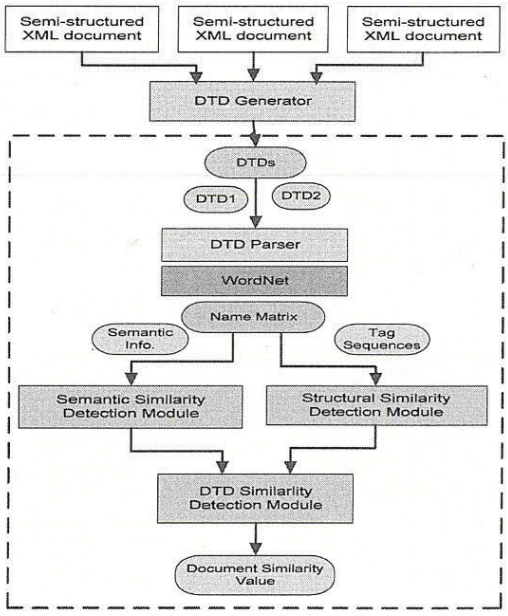


Fig. 1. Similarity Detection Scheme Layout

3.1 Sample Documents

For easy and intuitive understanding, we will use 4 sample DTDs. Figure 2 shows the DTDs. In the figure, *book1* and *book2* in (a) and (b) are semantically same but use few different tags. Except XClust, other algorithms cannot detect the semantic similarity between the semantically same tags. We can detect the similarity using the ontological function. Document *author1* in (c) is a almost subtree of document *book1* in (a). Document *article1* in (d) has lots of same tags with *book1* in (a).

(a) book1	(b) book2
<!ELEMENT xml (book)> <!ELEMENT book (title, author+, publisher)> <!ELEMENT title (#PCDATA)> <!ELEMENT author (name, affiliation)> <!ELEMENT publisher (#PCDATA)> <!ELEMENT name (first_name, last_name)> <!ELEMENT affiliation (#PCDATA)> <!ELEMENT first_name (#PCDATA)> <!ELEMENT last_name (#PCDATA)>	<!ELEMENT xml (book)> <!ELEMENT book (title, writer+, publishing_company)> <!ELEMENT title (#PCDATA)> <!ELEMENT writer (name, affiliation)> <!ELEMENT publishing_company (#PCDATA)> <!ELEMENT name (given_name, family_name)> <!ELEMENT affiliation (#PCDATA)> <!ELEMENT given_name (#PCDATA)> <!ELEMENT family_name (#PCDATA)>
(c) author1	**(d) article1**
<!ELEMENT xml (author+)> <!ELEMENT author (name, affiliation, email)> <!ELEMENT name (first_name, last_name)> <!ELEMENT affiliation (#PCDATA)> <!ELEMENT email (#PCDATA)> <!ELEMENT first_name (#PCDATA)> <!ELEMENT last_name (#PCDATA)>	<!ELEMENT xml (article+)> <!ELEMENT article (title, writer, para*)> <!ELEMENT title (#PCDATA)> <!ELEMENT writer (name, affiliation)> <!ELEMENT name (first_name, last_name)> <!ELEMENT affiliation (#PCDATA)> <!ELEMENT first_name (#PCDATA)> <!ELEMENT last_name (#PCDATA)> <!ELEMENT para (title, sentence*)> <!ELEMENT sentence (#PCDATA)>

Fig. 2. Sample DTD Documents

Figure 3 describes the trees of documents in Figure 1. We describe cardinality information of nodes in each node. Our method uses encoded tag sequences instead of real DTD tree traversals to reduce time and effort.

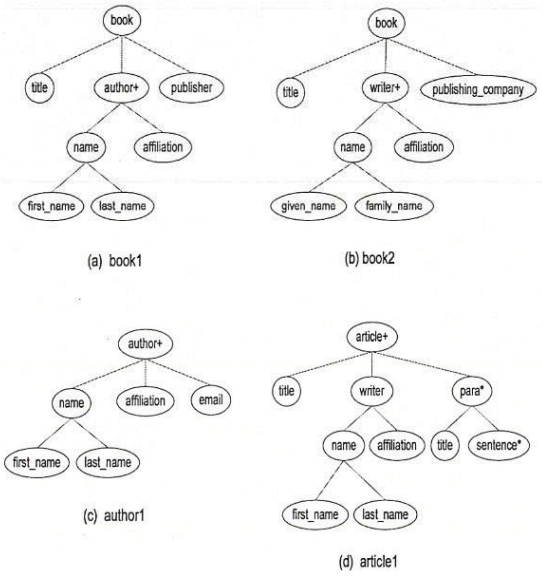


Fig. 3. DTD trees for Figure

3.2 Semantic Similarity Detection Mechanism

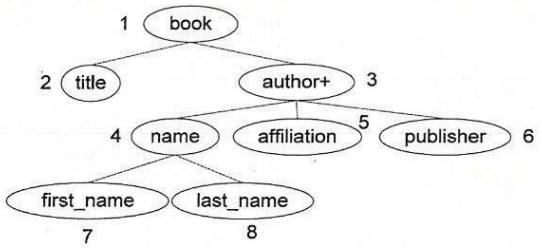
Before calculation of semantic and structural similarity, Name Similarity between nodes in two documents should be identified. If tag names are same, Name Similarity value is 1. If tag names are different, WordNet is called to check whether the tag names are synonyms to each other or not. Depending on the similarity quality of clusters, users can assign the number of calling. We limit the calling number less than 3, because we don't want to spend a lot of time for that. The more we call WordNet, the more time is noticeable needed. Based on the calculated Name similarity, Semantic Similarity value is calculated.
Algorithm for Semantic Similarity Detection Mechanism is shown in Table 1. Table 2 describes Name Matrix between document *book1* in Figure 2 (a) and *book2* in (b).

Table 1. SemanticSim Algorithm.

```
Algorithm SemanticSim(t1, t2, threshold1,
threshold2, map)
Input:
t1, t2 - DTD trees
threshold1 - The minimum value for OntologySim
having same index
threshold2 - The minimum value can be used in
LocalMatch
map - name to index map
Output:
        semantic similarity between two trees
Begin
a1 ← TreeTraverse(t1)
a2 ← TreeTraverse(t2)
next ← 0
matrix ← {}
for each e1 ∈a1 do
  for each e2 ∈a2 do
n1 ← e1.name
n2 ← e2.name
n = NameSim(n1, n2, 3)

if n > threshold then
 if both names are not exists in map then
          put (n1, next) into map
          put (n2, next) into map
next ← next + 1
 else if n1 is not exist in map then
          put (n1, index of n2) into map
  else if n2 is not exist in map then
          put (n2, index of n1) into map
else
 if n1 is not exist in map then
           put (n1, next) into map
           next ← next + 1
   if n2 is not exist in map then
           put (n2, next) into map
           next ← next + 1

 matrix ← matrix ∪(e1, e2, w1 * n
+ w2 * CardinalitySim(e1.card, e2.card))

MatchList ← LocalMatch(matrix, |a1|, |a2|,
threshold2)

return ∑sim / Max(|a1|, |a2|)
    sim ∈MatchList
End.


Function TreeTraverse(t)
Input:
t - DTD tree
Output:
element array using visitor pattern, which traverse
tree
in depth-first mechanism


Algorithm NameSim(n1, n2, maxDepth) = OntologySim
(w1, w2, maxDepth)
```

3.3 Structural Similarity Detection Mechanism

Figure 4 describes our encoding scheme to create tag sequences. Tag sequence scheme used in this paper is very simple but very powerful because it includes structural information including tag names, levels, hierarchical nesting structures, etc. We use Direct Invariant Encoding Scheme (DIES), therefore, a start tag and an end tag is encoded into the same numbers. When we create a tag sequence of a document, we traverse the tree in depth first matching method.
To explain the tag sequence *S(d)*, refer *book1* tree and tag sequence in Figure 4. Index number 1 is representing node named *book*. In a tag sequence, the first-placed tag 1 means start tag <*book*>, and the second-placed tag 1 means end tag </*book*>. Like this, if same index number takes place in odd time, it means start tag. If it takes place in even time, it means end tag.
We can easily see the nesting structure in the tag sequence.



S(*book1*)=(1,2,2,3,4,7,7,8,8,4,5,5,6,6,3,1)
= {1 {2, 2} (3 {4 {7 7} {8 8} 4} {5 5} {6 6} 3) 1}

Fig. 4. DTD trees and node encoding scheme

A tag sequence is an ordered encoding numbers that includes semantic and structural information same as XML trees. Therefore, comparing tag sequence instead of tree traversal using Top-Down Mechanism (TDM) and Bottom-Up Mechanism (BUM), especially more than hundreds of nodes, can be great solution. Our Structural Similarity algorithm is described in Table 2.
As a simple example, let's calculate the similarity between *book1* and *article1*. Before any tag sequence matching, the index unification between two trees should be completed. The tag sequences of *book1* and *article1* after index unification are shown as follows;

S(*book1*) = (1,4,4,5,6,7,7,8,8,9,9,5,13,13,1)
S(*article1*)=
(2,3,4,4,5,6,7,7,8,8,6,9,9,5,10,4,4,11,11,10,3,2)

When we match two tag sequences, we check how many numbers of ordered strings are taken place in both sequences. By using StructureSim algorithm, we can easily check it. The matching string set should be ordered, because the order includes the structural information. For example, the above two tag sequences have same ordered strings underlined. Therefore, the structural similarity between *book1* and *article1* is as follows;

StructureSim(*book1*, *article1*) = 12/22 = 0.5454546

Table 2. StructureSim Algorithm.

```
Algorithm StructureSim(t1, t2, map)
Input:
   t1, t2 - DTD tree
   map - name to index map
Output:
   structural similarity between two trees
Begin
   length ← 0
   x1 ← 0
   x2 ← 0

   seq1 ← TagSequence(t1)
   seq2 ← TagSequence(t2)

   for (match ← NextBestMatch(x1, x2)) ≠ nil do
     length ← match.length
     x1 ← match.i + match.length
     x2 ← match.j + match.length

   return length / Max(|seq1|, |seq2|)
End.


Function TagSequence(t)
   using visitor pattern


Function NextBestMatch(seq1, seq2, m1, m2)
Input:
   seq1, seq2: tag sequences
   m1, m2: start index
Output:
   Next maximum-length match
   if there's no more match then nil
Begin
   if m1 >= |seq1| or m2 >= |seq2| then return nil

   if seq1[m1] = seq2[m2] then
    return (m1, m2, Length(seq1, seq2, m1, m2))
   else
  a ← Search(seq1, seq2, m1, m2)
   b ← Search(seq2, seq1, m2, m1)

   if a = nil and b = nil then return nil
   else if a = nil then
     return (b.j, b.i, b.length)
   else if b = nil then
     return (a.i, a.j, a.length)
   else
     if a.length >= b.length then
       return (a.i, a.j, a.length)
     else
       return (b.j, b.i, b.length)
End.


Function Search(seq1, seq2, m1, m2)
Begin
   for each p_{m1<=i<|seq1|} ∈seq1 do
    for each q_{m2<=j<|seq2|} ∈seq2 do
      if p_i = q_j then return (i, j, Length(seq1, seq2, i,
j))
   return nil
End.

Function Length(seq1, seq2, m1, m2)
Input:
Output:
   The length matched from corresponding offset
   in each sequence
```

3.4 Similarity Between Two documents

The similarity between two DTD documents is calculated by adding semantic similarity and structural similarity with their weights. The total weight should be 1, and it is set 0.5 and 0.5 now. If we change the weights, then we can perform a different similarity check. Table 3 shows how to calculate document similarity algorithm in our method.

Table 3. DTDSim Algorithm.

```
Algorithm DTDSim(t1, t2)
Input:
      t1, t2 - DTD tree
Output:
      DTD similarity
Begin
      initialize map<as name to index>
      return 0.5 * SemanticSim(t1, t2, 0.7, 0.3, map)
        + 0.5 * StructureSim(t1, t2, map)
End.
```

4. Experiments

In this section, we show the experiment results on each pair of documents in Figure 2, Table 4 ~ 9 show the document similarity results using our similarity detection algorithms.

Table 4 shows that our similarity detection method has good and flexible semantic compatibility. Documents *book1* and *book2* in Figure 2 have same meaning but use some different tags; *author/writer*, *first_name/given_name*, *last_name/family_name*, or *publisher/publishing_company*. As you can see in figure, tags in same semantic meaning have same index numbers even though they have different tag names. The return value of *getSemanticSim* function shows the semantic similarity degree between the two documents as 0.84799993. It means the two documents are 84.799993% similar. If the detection method doesn't have flexible semantic compatibility, then above tags cannot be matched to each other, therefore, the result is changed to 0.5, i.e., 50% semantic similarity.

Table 4. Similarity detection result between *book1* and *book2*

```
book1 book2

Loading wnjn ...
[Name Table]
given_name : 4
family_name : 5
title : 1
book : 0
publishing_company : 7
affiliation : 6
writer : 2
publisher : 7
last_name : 5
name : 3
first_name : 4
author : 2

[TripletSet]
(last_name,family_name) => 0.84799993
(title,title) => 1.0
(affiliation,affiliation) => 1.0
(author,writer) => 0.84799993
(publisher,publishing_company) => 0.84799993
(book,book) => 1.0
(name,name) => 1.0
(first_name,given_name) => 0.84799993

Sequence1 = (1,2,2,3,4,5,5,6,6,4,7,7,3,8,8,1)
Sequence2 = (1,2,2,3,4,5,5,6,6,4,7,7,3,8,8,1)

getSemanticSim(threshold=0.3)=0.924 [4677ms]
getStructureSim()=1.0 [10ms]
getSim(alpha=0.5,beta=0.5)=0.962
```

Table 5 shows the similarity value between *book1* and *author1*. Document author1 can be said as a subtree of *book1* except one node, email. Structurally, *author1* is a subtree of *book1*. This result shows that our method easily detect the similarity when a tree is a subtree of another tree.

Table 5. Similarity detection result between *book1* and *author1*

```
book1 author1

Loading wnjn ...
[Name Table]
title : 8
book : 0
email : 7
affiliation : 6
publisher : 9
last_name : 5
xml : 1
name : 3
first_name : 4
author : 2

[TripletSet]
(affiliation,affiliation) => 1.0
(author,author) => 1.0
(name,name) => 1.0
(first_name,first_name) => 1.0
(last_name,last_name) => 1.0

Sequence1 = (1,9,9,3,4,5,5,6,6,4,7,7,3,10,10,1)
Sequence2 = (2,3,4,5,5,6,6,4,7,7,8,8,3,2)

getSemanticSim(threshold=0.3)=0.625 [4026ms]
getStructureSim()=0.625 [0ms]
getSim(alpha=0.5,beta=0.5)=0.625
```

Until now, we experiment on very similar trees or a subtree of another tree. Now, we try to apply our algorithm between somewhat different trees. Documents *book1* and *article1* in Figure 2 have same tags and different tags with another. Document *article1* have two different tags in same semantic perspective and the node 4 places in two places in *article1*. When we meet this kind of ambiguity, we choose the longer matching string and it should be ordered. Therefore, the matching strings are {4,4,5,6,7,7,8,8,6,9,9,5,}, and the structural similarity is 12/22=0.5454546, i.e., the two nodes 54% structurally similar to each other.

Table 6. Similarity detection result between *book1* and *article1*

```
book1 article1

Loading wnjn ...
[Name Table]
sentence : 10
title : 3
publisher : 12
xml : 1
last_name : 7
article : 2
book : 0
writer : 4
affiliation : 8
para : 9
name : 5
first_name : 6
author : 4

[TripletSet]
(first_name,first_name) => 1.0
(name,name) => 1.0
(affiliation,affiliation) => 1.0
(title,title) => 1.0
(author,writer) => 0.7879999
(last_name,last_name) => 1.0

Sequence1 = (1,4,4,5,6,7,7,8,8,6,9,9,5,13,13,1)
Sequence2 = (2,3,4,4,5,6,7,7,8,8,6,9,9,5,10,4,4,11,11,10,3,2)

getSemanticSim(threshold=0.3)=0.5261818 [5077ms]
getStructureSim()=0.54545456 [0ms]
getSim(alpha=0.5,beta=0.5)=0.5358182
```

In here, we need to test whether the string matching method is consistent when we changed the input order. We've already checked input order change tolerance with two documents. The similarity value between two documents is same even though the input order is changed.

Because *book1* and *book2* is semantically very similar, the similarity of *book1* and *author1* should be similar with the

similarity of *book2* and *author1*. In Table 7, you can check that the similarity value of *book2* and *author1* is 59.65%, and the similarity values of *book1* and *author1* is 62.5%.

Table 7. Similarity detection result between *book2* and *author1*

```
book2 author1

Loading wnjn ...
[Name Table]
given_name : 4
title : 8
publishing_company : 12
xml : 1
last_name : 5
family_name : 5
book : 0
affiliation : 6
email : 7
writer : 2
name : 3
author : 2
first_name : 4

[TripletSet]
(affiliation,affiliation) => 1.0
(writer,author) => 0.84799993
(name,name) => 1.0
(given_name,first_name) => 0.84799993
(family_name,last_name) => 0.84799993

Sequence1 = (1,9,9,3,4,5,5,6,6,4,7,7,3,13,13,1)
Sequence2 = (2,3,4,5,5,6,6,4,7,7,8,8,3,2)

getSemanticSim(threshold=0.3)=0.568 [2483ms]
getStructureSim()=0.625 [10ms]
getSim(alpha=0.5,beta=0.5)=0.59650004
```

Table 8 shows the similarity detection result between *book2* and *article1*, and the result is very similar with *book1* and *article1*.

Table 8. Similarity detection result between *book2* and *article1*

```
book2 article1

Loading wnjn ...
[Name Table]
sentence : 10
given_name : 6
title : 3
publishing_company : 13
xml : 1
last_name : 7
article : 2
family_name : 7
book : 0
writer : 4
affiliation : 8
para : 9
name : 5
first_name : 6

[TripletSet]
(given_name,first_name) => 0.84799993
(name,name) => 1.0
(affiliation,affiliation) => 1.0
(title,title) => 1.0
(writer,writer) => 0.94
(family_name,last_name) => 0.84799993

Sequence1 = (1,4,4,5,6,7,7,8,8,6,9,9,5,14,14,1)
Sequence2 = (2,3,4,4,5,6,7,7,8,8,6,9,9,5,10,4,4,11,11,10,3,2)

getSemanticSim(threshold=0.3)=0.5123637 [4417ms]
getStructureSim()=0.54545456 [10ms]
getSim(alpha=0.5,beta=0.5)=0.5289091
```

Table 9 shows the similarity between *author1* and *article1*. The result shows that *author1* and *article1* are 53.58% similar.

Table 9. Similarity detection result between *author1* and *article1*

```
author1 article1

Loading wnjn ...
[Name Table]
sentence : 9
article : 1
title : 2
email : 11
affiliation : 7
writer : 3
para : 8
last_name : 6
xml : 0
name : 4
author : 3
first_name : 5

[TripletSet]
(name,name) => 1.0
(author,writer) => 0.7879999
(last_name,last_name) => 1.0
(affiliation,affiliation) => 1.0
(first_name,first_name) => 1.0
(xml,xml) => 1.0

Sequence1 = (1,4,5,6,6,7,7,5,8,8,12,12,4,1)
Sequence2 = (1,2,3,3,4,5,6,6,7,7,5,8,8,4,9,3,3,10,10,9,2,1)

getSemanticSim(threshold=0.3)=0.5261818 [5779ms]
getStructureSim()=0.54545456 [0ms]
getSim(alpha=0.5,beta=0.5)=0.5358182
```

Through these experiments we can sure the speed and performance of our similarity detection method. We can have flexible and strong tag compatibility and also have practicality by using tag sequence. The fast speed can be checked in Table 4~9, and it's more than enough when we compare the speed with XClust.

5. Conclusions

XML format data has been widely adapted to store and retrieve Web data, and similarity detection between two XML documents is an important function in many applications. XML is a semi-structured language that uses user-defined self-describing tags. Semi-structure and user-defined tags make XML much stronger, but it also creates ambiguity when we interpret and group them into clusters.

We propose a fast and efficient similarity detection method that covers both advantages in XClust and in TSAs. Using our method, both flexible word compatibility and practicality can be achieved. These advantages can be achieved by dividing the semantic similarity detection mechanism and the structural similarity detection mechanism, and by using a very simple structural difference detection mechanism without any tree traversal. It is slower than TSAs but has much stronger and flexible in semantic compatibility. It is much faster than XClust but

there's no big difference on the structural information. If we consider further relationships between similarity detection algorithms and remarkably expensive additional ontological functions, this method can be one of the great solutions for the future.

In heterogeneous and dynamic environment and huge collections of web data, this method can be easily applied for fast and effective clustering or retrieval of data.

References

[1] Mong Li Lee et. al., "XClust: Clustering XML Schemes for Effective Integration," 11th ACM In Proc. International Conference of CIKM, 2002.

[2] Sergio Flesca et. al., "Fast Detection of XML Structural Similarity," IEEE Transactions on Knowledge and Data Engineering, vol.17, no.2, Feb. 2005.

[3] Byoung-Kee et. al., "Efficient Retrieval of Similar Time Sequences Under Time Warping," In Proc. of ICDE'98, pp.201-208, Feb. 1998.

[4] Raihan Al-Ekram et. al., "diffX: An Algorithm to Detect Changes in Multi-Version XML Documents," In Proc. of CASCON, 2005.

[5] Sergio Flesca et. al., "Detecting Structural Similarities between XML Documents," In Proc. of WebDB, 2002.

[6] Theodore Dalamagas et. al., "A Methodology for clustering XML Documents by Structure," Information Systems, 31, pp. 187-228, 2006.

[7] Tatsuya Asai et. al., "Efficient Substructure Discovery from Large Semi-structured Data,", In Proc. of SDM, 2002.

[8] Wang Lian et. al., "An Efficient and Scalable Algorithm for Clustering XML Documents by Structure," IEEE Transactions on Knowledge and Data Engineering, vol.16, no.1, Jan. 2004.

[9] Rui Yang et. al., "Similarity Evaluation on Tree-structured Data," In Proc. of ACM SIGMOD, pp.754-765, 2005.

[10] E. Bertino et. al., "Measuring the Structural Similarity among XML Documents and DTDs," http://www.disi.unige.it/person/MesitiM, 2001.

[11] Dina Q Goldin et. al., "On Similarity Queries for Time-Series Data: Constraint Specification and Implementation," In Proc. of CP, pp.137-153, 1995.

[12] Andrew Nierman et, al., "Evaluating Structural Similarity in XML Documents," In Proc. of WebDB, Jun, 2002.

[13] Mahbubul et, al., "Quality of Service among IP-based Heterogeneous Networks," IEEE Personal Communications, vol. 8, no.6, pp.18-24, Dec, 2001.

Authors

Hyunjoo Moon

Received the M.S. in computer science from Soongsil University, Korea, in 1993. She is working toward the Ph.D. degree in computer science at Soongsil University. Her research interests include Programming environments, Compilers, Multicast, System software, Ubiquitous computing and Sensor networks.

Kwanjoong Kim

Received the M.S., and Ph.D. degrees in computer science from Soongsil University, Korea, in 1988 and 1998, respectively. He is currently an Associate Professor in Department of Computer and Information at Hanseo University. His research interests include Multicast mobile network, computer architecture, B3G networks and sensor networks.

Gilcheol Park

Received M.S. degree in SoongSil University, Korea, in 1985 and Ph.D. degree in SungKunKwan University, Korea, in 1990, respectively. He is currently a Professor at Department of Multimedia Engineering Hannam University. His research interests include Multimedia Communication, Mobile Web Service, and Ubiquitous Network.

Chae-woo Yoo

Received M.S. degree in Soongsil University, Korea, in 1976 and Ph.D. degree in Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1985, respectively. He is currently a Professor at School of Computing, Soongsil University. His research interests include Programming Languages, Compilers, and Human and Computer Interaction.