Matching XML Documents in Highly Dynamic Applications

Adrovane Marques Kade
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970
Porto Alegre – RS – Brazil
adrovane@inf.ufrgs.br

Carlos Alberto Heuser
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970
Porto Alegre – RS – Brazil
heuser@inf.ufrgs.br

ABSTRACT

Highly dynamic applications like the Web and peer-to-peer systems require a great deal of effort in document management. Different sources can provide documents that contain data that, although having different structure or different contents, may be considered as representing the same conceptual information. One essential task in this scenario is the identification of complementary or overlapping documents that need to be integrated. In this paper, we deal specifically with documents represented in the XML format. XML document integration is an important process in highly dynamical applications, for the volume of data available in this format is constantly growing. It is also a challenging task, due to the flexible nature of the XML format, which leads to structure divergences and content conflicts between the documents. In this work, we present a novel approach to the *matching* problem, i.e., the problem of defining which parts of two documents contain the same information. Matching is usually the first step of an integration process. Our approach is novel in the sense it combines similarity information from the content of the elements with information from the structure of the documents. This feature, as our experiments confirm, makes our approach capable of dealing with content as well as structural divergences.

1. INTRODUCTION

Document management in highly dynamic environments like the Web and peer-to-peer systems is a hard task. In such environments, documents change very frequently, both in content and structure. Additionally, data sources in these environments usually have documents which contents overlap (partially or totally) the contents of documents in other data sources. Dealing with these overlaps and/or duplications in a dynamic environment is challenging in many aspects.

In this paper, we are interested in identifying which portions of different documents represent what may be regarded as the same information at a conceptual level. We focus on XML documents, since documents of this class mix structure and content, and are largely available (RSS data feeds, web service results, etc.). Since each data source may publish data in its very own format, XML integration must deal with the problem of structure and content inconsistencies between documents.

Figure 1 shows an example of two different XML documents. One can easily infer that the data on both documents (a) and (b) represent the same conceptual information, despite the differences in their content and structure. One example of content divergence occurs in the `name` element, which is spelled in document (a) as `John Smith`, while in document (b) it is `Smith, John`. To find out that `John Smith` and `Smith, John` represent the same name, a simple string comparison would not suffice. A string *similarity function* is required to discover that these two strings represent the same information. Such similarity functions get two strings as input and return a value indicating how similar (or how different) they are. Some well-known similarity functions are Edit Distance [15], Jaro-Winkler [10, 11, 26], Monge-Elkan [17] and SoftTF.IDF [4], among others. When such functions are used, we usually consider that the input strings match if their similarity value is above a certain threshold.

Figure 1 also shows a structural divergence between documents (a) and (b). The person's address in document (a) is represented as a single element, while in document (b) it is represented as individual elements for street, number, etc. However, if we merge all these individual elements from (b) into a single element, it will have (almost) the same content as the `address` element in (a). This kind of divergence cannot be detected by a similarity function, like the ones listed above. To correctly identify this case, the integration engine must be aware that one element's content is identical (or highly similar) to the merged contents of sub-elements of another element. The use of semantic information available in the element's tags and attributes can also be valuable in this process.

The identification and resolution of these kinds of structural and content divergences is the essence of the process known as *matching*. The matching process consists in determining which elements from two documents represent the same conceptual information. Matching is the first step in the process of integrating XML documents.

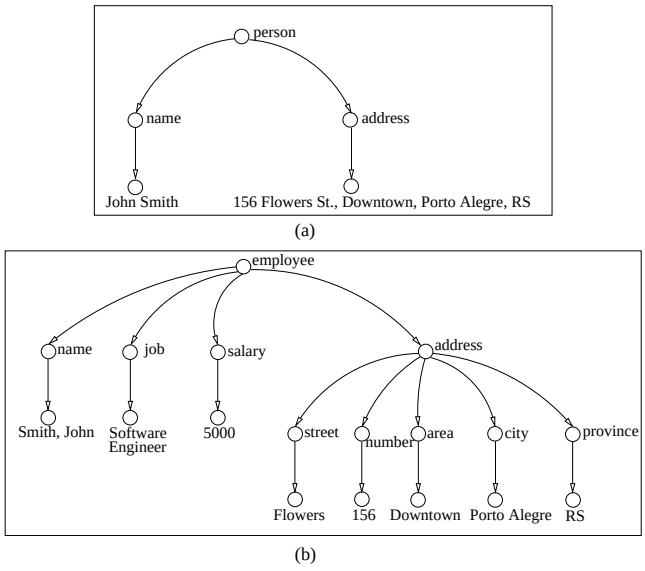

Figure 1: Two different ways of representing personal data in XML.

The problem illustrated above is relevant for several kinds of applications, such as query processing and data cleaning [19]. For example, a user would want to integrate data from his CD collection with other information gathered from the web (e.g., the price of each CD in a record store). Someone else would also need to build an integrated repository with data gathered from several on-line sources, which will then receive queries from several different users.

In this context, the integration process can be treated as two separate (but closely related) problems:

- *Matching*, which consists in the discovery of correspondences between elements of XML documents;
- *Merging*, which is the process of determining, based on the correspondences found in the matching phase, how the final document is going to look like.

This paper is restricted to the first problem, i.e., matching XML documents. In this context, we present a novel strategy to identify matchings between two XML documents. Our technique's main strength resides in the fact that it uses information from both content and structure of XML documents.

The remaining of this paper is organized as follows. In Section 2, we present and discuss the related work. In Section 3, we describe our approach to XML matching. In Section 4, we detail the performed experiments and the results obtained with three document sets. In Section 5, we draw our conclusions and point out some future works direction.

2. RELATED WORK

The research problems discussed in this paper permeate several areas of application, in many different ways. In this paper, we concentrate on the related work on two main areas: *duplicate detection* and *similarity metrics*.

Duplicate detection is the process of discovering the same real world object with different representations, in one or more databases. This is a broadly studied problem in data integration, under different names. In the database field, the problem is known as *merge-purge* [9], *data deduplication* [20] and *instance identification* [23]. In artificial intelligence, it is called *database hardening* [3] and *name matching* [1]. The terms *record linkage*, *record matching*, *co-reference resolution*, and *duplicate record detection* are also used to refer to the same problem. A recent survey in duplicate detection in relational databases can be found in [7].

Duplicate detection in XML, in some aspects, is similar to duplicate detection in relational databases. In both cases, one has to deal with strings of text, which must be compared in order to check if they carry the same content or not. For instance, errors in data input and/or interchange can produce inconsistencies between two strings. These so called *lexical* inconsistencies can be, for instance, swapping characters or words in a string. For example, the name of the city "Porto Alegre" could be written as "Porto Elegre". There can also occur standardization problems, like when, for example, one database has the value "Porto Alegre", and the other one has the value "POA" (which is the IATA identifier for the city's airport).

The same techniques used to handle lexical inconsistencies in relational database records can be used when we're working with XML. In the latter case, the lexical inconsistencies are found in the textual element's contents, and also in the element's tags.

A second kind of problem the duplicate detection engine must deal with is the existence of *structural* divergences. In the relational database context, structural divergences may occur in schema and table structures. Two records from two different databases can store the same information, but their fields can be organized in different ways. Consider, for instance, a person's address. It can be represented in one database as a single attribute, while in the other it can be represented as individual attributes for street name, city name, postal code, and so on.

The structural divergences occur because databases are created by different people. So, each database designer uses the modeling solutions (s)he prefers. Structural divergences in relational databases are considered in an early step called *data preparation*. This step precedes the duplicate record detection. The set of activities which composes the data preparation is also known as *Extraction, Transformation, Load* — ETL [13].

Structural divergences are more difficult to identify in XML data than in relational databases, due to the flexible hierarchical structure of XML documents. Several nesting levels can exist in an XML document. This does not occur in relational databases, since tables and attributes are flat. It is harder to find correspondences when comparing documents which use different hierarchical structures to convey the same information. As a consequence, the techniques used for relational databases cannot be readily applied for XML data sources.

In this context, new duplicate detection techniques suitable for XML have been proposed in the literature. A framework for duplicate detection in XML called DogmatiX is proposed in [24]. The detection mechanism compares the elements of the XML documents based on their own values and also uses the similarity from the parents and children of the element. A bayesian network based duplicate detection mechanism is proposed in [14]. The vector model, broadly used in information retrieval, is used for detecting duplicates in XML documents in [5].

Several work present metrics to measure the similarity between two XML documents. Many of them use the *tree edit distance* metric [21]. As an example, in [18] the tree edit distance is adapted to handle XML specific situations, such as optional and repeated elements.

The tree edit distance is also used in the work presented in [8]. In this case, it is used with a different focus: finding correlations between XML elements by the use of join operations. Tree edit distance algorithms are known to have severe performance limitations [6]. As a solution for this problem, they propose *bounds* to limit the number of elements which will be considered in the join operations. They also propose an indexed join which uses R-Trees to enhance the algorithms performance.

But performance limitation is not the only problem in of tree edit distance when used to match XML documents, as it is pointed out in [16]. The tree edit distance algorithms give more importance to the *tree's topology* than to the *semantic* information contained in the tree nodes' labels. The semantic information, however, is one of the XML's main strength. So, we would be throwing away a lot of information if we just ignore or underuse it, which is the case in tree edit distance techniques.

An alternative technique called *structure aware XML distance* is proposed in [16]. It employs a strategy to identify common structures—called *overlays*—in two XML documents. This technique solves some deficiencies of the tree edit distance. It is not very flexible, however, in the sense that it requires the matching nodes' paths to be exactly the same. This is a strong restriction when we consider XML's flexible nature.

Two techniques are proposed in [12]: *bag of paths* and *bag of XPaths*. The distance between two documents is calculated by the identification of common paths (which are XPath expressions in the last case), using clustering techniques. They only consider the parent/child relationships between the elements when separating the documents into clusters. So, for their technique to be effective it is necessary that the structure of the input documents be reasonably different. This can be a limitation if one wants to match datasources whose documents have a considerable degree of structural similarity.

The techniques described above use information from document's content and structure at various degrees, ranging from nodes' topology in the trees to parent/child relationships. None of them, however, consider both content and structure as first class features in the task of XML matching.

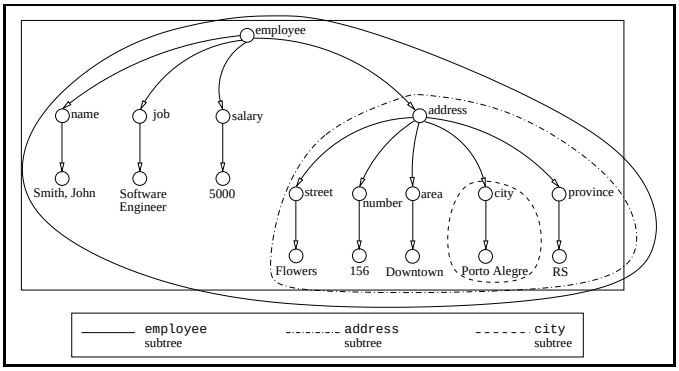

Figure 2: Sample subtrees highlighted in the tree (b) from the Figure 1.

This is the case in tree edit distance, for example, which considers first the tree's topology and just then the document's content.

In the next section, we present a novel technique to match XML documents. Our technique's main strength is that it uses information from both content and structure to discover the matchings.

3. XML MATCHING

In this paper, we propose a novel technique for matching two XML documents. This technique makes use of information from both structure and content of the elements of the documents.

The similarity between the elements of two XML trees is assessed in two steps. In the first step, we decompose every XML tree in *subtrees*. We call this the *top-down* process. We consider, one at a time, every non-leaf node in the tree, starting from the root and traversing down to the leaves. In this process, each node from the tree is considered as the root of its own subtree. For each subtree we produce one string, which is made up of the contents of all the subtree's leaf nodes merged together. The result of this process is a set of tuples of the form *(path, content)*. The subtrees contents of two XML trees are then compared against each other using a string similarity function.

Figure 2 highlights some subtrees produced by the application of this process to the tree (b) from the Figure 1. Three subtress are highlighted in the figure: the `employee` subtree (the continuous line); the `address` subtree (the dash-point line); and the `city` subtree (the dashed line). Note that the `employee` subtree in fact comprises all the elements of the document, for it is the root of the tree. Figure 3 shows the tuples for all the subtrees in the tree from Figure 2.

The decomposition process is shown in Algorithm 1. It receives as input an XML document tree and generates as result a document list, as described above. The document list starts empty (line 1). Then, we use the $DocIter()$ function to traverse the XML tree in document order and visit all its non-leaf nodes (line 2). For every node, we apply the $GetAbsolutePath()$ function to get its path from the root (line 3). We test the $nodeType$ node's property to see if it is

Path	Content
employee	Smith, John Software Engineer 50000 Flower St. 156 Downtown Porto Alegre RS
employee/name	Smith, John
employee/job	Software Engineer
employee/salary	50000
employee/address	Flower St. 156 Downtown Porto Alegre RS
employee/address/street	Flower St.
employee/address/number	156
employee/address/area	Downtown
employee/address/city	Porto Alegre
employee/address/province	RS

Figure 3: Set of tuples (path, content) for the subtrees from the tree from the Figure 2.

an *element* node or an *attribute* node (lines 4 and 7). Then, we use a suitable method to get the node's content. If it is an element node, we use the $GetStringValue()$ function to get a string which comprises all the contents of the nodes on its subtree (line 5). If it is an attribute node, we simply get its textual content, by using the function $GetNodeValue()$. Next, we append the pair (*path*, *content*) to the document list (line 10). In the end, we return the full document list (line 12).

```
   Data: an XML document tree
   Result: a document list with (path, content) tuples
1  docList ← [];
2  foreach v ∈ DocIter(d) do
3  |  path ← GetAbsolutePath(n);
4  |  if v.nodeType = ELEMENT-NODE then
5  |  |  content ← GetStringValue(n);
6  |  endif
7  |  else if v.nodeType = ATTRIBUTE-NODE then
8  |  |  content ← GetNodeValue(n);
9  |  endif
10 |  tuple = (path, content);
11 |  docList.append(tuple)
12 endfch
13 return docList
```

Algorithm 1: Decomposition process.

The structural divergences of the matching trees are minimized by the application of the decomposition process described above. Consider, for example, the trees presented in Figure 1. The person's name is represented by two different structures. In (a), it is represented by only one node, while in (b) it is represented as individual nodes for street name, city name, and so on. In an element-to-element content comparison, we would compare the terminal nodes in (a) with the terminal nodes in (b). It is easy to infer that this process would not be successful in finding matches, because no node in (b) has exactly (or very similar) content as the `address` node in (a). Consider now that we apply the decomposition process to both trees (a) and (b). When we compare the resulting tuples, we will find that the terminal node `address` in (a) matches the contents of the subtree rooted at the non-leaf node `address` in (b).

In the second step we compare the tuples of the document lists, searching for matching nodes. To do this we make use of three evidences for each node :

- the content of the subtree rooted at the node;
- the node's label (i.e., the tag of the XML element);
- the node's path up to the root of the document tree.

The first evidence comparison tells us how similar the contents of the nodes are, while the comparison of the two remaining evidences tell us about the structural similarity of the nodes. These two dimensions—content and structure—in an XML document complement each other, and we only get a partial document view if we consider only one of them. So, it is essential in a matching process not to underuse any one of these dimensions.

When we have the individual similarity values for all the evidences of two nodes (i.e., content, name, and path similarity), we use them to evaluate the similarity between two nodes. To do this, we use thresholds for content similarity, name similarity, and path similarity. If the similarity value for a pair of nodes are higher than or equal to the thresholds values, we consider that the nodes match.

As an example, consider that the process described above is applied to documents (a) and (b) in Figure 1. The document list from document (a) would contain, among others, a tuple for the `/person/address` element, whose contents would be `"156 Flowers St., Downtown, Porto Alegre, RS"`. On the other side, the document list for document (b) would contain, among others, a tuple for element `/employee/address`, whose contents would be `"Flowers 156 Downtown Porto Alegre RS"`. Suppose that we use the following similarity functions: Jaro-Winkler, for the textual content of the elements; Edit Distance, for the name of the elements; and PathSim [22] for the path of the elements. Suppose also we use 0.5 as the value for all thresholds. The application of the similarity functions to the tuples would result in the following values: 0.79 for the content of the elements; 0.5 for the paths; and 1.0 for the elements names. By calculating the average of these values we get 0.76.

We detail this process in Algorithm 2. The algorithm receives as input two document lists created by the application of the Algorithm 1, along with three threshold values. It starts by defining an empty *matchings* list (line 1). Then, each tuple from the first document list d_1 is compared against all the tuples from the second document list d_2 (lines 2–3). We first test the two tuples's *content* fields, by passing them to the $GetContentSim()$ function. Note that our algorithm does not impose the use of one specific similarity function, so we use a generic function name here. If the resulting similarity value surpasses the content threshold (line 5), we continue by testing the structural information. The name similarity is evaluated by the $GetNameSim()$ function (line 6). Again, our algorithm does not impose the use of a specific similarity function, so we use generic function names. If name similarity value is less than their threshold (line 7), we consider that the elements names are completely different (i.e., they can't be the same concept if their similarity is under the threshold), and set the similarity value to 0.0 (line 8). The path similarity is evaluated by the $GetPathSim()$ function (line 10). Then, we calculate the similarity between two nodes as the average between the content similarity, path similarity and name similarity (line 11). If the similarity of the nodes is above the threshold, we consider that we found a match and append the pair of elements to the matchings list, along with the similarity value

(lines 12–14). In the end, we return the list of all matching pairs (line 16).

```
   Data: two document lists D1, D2; CONTENT-THRESHOLD;
         NAME-THRESHOLD; NODE-THRESHOLD
   Result: a list containing pairs of matching nodes
1  matchings ← [];
2  foreach d1 ∈ D1 do
3      foreach d2 ∈ D2 do
4          contSim ← GetContentSim(d1.content, d2.content);
5          if contSim ≥ CONTENT-THRESHOLD then
6              nameSim ← GetNameSim(d1.path, d2.path);
7              if (nameSim ≥ NAME-THRESHOLD) then
8                  nameSim ← 0.0;
9              endif
10             pathSim ← GetPathSim(d1.path, d2.path);
11             nSim ← (contSim + pathSim + nameSim)/3;
12             if (nSim ≥ NODE-THRESHOLD) then
13                 pair ← (d1, d2, vSim);
14                 matchings.append(pair);
15             endif
16         endif
17     endfch
18 endfch
19 return matchings
```

Algorithm 2: Node similarity evaluation.

The list contains all the matching pairs for the documents we are comparing. It is then used to evaluate the similarity between two XML documents. In a way analogous to [16], we consider that only the pairs of elements which effectively match are significant in the calculation of the similarity between two XML documents. So, we calculate the document similarity as the average between all the similarity values in the matchings list. Consider that the number of elements in the matchings list is given by its length, represented as ℓ, and the similarity value measured for each pair of elements is given by σ_i, where i is the position of the pair in the matchings list. Then, the similarity of two XML documents, $XSim$, is given by:

$$XSim = \frac{\sum_{i=1}^{\ell} \sigma_i}{\ell} \quad (1)$$

In the documents of Figure 1, the nodes `job` e `salary` in document (a) does no match with any other node in document (b). The individual nodes which form the address in (b) partially match with the address node in (a), but this matching is superseded by the matching of address in (b) with address in (a). So, we find the following matches:

- `/person/name` in (a) with `/employee/name` in (b);
- `/person/address` in (a) with `/employee/address`.

The similarity values of each evidence for these matching pairs can be seen in Table 1. The first column lists the evidences, the second shows the first matching pair, and the third shows the second matching pair. The similarity values were computed by the application of the following similarity functions: Jaro-Winkler, for the contents; PathSim, for the paths; and Edit Distance for the names of the nodes. The last line of the table shows the global similarity of the nodes, calculated as the average between the evidences. Using similarity values from Table 1, we calculate the $XSim$ between documents (a) and (b) in Figure 1 as the average of 0.71 and 0.76, which gives 0.73.

Evidence	(`/person/name`, `employee/name`)	(`/person/address`, `employee/address`)
Content	0.64	0.79
Path	0.5	0.5
Node Label	1.0	1.0
Node	0.71	0.76

Table 1: Similarity values for the matching nodes in Figure 1.

4. EXPERIMENTS AND RESULTS

In order to evaluate our proposal's effectiveness we implemented a prototype system and performed some experiments with it. In this section, we report on the experiments that were performed.

4.1 Data generation

We were not able to find a real XML document set suitable for our experiments. A benchmark repository for XML matching is proposed in [25], but it was not implemented yet. So, we decided to work with synthetic document sets. We did that by using the data generator from a system called *Freely Available Biological Record Linkage*—Febrl [2].

Febrl is a *record linkage* framework. It has a dataset generator and several algorithms to detect duplicate records in relational databases. The dataset generator produces collections of records comprising fields with information about people, such as name, street, city, province, etc. The framework also provides tables with data extracted from the web (proper names, for instance) and geographical information (street names, suburbs, etc.).

The dataset generator also inject errors in the generated records, such as character transposition, insertion and removal. This way, it can generate one *original* record and several *copies* obtained by error injection. Figure 4 shows a subset of the data produced by Febrl's dataset generator. Each record has a unique `rec_id`, composed by the record number, an original/duplicate indicative, and the duplicate record number (when it applies).

The Febrl's data generator creates collections of records. In this work, however, we are working with XML documents. So, we implemented an XML Generator to create these instances from the Febrl's generated records. The XML Generator receives as input a template XML document along with a collection of records, and produces as a result a collection of XML documents. Figure 5 presents some sample DTD from documents created by the XML Generator.

4.2 Experimental method

We used the Febrl dataset generator to create a data file with 1,000 records, from which 200 are original records and

rec_id	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth
rec-2-org	maddison		94	apperly close		stanmore	2486	sa	
rec-3-org	samuel	mcelwee	500	nicholas street	corridella	valla	2207	nsw	19650416
rec-3-dup-0	samuel	mcelwee	500	nicholas street	corridella	vakls	2207	nsw	19650416
rec-3-dup-1	sajuel	mcelqee	500	nicholas street	corridella	valla	2207	nsw	19650416
rec-3-dup-2	samuel	mcelwee	500	nicholas street	corridella	vala	2270	nsw	19650416
rec-3-dup-3	samuel	mcelwde	500	hugh mckay crescent	corridella	valla	2207	nsw	19650416
rec-4-org	lochlan	white	23	hillebrand street		toorak	2033	wa	19620107
rec-4-dup-0	lochlan	white	22	hillebrans street		toorak	2033	wa	19620107
rec-4-dup-1	lochlan	white	21	hillebrand street		toorak	2033	wa	19620107
rec-4-dup-2	lochlan	whitd	23	hillebrand street		toora	2033	wa	19620107
rec-4-dup-3	lochlan	white	2	hillebrand street		toorak	2033	wa	19620107
rec-5-org	ellouise	white	62	nakala place		old beach	6018	vic	19150301
rec-5-dup-0	ellouise	wight	62	nakala place	urambi vlge	old beach	6018	vic	19150301
rec-5-dup-1	ello uie	white	62	nakala place		old beach	6018	vic	19150301
rec-5-dup-2	elysse	white	62	nakala place		old beach	6018	vic	19150311
rec-5-dup-3	ellouise	white	61	nakala place		old bach	6018	vic	19150301
rec-5-dup-4	ellouise	white	62	nakala place		old blesch	6018	vic	19150301

Figure 4: Sample data generated by the Febrl's dataset generator.

```
<!DOCTYPE person [
  <!ELEMENT person  (name, address, date-of-birth)>
  <!ATTLIST person id ID>
  <!ELEMENT name    (#PCDATA)>
  <!ELEMENT address (#PCDATA)>
  <!ELEMENT date-of-birth (#PCDATA)>
]>
```

(a)

```
<!DOCTYPE persons [
  <!ELEMENT persons  (person)>
  <!ELEMENT person   (#PCDATA)>
  <!ATTLIST person id ID>
]>
```

(b)

```
<!DOCTYPE employee [
  <!ELEMENT employee (name, address, job, salary)>
  <!ATTLIST employee id ID>
  <!ELEMENT name     (first, last)>
  <!ELEMENT address  (street, number, suburb, postcode, state)>
  <!ELEMENT first    (#PCDATA)>
  <!ELEMENT last     (#PCDATA)>
  <!ELEMENT street   (#PCDATA)>
  <!ELEMENT number   (#PCDATA)>
  <!ELEMENT suburb   (#PCDATA)>
  <!ELEMENT postcode (#PCDATA)>
  <!ELEMENT state    (#PCDATA)>
  <!ELEMENT job      (#PCDATA)>
  <!ELEMENT salary   (#PCDATA)>
]>
```

(c)

Figure 5: Sample DTDs generated for the experiments.

800 are duplicates. This data file was used as input for the XML Generator to create three distinct document sets. In each document set we generated an XML document for every record created by Febrl. We named these document sets according to the label of their root element:

- *Person*, whose documents conform with the DTD in Figure 5 (a);
- *Persons*, whose documents conform to the DTD in Figure 5 (b);
- *Employee*, whose documents conforme to the DTD in Figure 5 (c).

During the experiments, each document set was compared to the remaining two. In this process, each instance of the first document set was compared with all the documents of the other. This way, for each instance from the first document set we got a ranking with 1,000 similarity values, similar to the one shown in Table 2. Next, we used these ranking results to calculate the precision for each one of the eleven recall points (0.0, 0.1, ..., 1.0). Finally, the 1,000 resulting rankings were used to calculate the mean precision in each one of these points.

The functions used to test element's content, path and name were, respectively *Jaro-Winkler* [10, 11, 26], *PathSim* [22] and *EditDistance*. The values used for CONTENT-THRESHOLD, PATH-THRESHOLD, and NAME-THRESHOLD were, respectively, 0.5, 0.7 and 0.5.

4.3 Results

The recall-precision curves in Figure 6 shows the results of the experiments. They were produced by the application of the process we detailed in the previous section.

The figure shows three curves, one for each experiment. In the first experiment, we used only the document set named *Person*. The results appear in the figure as the curve labeled "Person x Person". Note that all the instances have the same structure, but their contents are different. We observed that all the relevant instances appeared in the beginning of the ranking, and the mean precision was 0.9978.

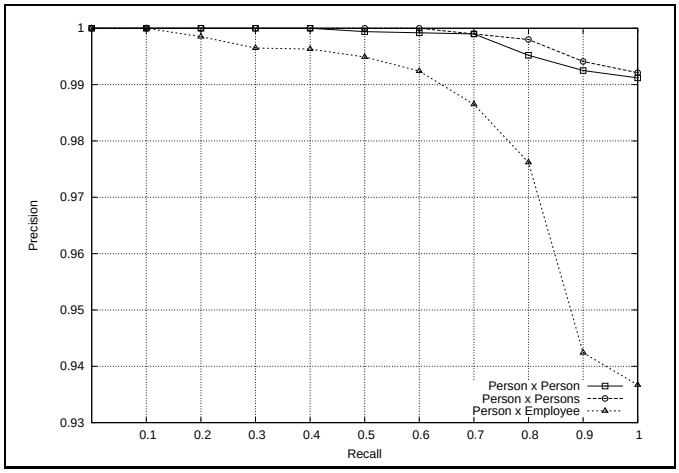

Figure 6: Recall-Precision curves from results of the experiments using the generated test document sets.

For the second experiment, we used the document sets named *Person* and *Persons*. The results appear in the figure as the curve labeled "Person x Persons". Each document was compared against the other 1,000 documents of the second document set. Note that this time the documents did not have the same structure, and their contents were also different. There was no significant change in the results. All the relevant documents appeared in the beginning of the ranking, and the mean precision was 0.9987.

For the third experiment, we used the document sets named *Person* and *Employee*. The results are shown in the figure as the curve labeled "Person x Employee". Each instance was compared against the other 1,000 instances of the second document set. Again, the documents didn't have the same structure, and their contents were also different. In this case, there was a slight change in the results when compared to the previous ones. However, the precision values were still high and the mean precision was 0.9836.

4.3.1 Future Improvements

The experiments results shown above allow us to say that our technique is effective in the task of matching XML documents. However, they also show that there is room for improvements. In this section, we analyze some situations where our technique did not perform well. We use this to highlight some weaknesses we had identified and to indicate further research directions.

Table 2 shows a partial ranking we got for document `rec-175-dup-8` in the "Person" document set, when compared with all documents in document set "Employee". Document `rec-175-dup-8` in "Person" document set was expected to match ten documents in document set "Employee": one original and nine duplicates.

In a perfect situation, all ten first positions in the ranking would be occupied by `rec-175-dup-8` original and duplicate documents. The experiment met this expectation up to the ranking position 5. All "Employee" documents in ranking positions 0 through 5 match the `rec-175-dup-8` document

Person Doc.	Employee Doc.	Rank	SimValue
rec-175-dup-8	rec-175-dup-8	0	0.816137566138
rec-175-dup-8	rec-175-dup-3	1	0.760846560847
rec-175-dup-8	rec-175-dup-6	2	0.760846560847
rec-175-dup-8	rec-175-org	3	0.741534391534
rec-175-dup-8	rec-175-dup-5	4	0.741534391534
rec-175-dup-8	rec-175-dup-4	5	0.73544973545
rec-175-dup-8	rec-91-org	6	0.722222222222
rec-175-dup-8	rec-175-dup-2	7	0.722222222222
rec-175-dup-8	rec-175-dup-7	8	0.721560846561
rec-175-dup-8	rec-175-dup-0	9	0.721560846561
rec-175-dup-8	rec-175-dup-1	10	0.718253968254

Table 2: Partial ranking for document `rec-175-dup-8` in the "Person x Employee" experiment.

```
<person>
   <name>patrick purodn</name>
   ...
</person>
```
(a)

```
<employee>
   <name>
      <first>patrick</first>
      <last>crouch</last>
   </name>
   ...
</employee>
```
(b)

Figure 7: Name element in two different documents from two distinct data sources : (a) `rec-175-dup-8` in `Person`; (b) `rec-91-org` in `Employee`.

in "Person" document set. The document `rec-91-org` in position 6, however, does not match with `rec-175-dup-8` and should not appear before the other `rec-175` documents in the ranking order.

In order to find out what went wrong in this process, we analyzed the `rec-175-dup-8` document in `Person` data source, along with the `rec-91-org` in the `Employee` data source. We discovered that the experiment found only one match between elements of the documents, which was the `name`.

Figure 7 shows the contents of both elements. We concluded that our system behave unexpectedly in this case because the similarity between the elements' content is above 0.5 (`patrick purodn` *versus* `patrick crouch`), and the elements also have a high structural similarity. The structural similarity is high because the path is half the same (`/person/name` *versus* `/employee/name`), and both comparing elements have the same label (`name`). Furthermore, since this was the only matching found, its value was assumed as the similarity value between the documents.

In order do reduce or avoid this kind of mistake, we are studying alternative techniques to compute the evidences. Currently, we are using a simple average to combine the evidences. Our analysis of the experiments tells us that we could get better results by using alternative techniques, such as weights to give more importance to one feature above the others. We already do this in a certain way by using the content threshold as a filter to select the elements for which we will evaluate the structure. We can also use "smarter" techniques to face this problem, including machine learning techniques such as bayesian networks. The inclusion of this techniques in our prototype and the execution of new experiments is work that still must be done.

Alternatively, we could penalize the document similarity value by decrementing the similarity value of those elements which does not match. This can be helpful in situations where only a few document nodes match, while the larger part of the document nodes does not.

Furthermore, our results can be also improved by using domain-specific structures, such as dictionaries and ontologies. The information provided by these structures can help us to infer correlations between the elements' contents, making easier to detect correspondences. Also, they can improve our techniques' performance with documents which use different vocabularies to label their elements.

5. CONCLUDING REMARKS

In this paper, we presented a novel technique for the problem of XML documents matching. This is an important step in the integration of XML documents. The integration of XML documents is a relevant task in several applications, such as data cleaning and XML query processing.

The main characteristic of this technique is the ability to make use of information from both the structure and the content of XML documents. We use three pieces of information (which we call *evidences*) to calculate the similarity between two nodes of XML trees: the element's content, the nodes' names and the nodes' path.

We implemented a prototype to evaluate our technique. The experiments were performed using synthetic databases. These databases consist of records generated by a system called Febrl [2] and converted to XML using an XML Generator we developed.

The experiments showed that our technique is capable of dealing with structural and content diversity which typically exists in XML documents create by distinct sources.

In the sequel of this work we are planning to include domain-specific structures to deal with vocabulary divergences. We are also considering the use of alternative techniques to combine the evidences we calculate from the documents.

6. REFERENCES

[1] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive Name Matching in Information Integration. *IEEE Intelligent Systems*, 18(5):16–23, Sept./Oct. 2003.

[2] P. Christen and T. Churches. *Febrl - Freely extensible biomedical record linkage (Manual, release 0.3)*, 0.3 edition, April 2005.

[3] W. Cohen, H. Kautz, and D. McAllester. Hardening Soft Information Sources. In *Proc. Sixth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '00)*, pages 255–259, 2000.

[4] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003*, 2003.

[5] J. C. P. de Carvalho and A. S. da Silva. Finding similar identities among objects from multiple web sources. In *Proceedings of the WIDM 2003*, pages 90–93, 2003.

[6] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, pages 83–95. Springer, 2003.

[7] A. K. Elmagarmid, P. G. Ipirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge And Data Engineering*, 19:1–16, 2007.

[8] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Integrating xml data sources using approximate joins. *ACM Transactions on Database Systems*, 31:161–207, 2006.

[9] M. A. Hernandez and S. J. Stolfo. Real world data is dirty: Data cleansing and the merge/purge problem. *Data Mining And Knowledge Discovery*, 2(1):9–37, Jan. 1998.

[10] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[11] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14(5–7):491–498, 1995.

[12] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi. A bag of paths model for measuring structural similarity in web documents. In *Proceedings of SIGKDD*, pages 577–582, 2003.

[13] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, 2004.

[14] L. Leitao, P. Calado, and M. Weis. Structure-based inference of xml similarity for fuzzy duplicate detection. In *Proceedings of the sixteenth ACM Conference on Information and Knowledge Management (CIKM)*, Lisboa, Portugal, November 2007.

[15] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original version in Russian in Doklady Akademii Nauk SSSR, 163(4):845-848, 1965.

[16] D. Milano, M. Scannapieco, and T. Catarci. Structure-aware xml object identification. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 29(2):67–76, 2006.

[17] A. E. Monge and C. P. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, (KDD)*, 1996.

[18] A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 2002.

[19] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. In *IEEE Bulletin of the Technical Committee on Data Engineering*, volume 23, dec 2000.

[20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. Eighth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '02)*, pages 269–278, 2002.

[21] D. Shasha and K. Zhang. Approximate tree pattern matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press, 1997.

[22] A. Vinson and C. Heuser. Similaridade de elementos xml baseada em caminhos. In *Anais do WTDBD2006 (Workshop do SBBD2006)*, Florianopolis, 2006.

[23] Y. Wang and S. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Proc. Fifth IEEE Int'l Conf. Data Eng. (ICDE '89)*, pages 46–55, 1989.

[24] M. Weis and F. Naumann. Dogmatix tracks down duplicates in xml. In *Proc. ACM SIGMOD international conference on Management of Data (SIGMOD'05)*, pages 431–442, 2005.

[25] M. Weis, F. Naumann, and F. Brosy. A duplicate detection benchmark for xml (and relational) data. In *Proceedings of SIGMOD Workshop on Information Quality for Information Systems (IQIS)*, 2006.

[26] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Wachington, DC, 1999.