University of Windsor

Scholarship at UWindsor

Computer Science Publications

School of Computer Science

2007

Xml Schema Matching

Jianguo Lu
University of Windsor

Ju Wang

Shengrui Wang

Follow this and additional works at: https://scholar.uwindsor.ca/computersciencepub

Part of the Computer Sciences Commons

Recommended Citation

Lu, Jianguo; Wang, Ju; and Wang, Shengrui. (2007). Xml Schema Matching. *International Journal of Software Engineering*, 17 (5), 575.
https://scholar.uwindsor.ca/computersciencepub/3

This Article is brought to you for free and open access by the School of Computer Science at Scholarship at UWindsor. It has been accepted for inclusion in Computer Science Publications by an authorized administrator of Scholarship at UWindsor. For more information, please contact scholarship@uwindsor.ca.

International Journal of Software Engineering
and Knowledge Engineering
Vol. 17, No. 5 (2007) 1–23
© World Scientific Publishing Company

XML SCHEMA MATCHING

JIANGUO LU
School of Computer Science, University of Windsor, Windsor ON, N9B 3P4, Canada
jlu@cs.uwindsor.ca

JU WANG
Open Text Corporation, Waterloo ON, N2L 0A1, Canada
jasonw@opentext.com

SHENGRUI WANG
Department of Computer Science, University of Sherbrooke,
Sherbrooke, Quebec J1K 2R1, Canada
Shengrui.wang@usherbrooke.ca

Received 18 May 2006
Accepted 16 October 2006

XML Schema matching problem can be formulated as follows: given two XML Schemas, find the best mapping between the elements and attributes of the schemas, and the overall similarity between them. XML Schema matching is an important problem in data integration, schema evolution, and software reuse. This paper describes a matching system that can find accurate matches and scales to large XML Schemas with hundreds of nodes. In our system, XML Schemas are modeled as labeled and unordered trees, and the schema matching problem is turned into a tree matching problem. We proposed Approximate Common Structures in trees, and developed a tree matching algorithm based on this concept. Compared with the traditional tree edit-distance algorithm and other schema matching systems, our algorithm is faster and more suitable for large XML Schema matching.

Keywords: Software reuse; software component search; schema matching; XML schema; tree matching algorithm; data integration.

1. Introduction

XML Schema has become an indispensable component in web application development. Schemas are used to represent all kinds of data structure in programming, and are often mapped to object models [28]. To some extent, we can think XML Schemas are similar to data types or classes in traditional programming language. What makes XML Schema different from traditional software components is that it is available on the web, encoded in XML, programming language independent, and adopted by all the major software vendors. All these features make XML Schema reuse not only imperative, but also have the potential to succeed beyond traditional software component reuse. We can envision that almost any data structure that you can think of will be available on the web. Programmers need a search tool to find the relevant schemas instead of developing the schema from scratch.

Our goal of research on schema matching has its root in software component search [20] and software agent search [29], both having a long history. [20] provides a good survey in component search, and [29] is the seminal paper on software agent matching, which also inspired numerous works on web service searching [22]. Since XML Schema is an inherent and major element of web services and modern software components, XML Schema matching is a foundation for the search of web serices, software agents, and software compoenents in general.

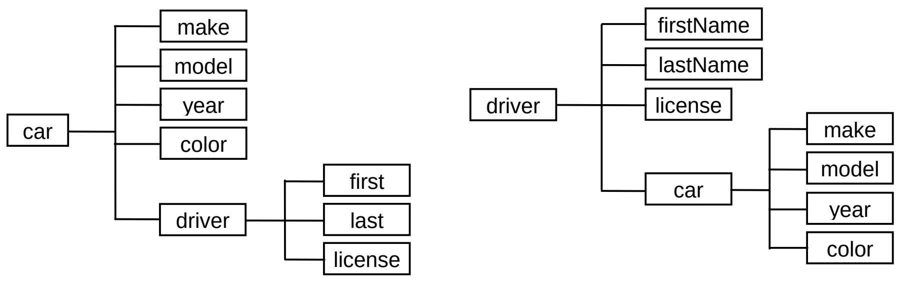
Schema matching is also widely studied in the database area [7, 16, 18, 19], with the aim to bridge relational and semi-structured data models, or to integrate data with either homogeneous or heterogeneous data models. [23] is a good survey of the works in this area.

There are a variety of schemas in schema matching research, ranging from XML related schema such as DTD and XML Schema, to relational and object schemas. We focus on XML Schema matching, instead of a hybrid matching system such as Cupid [18] that considers different schemas including relational database schema and XML Schema. The purpose of the matching is more on the overall similarity between two XML Schemas, instead of the concrete correspondence of the elements in two schemas.

Since schemas are usually modeled as trees [18, 27] or a similar format as directed acyclic graphs [7], tree matching has inevitably become one of the main issue in schema matching. Tree matching is an extensively studied problem. The classical tree edit distance matching algorithm [36] and many schema matching systems derived from this algorithm are not adequate for two reasons. One is that it is not fast enough as is shown in our experiment explained in Sec. 5. Another more important factor is that those algorithms must preserve the tree ancestor structure during the match, hence may miss better matches.

Take the two schemas in Fig. 1 for example. In those two schemas, there are two substrucrues that are very similar. One is about car information, the other one is driver information. Intuitively we would like to match those substructures. However, with the traditional tree edit distance algorithms, that kind of matching is not easy to achieve because shifting two sub-trees (e.g., exchange the position of driver information with car information in Schema 1) requires many edit operations.

Based on this observation, we generalized the concept of common substructures [3] between two trees to Approximate Common Sutstructures (ACS), and developed an efficient tree matching algorithm for extracting a disjoint set of the largest ACSs. This disjoint set of ACSs represents the most likely matches between substructures in the two schemas. In addition, the algorithm provides structural similarity estimate for each pair of substructures including, of course, the overall similarity between the two schemas. Using our algorithm to match the above car-



Schema 1, from car rental company Schema 2, from insurance company

Fig. 1. Car-driver schemas.

driver schemas, both driver and car nodes and their compoents can be matched, even though car is an ancestor of driver in schema one, and it is the other way around in schema two.

Some important features introduced in our algorithm include a search strategy and a recursive structural similarity computing used for comparing two substrees. These features are particularly adapted to schema matching. The search strategy allows a good trade-off between accuracy of structural similarity and time complexity. It focuses on comparing "root parts" of subtrees while still taking into account the structural similarity between other parts (those closer to leaves). This makes the algorithm very efficient and able to deal with large schemas.

Our system is designed with the objectives to work effectively and efficiently — generating good results in acceptable time, in order to be able to match real life schemas with several hundreds of nodes.

Figure 2 depicts the main phases of the processing performed by our matching system. First, an XML Schema is modeled as a tree. The second phase is the computation of node similarity, and the third is the computation of structural similarities between subtrees in the two Schemas. The following sections will discuss the three phases in order.

The system has been tested extensively using about 600 XML Schemas in total [17]. We evaluated both matching accuracy and computational efficiency of our system. Comparisons were made with the traditional edit distance tree matching algorithm [31] and a popular XML Schema matching system COMA[7]. The results show that our new tree matching algorithm outperforms these two methods, and can be used to match larger schemas that contain hundreds of elements.

Parts of the work in this paper are introduced in [17, 32].

2. Modelling XML Schemas as Trees

We model XML Schema as a *labeled unordered rooted tree*. In general, an XML schema corresponds to a directed graph in which recursive definitions are represented by loops and reference definitions are represented by cross edges. The graph

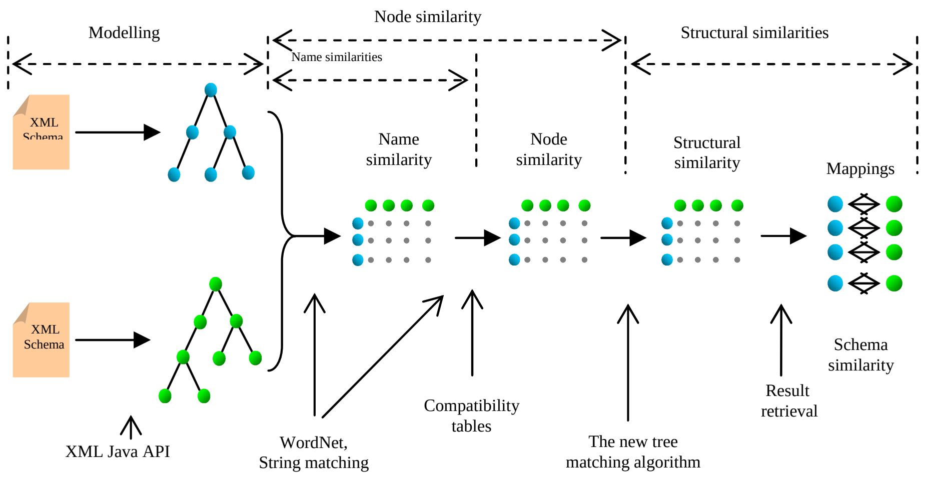


Fig. 2. Matching system developed.

representation is not adopted in our work for two reasons. First, intuitively the directed graph representation of an XML Schema still encompasses a hierarchical structure similar to a tree, with a few "loop" exceptions. Secondly and more importantly, approximate graph matching [3] is too computationally costly as we have investigated in [13]. Our recent algorithm in graph matching employed strong heuristics to reduce search space, but still can only deal with graphs with dozens of node [13]. Obviously, graph matching algorithms would be difficult to match XML Schemas with hundreds of nodes.

Each element or attribute of the schema is translated into a *node*. Attributes and elements that reside inside an element are translated as children of the element node. The names of elements and attributes, along with some optional information such as data types and cardinalities, are the labels of the nodes.

The tree structure reflects the nesting relations of elements and attributes in a schema. Although by XML Schema standard the order of the elements matters, it is ignored in our tree model based on the assumption that the order does not make differences as big as changing the labels. The modelled tree does not include every detail of an XML Schema. Excluded information falls into two categories. One is related to elements or attributes such as default value and value range. The other is relevant to structure, such as element order indicators.

Modelling XML Schema is a tedious task due to the complexity of XML Schema. During the modelling, we need to take care of the following constructs in XML Schema, to insure that a schema is modelled as a tree correctly.

Reference Definition

Reference definition is a mechanism to simplify schema through the sharing of common segments. To transform this structure into a tree, we duplicate the shared segment under the node that refers to it. By doing this, we increased the number of nodes. In implementation of the modelling, we create an array which contains the distinct node labels and establish connections from each node to this array. In subsequent processes, the node labels are handled based the array instead of the nodes themselves.

There are two types of references in XML Schema specification: data type reference and name reference. Data type reference is created by the clause 'type=dataTypeName' (where 'dataTypeName' is not a built-in data type), and the referred segment is a <*complexType*> or <*simpleType*>; while name reference is created by 'ref=elementName', and referred segment must be a <*element*>. All the referred types or elements must be top level such that they are nested in <*schema*> only. Therefore, our solution is that: build two lists called '*referred*' and '*referring*', list '*referred*' contained all the top level elements and types (both complex and simple), and list '*referring*' contain the elements having '*type*' or '*ref*' reference; then after scanning the schema file, for every element in '*referring*', we physically duplicate the segment which they refer. Solving those segments which are from outside of the schema file follows the same method as importing and inclusion.

Recursive definition

Recursive definition happens when a leaf element refers to one of its ancestors. This definition also breaks the tree structure, and it has to be solved differently from the way of solving reference definition, otherwise it falls into an infinite loop.

Matching recursively defined node is equivalent to matching the inner node being referred. So we utilize a detecting procedure, which scans the path from a node up to the root of the tree to find out whether this node refers to its ancestor or not. Once a node which has recursive definition is found, we cut the connection and mark the node with recursive property to distinguish it from its referred ancestor.

Namespace

Namespace is a way to avoid name ambiguity, such as two same data type names in one schema file, by assigning them to different vocabularies. This is accomplished by adding unique URIs and giving them aliases. The aliases serve as prefixes, such as 'xsd:' in the example, to associate the terms with certain vocabularies — namespaces. In our implementation, namespace affects reference definitions in three ways: built-in data type, user-defined data type, and element reference. To support this feature, our program tracks every prefix and its corresponding URI, takes them and the term right after the prefix as one unit, then put this unit into the reference solving.

Importing and including

Importing and *including* are mechanisms of reusing elements and attributes defined in other schema files. Including limits the sharing within the same namespace, and importing can cross different namespaces. When being imported, the imported schema file's information is provided in the <*import*> tag, including the file name, location and the imported namespace. Our program also parses and models this schema, then together with its namespace, brings its top level elements and types into the 'referred' list. If any of them are referred by the components in the original schema file, they will be handled by the reference solving process. For including, the included file's information is kept in <*include*> tag, and the same method is applied to solve including with the difference of namespace. The namespace for including is the same as the original schema file.

Extension

Extension allows new elements and attributes being added. For this situation, we first need to solve the type reference, so we treat the base clause the same as type reference. After getting the base type being duplicated, we process the newly added components, converting them to nodes and join them as siblings to the duplicated ones.

Grouping

Grouping is similar to complex type definition, providing a way of reusing predefined components. The most often used grouping is attribute grouping, which is specified by <*attributeGroup*> tag. We use the same way as type reference to solve this situation, i.e., add the <*attributeGroup*> definition and reference element to the '*referred*' list, then duplicate the referred group.

3. Node Similarity

Since a label of a node consists of name, datatype, and cardinality information, the node similarity is computed based on these entities. Among them the name similarity is the most complex one.

3.1. *Name similarity*

Name similarity is a score that reflects the relation between the meanings of two names, such as tag name or attribute name, which is usually comprised of multiple words or acronyms. The steps of computing name similarity include tokenization, computing the semantic similarities of words by WordNet, determing the relations of tokens by a string matching algorithm if they can not be solved by WordNet, and calculating the similarity between two token lists.

Tokenization

Quite often a tag name consists of a few words. It is necessary to split up the name into tokens before computing the semantic similarity with another one. This operation is called *tokenization.* A token could be a word, or an abbreviation. Although there are no strict rules of combining tokens together, conventionally, we have some clues to separate them from each other such as case switching, hyphen, under line, and number. For instance: 'clientName' is tokenized into 'client' and name, and 'ship2Addr' to 'ship', '2', and 'add'.

Computing semantic similarity using WordNet

Once a name is tokenized into a list of words, we use WordNet [32] to compute the similarity between the words.

WordNet builds connections between four types of POS (Part of Speech), i.e., noun, verb, adjective, and adverb. The smallest unit in WordNet is synset, which represents a specific meaning of a word. It includes the word, its explanation, and the synonyms of this meaning. A specific meaning of one word under one type of POS is called a sense. Each sense of a word is in a different synset. For one word, one type of POS, if there are more than one sense, WordNet organizes them in the order from the most frequently used to the least frequently used.

There are different kinds of relations between words, such as hypernym, hyponym, antonym, coordinate, etc., and these relations are connected on synsets. WordNet APIs for different programming languages have been developed by several groups [1, 2].

Based on WordNet and its API, we use synonym and hypernym relations to capture the semantic similarities of tokens. Given a pair of words, once a path that connects the two words is found, we determine their similarity according to two factors: the length of the path and the order of the sense involved in this path.

Searching the connection between two words in WordNet is an expensive operation due to the huge searching space. We impose two restrictions in order to reduce the computational cost. The first one is that only synonym and hypernym relations are considered, since exhausting all the relations is too costly. This restriction is also adopted in some related works [1, 2]. Another restriction is to limit the path searching process to a certain number of length. If a path has not been connected within a length limit, we stop further searching and report no path found.

In our implementation, we use the following formula to calculate the semantic similarity:

$$wordSim(s,t) = senseWeight(s)^{*} senseWeight(t)/pathLength$$

where s and t denote the source and target words being compared. *senseWeight* denotes a weight calculated according to the order of this sense and the count of total senses.

We performed a comparison with seven other approaches on the set of word pairs in [14]. In terms of correlation, ours exceeds four approaches and falls behind three of them. Considering that the method we use is simpler and scalable, our similarity measure is acceptable.

Similarity between words outside vocabulary

Words outside the English vocabulary are often used in schemas definition, such as abbreviations ("qty" for quantity) and acronyms ("PO" for purchase order). In this case WordNet is no longer applicable, and we use edit-distance string matching algorithm. By doing this, the measurement reflects the relations between the patterns of the two strings, rather than the meaning of the words.

Similarity between token lists

After breaking names into token lists, we determine the similarity between two names by computing the similarity of those two token lists, which is reduced to the bipartite graph matching problem [15]. It can be described as follows: the node set of a graph G can be partitioned into two subsets of disjoint nodes X and Y such that every edge connects a node in X with a node in Y, and each edge has a non-negative weight. The task is to find a subset of node-disjoint edges that has the maximum total weight.

When X and Y are two token lists and the edges are the similarities between the tokens, the token list matching problem is reduced to the bipartite matching problem. We use the efficient Hungarian method [15] to solve the weighted bipartite graph matching.

The semantic similarity between token lists is also normalized to a value between 0 and 1. As a result, we shall compute the average based on the summation of similarity, dividing the summation by the median of token counts.

3.2. *Similarity of built-in data type*

XML Schema has 44 built-in data types, including nineteen primitive ones and twenty-five derived ones. To reduce the number of combinations, we create seven data type categories, i.e., *binary*, *boolean*, *dataTime*, *float*, *idRef*, *integer*, and *string* that cover the 44 data types. The compatibility table is built for the seven categories. After this, when comparing two data types, first we check which category these types belong to, then extract the similarity measure from the category compatibility table.

3.3. *Similarity of cardinalities*

XML Schema allows the specification of minimum and maximum occurrences, i.e., cardinality, for elements. The range of cardinality is from 0 to unbounded. It is impossible and unnecessary to compare all the cardinalities in this range. As a result, we apply a threshold. When cardinalities are equal to or bigger than it, we treat the cardinality as this threshold.

4. Approximate Tree Matching

In schema matching, edit-distance based algorithms are not adequate solutions for two reasons. One is that it is not fast. Another more important factor is that the algorithm must preserve the tree ancestor structure during the match, hence may miss some better matches.

October 17, 2007 17:44 WSPC/117-ijseke 00344 **1st Reading**

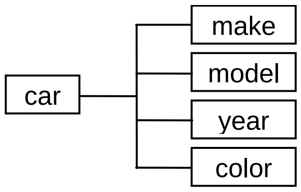


Fig. 3. A possible common substructure between the example Schemas.

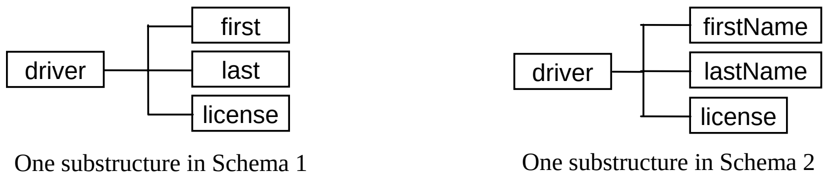


Fig. 4. Two similar substructures that are not common substructures.

To understand this latter point, let us consider the two schemas in Fig. 1. The two schemas are not similar because of the structural difference. However, the two pairs of substructures, shown in Figs. 3 and 4, are indeed similar. Figure 3 shows a common substructure of the two schemas, while Fig. 4 shows two very similar substructures although they are not a common substructure in the classic sense. Their extraction allows some further interesting comparison between the two Schemas. Using a classical tree matching algorithm to extract these common substructures would mean a very costly process of running the algorithm on each possible pair of subtrees, demanding many edit operations.

Based on this observation, we propose a concept of Approximate Common Substructures (ACS) between two trees and developed an efficient tree matching algorithm for extracting a disjoint set of the largest ACSs. This disjoint set of ACSs represents the most likely matches between substructures in the two schemas. Indeed, the algorithm provides structure similarity estimate for each pair of substructures including, of course, the overall similarity between the two schemas. Using our algorithm to match the above car-driver schemas, both driver and car nodes and their components can be matched, even though the car is an ancestor of driver in schema one, and it is the other way around in schema two.

Some important features introduced in our algorithm include a search strategy and a recursive structure similarity computing used for comparing two subtrees. These features are particularly adapted to schema matching. The search strategy allows a good trade-off between accuracy of structure similarity and time complexity. It focuses on comparing "root parts" (i.e. low level parts) of subtrees while still taking into account the structure similarity between other parts (i.e. higher level parts, those closer to leaves). This is the key heuristic that makes the algorithm efficient in time and enables it to deal with large schemas.

In the following, we use $nodeSim(u, p)$, computed at the second stage in our system, to represent the (semantic) similarity between the nodes u and p from the two trees. $nodeSim(a, p) = 1$ means that u and p are the same.

4.1. *Approximate Common Substructure (ACS)*

The concept of ACS generalizes the conventional concept of the *common substructure* [3]. Given two trees T_1 and T_2, the concept of ACS is related to subtree matching and every pair of subtrees can be considered as being an ACS. A quality measure, defined as the structure similarity, is necessary to distinguish between a "good" ACS and a "not very good" ACS. Formally, the structure similarity between the two substructures $subStr_1$ from T_1 and $subStr_2$ from T_2 can be defined as follows:

$$structSim(subStr_1, subStr_2) = \max_M C(M)$$

where M is any mapping between the node set of $subStr_1$, $Nodes(subStr_1)$, and the node set of $subStr_2$, $Nodes(subStr_2)$ satisfying the following conditions:

(1) If $(u, p) \in M$ and $(v, q) \in M$, and $u = v$ then $p = q$;
(2) If $(u, p) \in M$ and $(v, q) \in M$, then u is v's ancestor if and only if p is q's ancestor;

And the similarity measure $C(M)$ should satisfy the following conditions:

— $0 \leq C(M) \leq 1$, if M is not an isomorphism between $subStr_1$ and $subStr_2$, if and only if $subStr_1$ and $subStr_2$ are a common substructure (i.e. M is an isomorphism between $subStr_1$ and $subStr_2$ in which all the corresponding nodes are the same).
— $C(M)$ is positively proportional to the size of M (and negatively proportional to the number of unmatched nodes);

The M that gives rise to $structSim(subStr_1, subStr_2)$ defines the ACS between the two substructures. Obviously, the closer $structSim(subStr_1, subStr_2)$ is to 1, the more there are similar nodes structured in the same way as in $subStr_1$ and $subStr_2$, i.e. M is larger. Unfortunately, as in the case of computing $structSim(subStr_1, subStr_2)$ edit distance, the problem of computing is also NP. In our matching algorithm, $structSim(subStr_1, subStr_2)$ is replaced by an approximate similarity function $treeSim()$ while $subStr_1$ are limited to rooted subtrees.

Another important measure that we have used in our algorithm is the matching percentage defined as the ratio of the number of nodes in a potential ACS to the average number of nodes in the two trees. It is used together with $structSim(subStr_1, subStr_2)$ (or in practice $treeSim()$) in order to favor large ACSs. An optimal ACS reaches a trade-off between structural similarity and matching percentage.

We can now state more formally the objective of our matching algorithm. The algorithm aims to find a unified mapping M_{schema} composed of the set of all

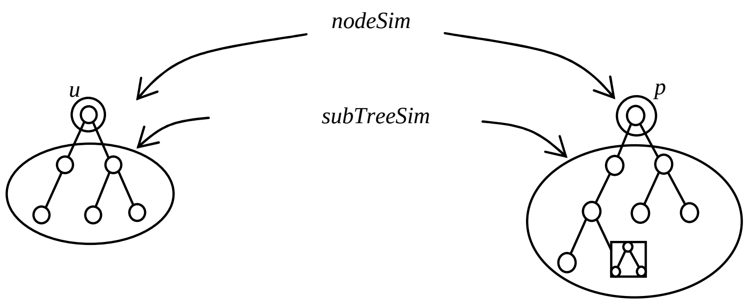


Fig. 5. Basic idea of the proposed matching algorithm.

mappings derived from **disjoint** ACSs

$$M_{\text{schema}} = m_{ACS1} \cup m_{ACS2} \cup \cdots \cup m_{ACS_i} \cup \cdots \cup m_{ACS_n}$$

such that each ACS_i has a combined score of the structural similarity and the matching percentage beyond a fixed threshold. It is important to notice that while ancestral relations are preserved within each ACS_i (or by the mapping m_{ACS_i}); they do not have to be preserved in the unified mapping M_{schema}. This is a distinctive feature of the algorithm proposed in this paper that makes the matching of the above car-driver Schemas possible.

4.2. *The matching algorithm*

For efficiency reason, we use the following definition of (sub-) tree similarity. Given a node pair (u, p), where $u \in T$ and $p \in T_2$ respectively, the structure similarity, $treeSim(u, p)$, between the subtrees rooted at u and p is defined as follows:

$$treeSim(u, p) = \alpha^* nodeSim(a, p) + (1 - \alpha)^* subTreeSim(u, p)$$

where $subTreeSim(u, p)$ represents the similarity computed based on the subtrees rooted at u and p, and is the major concern of the algorithm. α is a factor whose value is between 0 and 1, which reflects the weight of the two parts. This definition, easily justifiable for a large number of real applications, is deliberately in favor of the root parts of the two subtrees and suggests a recursive approach to matching the two trees. Figure 5 outlines the general idea of the approach adopted for the new algorithm.

The subtree similarity corresponds in fact to the similarity between two forests under u and p. Since the size of each subtree in the forests can be very large, we trim the size of each subtree to two levels by considering each subtree beyond the level 2 as a super-node as shown in the forest under (p) in Fig. 5. The choice of 2 levels has been made here based on a consideration of trade-off between complexity and accuracy. The procedure for computing $sinTreeSim(u, p)$ includes matching trees in the two forests and computing node-to-node similarities. The following conditions have been taken into account:

(1) Preservation of ancestor relation: if u_1 and u_2 from the forest under u are matched respectively to p_1 and p_2 from the forest under p, then u_1 is a parent of u_2 if and only if p_1 is a parent of p_2.
(2) Deletion operation: if a node u_1 is deleted, all the child nodes of u_1 will be moved up to become children of the parent of u_1. This (edit) operation makes matching of nodes at different levels possible. However, in order to further reduce the computational complexity without seriously affecting the matching optimality, this operation is applied only to the nodes at the root level of each subtree in the forest, i.e. the original child nodes of u (or p).
(3) Node-to-node similarity $\mathit{forestNodeSim}(u_1, p_1)$: when comparing two "normal" nodes u_1 and p_1, $\mathit{forestNodeSim}(u_1, p_1) = \mathit{nodeSim}(u_1, p_1)$. If at least one of the two nodes is a super-node, then $\mathit{forestNodeSim}(u_1, p_1) = \mathit{treeSim}(u_1, p_1)$. The recursive nature of this algorithm guarantees that the structural similarity $\mathit{treeSim}(u_1, p_1)$ is computed before $\mathit{treeSim}(u, p)$.

To compute $\mathit{treeSim}(u, p)$, we distinguish three cases: (1) both u and p are leaves; (2) one of them is a leaf and the other one is an inner node (the number of descendants is greater than 0); and (3) both of them are inner nodes (Fig. 5 shows only this general case).

Case 1: Both u and p are leaves. In this case, we define

$$\mathit{treeSim}(u, p) = \mathit{nodeSim}(u, p)$$

Case 2: One of the two nodes is a leaf, another one is an inner node. Suppose that u is a leaf, then

$$\begin{aligned} \mathit{treeSim}(u, p) &= \alpha^{*}\mathit{nodeSim}(u, p) + (1-\alpha)^{*}\mathit{subTreeSim} \\ &= \alpha^{*}\mathit{nodeSim}(u, p) + (1-\alpha)^{*}\frac{\beta}{\sqrt{1 + |\mathit{descendants}|}} \end{aligned} \tag{1}$$

where $|\mathit{descendants}|$ denotes the number of descendants of the non-leaf node. $\sqrt{1 + |\mathit{descendants}|}$ is a penalty factor that reflects the difference between the two forests under u and p. β is a user-defined parameter which is set to 0.3 in our experiments.

Case 3: Both u and p are inner nodes.

$$\begin{aligned} \mathit{treeSim}(u, p) &= \alpha^{*}\mathit{nodeSim}(u, p) + (1-\alpha)^{*}\mathit{subTreeSim} \\ &= \alpha^{*}\mathit{nodeSim}(u, p) + (1-\alpha) \\ &\quad \times \max_{M}\left\{\frac{\sum_{(i,j)\in M} \mathit{forestNodeSim}(i, j)}{\sqrt{1 + |\mathit{deletedChild}| \cdot |M|}} \cdot \frac{|M|}{\mathit{NbLF}}\right\} \\ &= \alpha^{*}\mathit{nodeSim}(u, p) + (1-\alpha) \\ &\quad \times \max_{m}\left\{\frac{\sum_{(i,j)\in M} \mathit{forestNodeSim}(i, j)}{\sqrt{1 + |\mathit{deletedChild}| \cdot \mathit{NbLF}}}\right\} \end{aligned} \tag{2}$$

In this equation, M is any mapping built following the above conditions 1 and 2, i.e. M is an ancestor order preserving mapping between the remained forest under u and the remained forest under p once certain immediate children of u and p are deleted. $NbLF$ is the number of nodes in the larger (remained) forest. Formula (3) can be interpreted as follows. $\frac{\sum_{(i,j)\in M} forestNodeSim(i,j)}{|M|}$ is the average similarity between the matched nodes. This average similarity is penalized by two factors. One is related to deleted nodes (division by $\sqrt{1+|deletedChild|}$) and the other one is related to percentage of non-matched nodes (multiplication by $\frac{|M|}{NbLF}$). This formula materializes the goal of the matching, which is to search the best ancestor order preserving correspondence between the two forests in terms of the similarity and the number, while limiting the number of deletions.

There is a gap between **Case 1** and **Case 3**, since the two cases represent completely different situations. While it is relatively easy to justify the design principles for formulas (1) and (3), it is not as easy to justify Formula (2) for **Case 2**, which represents the middle situations. Formula (2) designed for the current system is more consistent with Formula (3) for **Case 3**, if we consider that since one of the nodes (say u) is a leaf node, we have to delete all the nodes in the forest under p in order to obtain two identical "remained forests". This design provides a fine grading of the difference between the structures of the two forests. One might want to adopt, for **Case 2**, the formula $treeSim(u,p) = \alpha^{*} nodeSim(u,p)$, which is more consistent with formula (1).

The most difficult task in computing $treeSim(u,p)$ is to generate all the mappings M in **Case 3**. An enumerative approach could be used, particularly if the optimal solution is necessary. A more efficient approach is to consider M as a state in a state space and to search a good, if not optimal, solution by exploring the space. Given an M, it can be altered by a number of actions on the forest under each root. These actions are

(1) Deleting an original child of each root;
(2) Add back a deleted child node;
(3) Matching/re-matching nodes between the two forests.

We have adopted the same *Iterative Improvement* heuristic [25] to searching the state space. Since this hill climbing method can easily result in local maximum, we have adopted the strategy of running the algorithm with a different initial state. The number of restarts has been fixed to be 5 or less in our experimentations.

The matching process starts with leaf nodes of the two trees and goes upwards. Each pair of subtrees will be matched after all their pairs of subtrees have been matched. The output of the matching algorithm is all the similarity and the corresponding mapping for each pair of subtrees.

4.3. *Identifying ACSs*

To identify the ACSs, each node pair which represents two matching subtrees is used to represent a potential candidate. The qualified ACSs are identified from

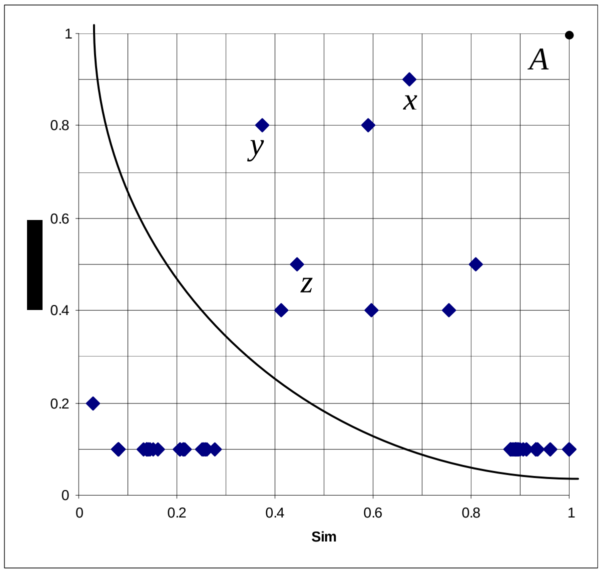


Fig. 6. Distance to perfect matching point as threshold.

these candidates by applying a combination of thresholds of structural similarity and matching percentage. Here the matching percentage for an ACS candidate, denoted as $mPer(u,p)$, is computed as the total number of matched nodes enclosed in this ACS candidate divided by the average number of nodes for the two trees. If we use $ACS(u,p)$ to denote the ACS candidate rooted at u and p, $|ACS(u,p)|$ to denote the number of matched node pairs, then we have:

$$mPer(u,p) = \frac{|ACS(u,p)|}{|T_1| + |T_2|)/2} = \frac{2|ACS(u,p)|}{|T_1| + |T_2|}\,.$$

Both *treeSim*(u,p) and *mPerf*(u,p) should be considered in determining the qualified ACSs which reach the balance of high structural similarity and matching percentage. If we project these two values for every node pair into a two-dimensional plane, we will get a scatter chart similar to the one in Fig. 6. In this chart, the horizontal axis represents the structural similarity, the vertical axis is the matching percentage, and each point denotes the result of an ACS candidate. Point $A(1,1)$ represents a perfect matching — both the structural similarity and matching percentage reach the maximum value. Obviously, the points near A reflect the good ACS candidates. Points near $D(0,0)$, in this chart, represent the poorest situation — low similarity and low matching percentage, and points near $B(0,1)$ and $C(1,0)$ represent the situations that only one of the values is high. Generally, most points fall into the area in between.

To determine good ACSs from the scatter chart, we use the Euclidean distance from the node to the perfect matching point A. The idea of this approach is shown in Fig. 6: the arc represents those points whose distance to point A is equal to the threshold, therefore the points covered by the sector will be considered as admissible candidates to generate the ACSs.

Extracting the ensemble of disjoint ACSs is done in the following way. By ordering all the admissible candidates according to their distances to Point $A(1,1)$, a list of candidates is established. Candidates at the top of the list are those that are closer to $A(1,1)$ and are considered to be better candidates. The extraction consists in comparing each candidate from the list with all the subsequent candidates from the lists and eliminating those conflicting candidates. Using $ACS(u,p)$ and $ACS(u_1,p_1)$ to represent two such candidates, the comparison/elimination is done by applying the following rules:

(1) If the pair of subtrees corresponding to (u,p) and the pair of the subtrees corresponding to (u_1,p_1) are disjoint, do nothing;
(2) If one of the subtrees is included in another, i.e. either $subtree(u_1)$ is a subtree of $subtree(u)$ or vice versa, or $subtree(p_1)$ is a subtree of $subtree(p)$ or vice versa, two situations should be dealt with. We suppose $subtree(u_1)$ is a subtree of $subtree(u)$, then
    (i) If $subtree(p_1)$ is a subtree of $subtree(p)$ and there is at least one node involved in $ACS(u_1,p_1)$ that is also involved in $ACS(u,p)$, discard $ACS(u_1,p_1)$, i.e. eliminate $ACS(u_1,p_1)$ from the above list;
    (ii) If $subtree(p_1)$ is not a subtree of $subtree(p)$ then
        1. if $subtree(u_1)$ is not involved in $ACS(u,p)$, do nothing;
        2. if there is at least one node from $subtree(u_1)$ involved in $ACS(u,p)$, discard either $ACS(u,p)$ or $ACS(u_1,p_1)$ depending on their distance to $A(1,1)$.

These rules are applied to all the candidate pairs iteratively until no more candidates are eliminated. All the remaining candidates constitute the final ensemble of disjoint ACSs that forms the unified mapping between the two Schemas, i.e. $M_{\text{schema}} = m_{ACS_i} \cup m_{ACS2} \cup \cdots \cup m_{ACS_i} \cup \cdots \cup m_{ACS_n}$. The number of these ACSs is obviously dependent on the value of threshold distance. As shown in Fig. 6, the threshold distance ranges from 0 to $\sqrt{2}$. The best value is problem specific and depends on the number of nodes in the two trees and how close the two trees are. Usually, it is easy to determine this after a few tests. In our experiments, the value of threshold ranges from 0.88 to 1.01.

4.4. *Reporting results — mappings and Schema similarity*

Retrieving mappings is relatively straightforward once M_{schema} is identified. Each mapping is reported as two strings in XPath format, i.e. a string of names from the root element (always 'schema') to this matched element, and the names are delimited by slash, e.g. schema/car/driver/first and schema/driver/firstName. Note that the root element of an XML Schema is always 'schema', so we do not treat the root to root as a mapping. The similarity of the two Schemas is simply the structural similarity of the two roots of the trees, as $ACS(r_1, r_2)$ has been computed indeed during the Schema matching process. It is to be pointed out that in general, there is no similarity value associated to M_{schema}.

5. Experiment

Our system is compared with the traditional edit distance tree matching algorithm for labeled unordered trees [25] that is implemented by us, and the popular schema matching system COMA [7].

5.1. *Data*

The experiments are performed on the XML Schemas which we collected from various sources. The first group comprises five purchase order schemas which are used in the evaluation of COMA [7]. We choose the same test data to compare with COMA. The second group includes 86 large schemas from www.xml.org. These are large schemas that are proposed by companies and organizations to describe the concepts and standards for particular areas. We use these large schemas to evaluate system efficiency. The third group consists of 95 schemas that are collected from HITIS [12]. These schemas are designed to be the standards of interfaces between hospitality related information systems, such as hotel searching, room reservation, etc. Group four consists of 419 schemas extracted from WSDL files that describe the schemas of the parameters of web service operations. These schemas are small in general. Groups three and four are used to test the accuracy of our matching system. Since most of them are relatively small, they are easy to read and judge manually.

5.2. *Accurary*

5.2.1. *Comparison with edit-diatance algorithm*

Figure 7 compares the precision and recall between our algorithm (method 1) and edit distance algorithm (method 2). The test cases are from data group 1, which consists of 5 purchase orders that are also used in COMA.

The figure shows that our algorithm outperforms the edit distance tree matching algorithm consistently. Both algorithms adopt node removal operation and use iterative improvement heuristic to search the approximate result. The major difference between these two algorithms is that we deal with two nodes (one for each tree) each time, recursively match two trees from leaves to roots, and the node removal operation is limited to the child level of current nodes only. The edit distance tree matching algorithm always takes two trees, tries to remove some nodes in the range of entire trees each time, compares and keeps the state with smallest distance. Reviewing these five purchase order schemas supports our schema properties observation again — similar concepts described by XML are made up of similar elements, and these elements are constructed in similar ways. Simply speaking, good

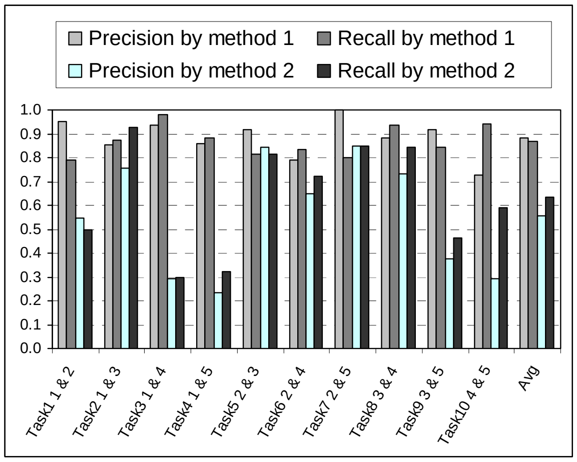


Fig. 7. Precision and recall for our method (method 1) and edit-distance algorithm (method 2).

Table 1. COMA and our algorithm.

	COMA (All)	COMA (All+SchemaM)	Our algorithm
Precision	0.95	0.93	0.88
Recall	0.78	0.89	0.87
Overall	0.73	0.82	0.75

mappings between two similar schemas could be found by a few node removal operations. Our algorithm takes advantage of this condition and limits the range of node removal. Therefore it removes less nodes, but achieves better result. On the other hand, for the edit distance tree matching algorithm, when the input size is large, the wide range of node removal increases the searching space and decreases the chance of getting good mappings.

5.2.2. *Comparison with COMA*

COMA maintains a library of different matchers (matching methods) and can flexibly combine them to work out the result. It introduced a manual reuse strategy which can improve the results but needs human assistance. Besides precision and recall, COMA adopts the overrall measurement that combines precision and recall.

We focus on two matcher combinations in COMA, i.e., 'All' — the best no-reuse combination, and 'All+SchemaM' — the best reuse involved combination. Together with the result of our matching system, the precision, recall and overall measure are compared in Table 1.

From this table, we can conclude that in terms of overall accuracy, our matching system outperforms COMA 'All' combination, and falls behind 'All+SchemaM'

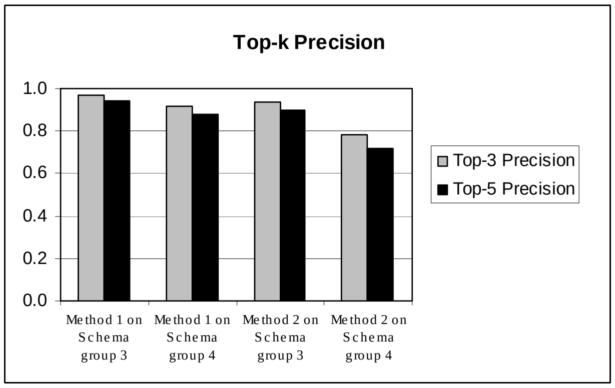


Fig. 8. Comparison of top-3 and top-5 precision.

combination on matching the given five purchase order schemas. Considering the 'All+Schema' needs human assistance, our matching system works well.

5.2.3. *Top-k precision*

We use Top-k precision method to assess the schema relations reported by our algorithm and tree edit distance algorithm. Top-k precision is defined as

$$p_{\text{Top-k}} = |ReportCorrect_k|/k\,.$$

where $ReportCorrect_k$ is the set of correct results in the top-k return ones. The experiment for assessing the schema relations is performed on data group three and four, and is designed as follows: in each group, we randomly pick a schema; compare it with every schema in this group using both of the algorithms; then we sort the returned schemas. Next, we take the union of top-k schemas from the two lists, subsequently, based on the union set, we manually determine which schema(s) should not be ranked in top-k, and finally compute the top-k precision for each algorithm. In order to get better overall measurement, we compute top-3 and top-5 precisions, repeat above process, and take averages. Figures 8, 9 and 10 summarize the evaluation results which are based on 10 random schemas in group 3 and 20 schemas in group 4.

The result shows that (1) using either algorithm to matching a schema group, top-3 precision is better than top-5 precision; (2) both algorithms get better precision on schema group 3; and (3) our algorithm gets better overall results than the edit distance algorithm.

The reason of better top-3 and top-5 precisions for group 3 is that all the schemas in this group are collected from one domain. Most files have similar pieces of information, a few of them are even identical.

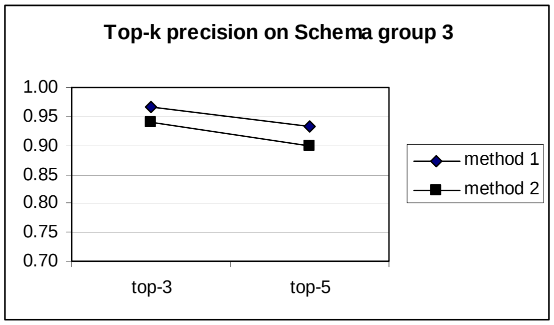


Fig. 9. Top-k precision on data group 3.

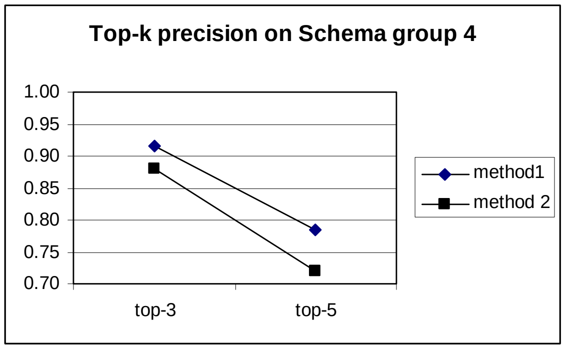


Fig. 10. Top-k precision on data group 4.

5.3. *Performance*

The performance is assessed using group two that consists of 86 large schemas. This experiment is performed on a computer with single Intel Pentium 4 3.0GHz CPU and 1G memory. The operating system is Red Hat Linux release 9. Every two schemas in this group are matched, so there are 3655 matching tasks in total. Due to the high computation cost of method 2, we bypass this method for schemas that exceed 150 nodes. Therefore, the count of matching tasks that the two algorithms participate is different.

Figure 11 shows the execution times of the three methods. We divide the input size, represented by the multiplication of node count of the two trees into several intervals, then count the number of matching tasks, and calculate the average execution times for each interval. As we can see, for method 2, there are only six matching tasks when input size is from 16 k to 20 k, and there is no task when input size is over 20 k.

It illustrates the increasing trend for all of the three execution times while the input size gets large. Besides, we can conclude that the preparation part is a heavy

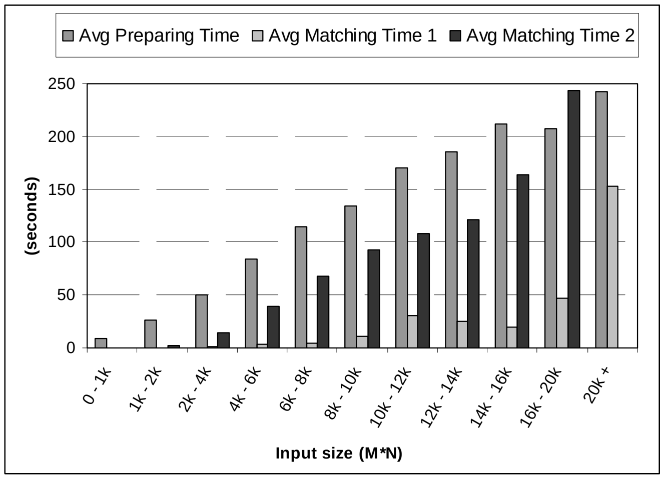


Fig. 11. Execution time.

job, and the new tree matching algorithm is faster than the edit distance tree matching algorithm.

There are some tasks in preparing part, including modelling, computing node similarity, and preparing related data structures for later matching. Clearly, the majority cost is spent on computing node similarity, and more specifically, on computing semantic similarities. Computing semantic similarities is a very expensive task: given two words, the program exhausts their relations stored in WordNet, and tries to find the highest ranked connection. Even through we restrict the relation to synonymy and hypernym only, the searching space is still huge. However, we could adopt some alternatives to reduce the dependence of WordNet, such as reuse pre-calculated result and build user-specified similarity tables.

Our tree matching algorithm is faster than the edit distance tree matching algorithm. Due to the same reason described in the previous section, our tree matching algorithm limits node removal operation, therefore it reduces the searching space.

In conclusion, compared with the edit distance tree matching algorithm, our algorithm generates better results in shorter time for most of the matching tasks, especially when input size is large. Therefore it is more applicable in real life schema matching problems.

5.4. *Implementation of the matching system*

This matching system is developed using Java. SAX XML parser in Sun JAXP package is used to parse XML schema, and WordNet API JWNL is used to access WordNet's dictionaries. The experiments generate huge amount of result data, therefore, we employ Oracle database to manage the data. In addition, after creating proper indices, we benefit from Oracle database for quick searching and retrieval operations. There are two types of user interfaces, i.e., command line and web-based. Command line interfaces are used to debug the system and conduct experiments, while the Web-based one is used to show the experimental results in a user-friendly way so that the evaluation work is easier.

6. Discussions and Conclusion

This paper presents our XML Schema matching system to support schema reuse. There are already hundreds of thousands of XML Schemas on the web, which need to be collected, classified, indexed, and searched upon. We are developing an XML Schema repository, and provide various search mechanisms ranging from simple keyword search to the sophisticated tree matchings as described in this paper.

To achieve this goal, one salient feature of our system is our exhaustive approach to each step in the matching process, coping with the engineering details in real application scenario, with the ultimate goal for practical applicaiton. For example, we considered the details of modelling an XML Schema as a tree, and the practical issues in using WordNet to compute the name similarity. Most existing schema matching systems are prototypes that omitted those details.

We also implemented a classical tree matching algorithm for labelled unordered rooted trees. Compared with the edit distance tree matching algorithm, our new tree matching algorithm is more applicable in schema matching problems, because it is designed for matching the trees modelled from schemas and it uses heuristics to reduce the searching space.

Compared with COMA, the performance of our matching system is also satisfying. COMA maintains various types of matching methods including a user-assistant reuse mechanism, and can flexibly combine them to generate the result. Based on the same five purchase order schemas, our experimental results show that in term of overall measurement, our matching system exceeds the best matcher combination without manual reuse, but falls behind the best matcher combination that includes manual reuse. Under the condition of no human interference, our matching system works better than COMA in matching the five purchase order schemas.

The experimental results also show that our new tree matching algorithm can match large trees with hundreds of nodes effectively and efficiently. In a matching task, most executing time is spent on computing node similarities, especially the connection time with wordNet. We are improving this by precalculating and storing the word relationships.

We are also applying schema matching system in web service searching, since the major components in web services are XML Schemas which defines the parameters in the operations of a web service.

Acknowledgments

The work is supported by NSERC (Natural Sciences and Engineering Research Council of Canada), CITO (Communications and Information Technology Ontario), NSERC CRD and NCE (Network Centers of Excellence) Auto 21.

References

1. S. Banerjee and T. Pedersen, Extended gloss overlaps as a measure of semantic relatedness, *IJCAI 2003*.
2. A. Budanitsky and G. Hirst, Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures, in *Proc. NAACL 2001 Workshop on WordNet and Other Lexical Resources*, Pittsburgh, June 2001.
3. H. Bunke, On a relation between graph edit distance and maximum common subgraph, *Pattern Recognition Lett.* **18**(8) (1997) 689–694.
4. H. Bunke, Recent developments in graph matching, in *Proc. 15th Int. Conf. on Pattern Recognition*, Barcelona, 2000, Vol. 2, pp. 117–124.
5. P. V. Biron and A. Malhotra (eds.), W3C, April 2000, 'XML Schema Part 2: Datatypes', http://www.w3.org/TR/xmlschema-2.
6. A. Budanitsky and G. Hirst, Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures, in *Proc. NAACL 2001 Workshop on WordNet and Other Lexical Resources*, Pittsburgh, June 2001.
7. H. Do and E. Rahm, COMA: A system for flexible combination of schema matching approaches, *VLDB 2002*.
8. H. Do, S. Melnik, and E. Rahm, Comparison of schema matching evaluations, in *Proc. GI-Workshop on Web and Databases*, Erfurt, October 2002.
9. A. Doan, P. Domingos, and A. Halevy, Reconciling schemas of disparate data sources: A machine-learning approach, in *Proc. SIGMOD Conference*, 2001.
10. X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, Similarity search for web services, *30th VLDB Conference*, August–September 2004.
11. A. Gupta and N. Nishimura, Finding largest subtrees and smallest supertrees, *Algorithmica* **21** (1998) 183–210.
12. HITIS — Hospitality Industry Technology Integration Standard, http://www.hitis.org.
13. A. Hlaoui and S. Wang, A Node-Mapping-Based Algorithm for Graph Matching, submitted (and revised), *J. Discrete Algorithms*.
14. M. Jarmasz and S. Szpakowicz, Roget's Thesaurus and Semantic Similarity, *RANLP 2003*.
15. H. W. Kuhn, The Hungarian method for the assignment problem, *Naval Research Logistics Quarterly* **2** (1955) 83–97.
16. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang, XClust: Clustering XML schemas for effective integration, in *Proc. 11th Int. Conf. on Information and Knowledge Management*, 2002, pp. 292–299.
17. J. Lu, S. Wang, and J. Wang, An experiment on the matching and reuse of XML schemas, *5th Int. Conf. on Web Engineering (ICWE 2005)*, July 2005, Sydney, pp. 273–284.
18. J. Madhavan, P. A. Bernstein, and E. Rahm, Generic schema matching with Cupid, *VLDB* 2001.
19. S. Melnik, H. Garcia-Molina, and E. Rahm, Similarity flooding: A versatile graph matching algorithm and its application to schema matching, *ICDE 2002*.
20. A. Mili, R. Mili, and R. T. Mittermeir, A survey of software reuse libraries, *Annals of Software Engineering*, 1998.
21. P. Mitra, G. Wiederhold, and M. Kersten, A graph-oriented model for articulation of ontology interdependencies, *EDBT 2000*, LNCS, Vol. 1777, Springer Verlag, 2000, pp. 86–100.
22. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, Semantic matching of web services capabilities, *1st Int. Semantic Web Conf.*, 2002, pp. 333–347.

23. E. Rahm and P. A. Bernstein, A survey of approaches to automatic schema matching, *VLDB J.* **10**(4) (2001) 334–350.
24. T. Schilieder and F. Naumann, Approximate tree embedding for querying XML data, *ACM SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, July 28, 2000.
25. D. Shasha, J. Wang, K. Zhang, and F. Y. Shih, Exact and approximate algorithms for unordered tree matching, *IEEE Trans. on Systems, Man, and Cybernetics*, **24**(4) (1994).
26. D. Shasha, J. T. L. Wang, and R. Giugno, Algorithmics and applications of tree and graph searching, in *Proc. PODS'02*, June 3–5, 2002.
27. H. Su, S. Padmanabhan, and M. Lo, Identification of syntactically similar DTD elements for schema matching, *WAIM*, 2001.
28. Sun, Java Architecture for XML Binding (JAXB), www.sun.com, 2002.
29. K. Sycara, J. Lu, and M. Klusch, Interoperability among Heterogeneous Software Agents on the Internet, Technical Report CMU-RI-TR-98-22, CMU, Pittsburgh.
30. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (eds.), W3C, April 2000, XML Schema Part 1: Structures, http://www.w3.org/TR/xmlschema-1/.
31. J. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and K. M. Currey, An algorithm for finding the largest approximately common substructures of two trees, *IEEE Trans. PAMI* **20** (1998) 889–895.
32. S. Wang, J. Lu, and J. Wang, Approximate common structure in XML schema matching, *WAIM 2005*, pp. 900–905.
33. WordNet — a lexical database for English. http://www.cogsci.princeton.edu/~wn/.
34. J. T. Yao and M. Zhang, A fast tree pattern matching algorithm for XML query, in *Proc. IEEE/WIC/ACM Int. Conf. on Web Intelligence*, Beijing, China, September 20–24, 2004, pp. 235–241.
35. K. Zhang and D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, *SIAM J. Computing* **18**(6) (1989) 1245–1263.
36. K. Zhang, D. Shasha, and J. T. L. Wang, Approximate tree matching in the presence of variable length don't cares, *J. Algorithms* **16**(1) (1994) 33–66.