Exploitation of Similarity and Pattern Matching in XML Technologies
(Technical Report)

Irena Mlynkova and Jaroslav Pokorny

Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske nam. 25, 118 00 Prague 1, Czech Republic
{irena.mlynkova,jaroslav.pokorny}@mff.cuni.cz

Abstract. As XML technologies have undoubtedly become a standard for data representation, it is inevitable to provide efficient implementations of W3C recommendations. A possible optimization of particular types of techniques can be found in exploitation of similarity of XML data and/or matching of XML patterns. In this paper we provide an overview and classification of such techniques from various points of view. We also briefly describe the best known representatives of particular ideas and we discuss their key advantages and disadvantages. The text should serve as a good starting point for proposing an appropriate similarity-based optimization.

1 Introduction

Without any doubt the eXtensible Markup Language (XML) [8] has become a standard for data representation and manipulation. Its popularity is probably given by the fact that the recommendations are well-defined, easy-to-learn, and at the same time still enough powerful. The popularity naturally invoked a boom of efficient implementations of the W3C proposals as well as possible improvements of the existing ones. A possible optimization of particular methods can be found in exploitation of similarity of XML data and mutual matching of XML patterns. In general it enables to treat similar data in a similar way or to extend appropriate approaches known for particular data to the whole set of similar ones.

Under a closer investigation we can see, that the variety of exploitation of similarity in XML technologies is surprisingly wide. It can be found in various XML technologies, such as, e.g., document validation, query processing, data transformation, storage strategies based on clustering, data integration systems, dissemination-based applications, etc. We can search for similarity among XML documents, XML schemes, or between the two groups. Furthermore, we can distinguish several levels of similarity – a structural level (i.e. considering the structure of the given XML fragments), a semantic level (i.e. taking into account the meaning of element / attribute names), a constraint level (i.e. taking into account text value constraints), or their various combinations.

As the number of existing works and approaches is enormous, in a situation when a certain similarity evaluation is needed it is hard to say whether there exists an appropriate solution, or a new one should be proposed. Thus in this paper we provide an overview and classification of these techniques from various points of view. We briefly describe the best known representatives of particular ideas and we discuss their key advantages and disadvantages. The text should serve as a good starting point for proposing an appropriate similarity-based optimization, no matter if based on an existing approach, its slight modification, or a brand new one.

The paper is structured as follows: Section 2 contains a brief introduction to the area of similarity evaluation in connection with XML technologies. Section 3 states several definitions used throughout the paper. Section 4 describes and analyzes the existing proposals of similarity metrics for XML documents, Section 5 does the same for XML documents and their schemes, and Section 6 describes the techniques for XML schemes. Section 7, finally, provides brief conclusions.

2 Similarity, Pattern Matching, and XML

The idea of similarity evaluation and/or pattern matching (i.e. searching for patterns similar to the given ones) in connection with XML can be found in various areas. Hence neither its classification is not and easy task since there are various points of view of the problem and thus various ways of classification and categorization.

Probably the most popular exploitation of pattern matching can be found in query evaluation, document validation, and document transformation. In the first case the given query is modeled as a labeled tree representing the structural and content constraints a document or its part should follow to be considered as the answer to a query. The second approach has a similar aim but in this case the schema is regarded as a template a valid document should correspond to, i.e. no other but the specified elements are allowed. The third approach can be placed between the two previous ones, since we search for the given pattern regardless the other possibly existing elements but, on the other hand, the set of transformation patterns must (usually) cover the whole document. In all the three cases the conformance to the given patterns has to be 100%, though in case of query evaluation we can distinguish so-called *approximate query evaluation*, which eases the requirement and searches for results similar to the given pattern. But in this paper we focus on slightly different ways of similarity and/or pattern matching evaluation, as described in the following text.

An obvious area of exploitation of data similarity (not only for the XML case) are storage strategies based the idea of clustering XML documents or XML schemes. It enables to store structurally similar data in a similar way or "close" to each other to enable fast retrieval and reduces processing of the whole set of stored data to the relevant ones.

Another large area covers so-called *dissemination-based applications* (e.g. [1], [51]), i.e. applications which timely distribute data from the underlying data

sources to a set of customers according to user-defined profiles. These systems also use the approximate query evaluation, since the user expects that the data conform to the profile up to particular similarity threshold.

Last but not least wide area of similarity-based techniques are so-called *data integration systems*, or when concerning directly XML schemes *schema integration systems*. They enable to provide a user with a uniform view of the data coming from different sources and thus having different structure, identifiers, etc. Hence such system must be able to analyze the source data and find corresponding similarities which are expected to appear often.

There are also other possible areas of similarity exploitation [36], such as, e.g., data warehousing [34] (which needs to transform the data from source format to the warehouse format) or e-commerce (where message translation is necessary). But these are a bit too far from the scope of this paper.

2.1 Classification of Similarity-Evaluation Approaches

There are various points of view for classification similarity evaluation approaches. Apart from classification according to the above-described purpose of the similarity evaluation the key classifications are the type of the source data, the precision of the similarity, and the depth of similarity.

From the point of view of the source data the evaluation can be done on several levels:

- at data level, i.e. among XML documents,
- at data type level, i.e. among DTDs [8] or XML Schema definitions (XSDs) [44] [7], or
- between the two levels, i.e. between XML data and schemes.

The first two sets are obvious and depend on the type of the data a particular system focuses on. The last mentioned level is, at least at first glance, a bit surprising since it compares two different types of data. But on the other hand there are cases when such approach can be very useful, e.g. the situation when the system is given both schema-conforming and schema-less data.

The second classification can be done on the basis of the required precision of the similarity. The measure of similarity is usually expressed as a real value from $[0, 1]$, where 0 represents strong dissimilarity and 1 strong similarity. A threshold $T_{sim} \in [0, 1]$ expresses the required precision. Thus we can classify the techniques according to the threshold, though it seems to be advisable to distinguish $T_{sim} = 1$ and $T_{sim} < 1$, i.e. precise and approximate similarity.

Conversely, the depth of similarity evaluation expresses the amount of information that are taken into account during the search process. The best-known levels are:

- a structural level,
- a tag name level,
- a constraint level, or

– their combinations.

In the first case we consider only the structure of the given XML fragments, regardless the element and attribute names or text values. This way it is possible to cluster data from different areas, but having the same or similar structure. We can also further distinguish various sublevels of structural similarity given by, e.g., disregarding element order, complexity of mixed contents, etc. Conversely, the second approach takes into account the tag names and in combination with the first one it can be exploited in the area of processing simple queries that do not contain value constraints. In this case we can also distinguish several sublevels, e.g. if we do not search for equal element/atribute names but we exploit a kind of thesaurus, i.e. similar semantics of element names. The last, and in combination with the first two, most strict approach takes into account also various value constraints and thus it is appropriate for complex query processing and data validation.

3 Definitions and Formalism

Before we begin to describe and classify the methods, we state several basic terms used in the rest of the text.

An *XML document* is usually viewed as a directed labelled tree with several types of nodes whose edges represent relationships among them.

Definition 1. An XML document *is a directed labelled tree* $T = (V, E, \Sigma_E, \Sigma_A, \Gamma, lab, r)$*, where* V *is a finite set of nodes,* $E \subseteq V \times V$ *is a set of edges,* Σ_E *is a finite set of element names,* Σ_A *is a finite set of attribute names,* Γ *is a finite set of text values,* $lab : V \rightarrow \Sigma_E \cup \Sigma_A \cup \Gamma$ *is a surjective function which assigns a label to each* $v \in V$*, whereas* v *is an element if* $lab(v) \in \Sigma_E$*, an attribute if* $lab(v) \in \Sigma_A$*, or a text value if* $lab(v) \in \Gamma$*, and* r *is the root node of the tree.*

A *schema* of an XML document is usually described using DTD or XML Schema[1]. Both the languages use a similar approach and describe the allowed structure of an element using its *content model*.

Definition 2. A content model α *over a set of element names* Σ'_E *is a regular expression defined as* $\alpha = \epsilon \mid pcdata \mid f \mid (\alpha_1, \alpha_2, ..., \alpha_n) \mid (\alpha_1|\alpha_2|...|\alpha_n) \mid \beta* \mid \beta+ \mid \beta?$*, where* ϵ *denotes the empty content model, pcdata denotes the text content,* $f \in \Sigma'_E$*, "," and "|" stand for concatenation and union (of content models* α_1*,* α_2*, ...,* α_n*), and "*", "+", and "?" stand for zero or more, one or more, and optional occurrence(s) (of content model* β*), respectively.*

Definition 3. An XML schema S *is a four-tuple* $(\Sigma'_E, \Sigma'_A, \Delta, s)$*, where* Σ'_E *is a finite set of element names,* Σ'_A *is a finite set of attribute names,* Δ *is a finite set of declarations of the form* $e \rightarrow \alpha$ *or* $e \rightarrow \beta$*, where* $e \in \Sigma'_E$*,* α *is a content model over* Σ'_E*, and* $\beta \subseteq \Sigma'_A$*, and* $s \in \Sigma'_E$ *is a start symbol.*

[1] We state the definitions for DTD only for the paper length.

To simplify processing an XML schema is often transformed into a graph representation.

Definition 4. A schema graph *of a schema* $S = (\Sigma'_E, \Sigma'_A, \Delta, s)$ *is a directed, labelled graph* $G = (V, E, lab')$*, where* V *is a finite set of nodes,* $E \subseteq V \times V$ *is a set of edges,* $lab' : V \rightarrow \Sigma'_E \cup \Sigma'_A \cup \{$ "|", "*", "+", "?", ","$\} \cup \{pcdata\}$ *is a surjective function which assigns a label to* $\forall\ v \in V$*, and* s *is the root node of the graph.*

4 Similarity among XML Documents

Currently, there is a huge amount of works focusing on measuring similarity among XML documents, which is probably caused by the fact that XML documents can be viewed as directed labeled (un)ordered trees.

We can distinguish two main approaches. On the one hand, there are techniques which express the similarity of two documents D_1 and D_2 measuring "how difficult" is to transform D_1 into D_2. On the other hand, there are techniques which define a reasonable representation of D_1 and D_2 that enables their efficient comparison and similarity evaluation.

The core ideas of all the existing approaches also strongly vary from the natural ones, such as measuring tree edit distance or the size of the set of paths to obscure ones such as comparing signals of XML documents.

4.1 Tree Edit Distance

A quite natural idea for measuring the difficulty of transforming document D_1 into D_2 is to represent the documents as trees T_1 and T_2 and measure so-called *tree edit distance*. The idea results from methods for measuring similarity of strings and computes the amount of edit operations necessary for transforming tree T_1 into T_2.

Approach I. One of the first approaches for measuring similarity of XML documents on the basis of tree edit distance can be found in [33]. There are also older approaches measuring a general labeled tree edit distance (e.g. [43], [38], [50], [10], etc.), but they consider only simple edit operations on a single node. But as two documents created from the same DTD can have radically different structure (due to repeatable, optional, and alternative elements), they compute undesirably high distance. On this account authors of paper [33] allow more complex edit operations, each having its particular non-negative cost, as follows:

- *Insert* – a single node n is inserted to the position given by parent node p and ordinal number expressing its position among subelements of p
- *Delete* – a leaf node n is deleted
- *Relabel* – a node n is relabeled

- *InsertTree* – a whole subtree T is inserted to the position given by parent node p and ordinal number expressing position of its root node among subelements of p
- *DeleteTree* – a whole subtree rooted at node n is deleted

As it is obvious, for given trees T_1 and T_2 there are usually several possible transformation sequences for transforming T_1 into T_2. A natural approach is to evaluate all the possibilities and to choose the one with the lowest cost. But such approach can be quite inefficient. Thus the authors define so-called *allowable sequences* of edit operations, which significantly reduce the set of possibilities and, at the same time, speed up their cost evaluation.

Definition 5. *A sequence of edit operations is* allowable *if it satisfies the following two conditions:*

1. *A tree T may be inserted only if T already occurs in the source tree T_1. A tree T may be deleted only if it occurs in the destination tree T_2.*
2. *A tree that has been inserted via the InsertTree operation may not subsequently have additional nodes inserted. A tree that has been deleted via the DeleteTree operation may not previously have had children nodes deleted.*

The first restriction forbids undesirable operations like, e.g., deleting whole T_1 and inserting whole T_2, etc., whereas the second one enables to efficiently compute the costs of the operations.

The evaluating algorithm is based on the idea of determining the minimum cost of each required insert of every subtree of T_2 and delete of every subtree of T_1. They can easily be evaluated using a bottom-up procedure. For example in case of inserts, for each node $n \in T_2$ the cost of inserting whole T_n and the sum of the cost of inserting node n and costs of inserting its subtrees $T'_1, T'_2, ..., T'_k$ are first evaluated. If the tree T_n rooted at n does not occur in T_1 (i.e. we cannot insert whole T_n), the sum is the demanded result, otherwise minimum of the two values is the demanded result. With this preprocessing the resulting edit distance is determined using classical dynamic programming.

Approach II. A slightly different algorithm based on tree edit distance, called *MH-DIFF*, can be found in [11] (which extends ideas of [12]). The main difference is in the set of edit operations:

- *Insert* – a new node n is inserted to the position given by its parent node p, whereas child nodes of p become child nodes of n
- *Delete* – a node n is deleted, while its child nodes become child nodes of its parent node p
- *Update* – a node n is relabeled
- *Move* – a tree T rooted at n is moved to the given position
- *Copy* – a copy of tree T rooted at n is inserted to the given position
- *Glue* – a tree T isomorphic to a tree T' is deleted

As it is obvious, the operations differ from the previous case especially in the Insert and Delete operation. In this case a node can be inserted as an internal node of the tree, whereas in the previous case the inserted node always becomes a leaf. A similar situation can be detected in case of deleting a node. Considering operations on whole trees we can see that the main difference is in the fact that a tree T can be deleted only if there exists a tree T' isomorphic to T. In other words Glue is an inverse operation to Copy.

The aim of the algorithm is to find a minimal transformation sequence (in paper [11] called *edit script*), i.e. the sequence of edit operations, where the sum of their costs is minimal. The result of the algorithm is not only the degree of difference of the two documents, but also the corresponding sequence of edit operations.

The algorithm can be summed up as follows:

1. A complete bipartite labeled graph G_1 consisting of nodes of T_1 on one side and nodes of T_2 on the other side is created, whereas the edges are labeled with all INS, DEL, UPD, MOV, CPY, or GLU labels expressing the corresponding operations.
2. The lower and upper costs are evaluated for each edge $e \in G_1$. They can be computed without the knowledge of which other edges will be in the result.
3. Using a set of *conservative pruning rules* the unnecessary edges are removed from G_1, resulting in G_2. The pruned edges are those which certainly cannot be part of the minimum cost edit script, e.g. an edge e whose cost is higher than deleting and inserting the corresponding nodes.
4. Using the lower edge costs a standard technique for bipartite weighted matching problem searching the minimum edge cover is applied.
5. According to the the minimum edge cover the edit script is generated.

Furthermore, the authors mention that the problem of change detection is in general case NP-hard and thus the proposed algorithm is a heuristic one.

Tree Alignment Paper [21] proposes an algorithm which should serve as a more efficient version of the tree edit distance algorithms, though in this case determined for general ordered labeled trees. It is based on the idea of constructing an *alignment* A of the given trees T_1 and T_2, which is obtained by first inserting nodes labeled with λ into both T_1 and T_2 such that the two resulting trees T_1' and T_2' have the same structure (ignoring the node labels) and then "overlaying" T_1' on T_2'. A λ-node l can be inserted similarly to the previous case, i.e. to the position given by parent node p, whereas child nodes of p become child nodes of l.

The algorithm can be summed up as follows:

1. An alignment A is constructed for trees T_1 and T_2.
2. Each pair of the "overlaying" labels is assigned a score.
3. The distance of the alignment, i.e. the sum of the scores of its pairs of labels, is computed.

Naturally, there exist several alignments for the two given trees depending on the several possible positions λ-nodes can be inserted to. The algorithm searches for the optimal one, i.e. the one with minimum alignment distance. It recursively traverses the nodes of the tree, analyzes all the possible subalignments at each level, and chooses the least expensive one. The difference between ordered and unordered case is that in the unordered case all possible orderings are analyzed, i.e the number of possibilities is much higher.

Furthermore, the authors show that the minimal alignment distance between two unordered trees can be computed in polynomial time if the trees have bounded degrees[2], otherwise becomes NP-hard.

There are numerous other methods exploiting the idea of tree edit distance or tree alignment (e.g. [48], [13], [46], [47], etc.). They usually differ in the set of edit operations, the way their costs are evaluated, and/or the purpose the algorithm is slightly modified to. Similarly, there is even a wider range of methods which consider general rooted ordered trees. We do not describe them here for the paper length and we rather focus on similarity-evaluation strategies based on different ideas. For more complex description and comparison of the tree-edit or tree-alignment approaches see, e.g., [5] or [6].

4.2 Path Sets and Root Paths

A recent, surprisingly simple and at the same time efficient technique for similarity evaluation between XML documents can be found in [35]. The authors first define an abstraction of an XML document called a *structure tree*, which is quite similar to well-known *dataguides* [19].

Definition 6. A structure tree *of an XML document D is a tree T such that for every path in D there is a corresponding path in T and vice versa and T is minimal, meaning that no edges in T can be removed while still preserving the same relationship.*

The approach is based on the idea to compare the respective structure trees of two XML documents, in particular to compare the set of corresponding root paths that can be, undoubtedly, gathered more efficiently than the tree edit sequences.

Definition 7. A root path *of an XML document is a path $e_1e_2...e_n$ in its respective structure tree, where e_1 is the root element and e_n is a leaf element.*

Furthermore, to consider not only identical but also similar paths in XML documents also the respective sets of all subpaths (i.e. consecutive subsequences of elements) of root paths are compared.

[2] A degree of a tree is the maximum degree of nodes in the tree. A degree of a node is the number of its child nodes.

Definition 8. A path set *of an XML document is the union of all of its root paths and all subpaths of the root paths.*

And finally, to take into account also paths which appear more frequently, a frequency of each path is computed.

Having the path sets of two documents together with the corresponding path frequencies, the problem of similarity evaluation is transformed to the problem of finding the intersection of the path sets and measuring its size. Note that, the approach omits order in which elements occur in the documents as well as particular data values.

4.3 Set, Linear, Cost, and Machine-Learning Metrics

Paper [52] discusses advantages and disadvantages of several metrics for evaluating similarity of the given documents D_1 and D_2, i.e. trees T_1 and T_2. It considers the following approaches:

- *Set metrics* based on comparison of edge or path sets of XML documents:
  - $sim_e(T_1, T_2) = \frac{|e^{T_1} \cap e^{T_2}|}{|e^{T_1} \cup e^{T_2}|}$, where e^{T_i} is a set of edges of tree T_i
  - $sim_p(T_1, T_2) = \frac{|p^{T_1} \cap p^{T_2}|}{|p^{T_1} \cup p^{T_2}|}$, where p^{T_i} is a set of root paths of document T_i
- *Linear metric* based on Euclidean distance of paths of trees T_1 and T_2:

  $sim_{lin}(T_1, T_2) = 1 - \sqrt{\sum_{i=1}^{m} |Q_{T_1}(p_i) - Q_{T_2}(p_i)|^2}$

  where $Q_T(p_i) = t_T(e_1)/\beta + t_T(e_2)/\beta^2 + ... + t_T(e_n)/\beta^n$ maps a path $p_i = e_1 e_2 ... e_n$ to a rational number, where $t_T(e_k)$ returns an established order value of name of element e_k, $\beta = 2\alpha + 1$, and α is the number of element names in T
- *Cost metric* based on tree edit distance given by a set of operations:

  $sim_{cost}(T_1, T_2) = \sum_{i,j} cost(op_i(e_j))$

  where operator op_i is applied on element e_j. Its cost can be further influenced by:
  - the type of the operator op_i,
  - the level of element e_j,
  - the number of subelements of element e_j, or
  - the semantics of element e_j.

  and thus the similarity can be rewritten into:

  $sim_{cost}(T_1, T_2) = \frac{1}{\Delta}(\sum_i w_{op_i} cost(op_i) + \sum_j w_{e_j} cost(e_j))$

  where w_{op_i} and w_{e_j} are weights of operators and elements, respectively, and Δ is a normalization factor.

– *Machine-learning metric* exploiting XML querying using a set of training instances $I_1, I_2, ..., I_n$, i.e. 3-tuples each consisting of a query Q_k, its approximate answer A_k, and $L_k = +1$ if A_k is relevant to Q_i or -1 otherwise. The algorithm in turns adapts weights w_{op_i} and w_{e_j} to particular instances I_k so that the result of the similarity sum approximates the value of L_k.

4.4 Document Signal

A considerably different approach, which is also trying to avoid expensive evaluation of tree edit distance, can be found in [18]. It is based on the idea of representing an XML document as a time series in which each occurrence of a (start or end) tag represents an impulse. In other words, each document is assigned a signal corresponding to occurrences of elements in the depth-first left-to-right traversal order.

The algorithm for transforming an XML document D into a series of impulses $S = [I_1, I_2, ..., I_n]$ can be summed up as follows: The distinct tag names are randomly ordered and each start tag t_i is assigned its position in the sequence $\gamma(t_i)$, whereas end tags are assigned symmetric minus values. Each (start or end) tag occurrence t_i in the document is assigned an impulse I_i as follows:

$$I_i = \gamma(t_i) * (N-1)^{D_{depth} - l_{t_i}} + \sum_{t_j \in anc(t_i)} \gamma(t_j) * (N-1)^{D_{depth} - l_{t_j}} \qquad (1)$$

where N is the number of distinct tag names, D_{depth} is the depth of the document D, l_{t_i} is the level of tag occurrence t_i in document D, and $anc(t_i)$ is the set of ancestors of tag occurrence t_i.

As it is obvious, the impulse of an element represents its position in the document giving higher impulse values to elements appearing at higher levels.

So the problem of similarity of documents D_1 and D_2 is transformed into the problem of similarity of their signals $S_1 = [I_1^1, I_2^1, ..., I_n^1]$ and $S_2 = [I_1^2, I_2^2, ..., I_m^2]$. As the signals can have different lengths (depending on the length of the documents) and intensity (depending on the chosen encoding schemes), the signals are periodically extended, the *Discrete Fourier Transform (DFT)* is applied on them, and the result is linearly interpolated having $M = N_{D_1} + N_{D_2} - 1$ points, where N_{D_i} is the number of (start and end) tags in document D_i. This way we get new signals $S_1' = [J_1^1, J_2^1, ..., J_M^1]$ and $S_2' = [J_1^2, J_2^2, ..., J_M^2]$ and the resulting distance of XML documents D_1 and D_2 is defined as follows:

$$dist(D_1, D_2) = \sqrt{\sum_{k=1}^{M/2} (|J_k^1| - |J_k^2|)} \qquad (2)$$

In other words, the distance of the documents is the approximation of the difference of magnitudes of the two signals S_1' and S_2'.

5 Similarity among XML Documents and XML Schemes

Another set of similarity-evaluation algorithms focuses on measuring similarity between an XML document D and an XML schema S. The main exploitation is again in the area of clustering XML data, where these techniques enable to cluster together schema-less and schema-conforming XML documents having a similar structure. Surprisingly, the set of existing approaches is relatively small, especially in comparison with the previous one.

In this case we cannot transform the problem to measuring similarity of two ordered labeled trees, since, on the one hand, we have a tree, but we have to match it with a schema, i.e. a set of regular expressions. Thus the problem is much complicated.

A natural idea could be to exploit techniques for extracting a schema from a sample set of XML documents (e.g. [30], [31]) and thus to transform the problem of similarity of an XML document and a schema to the problem of similarity of two schemes. But, to our knowledge, there is no such approach yet. The main question is whether the automatically generated schema would not be too artificial and, furthermore, for which applications would be such approach useful as long as it requires a nontrivial set of "training" XML documents.

Among the small set of existing works we can distinguish two types of strategies – techniques which measure the number of elements which appear in D but not in S and vice versa and techniques which measure the closest distance between D and all documents valid against S.

5.1 Common, Plus, and Minus Elements

Papers [3] and [4] propose an algorithm for evaluating structural similarity between XML documents and DTDs. It is based on the fact that while matching a document against a DTD, some attributes and subelements of an element in the DTD can be missing from the corresponding element in the document, or the document can contain some additional attributes and/or subelements. On this account the authors define three types of elements:[3]

- *common* elements, which appear both in the document and the DTD,
- *plus* elements, which appear only in the document, and
- *minus* elements, which appear only in the DTD.

Naturally, the lower number of plus and minus elements and higher number of common elements is, the higher similarity level the corresponding XML document and DTD has.

The proposed algorithm exploits the idea that elements at higher levels as well as elements which are more complex have greater impact on the evaluation of similarity. While the depth of an XML document is the number of nodes along its longest path, in case of a DTD only nodes not labeled by an operator

[3] For the sake of simplicity the approach does not consider attributes, empty elements, sequences of operators, and cycles.

(i.e. element nodes) are considered. The complexity of an element depends on its content, whereas in case of minus elements only the simplest mandatory structure that can be derived from the given DTD fragment is considered. For this purpose, the weight of a given (document or DTD) (sub)tree $T = (r, C)$ (where r is the root node of T and $C = \{T_1, ..., T_n\}$ is set of its subtrees) and its level l_T is defined as follows:

$$weight(T, l_T) = \begin{cases} l_T & C = \emptyset \\ 0 & r \in \{*, ?\} \\ weight(T_1, l_T) & r = \ + \\ \sum_{i=1}^{n} weight(T_i, l_T) & r = \ , \\ \min_{i=1}^{n} weight(T_i, l_T) & r = \ | \\ \sum_{i=1}^{n} weight(T_i, \frac{l_T}{\gamma}) + l_T & otherwise \end{cases} \quad (3)$$

where γ is *level relevance factor* which can decrease the weight at each level (e.g. $\gamma = 2^k$ for $k = 0, 1, ...$ at $0^{th}, 1^{st}, 2^{nd}, ...$ level of the subtree).

The algorithm matches elements at each level of the document tree T_{doc} with corresponding DTD tree T_{dtd}. Due to several possible matches (caused by optional, repeatable, and alternative elements) it evaluates all the possibilities (using the *weight* function) a chooses the one, where the number of plus (p) and minus (m) elements is minimal and the number of common (c) elements is maximal. The overall evaluation function f_{eval} is defined as follows:

$$f_{eval}(p, m, c) = \begin{cases} 0 & (p, m, c) = (0, 0, 0) \\ \frac{c}{\alpha * p + c + \beta * m} & otherwise \end{cases} \quad (4)$$

where α and β are relevance factors which influence the impact of plus and minus elements, respectively.

5.2 Edit Distance

Paper [32] proposes an algorithm for measuring similarity of an XML document D and a DTD S on the basis of local similarity of elements[4] defined using their edit distance from the corresponding declaration in the DTD. The local similarities are then weighted using the size of the element subtree and aggregated to the total similarity value.

The edit distance of an element $e \in D$ and corresponding element declaration $f \in S$ is defined as $dist_e = min\{dist(e, e') \mid e' \text{ matches } f\}$, whereas the function $dist(e, e')$ is evaluated using a classical tree edit distance algorithm and the resulting minimum using Thompson's algorithm for automaton construction [45].

As the edit distances for different elements in the same document may not be comparable (due to the varying complexity of corresponding element declarations), they are normalized using the maximum edit distance defined as $distmax_e = max(len(e), minlen(f))$, where $len(e)$ is the length of string containing e and $minlen(f) = min\{len(e') \mid e' \text{ matches } f\}$.

The algorithm can be summed up as follows:

[4] For the sake of clarity the algorithm omits attributes.

1. The weight w_e, i.e. the number of descendants of e, is calculated for every element $e \in D$.
2. The edit distance $dist_e$ and the maximum edit distance $distmax_e$ is calculated for every $e \in D$ for which a corresponding element declaration $f \in S$ exists.
3. The local similarity sim_e is evaluated for each element $e \in D$, whereas $sim_e = 1 - \frac{dist_e}{distmax_e}$ if e has a corresponding declaration $f \in S$, or $sim_e = 0$ otherwise.
4. The total similarity $sim(D, S) = \frac{\sum_{e \in D} w_e * sim_e}{\sum_{e \in D} w_e}$ is evaluated.

As we can see, contrary to the previous case the algorithm preserves element order and it is able to match elements at different levels.

5.3 Regular Hedge Grammar

An idea similar to the previous case can be found in paper [9]. The main differences are that the authors analyze similarity of XML documents and XSDs and instead of regular expressions a grammar, so-called *Regular Hedge Grammar (RHG)*, is used to express the schema constructs. Similarly to the previous case the distance of XML document D_1 and an RHG G is defined as the minimum tree edit distance to a document D_2 which is valid against G. To compute the distance a weighted dependency graph is constructed for the grammar rules and using the Dijkstra's shortest path the required minimum is found.

6 Similarity among XML Schemes

The last considered set of approaches focuses on measuring similarity among XML schemes expressed either in DTD or XML Schema. In this case we consider a problem of similarity comparison of two sets of regular expressions, typically called a *schema matching problem*. In comparison with the previous case the amount of existing techniques is enormous. We again focus on the key representatives of a particular idea, mentioning the similar methods but omitting their detailed description for the paper length.

The main exploitation of these techniques is undoubtedly in schema integration systems, where various subsystems provide a schema of their data (e.g. XML, SGML, relational, object-oriented, etc.) and the aim of the system is to provide a uniform schema over which queries are posed or operations are carried out. A second key exploitation is clustering the data on the basis of similarity of their schemes.

The general idea of schema matching is to exploit various supplemental *matchers* [36], i.e. functions which evaluate similarity of a particular feature of the given schema fragments, such as, e.g., similarity of leaf nodes, similarity of root element names, similarity of context, etc. which are combined into the resulting similarity value. In a significant amount of existing works also the idea

of machine learning is exploited, though it can be used only in cases when a large-enough training set is available.

As the variety of approaches is really wide and based on exploitation of various ideas, techniques, and/or information – such as, e.g., element and attribute names, data types, structure, dictionaries, thesauri, results of previous matchings, etc. – there are also papers, which compare them from various points of view (e.g. [14], [36]) as well as papers which analyze the matching problem from theoretic point of view (e.g. [40]). We will mention their key findings too.

6.1 Schema Integration

TranScm One of the first proposals of a schema-matching algorithm can be found in system called *TranScm* [27]. Its main aim is integration of data from various sources which are clustered according to similarity of their schemes. The given schemes of various types (e.g. SGML, OODB, etc.) are first transformed into *middleware schemes* similar to W3C DTDs or XSDs. Starting from the root nodes for each schema fragment $f_1 \in S_1$ the system searches the best possible matching fragment $f_2 \in S_2$ using a *rule-based method*. If there is no matching fragment in the target schema or there are several possibilities, a user interference is required.

Each rule r in the set of mapping rules R defined over two fragments $f_1 \in S_1$ and $f_2 \in S_2$ rooted at r_1 and r_2, respectively, consists of:

- *Priority* p_r, which enables to order the rules if more than one is possible,
- *Matching part*, which consists of the following two operations:
  1. *Match* – examines whether r_1 and r_2 match the rule and
  2. *Descendants* – examines whether the sets of (un)direct child nodes of r_1 and r_2 match the rule,

  and defines the possible common matching, and
- *Translation part*, which defines how to transform f_1 into f_2.

An example of a rule can be, e.g., matching ordered and unordered tuple-like structures, where the Match function simply compares names of r_1 and r_2 (using a given set of synonyms) and the number of child nodes they have. The Descendants function similarly compares the sets of child nodes. The Translation part simply creates a node representing r_2 and attaches child nodes to it.

The system contains a set of predefined rules, that can be removed, added, overridden, or disabled.

Similarity Flooding Another schema integration approach – so called *similarity flooding (SF)* algorithm – is proposed in paper [26]. Also in this case the input schemes (relational, XML, or any other) are transformed into common internal graph representation which enables to process heterogenous data. The key idea of the matching algorithm is that two elements are similar when their adjacent elements are similar, i.e. the part of the similarity of two elements propagates to their neighborhood. The algorithm for two given schemes S_1 and S_2 works in the following steps:

1. The schemes S_1 and S_2 are translated from their native formats into common graph representations G_1 and G_2.
2. Initial similarity mapping between nodes of G_1 and G_2 is defined using similarity of node names (based on analyzing of common prefixes and suffixes).
3. Using the *similarity flooding algorithm* (see below) the initial similarities are propagated through the graphs until the *fix point* is reached, i.e. the similarity values *stabilize*.
4. The resulting matching candidates are further filtered using a required threshold.

The similarity flooding algorithm is iterative. At each iteration the similarity of each map pair of nodes (n_1, n_2), where $n_1 \in G_1$ and $n_2 \in G_2$, is incremented by similarity values of its neighbors multiplied by predefined *propagation coefficients*. The coefficients are assigned so that each type of edge has an equal contribution of 1.0 which is distributed among all its occurrences. The algorithm terminates either if the similarity increment vector is small enough or after a certain number of iterations.

Cupid Paper [25] proposes a system called *Cupid*, which also evaluates the similarity of two XML schemes. The evaluation consists of the following two phases:

1. *Linguistic phase* of the algorithm matches individual schema elements based on their names, data types, domains, etc., resulting in a *linguistic similarity coefficient* $lsim \in [0, 1]$ for each pair of elements.
2. *Structural phase* of the algorithm evaluates the element similarity on the basis of their context and vicinity resulting in a *structural similarity coefficient* $ssim$ for each pair of elements.

The total element similarity is the *weighted similarity* $wsim = w_{struct} * ssim + (1 - w_{struct}) * lsim$, where $w_{struct} \in [0, 1]$.

The linguistic phase consists of the following three steps:

1. *Normalization* – element names are tokenized (i.e. parsed into tokens based on punctuation, case, etc.), expanded (i.e. abbreviations and acronyms are identified), and eliminated (i.e. prepositions, articles, etc. are discarded)
2. *Categorization* – element names are clustered into categories based on their data types, schema hierarchy, and linguistic content
3. *Comparison* – similarity of elements is computed by comparing the tokens using a thesaurus that has synonymy and hypernymy relationships and a substring matching considering elements that belong to similar categories

The matching algorithm of the structural phase is based on the following three observations:

1. Leaf elements are similar if they are linguistically and data-type similar and if their ancestors and siblings are similar.

2. Two non-leaf elements are similar if they are linguistically similar and the corresponding subtrees are similar.
3. Two non-leaf schema elements are structurally similar if their leaf sets are highly similar, even if their immediate children are not.

For each pair of elements *ssim* is evaluated using a bottom up strategy. The similarity of two inner nodes is expressed as the number of pairs of so-called *strong-link leaf nodes* in the corresponding subtrees, i.e. pairs of leaf nodes whose weighted similarity exceeds a given threshold.

The resulting set of similarity mapping elements selects pairs of elements with highest similarity values. Note that the mapping can be 1:n since several elements can satisfy the condition.

Since the above described algorithm considers schemes being trees whose nodes are elements, it is further extended to handling shared types (by creating a copy of each shared element for each sharer), optionality (by decreasing weights for optional elements), referential constraints (by augmenting the schema using "view nodes"), etc. On the other hand, it does not consider repetition operators at all.

SPL Paper [42] proposes an algorithm (in some literature denoted as *SPL* according to authors name) for evaluation similarity among DTDs. It is again intended for a schema-integration system, proposed as a reasonable starting point for further human correction.

The given DTDs are first preprocesses as follows:

1. The DTDs are simplified leading to a certain loss of information as follows:
   - Non-null constraints are "blurred" using rules $e? \rightarrow e$ and $e^* \rightarrow e^+$,
   - The content models are flattened using rules such as, e.g., $(e_1|e_2) \rightarrow (e_1, e_2)$, $(e_1, e_2)^+ \rightarrow (e_1^+, e_2^+)$, etc., and
   - Sub-elements having the same name in a content model are grouped into a single one with a corresponding occurrence,
2. The simplified DTDs are modelled as graphs, similar to *DTD graphs* [39] with weights assigned to edges expressing the number of occurrences of the particular element relationship.
3. Cyclic graphs are converted into acyclic by creating a copy of each recursive element, so-called *leaf recursive node*, and redirecting all edges to the copy.

The algorithm exploits two key matching criteria:

1. *Non-recursive leaf matching* – two non-recursive leaf nodes match if both are element nodes having the same name, or both are attribute nodes having the same name and type, or the first node is an attribute node, the second one is an element node having CDATA type and both are having the same name
2. *Graph distance* – represents the portion of elements in the two graphs that match and the total number of elements.

The algorithm is proposed either for searching semantically equivalent or semantically similar schema fragments. In both cases it uses a bottom-up strategy. In case of semantic equivalence it takes into account the level of each node and considers two nodes as matching candidates if

1. There is at least one pair of child nodes that match, and
2. The nodes have the same *reduced topology*, i.e. an ordered list of pairs (l, o), where l represents the level and o an out-degree of each node in the subtree.

As it is obvious, at each step there can be various possible matching candidates resulting in a set of *matching plans*, i.e. sets of matching pairs. The algorithm chooses the one having the lowest distance.

In case of semantic similarity, the algorithm is slightly modified. It drops the condition that the two matching elements must have the same reduced topology and must occur at the same level. Thus the set of possible matching candidates is at each step bigger and the authors recommend a greedy strategy which chooses the best local matching plan at each step.

COMA Paper [15] proposes a system called *COMA* based on the idea of combining various matchers with user interaction. The processing is iterative, where each iteration consists of the following phases:

1. An optional user feedback phase, where a user can:
   - specify the required match strategy, i.e. the set of matchers and the strategy of combining the individual match results and
   - accept of reject match candidates proposed in the previous iteration.
2. The execution of all allowed matchers which results in a *similarity cube* consisting of 3-tuples (fragment $f_1 \in S_1$, fragment $f_2 \in S_2$, $sim_j(f_1, f_2) \in [0, 1]$) for j-th matcher, and
3. The combination of the individual match results in the cube, i.e.
   (a) Aggregation of the results of particular matchers using, e.g., the maximum / minimum value, the average value, or the weighted sum
   (b) Selection of match candidates according to, e.g., the highest similarity value, the similarity value exceeding a given threshold, or differing at most by a given tolerance, etc.
   (c) Aggregation of the similarities of elements into the total similarity using, e.g., the average value or the ratio of number of elements which can be matched and the total number of elements

The set of matchers consists of the following types:

- *Simple matchers* – analyze the similarity of element name strings or the similarity of their meaning:
  - *Affix* – analyzes the sets of affixes, i.e. prefixes and suffixes
  - *n-gram* – analyzes the sets of n-grams, i.e. sequences of n characters
  - *EditDistance* – computes the number of edit operations to transform one string to another one

    - *Soundex* – computes the phonetic similarity between names from their corresponding soundex codes
    - *Synonym* – uses relationship-specific similarity values, e.g. 1.0 for a synonymy, 0.8 for a hypernymy relationship, etc.
    - *DataType* – uses a synonym table specifying the degree of compatibility between a set of predefined generic data types, to which data types of schema elements are mapped in order to determine their similarity
- *Hybrid matchers* – use a fixed combination of simple matchers or other hybrid matchers, combined similarly to the phase 3:
    - *Name* – combines different simple matchers
    - *NamePath* – applies Name matcher to paths of the elements
    - *TypeName* – combines Name and DataType matchers
    - *Children* – combines the similarities of child nodes
    - *Leaves* – combines the similarity of leaf descendant nodes
- *Reuse-oriented matchers* – all schemes S'_i for which there is a previous match result with both S_1 and S_2 are identified and all the corresponding pairs of match results are combined similarly to the phase 3

As we can see, the algorithm has many options (e.g. the ways of combining the partial results) and situations (e.g. the choice of match candidates), where the user interaction can be helpful. Otherwise a default strategy is selected.

CMC Paper [49] proposes a system called *CMC* which focuses on the fact that the way of combination of the base matchers, in particular their weights, highly influence the resulting similarity and performs differently on different schema fragments. For instance, structure matchers are obviously more credible if the schema fragments have rich structure. Thus the key concern of the approach is estimation of credibility of particular matchers for each pair of schema fragments being matched. These credibilities are then used as weights of the base matchers when being combined.

The base matcher's credibility prediction is based on the idea that it is correlated with several features of currently given pair of schema fragments (so-called *matching task*). For structural matcher, e.g., the number of edges can serve as such feature, since more edges indicate more structural information leading to higher matching accuracy. The prediction consists of two steps:

1. *Accuracy predicting* – determines the accuracy of the matcher as the mean accuracy of the set of tasks bearing the same features as the current task, i.e. the mean square error (MSE) of the tasks
2. *Converting accuracy to credibility* – converts the value of accuracy (MSE) to credibility according to the expression $e^{-C \cdot MSE}$, where C is a non-negative constant determining how fast the credibility falls with the increase of MSE

For simple matchers the MSE value can be determined manually, for more complex ones machine learning and an appropriate training set is used.

6.2 Clustering

Paper [23] describes a system called *XClust* which exploits the idea of clustering DTDs similar in semantics, content, and context. First, the DTDs are simplified using both *information-preserving rules* (e.g. $(a|b)* \Leftrightarrow (a*, b*)$) having high priority and *information-loss rules* (e.g. $(a|b)? \Rightarrow (a?, b?)$) having low priority. The similarity evaluation then consists of the following matchers:

- *Basic similarity*
  $BasicSim(e_1, e_2) = w_1 * OntSim(e_1, e_2) + w_2 * ConsSim(e_1.card, e_2.card)$
  where
  - $OntSim$ is an *ontology similarity* of element names based on a kind of thesaurus,
  - $ConsSim$ is a *cardinality similarity* defined by cardinality compatibility table, e.g. $ConsSim(*, *) = 1$, $ConsSim(*, ?) = 0.7$, $ConsSim(?, \emptyset) = 0.8$, etc., and
  - w_1, w_2, $w_1 + w_2 = 1$ are corresponding weights.
- *Path context coefficient* $PCC(p_1, p_2)$ expressing the similarity of paths p_1 and p_2 applying the following *PCC algorithm* on the sets of elements in the paths:
  1. Basic similarity $BasicSim$ is determined for each pair of elements of the two sets.
  2. The pairs of elements with the highest similarity are iteratively found producing a one-to-one mapping.
  3. The total PCC similarity is obtained by summing up all the similarities from the one-to-one mapping normalized (i.e. divided) by the maximum size of the element sets.

The total element similarity $ElementSim$ of elements e_1 and e_2 is expressed as a weighted sum of the following three similarity values:

- *Semantic similarity*
  $SemSim(e_1, e_2) = PCC(e_1.path_{root}, e_2.path_{root}) * BasicSim(e_1, e_2)$
- *Immediate descendant similarity* expressing the similarity of the sets of immediate descendants of e_1 and e_2 sing the PCC algorithm.
- *Leaf-context similarity* expressing the similarity of the sets of leaf subelement of e_1 and e_2 using the PCC algorithm, where $LeafSim$ instead of $BasicSim$ is considered:
  $LeafSim(l_1, l_2) = PCC(l_1.path_{e_1}, l_2.path_{e_2}) * BasicSim(l_1, l_2)$

Finally, the DTD similarity is evaluated applying the PCC algorithm on the set of elements of the two DTDs, where $ElementSim$ instead of $BasicSim$ is considered.

6.3 Machine Learning

The idea of machine learning can easily be exploited for the schema matching problem, since it enables to train the system on sample known data and than use for an arbitrary input. Nevertheless, the key shortcomings are evident. Firstly, there are cases when such training is not available. And secondly, if a particular type of schema was not present in the training set, its evaluation could be misleading.

LSD Paper [16] proposes a schema-integration system called *LSD (Learning Source Descriptions) system*, which enables to store and query semantically similar but structurally different data from various sources. It provides users with a *mediated schema*, to which schemes of the underlying systems are semantically mapped and over which users pose their queries. The mapping process is semi-automatical and exploits machine-learning techniques. In the *training phase* the systems asks a user to provide similarity mapping between sample schemes. In the *matching phase* the training sets are used to match new source schemes.[5]

The key features of the system were later exploited in system *GLUE* [17], which instead of XML schemes focuses on semantic mapping of ontologies.

MKB Paper [24] further enhances the previous work with the idea of deeper exploitation of experience from previous matching tasks. The key component of the system – so-called *Mapping Knowledge Base (MKB)* – captures the previously gathered knowledge of each element and exploits it in later matches.

Being given two schemes S_1 and S_2 that have to be matched, the system carries out the following steps:

1. Each element $s_i \in S_1$ and $t_j \in S_2$ is compared against all elements $e_k \in$ MKB by applying partial matchers (see below) and combining their results.
2. The results of the comparisons are formed into a vector $P_i = \langle p_{i1}, p_{i2}, ..., p_{in} \rangle$ for each $s_i \in S_1$ and $Q_j = \langle p_{j1}, p_{j2}, ..., p_{jn} \rangle$ for each $t_j \in S_2$.
3. The similarity of elements $s_i \in S_1$ and $t_j \in S_2$ is given by similarity of vectors P_i and Q_j measured using an *average weighted difference (AWD)*:

   $$AWD(P_i, Q_j) = \sum_{k=1}^{n} |p_{ik} - p_{jk}| * max(p_{ik}, p_{jk})$$

4. The result of the algorithm is a similarity matrix containing the similarity values for all pairs of elements in schemes S_1 and S_2.

The partial matchers for two elements (so-called *base learners*), which are combined (using a *meta-learner*) into the resulting similarity value, involve:

– *Name learner* – analyzes names of elements, in particular corresponding n-grams and their frequency,

[5] We do not describe the details of the machine learning strategy for different aim of the paper.

– *Description learner* – analyzes descriptions often available with the corresponding schemes,
– *Instance learner* – analyzes features of data instances (e.g. number of words, special symbols such as $, %, etc.),
– *Data type learner* – analyzes the data types of the elements, and
– *Structure learner* – analyzes other elements that co-occur with the element in its neighborhood, i.e. parent, child, and sibling elements.

6.4 Matching with Specific Conditions

Besides proposals of general similarity-evaluation algorithms, there are also papers which either modify the existing approaches to a specific situation or directly propose a brand new one strategy suitable for it.

Matching Large Schemes Paper [37], which extends the previously mentioned system COMA [15], focuses on matching very large schemes, in particular XSDs, and their specific features such as:

– type system, i.e. simple types, complex types, and subclassing,
– components reuse and sharing, i.e. locally and globally defined items, and
– distributed schemes, i.e. the usage of namespaces.

It proposes a fragment-oriented matching approach which enables to decompose the problem into smaller ones and, at the same time, to reuse previous match results. The algorithm works in the following steps:

1. *Schema decomposition* – the input schemes are decomposed into fragments, which can be either *subschemes* (i.e. schema parts which can be separately instantiated) or *inner fragments* (i.e. inner elements or complex types and their descendants).
2. *Identifying fragment-pair candidates* – to avoid trying to find correspondences between irrelevant fragments the similarity between fragments is first determined on the basis of easily-collected fragment statistics (e.g. name of fragment root, type of fragment root, size and depth of the fragment, etc.).
3. *Fragment matching* – the candidate pairs from the previous step are fully matched similarly to the COMA system and with the emphasis on reuse of previous fragment matches which are stored in the system too.
4. *Result combination* – the total match result for the two complete schemes is determined using the match correspondences for the inner fragments using a bottom-up propagation strategy similarly to the previous case(s).

Matching with Large Number of Schemes Paper [41] proposes an algorithm for efficient matching of a user-defined schema with a large number of schemes stored in a schema repository in the form of a *schema repository graph*, i.e. a forest of schemes. It is proposed as an enhancing of previously mentioned techniques. The algorithm exploits the idea of clustering which enables

to quickly identify regions, i.e. clusters, of schemes which are likely to produce good matching results. The classical schema matcher is then applied only to the clusters instead of searching through the whole repository.

The key problem is how to define the clusters. The authors of the paper propose an adaptation of the classical *k-means clustering algorithm*. Each cluster consists of elements involved in any previous matching step (so-called *mapping elements*). The algorithm works in the following steps:

1. Initialization of centroids – using a simple heuristic the initial centroids c_i are placed in areas which produce the most matching results
2. Distance computation – for each mapping element e_j and each centroid c_i the distance is computed as the length of corresponding path in the schema repository graph
3. Cluster assignment – each element e_j is assigned to the nearest centroid
4. Determining new centroids – for all clusters new centroids are determined, i.e. elements in the center of weights
5. Reclustering – dynamical change of the number of clusters, i.e. a join is applied if two centroids are near and a removal is performed if a cluster is too small, whereas its elements are joined with neighboring clusters

Matching "Opaque" Column Names and Data Types Paper [22] is trying to solve the situations when the previously described techniques fail. In particular in focuses on situations when:

1. The names of similar schema items (in particular considering database table columns) are not similar or
2. The data types of similar schema items are not similar.

The authors denote such column names or data types as "opaque" and define so-called *uninterpreted matching*, i.e. matching that does not depend on data interpretation. The key idea of the algorithm is to exploit other schema information – the dependency relationships between the data.

The proposed algorithm consists of the following two stages:

1. Construction of dependency graph – pair-wise correlations (see below) are estimated for all pairs of attributes in a table and structured into an undirected graph, where:
   - Each node is labelled with the value of entropy of the corresponding attribute and
   - Each edge is labelled with the value of mutual information of corresponding attributes.
2. Graph matching algorithm – using an appropriate graph matching strategy (depending on the required mapping, i.e. one-to-one, onto, or partial) the produced dependency graphs are matched

The correlation between two attributes is evaluated using mutual information and entropy. Having two attributes X and Y with alphabets $\overline{X}$ and $\overline{Y}$, a joint probability distribution $p(x, y)$, and marginal probability distributions $p(x)$ and $p(y)$ for $\forall x \in \overline{X}$ and $\forall y \in \overline{Y}$, the mutual information (MI) of X and Y is defined as:

$$MI(X,Y) = \sum_{x \in \overline{X}} \sum_{y \in \overline{Y}} p(x,y) \cdot log \frac{p(x,y)}{p(x) \cdot p(y)} \tag{5}$$

It can be rewritten using entropies

$$H(X) = -\sum_{x \in \overline{X}} p(x) \cdot \log p(x) \; ; \; H(Y) = -\sum_{y \in \overline{Y}} p(y) \cdot \log p(y) \tag{6}$$

and conditional entropy

$$H(X|Y) = -\sum_{x \in \overline{X}} \sum_{y \in \overline{Y}} p(x,y) \cdot \log p(x|y) \tag{7}$$

into the form:

$$MI(X,Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{8}$$

As it is obvious MI does not depend on actual values of the attributes but on probability distributions which form the input of the algorithm. The resulting value represents the amount of information captured in one attribute about the other.

Holistic Schema Matching Contrary to the above described approaches, the authors of paper [20] propose an idea of *holistic schema matching*, i.e. an approach which matches several schemes at the same time. Such approach enables to exploit the *context* information among the schemes (i.e. information already known from the other schemes), which cannot be exploited in pair-wise matching approaches. It takes a set of n schemes as an input and outputs a set of all the matchings among the schemes (so-called *semantic model*), i.e. not pairs, but n-tuples of matching objects. The authors propose two approaches – global and local evaluation.

Global evaluation exhaustively evaluates all possible models and selects the best one. It is based on the idea that each schema is viewed as an instance generated from a "hidden" model the approach is trying to find. The system takes as input a general structure of such models which captures specific target questions. For instance, if finding synonyms is the target, a model should explicitly express the grouping of synonyms. Then it generates all models that instantiate all the given schemes and chooses the best one.

On the contrary, *local evaluation* assesses every single matching and then incrementally constructs a model as a set of all matchings. It is based on an observation that *grouping attributes* (e.g. `fist_name` and `last_name` are co-present

often), while *synonym attributes* are not. Thus the system first builds both the groups, combines the results and finally, using a greedy strategy, iteratively chooses the (sub)optimal matching.

6.5 Theoretic Studies and Comparisons

As the amount of similarity-evaluation strategies is really high, there are of course papers which analyze the problem in general. They either focus on theoretical analysis or analyze a subset of the existing works together.

A Theoretic Study Paper [40] studies the problem of semantic XML schema matching, i.e. identifying semantically similar components within two schemes, from the theoretic point of view. It specifies the problem formally discovering that it can be described as a *constraint optimization problem (COP)* and thus corresponding existing general algorithms for its solving can be exploited.

The authors describe the given problem as follows: In a *matching problem* a template object T is matched against a set of target objects $R = \{T_1, ..., T_k\}$. If T is related to T_i through some desired relation, i.e., $T \approx T_i$, it is said that they *match* and the (T, T_i) pair forms a *mapping*. The solution of a matching problem is a list of mappings. In some matching problems, an *objective function* $\Delta(T, T_i)$ can be defined. The objective function evaluates to what extent the desired relation between the matching objects is met and is used to order the mappings in the solution.

A *semantic matching problem* is a matching problem where objects are matched on the basis of their semantics. As such evaluation can be reliably done only by humans, the corresponding algorithms compute an *approximate semantic matching*. The desired relation can be divided into a *semantic predicate function* and the *semantic objective function*. The template and target objects are represented by corresponding XML schemes, which are often transformed into directed labelled graphs, so-called *schema graphs*. The predicate function can be expressed as a composition of a number of predicates $C(T, T_i) = \bigwedge_{i=1}^{k} c_i(T, T_i)$ and the objective function is approximated as $\Delta(T, T_i) \in R$, often normalized to $[0, 1]$.

The semantic matching problem can be generally described as a COP defined as follows:

Definition 9. *A* constraint optimization problem (COP) *P is a 4-tuple $P = (X, D, C, \Delta)$ where:*

- *$X = (x_1, x_2, ..., x_n)$ is a list of variables,*
- *$D = (D_1, D_2, ..., D_n)$ is a list of finite domains, such that variable x_i takes values from domain D_i and D is called the* search space *for problem P,*
- *$C = \{c_1, c_2, ..., c_k\}$ is a set of constraints, where $c_i : D \rightarrow \{true, false\}$ are predicates over one or more variables in X, and*
- *$\Delta : D \rightarrow R$ is an objective function assigning a numerical quality value to a solution.*

A complete variable assignment is called *valuation* Θ. A valuation Θ for which constraints $C(X) = true$ is called a *solution*. The quality of a solution is determined by the value of the objective function, i.e., $\Delta(\Theta)$.

Taxonomy Paper [36] analyzes (and briefly describes) several existing works (from the above described LSD [16], TranScm [27], CUPID [25], Similarity Flooding [26] plus few others, bit more off from the focus of this paper). The main contribution of the paper is taxonomy of the existing approaches and their respective classification.

It focuses on exploitation of similarity evaluation of a special kind: It defines the *match operation* as a function that takes two schemes S_1 and S_2 as an input and returns a mapping between the two schemes, called the *match result*. It consists of so-called *mapping elements* each of which indicates that certain elements of schema S_1 are mapped to certain element of S_2. Furthermore, each mapping element can have a *mapping expression* which specifies how the corresponding elements are related.

The proposed classification of schema mapping approaches distinguishes the criteria for the individual matchers and for their combination. As for the individual matchers the criteria are:

1. *Instance vs. schema matching* – matching can be performed on data or schema level
2. *Element vs. structure matching* – matching can be performed on single elements or their complex combinations
3. *Language vs. constraints* – matching can use linguistics-based approach (e.g. element names) or constraints-based approach (e.g. keys or relationships)
4. *Matching cardinality* – the match result may relate one or more elements resulting in four matching cardinalities – 1:1, 1:n, n:1, or n:m
5. *Auxiliary information* – e.g. dictionaries, thesauri, user interaction, previous results, etc.

As for the combination of the matchers the authors distinguish two cases:

1. *Hybrid matcher* – a matcher that directly combines several matching approaches to determine the match candidates (e.g. combining name matching with data type matching)
2. *Composite matcher* – a matcher that combines results of several independently evaluated matchers

Efficiency Evaluation Last but not least, the authors of paper [14] also analyze and compare existing approaches, but in this case focussing on efficiency of automatic schema matching. They result from the fact, that all the existing works were evaluated on diverse methodologies, metrics, and data and thus it is impossible to compare the systems with each other. But on the other hand, not all the systems are publicly available to enable simple common testing on

an appropriate benchmark. Consequently, the authors analyze the existing public evaluations of the systems. They introduce several criteria influencing the effectiveness and discuss them in relation to the particular systems:

1. *Input* – a kind of input data used:
   - Schema language – i.e. relational, XML, etc.,
   - Number of schemes and match tasks,
   - Schema information – i.e. exploitation of specific facets,
   - Schema similarity – i.e. the ratio between the number of matching elements and number of elements in both schemes, and
   - Auxiliary information – e.g. dictionaries, thesauri, etc.
2. *Output* – information included in the match result:
   - Element representation – e.g. mapping between attributes or whole table, nodes or paths, etc.
   - Cardinality – i.e. 1:1, 1:n, n:1, or n:m
3. *Quality measures* – the match tasks are first solved manually and than compared with the automatic ones exploiting the following two measures:
   - $Precision = \frac{|B|}{|B|+|C|}$
   - $Recall = \frac{|B|}{|A|+|B|}$

   and their various combinations, where
   - A is the set of *false negatives*, i.e. matches needed but not automatically identified,
   - B is the set of *true positives*, i.e. matches identified by both manual and automatic processing, and
   - C is the set of *false positives*, i.e matches falsely proposed by the automatic processing.
4. *Effort* – how much savings of manual effort are obtained, whereas it can be divided to
   - Pre-match effort – e.g. training the machine learning-based matchers, configuration of parameters (e.g. thresholds, weights, etc.), specification of auxiliary information (e.g. domain synonyms, constraints, etc.)
   - Post-match effort – correction of the results

6.6 Summary

As can be seen from the overview, there are several interesting observations and conclusions that can be made.

Firstly, we can say that the area of similarity evaluation among XML documents is well analyzed. This is probably caused by the fact that XML documents can be viewed simply as general trees and thus the problem significantly simplifies. Another reason can bring analyses of real XML data which show that schemes are not used quite often – a significant portion of real XML documents (52% [2] of randomly crawled or 7.4% [29] of semi-automatically collected[6]) have no schema at all.

[6] Data collected with interference of a human operator who removes damaged, artificial, too simple, or otherwise useless XML data.

Second obvious finding is that the amount of works for evaluating similarity between an XML document and an XML scheme is quite low. The question is whether the task is too complicated or it is not as important as in the other two cases. And as we have mentioned, another question is whether the corresponding similarity evaluation algorithm could exploit techniques for automatic schema generation from a sample set of XML documents.

Another observations can be done for similarity evaluations among XML schemes. The existing works focus mainly on semantic similarity of elements, whereas the structural similarity seems to be a marginal part. It is probably caused by the fact that similarity evaluation on schema level is exploited mainly in schema-integration systems, where the semantics of the elements is the key aspect. But especially for various storage strategies would be much reasonable to focus rather on precise structural similarity of schema fragments. A possible solution could follow two different ways. On one hand, the similarity metric could use the idea of partial matchers and their composition, whereas the matchers will describe the structure of the given fragment as precisely as possible. On the other hand, it could be interesting to exploit the idea of tree edit distance for XML documents and define appropriate edit operations and their costs for XML schemes.

7 Conclusion

The main aim of this paper was to enable readers to become acquainted with the currently existing works proposing algorithms for evaluating similarity of XML data. We have described the best known approaches and corresponding representatives and pointed out their most striking advantages and/or shortcomings. For better lucidity the approaches were first classified and described within their categories. We hope that this paper can serve as a good starting point either for selecting an appropriate approach or for proposing a brand new one.

Our future work will focus especially on exploitation of similarity of XML data for enhancing storage techniques based on (object-)relational databases. For this purpose we will focus especially on the above mentioned problem of structural similarity within XML schemes which can be widely exploited especially in so-called *schema-driven* XML-to-relational mapping methods [39] [28].

Acknowledgement

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

References

1. M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *The VLDB Journal*, pages 53–64, 2000.

2. D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 8(4):413–438, 2005.
3. E. Bertino, G. Guerrini, and M. Mesiti. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Inf. Syst.*, 29(1):23–46, 2004.
4. E. Bertino, G. Guerrini, M. Mesiti, I. Rivara, and C. Tavella. *Measuring the Structural Similarity among XML Documents and DTDs.* Technical Report DISI-TR-02-02, Dipartimento di Informatica e Scienze dell'Informazione, Universita' di Genova, December 2001.
5. P. Bille. A Survey on Tree Edit Distance and Related Problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.
6. P. Bille. *Tree Edit Distance, Alignment Distance and Inclusion.* Technical report TR-2003-23, IT University Technical Report Series, Copenhagen, March 2003.
7. P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition).* W3C, October 2004.
8. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition).* W3C, February 2004.
9. E. R. Canfield and G. Xing. Approximate XML Document Matching. In *SAC '05: Proc. of the 2005 ACM symposium on Applied computing*, pages 787–788, New York, NY, USA, 2005. ACM Press.
10. S. S. Chawathe. Comparing Hierarchical Data in External Memory. In *VLDB '99: Proc. of the 25th Int. Conf. on Very Large Data Bases*, pages 90–101, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
11. S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD '97: Proc. of the 1997 ACM SIGMOD Int. conference on Management of data*, pages 26–37, New York, NY, USA, 1997. ACM Press.
12. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 493–504, 1996.
13. G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proc. of the 18th Int. Conf. on Data Engineering*, pages 41–52, San Jose, CA, USA, 2002.
14. H. Do, S. Melnik, and E. Rahm. Comparison of Schema Matching Evaluations. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 221–237, London, UK, 2003. Springer-Verlag.
15. H. H. Do and E. Rahm. COMA – A System for Flexible Combination of Schema Matching Approaches. In *VLDB*, pages 610–621. Morgan Kaufmann, 2002.
16. A. Doan, P. Domingos, and A. Y. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *SIGMOD '01: Proc. of the 2001 ACM SIGMOD Int. Conf. on Management of Data*, pages 509–520, New York, NY, USA, 2001. ACM Press.
17. A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to Map between Ontologies on the Semantic Web. In *WWW '02: Proc. of the 11th Int. conference on World Wide Web*, pages 662–673, New York, NY, USA, 2002. ACM Press.
18. S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Detecting Structural Similarities between XML Documents. In *Proc. of the Int. Workshop on The Web and Databases – WebDB 2002*, pages 55–60, 2002.
19. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB '97: Proc. of the 23rd Int. Conf.*

on Very Large Data Bases, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
20. B. He and K. C.-C. Chang. A Holistic Paradigm for Large Scale Schema Matching. *SIGMOD Rec.*, 33(4):20–25, 2004.
21. T. Jiang, L. Wang, and K. Zhang. Alignment of Trees – An Alternative to Tree Edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.
22. J. Kang and J. F. Naughton. On Schema Matching with Opaque Column Names and Data Values. In *SIGMOD '03: Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 205–216, New York, NY, USA, 2003. ACM Press.
23. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: Clustering XML Schemas for Effective Integration. In *CIKM '02: Proc. of the 11th Int. Conf. on Information and Knowledge Management*, pages 292–299, New York, USA, 2002. ACM Press.
24. J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-Based Schema Matching. In *ICDE '05: Proc. of the 21st Int. Conf. on Data Engineering (ICDE'05)*, pages 57–68, Washington, DC, USA, 2005. IEEE Computer Society.
25. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB '01: Proc. of the 27th Int. Conf. on Very Large Data Bases*, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
26. S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *ICDE '02: Proc. of the 18th Int. Conf. on Data Engineering*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.
27. T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *VLDB '98: Proc. of the 24rd Int. Conf. on Very Large Data Bases*, pages 122–133, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
28. I. Mlynkova and J. Pokorny. From XML Schema to Object-Relational Database – an XML Schema-Driven Mapping Algorithm. In *ICWI'04: Proc. of IADIS Int. Conf. WWW/Internet*, pages 115–122, Madrid, Spain, 2004. IADIS.
29. I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.
30. C.-H. Moh, E.-P. Lim, and W. K. Ng. DTD-Miner: A Tool for Mining DTD from XML Documents. In *WECWIS'00: Proc. of the 2nd Int. Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, pages 144–151, Milpitas, CA, USA, 2000. IEEE.
31. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In *SIGMOD'98: Proc. of the ACM Int. Conf. On Management of Data*, pages 295–306, Seattle, Washington, DC, USA, 1998. ACM Press.
32. P. K.L. Ng and V. T.Y. Ng. Structural Similarity between XML Documents and DTDs. In *Springer Berlin / Heidelberg*, pages 412–421. Lecture Notes in Computer Science, 2003.
33. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Proc. of the Fifth Int. Workshop on the Web and Databases – WebDB 2002*, Madison, Wisconsin, USA, 2002.
34. J. Pokorny. XML Data Warehouse: Possibilities and Solutions. In *Constructing the Infrastructure for the Knowledge Economy: Methods & Tools, Theory & Practice*, pages 531–542, Dordrecht, The Netherlands, 2004. Kluwer Academic Publishers.
35. D. Rafiei, D. L. Moise, and D. Sun. Finding Syntactic Similarities Between XML Documents. In *DEXA '06: Proc. of the 17th Int. Conf. on Database and Expert*

Systems Applications, pages 512–516, Washington, DC, USA, 2006. IEEE Computer Society.
36. E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.
37. E. Rahm, H.-H. Do, and S. Massmann. Matching Large XML Schemas. *SIGMOD Rec.*, 33(4):26–31, 2004.
38. S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, 1977.
39. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99: Proc. of 25th Int. Conf. on Very Large Data Bases*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
40. M. Smiljanic, M. van Keulen, and W. Jonker. Formalizing the XML Schema Matching Problem as a Constraint Optimization Problem. In *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 333–342. Springer, 2005.
41. M. Smiljanic, M. van Keulen, and W. Jonker. Using Element Clustering to Increase the Efficiency of XML Schema Matching. In *Proc. of the 22nd Int. Conf. on Data Engineering Workshops*, page 45. IEEE Computer Society, 2006.
42. H. Su, S. Padmanabhan, and M.-L. Lo. Identification of Syntactically Similar DTD Elements for Schema Matching. In *WAIM '01: Proc. of the Second Int. Conf. on Advances in Web-Age Information Management*, pages 145–159, London, UK, 2001. Springer-Verlag.
43. K.-C. Tai. The Tree-to-Tree Correction Problem. *J. ACM*, 26(3):422–433, 1979.
44. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004.
45. K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, 1968.
46. A. Torsello and E. R. Hancock. Computing Approximate Tree Edit Distance Using Relaxation Labeling. *Pattern Recogn. Lett.*, 24(8):1089–1097, 2003.
47. H. Touzet. Tree Edit Distance with Gaps. *Inf. Process. Lett.*, 85(3):123–129, 2003.
48. H. Touzet. A Linear Tree Edit Distance Algorithm for Similar Ordered Trees. In *CPM*, volume 3537 of *Lecture Notes in Computer Science*, pages 334–345. Springer Verlag, 2005.
49. K. Tu and Y. Yu. CMC: Combining Multiple Schema-Matching Strategies Based on Credibility Prediction. In *DASFAA*, volume 3453 of *Lecture Notes in Computer Science*, pages 888–893, 2005.
50. L. Wang, K. Zhang, K. Jeong, and D. Shasha. A System for Approximate Tree Matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.
51. T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Transactions on Database Systems*, 24(4):529–565, 1999.
52. Z. Zhang, R. Li, S. Cao, and Y. Zhu. Similarity Metric for XML Documents. In *Proc. of FGWM03: Workshop on Knowledge and Experience Management*, Karlsruhe, Germany, 2003.