Integration of Semantic and Structure Similarity for XML Data Ranking

Wilma Penzo

DEIS - CSITE-CNR
University of Bologna, Italy
wpenzo@deis.unibo.it

Abstract. In this paper we present a generic measure that integrates semantic and structure similarity to evaluate results of queries against XML documents. Differently from other proposals, we follow a set-oriented approach to results, that, depending on different relaxations on requirements, return possible alternatives that the user may be interested in. Approximate answers present both structure and semantic loosening, and solutions are ranked with respect to contextual factors, such as the coverage rate of the query and the cohesion of data retrieved. Finally, we present an algorithm that implements our proposal.

1 Introduction

Because of the expected diffusion of XML as a standard for future data representation, efficient and effective treatment of XML documents are important issues to be faced. At present, it is widely acknowledged the necessity of a full-fledged approach to query XML documents [6]. Several proposals [5, 10, 11, 16–20] have extensively emphasized that, besides content, also structure has a crucial role in the effectiveness of retrieval, and in most cases similarity techniques are applied as well. However, these proposals show some lacks. For instance, most of them do not cope with the problem of providing *incomplete results* when parts of query conditions are not satisfied. Also, they do not capture the presence of multiple occurrences of query results in a document, only returning the "best match". Instead, this information could be exploited, for instance, to strengthen the relevance of documents. As to this latest point, most of the existing works provide scoring methods that simply specify a ranking with respect to other solutions. Further, in these proposals, each single score assigned to a document is not enough informative as it is, since it does not indicate "how well" the single document *per se* satisfies query requirements.

This paper presents a generic similarity measure to support the effective retrieval of data in XML format. The main goal is to cover also common cases where providing specific relaxations on structure proves to be a powerful feature in absence of knowledge on data organization. Our method relies on a set-oriented approach, so as not to limit results to the best one, but also to present alternative solutions to satisfy user's requirements. We also present a scoring method that, besides ranking, gives a flavor of the *quality* of data retrieved.

The outline is as follows: In Section 2 we discuss some sample queries to show the limits of existing proposals. Section 3 introduces a starting query language to express user requirements on XML data. A tree representation for queries and documents is presented. In Section 4 tree embedding is the method we use to compare a query and a data tree, and we extend it in two directions: 1) to capture also partial matching on query structure, and 2) to assign a score to embeddings. This introduces the concept of *scored approximate tree embedding*. Then, we show how some "critical" queries are effectively managed by our method, by elegantly handling structural discrepancies. In Section 5 an algorithm that implements our measure is introduced, and implementation is briefly presented. Then, we compare our method with other approaches in Section 6 and, finally, in Section 7 we conclude and discuss future features we intend to work on.

2 Motivation

A common limit of most of current methods for querying XML data is that they do not return results which are *incomplete* with respect to query requirements. For instance, let consider a user looking for CD's authored by Sting, and relating to a concert held in a stadium in the city of Orlando. If data is organized as in documents in Fig. 1, only `Doc2` would be retrieved, since it is the only document that contains a `city` element, thus making condition on Orlando checkable. Nevertheless, it is easy to notice that also `Doc1` is relevant for the query, and should be returned. To our knowledge, this functionality, that we call *partial match on query structure*, is provided only in [17].

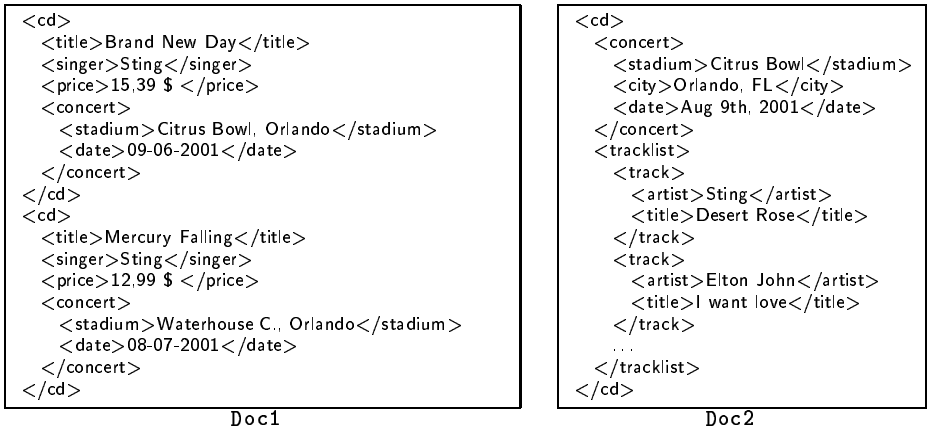

Fig. 1. Sample XML documents

However, in [17] only the best approximate result is retrieved. In many cases, this seems to be a too restrictive condition. In fact, consider `Doc1` in Fig. 1. Two relevant CD's are present, but according to [17] only one of them would appear in the final result. This approach looses an important information useful to score higher documents containing multiple occurrences of results. To this

end, a set-oriented approach seems to be more adequate, provided that, in order to reduce the amount of retrieved solutions, the user is given the possibility of defining proper thresholds to exclude results scoring least.

Then, an effective retrieval system should also capture ordinary structural dissimilarities that may occur on possibly relevant results. As an example, let suppose to ask for Elton John's CD's containing love songs. `Doc2` in Fig. 1 represents a collection of songs of several authors, where also an Elton John's song appears. This CD should be retrieved, but it should be pointed out that the author is related to the song and not to the CD.

None of the above proposals treat these cases, the latter covered only in [10]. Also, when result ranking is provided, the data retrieved is usually scored in a "absolute" fashion, in that constraint relaxations are assigned specific costs to be combined according to formulas that do not indicate the *goodness* of results with respect to requirements. For instance, let consider two queries with 10 and 5 conditions to be satisfied, respectively. Suppose, in both cases, the best result is obtained by the relaxation of 3 conditions. We expect the "quality" of the former solution (7 matching conditions out of 10) to be quite good, whereas the latter one (2 matching conditions out of 5) to be rather poor. Absolute scoring does not provide this information. In order to cope with these situations, we make scores dependent of factors such as the *coverage rate* of the query and the *cohesion* of data retrieved. This has the effect of "normalizing" the scores to the context they are applied in. The method we propose, the $SATES$ (*Scored Approximate Tree Embedding Set*) function, provides the features described above.

3 Query Language and Tree Representation

3.1 Preliminaries

Our approach deals with *typed* and *labelled* trees. As to tree representation of XML documents, we follow the XML Information Set standard [3] (with *parent* and *children*, *root* as `document element` property, and *label* as `local name` property). We also define some additional functions: *nodes(n)* which returns the set of children of a node n, and *type(n)* that returns the type (`element, attribute`, or `PCDATA`) of a node n. Further, we define *leaf(n)* iff $children(n) = \emptyset$, and $ancestor(n_1,n_2)$ iff ($parent(n_1,n_2) \vee \exists\, n \in N$ s.t. ($parent(n_1,n) \wedge ancestor(n,n_2)$). We denote with $\mathcal{T}$ the set of such trees. Then, given a node n in a tree $T \in \mathcal{T}$, we define *support(n)* the subtree of T rooted at n, and we call it the *support* of n in T.

3.2 The $XQuery^{-}$ Query Language

As a starting point, we consider a subset of XQuery1.0 grammar [4], that we call $XQuery^{-}$, and we report it in appendix A. The portion we consider is limited to path expression queries of the form:

```
["/"|"//"](<StepExpr>("/"|"//"))*<PrimaryExpr>"["<Expr>"]"
```

where the part before the square brackets denotes the context where conditions in <Expr> have to be checked. For simplicity, predicates are restricted to equality predicates on attribute values, and can be combined through the `and` and `or` logical connectives. Then, we neglect expressions that consider position of elements inside documents as well as dereferences. Queries are modeled as trees made of typed labelled nodes and (possibly labelled) edges: a legenda of symbols is provided in Fig. 3. The following is an example of complex query, with its tree representation.

Example 1. Retrieve authors of books, having lastname "Spencer" and either 1) having a phone number with value "1392", and a secretary (attribute) or 2) living in "Chicago":

```
book/author[lastname=''Spencer'' and ((.//phone=''1392'' and
          @secretary) or address/*=''Chicago'')]
```

Two separate queries are generated, each one expressing an alternative condition expressed by the `or` operator; final result is obtained by merging results coming from both. The corresponding trees are described in Fig.2 where the '+' sign stands for the merging operation.

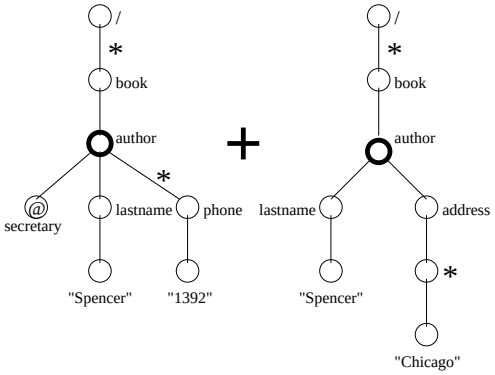

Fig. 2. Query trees of Example 1

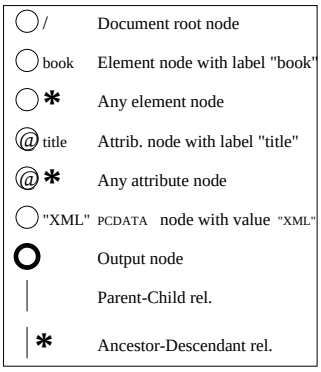

Fig. 3. Query and data tree symbols

For representation purposes, an output node is also present to point out the target of the query. Let $\mathcal{T}_Q \subset \mathcal{T}$ be the set of all query trees. With regard to XML documents, we intentionally neglect the presence of namespaces and IDREFs and we assume data is well-formed, but we do not require validity with respect to a DTD. Let $\mathcal{D}$ be the set of well-formed XML documents. Given a document $d \in \mathcal{D}$, its corresponding tree is a tree $t_d \in \mathcal{T}$. Let $\mathcal{T}_D \subset \mathcal{T}$ be the set of all data trees.

4 The Tree Embedding Problem

Satisfying a query q on a document d may lead to different results, depending on how much we are willing to relax constraints on semantics and requirements dependencies. When a query has to be completely satisfied, we are looking for

a "tree embedding" of the query tree t_q in the data tree t_d. This means that all query nodes must have a corresponding node in the document tree, and each parent-child relationship should be guaranteed at least by an ancestor-descendant one in the data tree. In many cases, this approach is not satisfactory, since it limits the set of relevant results. In order to overcome these limitations, our work basically relies on:

1. relaxation on the concept of *total* embedding of t_q in t_d, in that we admit *partial* structural match of the query tree.
2. approximate results, that are *ranked* with respect to *cohesion* of retrieved data, to relaxation of semantic and structural constraints, and to the *coverage rate* of the query. The ranking function we use takes into account the "query context", in that a score provides a ranking value but also a "quality measure" of results.
3. a set oriented approach to results that, depending on different relaxations on requirements, returns possible alternatives that the user may be interested in. This also possibly strengthens the relevance of data presenting multiple occurrences of query requirements.

In our view, finding an embedding for a query tree t_q in a document tree t_d is a problem that can be defined in a more flexible way. The following definition introduces our interpretation of an *approximate tree embedding.*

Definition 1 (Approximate Tree Embedding (ATE)). Given a query tree $t_q \in \mathcal{T}$ and a document tree $t_d \in \mathcal{T}$, an *approximate tree embedding* of t_q in t_d is a *partial* injective function $\tilde{e}[t_q, t_d] : nodes(t_q) \nrightarrow nodes(t_d)$ such that $\forall q_i, q_j \in dom(\tilde{e})$:

1. $sim(label(q_i), label(\tilde{e}(q_i))) > 0$, where *sim* is a similarity operator that returns a score normalized in [0,1] stating the semantic similarity between the two given labels
2. $parent(q_i, q_j) \Rightarrow ancestor(\tilde{e}(q_i), \tilde{e}(q_j))$

Let $\mathcal{E}$ be the set of approximate tree embeddings. Now we define how scores can be assigned to embeddings so that they can be ranked according to a similarity measure ρ.

Definition 2 (Scored Approximate Tree Embedding). A *scored approximate tree embedding* $\tilde{e}_s$ is an approximate tree embedding extended with *scoring data* in a domain $\mathcal{SD}$. The domain $\mathcal{SD}$ of scoring data is a set of tuples containing information on the coverage of the embedding with respect to the query tree, on the overall semantic similarity and structural correctness of the embedding, and on the cohesion of the resulting embedding with respect to the document tree. Details of $\mathcal{SD}$ elements are provided in Section 5. Formally: $\tilde{e}_s : \mathcal{SD} \times \mathcal{E}$.

Finally, in order to rank the embeddings we define our ranking fuction:

Definition 3 (Ranking Function). We denote with ρ a *ranking function* that, given a scoring data value in $\mathcal{SD}$ returns a normalized score in $[0, 1]$ that summarizes the overall similarity denoted by the input scoring data. Formally: $\rho : \mathcal{SD} \rightarrow [0, 1]$.

In the remainder of the paper, we will use the simplified notation: $\forall t_q \in \mathcal{T}_Q, \forall t_d \in \mathcal{T}_D, t \in \mathcal{T}$, let $s(t_q, t_d)$ be $sim(label(root(t_q)), label(root(t_d)))$, let $r(t)$ stand for $root(t)$, and let $t(t_q, t_d)$ be the equality predicate $type(t_q) = type(t_d)$ that evaluates to true iff types of $r(t_q)$ and $r(t_d)$ are the same. Let $\mathcal{M}_{t_q}^{t_d}$ be a *Bipartite Graph Matching*[1] between the two sets $children(r(t_q))$ and $children(r(t_d))$. When clearly defined in the context, we will use $\mathcal{M}$ in place of $\mathcal{M}_{t_q}^{t_d}$.

4.1 The *SATES* function

The similarity measure we propose to compare query trees over document trees is provided by the *SATES function*, that we define below.

Definition 4 (Scored Approximate Tree Embedding Set Function). (SATES Function) Let $\mathcal{T}_Q$ be the set of query trees, $\mathcal{T}_D$ the set of document trees, such that $\mathcal{T} = \mathcal{T}_Q \cup \mathcal{T}_D$. We define the *scored approximate tree embedding set function* as:

$$SATES : \mathcal{T}_Q \times \mathcal{T}_D \rightarrow 2^{\mathcal{SD} \times \mathcal{E}}$$

such that, $\forall t_q \in \mathcal{T}_Q, \forall t_d \in \mathcal{T}_D$, $SATES(t_q, t_d)$ returns a set of approximate tree embeddings for t_q in t_d. Each returned embedding is assigned a "scoring data" value sd in $\mathcal{SD}$ that denotes the semantic and structural similarity of t_q in t_d with respect to that specific embedding. The $SATES$ function returns a result set, thus capturing the possibility of having more than one embedding between a query tree and a document tree. The scoring data value sd is used as the input of the ρ function to rank the embedding with respect to all the different alternatives.

Intuitively, when comparing a query tree and a data tree, say t_q and t_d, respectively, the $SATES$ function determines "how well" t_d fits t_q, by showing each possible fitting for t_q. The $SATES$ function has a recursive definition that we briefly explain to help the reading of its formal representation shown in Fig. 4. In order to determine the scored embeddings, the function examines different cases:

Both leaves. This is the base case, where the two trees t_q and t_d are simple nodes without children. If labels are similar, the resulting embedding is simply made of the two nodes, and a scoring data that depends on the overall similarity of nodes (on labels and type). When dealing with leaves, if an embedding is returned, the structural similarity can be considered "perfect", thus ascribing to semantic similarity the key role of determining the final score for the embedding through the ρ function.

Query leaf and Data tree. If the semantic similarity between t_q and t_d (root) labels is positive, an embedding is found and its scoring data is determined as in the previous case. Here, note that, even if the structure of the two trees is

[1] Consider a graph $G = (V, E)$. G is *bipartite* if there is a partition $V = A \cup B$ of the nodes of G such that every edge of G has one endpoint in A and one endpoint in B. A *Matching* is a subset of the edges no two of which share the same endpoint. In our case $A = children(r(t_q))$ and $B = children(r(t_d))$.

$\forall t_q \in \mathcal{T}_Q, t_d \in \mathcal{T}_D, SATES(t_q, t_d)$ is defined as:

case $leaf(r(t_q)) \wedge leaf(r(t_d))$:
  if $s(t_q, t_d) > 0$
    $SATES(t_q, t_d) = \{[(s(t_q, t_d), t(t_q, t_d)), \{(r(t_q), r(t_d))\}]\}$
  else $SATES(t_q, t_d) = \emptyset$

case $leaf(r(t_q)) \wedge \neg leaf(r(t_d))$:
  if $s(t_q, t_d) > 0$
    $SATES(t_q, t_d) = \{[(s(t_q, t_d), t(t_q, t_d)), \{(r(t_q), r(t_d)\})]\}$
  else $SATES(t_q, t_d) = \bigcup_{c \in children(t_d)} \ominus_d(SATES(t_q, support(c)))$

case $\neg leaf(r(t_q)) \wedge leaf(r(t_d))$:
  if $s(t_q, t_d) > 0$
    $SATES(t_q, t_d) = \{[(s(t_q, t_d), t(t_q, t_d)), \{(r(t_q), r(t_d)\})]\}$
  else $SATES(t_q, t_d) = \bigcup_{c \in children(t_q)} \ominus_q(SATES(support(c), t_d))$

case $\neg leaf(r(t_q)) \wedge \neg leaf(r(t_d))$:
  if $s(t_q, t_d) > 0$ $SATES(t_q, t_d) =$

$$\bigcup_{\mathcal{M}_{t_q}^{t_d}} \bigcup_{\substack{(t_i^k, t_j^k) \in \mathcal{M} \\ k \in [1..|\mathcal{M}|] \\ (s_{l_k}^k, m_{l_k}^k) \in SATES(t_i^k, t_j^k) \\ l_k \in [1..|SATES(t_i^k, t_j^k)|]}} [\otimes(s(t_q, t_d), t(t_q, t_d), s_{l_1}^1, \ldots, s_{l_{|\mathcal{M}|}}^{|\mathcal{M}|}), \{(r(t_q), r(t_d))\} \cup m_{l_1}^1 \cup \ldots \cup m_{l_{|\mathcal{M}|}}^{|\mathcal{M}|}]$$

  else $SATES(t_q, t_d) = \bigcup(\bigcup_1, \bigcup_2, \bigcup_3)$

$$\text{where } \bigcup_1 = \bigcup_{\mathcal{M}_{t_q}^{t_d}} \ominus_q \circ \ominus_d \bigcup_{\substack{(t_i^k, t_j^k) \in \mathcal{M} \\ k \in [1..|\mathcal{M}|] \\ (s_{l_k}^k, m_{l_k}^k) \in SATES(t_i^k, t_j^k) \\ l_k \in [1..|SATES(t_i^k, t_j^k)|]}} [\otimes(s_{l_1}^1, \ldots, s_{l_{|\mathcal{M}|}}^{|\mathcal{M}|}), m_{l_1}^1 \cup \ldots \cup m_{l_{|\mathcal{M}|}}^{|\mathcal{M}|}]$$

$$\bigcup_2 = \bigcup_{c \in children(t_d)} \ominus_d SATES(t_q, c)$$

$$\bigcup_3 = \bigcup_{c \in children(t_q)} \ominus_q SATES(c, t_d)$$

Fig. 4. The $SATES$ Function

different (one of them is indeed a leaf), the embedding's scoring data should not be influenced from this. In fact, we can evidently conclude that the *structural coverage* is complete.

More interesting is the case of null semantic similarity between labels. In this case, the children of t_d are entrusted to determine among them some possible embeddings for the leaf t_q. If an embedding is found, even if we can say that t_d *covers* t_q, indeed it provides a more generic context. Thus, the final score should reflect the skip of t_d's root, that did not match with the query node. To this end a *lowering function* is used, as follows:[2]

[2] The definition presents also the dual function used in the case of unsuccessful match for a query node (see later in this section).

Definition 5 (Lowering Functions). Let $\ominus_q$ and $\ominus_d$ two lowering functions that, given a set of scored approximate tree embeddings, change their scoring data according to a lowering factor. New scoring data captures the unsuccessful match of a query node and a data node, respectively.

Formally: $\ominus_q$ ($\ominus_d$, resp.): $2^{\mathcal{SD}\times\mathcal{E}} \rightarrow 2^{\mathcal{SD}\times\mathcal{E}}$ such that $\forall \tilde{e}_s = (sd, \tilde{e}) \in dom(\ominus_q)$ ($\ominus_d$, resp.) $\ominus_q(\tilde{e}_s) = (sd', \tilde{e}) \wedge \rho(sd') \leq \rho(sd)$.

Query tree and Data leaf. This is the case that concludes recursion in the next step. Its role is made evident by the following case.

Query tree and Data tree. The final embedding set is basically made of the combinations of each single embedding that can be obtained for children of query root in children of data root, possibly extended with the coupling of the roots (depending on their semantic and type similarity). In order to capture all the existing mappings, all possible bipartite graph matchings are considered between query root's children and data root's children. Then, two cases may occur: either 1) semantic similarity of roots' labels exists, or 2) no similarity is found. The former is the simplest case, with embeddings including roots' coupling.

Example 2. Consider trees in Fig. 5. The set of scored approximate tree embed-

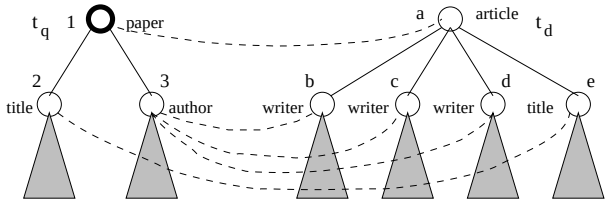

Fig. 5. Bipartite graph matchings between children nodes

dings resulting from $SATES(t_q, t_d)$ is obtained as follows. Without knowledge on query and data subtrees shadowed in Fig. 5, let consider the more promising bipartite graph matchings between children of t_q and t_d: {(1,a),(2,e),(3,b)}, {(1,a),(2,e),(3,c)}, and {(1,a),(2,e),(3,d)}. Let consider the first matching. Depending on the satisfiability of conditions in the supports of "title" and "author" query nodes in the supports of "title" and "writer" data nodes, respectively, $SATES(2, e)$ and $SATES(3, b)$ may return either empty results or some embeddings. The embeddings returned by children are completed with the pair ("paper","article") of roots, and each one is finally assigned a scoring data that depends on the scoring data of the source embeddings, and the semantic similarity between terms "paper" and "article", both being of type `element`.

In order to combine the scoring data coming from children embeddings, we provide the following function:

Definition 6 (Combine Function). We denote with $\otimes$ a n-ary function that, given a set of n scoring data values in $\mathcal{SD}$ returns a new scoring data value in $\mathcal{SD}$ obtained from the combination of the n input arguments. More formally: $\otimes : 2^{\mathcal{SD}} \rightarrow \mathcal{SD}$.

A more complex computation is required if root labels do not present any similarity. In this case, the final embeddings are computed as the union of three quantities, indicated in Fig. 4 by $\bigcup_1$, $\bigcup_2$, and $\bigcup_3$. $\bigcup_1$ captures the unsuccessful match of the roots, thus changing the scoring data by means of the $\ominus_q$ and the $\ominus_d$ lowering functions.[3] On the other hand, $\bigcup_2$ and $\bigcup_3$ capture the presence of possible "swaps", as well as sibling vs. hierarchical relationships, between nodes. Consider Fig. 6. Assume semantic similarity between "song" and "cd" is 0. In

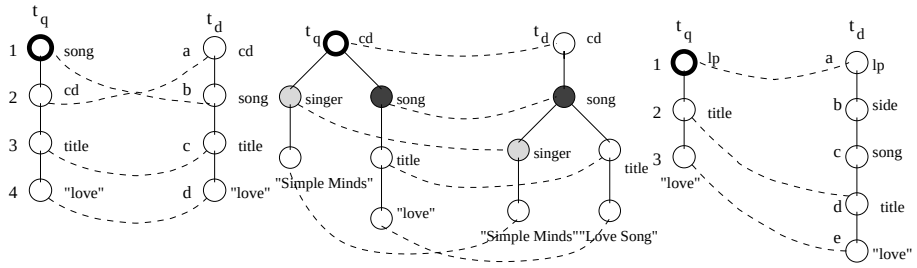

Fig. 6. Swap of nodes **Fig. 7.** Structure unbalance **Fig. 8.** Loss of cohesion

order to provide flexibility and to avoid to assert that certainly no embeddings exist, in the evaluation of $SATES(t_q, t_d)$ some constraints are "relaxed" at both query and document level. In fact: 1) we try to discover an embedding for the query tree t_q in the support of the "song" node in t_d (i.e. in the child of the root of t_d, since this did not match the query root) [$\bigcup_2$ **computation**]; 2) try to partially satisfy the query by attempting a research of an embedding for the support of the "cd" node in t_q [$\bigcup_3$ **computation**].

This "crossed comparison" may point out possible "swaps" between data nodes, due to a different organization of data. In fact, in our example, two embeddings are retrieved: {(1,b),(3,c),(4,d)} and {(2,a),(3,c),(4,d)}. Query requirements are only partially satisfied in both cases.[4] Thus, in the computation of $\bigcup_3$ the $\ominus_q$ function has to be applied. On the other hand, the use of the $\ominus_d$ function in the construction of $\bigcup_2$ is more evident in intermediate recursive steps. In fact, the computation of $\bigcup_2$ also captures structural dissimilarites like that shown in Fig. 7. Here, the "song" and "singer" nodes are siblings in t_q and parent-child in t_d. This also penalizes the final score of the embedding, and it is captured by the $\ominus_d$ function when computing the embedding of *support*("singer") in t_q in *support*("song") in t_d.

Further, when comparing subtrees as an effect of recursive calls of the $SATES$ function, the event of unsuccessful match for the document (current) root may

[3] The expression $\ominus_q \circ \ominus_d \bigcup[\otimes(s^1_{l_1}, \ldots, s^{|\mathcal{M}|}_{l_{|\mathcal{M}|}}), \ldots]$ used in Fig. 4 can be considered equivalent to expression $[\otimes(s_0, s^1_{l_1}, \ldots, s^{|\mathcal{M}|}_{l_{|\mathcal{M}|}}), \ldots]$, where s_0 is a scoring data value that denotes the null similarity between root labels. Indeed, we chose the first form because it better emphasizes the lowering of scoring data due to the unsuccessful match of query and document roots.

[4] Note that, this is not always the case. In fact, sometimes query requirements are completely satisfied since, in the computation of $\bigcup_2$, the query tree can be totally found at a deeper level in the document tree.

be interpreted as a sign of low cohesion of the global result. As an example, let consider the trees of Fig. 8. The evaluation of $SATES(t_q, t_d)$ leads to the embedding {(1,a),(2,d),(3,e)}. In this case, although the query tree is totally embedded in the document tree, cohesion of retrieved data is not fully satisfied. This is captured by the double[5] application of the $\ominus_d$ function that lowers the final scoring data to be assigned to this embedding. Note that the $SATES$ function is not symmetric. Consider trees in Fig. 8 again. Finding an approximate embedding of t_d in t_q would result in a partial satisfaction of query requirements since "side" and "song" current query elements do not have a correspondance in t_q. This means that the $\ominus_q$ lowering function "weighs" differently (much more indeed) from $\ominus_d$. Thus, the retrieved approximate embeddings would be scored differently. This captures the intuition that priority is on complete satisfaction of query requirements, and then on cohesion of results.

5 A Flexible Tree Similarity Algorithm

We implemented the $SATES$ function through an algorithm that evaluates trees in a bottom-up fashion. This is mainly due to the need of reducing the computational complexity of tree embedding methods that, for unordered labeled trees, is known to be NP-hard [15]. In order to limit the portions of a data tree to be considered during evaluation, we start from the bottom, looking for similarity matches between query and data leaves. Branches that do not present any similarity with any query leaf are not explored at all. The algorithm proceeds evaluating nodes in postorder.

Leaves. Depending on the type of query leaves, two strategies are followed: 1) *query leaf is a* `PCDATA` *node.* This is compared in turn with all *PCDATA* leaves of the document tree, looking for similar data leaves; 2) *query leaf is an* `element` *or* `attribute` *node.* This is compared in turn with all data nodes of type `element` and `attribute`, starting from nodes at lowest levels, and ascending in each branch until a similarity is found with some node label.

Intermediate Nodes. Non-leaf query nodes are compared with intermediate data nodes under the following rule: the current query node try a match with the lowest common ancestor of data nodes belonging to each embedding found for query node's children. Two cases may occur:

1. *nodes are similar.* In case data node's children are ancestors of data nodes belonging to the embeddings previosly found for query node's children, existing intermediate data nodes contribute to lower the cohesion of data retrieved, and are considered in the overall score.[6]
2. *nodes do not match.* The unsuccessful match of current roots is reported through the use of the $\ominus_q$ and $\ominus_d$ functions. Then, the algorithm starts

[5] Because of the two recursive steps with unsuccessful match of (2,b) and (2,c)

[6] Note that this holds if a parent-child relationship exists between corresponding query nodes; exceeding data nodes are neglected otherwise.

looking for a possible match of the query root with the parent of the current data root and it proceeds recursively until either a match is found or current data branch ends. If document root is reached without a match, the algorithm "gives up" looking for a match of the current query root, and examines the further query node. Thus, instead of declaring that no embeddings exist for the current query tree, we are willing to relax some conditions with the aim of finding future matches for the remaining query nodes.

In case a query leaf does not find any correspondence in the data tree, the path connecting the leaf with its first ancestor having more than one child is not traversed, but information on skipped nodes is used in the computation of the overall scoring data.

5.1 Computing the Scoring Data

The embeddings returned by the $SATES$ function are equipped with scoring information build up during data tree exploration. This scoring data represents a complex information that capture several properties of the retrieved embedding. These properties reflect semantic and structural completeness and correctness of results, as well as their cohesion. When a relaxation is applied, what we neglect should be taken into account to determine the final overall score. We chose to model a set of properties for each embedding. All these properties are normalized in the interval [0,1]. Properties modelled are shown below, where values close to 1 denote high satisfaction:

Semantic Completeness. It measures how much the embedding is semantically complete with respect to the given query. We compute it as the ratio between the number of query nodes in the embedding, and the total number of query nodes.

Semantic Correctness. It states how well the embedding satisfies semantic requirements. This is computed as a combination of label similarities of matching nodes, possibly lowered by type mismatches.

Structural Completeness. It represents the *structural coverage* of the query tree. It is computed as the portion of node pairs in the image of the embedding that satisfy the same hierarchical[7] relationship of the query node pairs which are related to.

Structural Correctness. It is a measure of how many nodes respect structural constraints. It is computed as the complement of the ratio between the number of structural penalties (i.e. swaps and unbalances) over the number of examined query nodes.

Cohesion of results. It represents the grade of fragmentation of the resulting embedding. It is computed as the complement of the ratio between the number of skipped data nodes and the total number of examined data nodes.

Since all properties are to be considered in conjunction, a t-norm proves to be a good candidate for the ρ ranking function.

[7] Either parent-child or ancestor-descendant relationship.

5.2 Implementation

We implemented a prototype and experienced our similarity measure on a real collection of XML documents. The dataset used is provided by the Astronomical Data Center [1]. The prototype has been developed in Java and uses the API for XML parsing, and the Xerces 1.4.4 parser to process documents. As to semantic similarity, the system refers to the WordNet semantic network [2] and accesses it through the Java WordNet Library (JWNL). As a similarity measure we used an adaptation of the Sussna's formula [8]. We experienced several queries on our prototype, which proves to fully satisfy effectiveness in data retrieval. It captures results presenting both semantic and structure dissimilarities as those shown in Section 4.1, and not retrieved by other approaches. Then, it provides them with a "fine-grained" ranking that also takes into account possible low cohesion of results. However, much work still needs to be done on efficiency. We plan to study and introduce specific indexing techniques to reduce the prototype's response time. We also intend to experience and refine our method, in order to find the better rules to be used for our *combine* and *ranking* functions.

6 Comparison with Related Approaches

Several works deal with the management of XML data. Most of them present query languages [7, 13] and query processing methods [10, 11, 16, 17, 19], others investigate new storage and indexing techniques [9, 12, 14, 18, 20]. However, all of them try to capture both structure and semantics of XML documents to improve effectiveness of data retrieval [6].

Our work belongs to the former class of approaches. As for these, approximate matching techniques are acknowledged to be powerful tools for querying blindly XML data. Thus, many proposals provide similarity query capabilities and, consequently, result ranking [10, 11, 17, 19]. In [19] vague predicates are introduced in the query language, and similarity scores are interpreted as a relevance probability, where basic scores are multiplied in conjunctions and along paths. This leads to a similarity measure that only depends on content, using structure information only for semantic score combination. This approach does not support partial match on query structure. In [11] the concept of *index object* is introduced to specify a context type in an XML document. This is considered as an "atomic unit" where term weighting is applied locally. This technique does not show how structure contributes to similarity score computation, except for providing a term weighting domain. Partial match on query structure is not discussed. In [10] documents and queries are represented as graphs. The result of graph embedding is scored according to weights attached to the matching arcs. However, neither semantic similarity nor partial match on query structure are considered. The approach most similar to ours is that presented in [17], basically because it provides partial match on query structure. However, several differences can be found. Similarity scores depend on the costs of basic transformations applied on the query tree. These costs do not depend on data. Thus, semantic relationships, like synonymy, are not exploited, and results that differ

in the content from the exact match are given the same score. Other approaches use tree matching techniques [5, 15, 16]. Some of them do not consider result ranking [15, 16], others provide limited relaxations on structure [5]. Further, to our knowledge, all proposed methods return only the *best* (in some cases approximate) matchings. The set-oriented approach used in the $SATES$ function also captures the relevance given by multiple occurrences of query requirements in the retrieved data.

7 Conclusion and Future Research

We have presented a generic similarity measure to evaluate results of queries against XML documents. The main novelties of our proposal are: 1) a set-oriented approach to results; 2) a ranking function that provides a "quality measure" to evaluate the adequacy of results with respect to user requirements; 3) partial match on query structure, in order to support at least partial satisfaction of user requirements, when exact match is not possible. Several issues urge to be investigated. In order to augment query flexibility, we plan to refine our ranking function so as to take care of user preferences on either semantics or structure of a query, thus assigning different priorities on results. As to queries with conditions referring to ordered data, we intend to study constraints on our generic unordered tree embedding method. With regard to implementation, in order to cope with the possible hugeness of approximate results returned, we plan to use threshold conditions expressed by the user, to keep only the most relevant results. This would provide a lower bound that would help in pruning branches during exploration, then reducing complexity and lightening system execution. To strengthen optimisation, we also plan to study proper storage and indexing techniques to further improve efficiency of retrieval.

References

1. ADC XML Resources Home Page. `http://xml.gsfc.nasa.gov/`.
2. WordNet Home Page. `http://www.cogsci.princeton.edu/ wn/`.
3. XML Information Set. `http://www.w3.org/TR/xml-infoset`.
4. XQuery 1.0: An XML Query Language. W3C Work. Draft, 2001, `http://www.w3.org/TR/xquery`.
5. S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the 8th Int. Conf. on Extending Database Technology (EDBT 2002)*, Prague, March 2002.
6. R. Baeza-Yates and G. Navarro. Integrating Contents and Structure in Text Retrieval. *SIGMOD Record*, 25(1), 1996.
7. A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1):68–79, 2000.
8. A. Budanitsky. Lexical Semantic Relatedness and its Application in Natural Language Processing. Technical Report CSRG-390, Computer System Research Group, University of Toronto, 1999.

9. Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE 2001)*, pages 595–604, Heidelberg, Germany, April 2001.
10. E. Damiani and L. Tanca. Blind Queries to XML Data. In *In Proc. of Int. Conf. on Database and Expert Systems Applications (DEXA)*, pages 345–356, 2000.
11. N. Fuhr and K. Großjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Proc. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2001.
12. Y. Hayashi, J. Tomita, and G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. In *Proc. ACM SIGIR 2000 Workshop on XML ad Information Retrieval*, Athens, Greece, July 2000.
13. H.V. Jagadish, L.V.S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. of Workshop on Data Bases and Programming Languages (DBPL 2001)*, Italy, 2001.
14. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proc. of the 18th Int. Conf. on Data Engineering (ICDE 2002)*, 2002.
15. P. Kilpeläinen. *Tree Matching Problems with Application to Structured Text Databases.* PhD thesis, Dept. of Computer Science, University of Helsinki, Helsinki, Finland, 1992.
16. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proc. of the 27th VLDB Conf.*, pages 49–58, Rome, Italy, 2001.
17. T. Schlieder. Similarity Search in XML Data Using Cost-Based Query Transformations. In *Proc. of 4th Int. Workshop on the Web and Databases (WebDB01)* , 2001.
18. T. Schlieder and H. Meuss. Result Ranking for Structured Queries against XML Documents. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.
19. A. Theobald and G. Weikum. Adding Relevance to XML. In *Proc. 3rd Int. Workshop on the Web and Databases (WebDB 2000)*, pages 35–40, Dallas, Texas, May 2000.
20. J. Wolff, H. Florke, and A. Cremers. Searching and Browsing Collections of Structural Information. In *Proc. of the IEEE Advances in Digital Libraries*, pages 141–150, USA, May 2000.

A *XQuery*$^-$: A Subset of XQuery1.0 Grammar

Excerpt of the XQuery1.0 grammar we used to express path expression queries (EBNF form), where `Literal` and `QName` are defined as in XQuery1.0 [4].

```
Query ::= PathExpr
PathExpr ::= AbsolutePathExpr | RelativePathExpr
AbsolutePathExpr ::= (‘‘/’’ RelativePathExpr?) | (‘‘//’’ RelativePathExpr)
RelativePathExpr ::= StepExpr ((‘‘/’’ | ‘‘//’’) StepExpr)*
StepExpr ::= (PrimaryExpr StepQualifiers) | (‘‘.’’ | ‘‘..’’ | (‘‘@’’ NameTest))
PrimaryExpr ::= Literal | QName | ‘‘*’’ | ‘‘(’’ Expr ‘‘)’’
StepQualifiers ::= (‘‘[’’ Expr ‘‘]’’)*
NameTest ::= QName | ‘‘*’’
Expr ::= OrExpr | AndExpr | GeneralComp | PathExpr
OrExpr ::= Expr ‘‘or’’ Expr
AndExpr ::= Expr ‘‘and’’ Expr
GeneralComp ::= Expr ‘‘=’’ Expr
```