On a method for XML Schema matching

*Using edit distance and schema trees for structural matching of elements*

M.M. Meijer (Michael)
m.m.meijer@student.utwente.nl

**ABSTRACT**

In this paper a schema is represented using schema trees. Structural matching is performed by matching the local vicinity of each vertex in a schema tree to the vicinity of each vertex in another schema tree using edit distance. A simulated authority is used to determine the quality of the matches by the matcher. The results show that structure indeed plays a role in matching and demonstrate that the method has potential. It produces more effective mappings for larger vicinities, though at the expense of the efficiency.

**Keywords**


1. **INTRODUCTION**

1.1 Data and schema

Today's information technological environment involves various kinds of digital data represented in some sort of format. One might think of diagrams used in software engineering, like databases and software in Entity-Relationship and UML diagrams [OMG07] respectively. Another example can be found in protocols defined for web services to allow heterogeneous applications exchange information [RB01], like SOAP based on XML [W3C07]. Although the examples given seem rather distinct, they are expressed using schemas: "representations of something in the form of a diagram, plan, theory or outline" [PG00a].

**Code 1. Example of an XML Schema fragment**

```xml
<element name="Person">
    <complexType>
        <all>
            <element ref="StreetName" />
            <element ref="HouseNumber" />
            <element ref="PostalCode" />
            <element ref="City" />
        </all>
    </complexType>
</element>
```

The fragment of code above shows part of an XML Schema modeling a person and address elements. XML Schema is utilized to describe the structure of XML documents [Do05]. Its most important components are elements, attributes and (data) types [Do05]. For simplicity, the XML Schema is abstracted to schema elements connected (as a graph like structure) using relationships defined in types.

1.2 Schema integration

The need for data depends on the domain in which it is used. Intuitively, it is understandable that an account manager of a large firm requires different data than a teacher of a high school. Differences give rise to different schemas and various human designers tend to represent the same data using different schemas. To visualize the mentioned issue, an example is given.

When thinking of a person, one often is interested in a person’s address. The question is, how should an address be modeled? Are the individual elements of an address elements of a person (see figure 1) or is the address itself an element of a person (see figure 2) [BRM01]?

![Figure 1. Address elements modeled as person elements.](image1)

![Figure 2. Address modeled as an element of person.](image2)

The example above clearly demonstrates the problem of different representations of (part of) the same domain. Probably, the semantic meaning of both representations is near equivalent, but their syntactic representation is not due to differences in nesting of the elements. Thinking again of the example, assume that the two representations should be integrated. The problem is that the ‘person’ entity modeled differently. This leads to the area of schema integration. The problems of schema integration can be traced back to the early 1980s [RB01]. In the 1990s data warehousing faced (and still faces) a similar problem when extracting and transforming data from the various data sources into the warehouse format [RB01]. Other difficulties can be found when users specify some sort of output they need in a schema, using concepts familiar to them, and the system needs to map those to schema elements in a schema repository [Smi06, RB01].
1.3 Schema matching
A common part in the schema integration process is concerned with a so called match operation. Basically it takes two given schemas as input and how they match as output [RB01, BMR01]. Any kind of available information can be exploited to match schemas [Do05]. One might think of element names (linguistics), data types or schema structure [Do05].

Various schema matching systems exist and are still under research (e.g. COMA, Artemis, Cupid [RB01, SE05]). They consist of one or more matchers. Each matcher performs a match operation [RB01, BMR01, DR02]. For schema-only based matching, one can differentiate between element-level and structure-level matchers [RB01], in which linguistics, type similarity and graph (structural) matching play important roles. Linguistics, for example, exploit properties like finding synonyms and homonyms of element names, while graph matching is concerned with isomorphism (structure similarity) [RB01, BRM01, SE05]. To be more effective, several matchers are combined into a hybrid matcher or a composition of matchers to exploit several properties. A hybrid matcher is a matcher using a mixture of properties at the same time, e.g. linguistic properties intertwined with structural properties [RB01, SE05]. Cupid is a concrete example, being a hybrid matcher using the similarity of element names (linguistic) and matching tree structure biased toward leaves (structural) [BRM01]. A composition is the use of several matchers in some sort of sequence, e.g. first a match on linguistic properties succeeded by matching structural properties (usually combining the [weighted] results of the matches) [RB01, SE05].

1.4 Research
The research conducted aimed at answering the main question:

*Can a method using edit distance and schema trees be of use for structural matching of XML Schema elements?*

A method is “a systematic procedure for doing something” [PG00b] and it is of use when it has “practical worth or application” [PG00c]. From a computer science point of view, one might say, the method (in the form of an algorithm) ideally is both efficient concerning space and time complexity and effective concerning the quality of the results (the matches found). The method is designed with XML Schema in mind. XML Schemas are relatively close to tree structures, allowing simpler and faster computation than graphs in general. An XML Schema can be represented by a graph from which a tree can be derived; a schema tree. The method provides a structure-level approach for an individual (no hybrid/composition) schema-only matcher, because it concerns tree structures of schemas only. The matches will be limited to 1:1 cardinality matches, meaning that the method tries to match one element of schema A with one element of schema B [RB01]. The 1:n (one-to-many), n:1 (many-to-one) and n:m (many-to-many) cases tend to be difficult to automate [RB01] and are left out. The research demonstrates the potential of a method using purely structural matching. It is an initial research on a method to improve schema matching (from a structural point of view).

2. DEFINITIONS

2.1 Function and graph theory
This section states definitions of a schema tree and (local) vicinity as they play a central role throughout this paper. Basically they rely on function and graph theory. From function theory, the definition of functions [Wei07a] and an injective [Wei07b] and surjective function [Wei07c] combined to form the bijective function [Bar07] is presumed to be common knowledge and therefore will not be defined in this paper.

From graph theory the definitions of a graph [Gri04a], tree [Gri04b], in and out degree [Gri04a], path [BV00], cycle [BV00] and graphs being connected [BV00], rooted [Rod07] and labeled [SKJ04] are all considered common knowledge and therefore will not be defined in this paper. In this paper a subgraph [BV00], with a subset of vertices and edges, is analogous to the partial subgraph referred to in some literature [SKJ04]. Throughout this paper, the words graph and directed connected graph are used interchangeably.

**Definition: Schema tree.** Let T denote a labeled rooted connected directed tree $T = (V, E, R, \Sigma)$ with vertices (or nodes) V, edges E, a root vertex R and an alphabet of vertex labels $\Sigma$. The ordering of the children (or siblings) is not important, the tree is said to be unordered. Let $\Sigma$ denote the alphabet or set of unique labels for the vertices, labeled by the bijective function $L: V \rightarrow \Sigma$. This means that each vertex has a unique label. In this paper whenever a schema tree is mentioned, this definition is meant.

**Definition: (Local) vicinity.** A vertex w is said to be in the local neighborhood or vicinity of a vertex v if v can be reached by a path from v of length or depth k, when looking at the graph as if it were undirected. The vicinity around a vertex is a rooted subgraph itself.

![Figure 3. A schema tree rooted at u1. Two vicinities, around u2 and u5, with depth 1 are marked by the dotted lines.](image)

Some examples in case of a schema tree will follow. For $k = 0$ the vicinity of v consists solely of v. The vicinity of v for $k = 1$ is the subtree v, its parent (root) and its children. For $k = 2$ the vicinity of v is the subtree v, its grandparent (root), its parent, its siblings, its children and its grandchildren. The vicinity of Address in figure 2 for $k = 1$ is: Person (root), Address and the children of Address. Another example can be found in Figure 3 above. A schema tree is depicted which is rooted at u1. When looking at the schema tree as if it were undirected, the marked area’s are the vicinities around u2 and u5 for $k = 1$. For $k = 2$, the vicinity around u2 would also include the vertices u1 and u4. In case of the vicinity around u5, the vertex u1 would be included.

2.2 Edit Distance
Graph matching can be used to determine the structural similarity between graphs [NRB06]. As seen in figure 1 and 2, graphs are not always similar (isomorphic). An intuitive and flexible error-tolerant graph matching method is the edit distance computation [NRB06]. Edit distance is a metric counting the number of edit operations to transform one graph into another. There are three edit operations: substitution, insertion and deletion applicable to vertices and edges [NRB06]. Let $\lambda$ denote null (empty). An edit operation in the form $a \rightarrow b$ is a substitution; a is replaced by b. Insertion of a is denoted by $\lambda \rightarrow a$ and deletion by $a \rightarrow \lambda$ [SZ97]. Each edit
operation has a certain cost [NRB06]. For example the cost of a substitution $c_{sub}$ is 1 and for insertion $c_{ins}$ or deletion $c_{del}$ is $\frac{1}{2}$, then substitution is as expensive as a deletion followed by an insertion. A sequence of edit operations is called an edit path [NRB06]. The ultimate goal of the edit distance is to find a minimum cost edit path to transform one graph into another [NRB06].

A main drawback of the edit distance is its time and space complexity, because it is exponential in the number of vertices of the graphs involved [NRB06]. When restricting graphs to unordered labeled trees, like schema trees, complexity is proven to be NP-complete (even MAX SNP-hard) [SZ97] and thus is hard or unknown how to solve efficiently. This basically limits the size of the graphs that can be matched with edit distance (to make it of use).

3. STRUCTURAL MATCHING

3.1 XML Schema structure representation

The first step in structural matching is to represent XML Schemas using graphs. Here schema trees are used. For structural matching it is not necessary to include all syntax defined for XML Schema. As stated in the introduction, XML Schema can be abstracted to elements and their relationships (defined in their types) as it conforms to the intention of structural matching of elements. The ordering of children should not matter. For example, swapping City and PostalCode in figure 1 should not make it a different tree.

For structural matching of elements there is no need to involve documentation, notation, annotation, attribute, grouping and include/import syntax (the latter two can be resolved by rewriting the XML Schema or during the transformation of the XML Schema into the schema tree). Attributes are left out, because they are not part of the structure-level under consideration. A schema tree requires four ingredients: vertices, edges, a root vertex (just one of the vertices) and labels. Vertices are the elements themselves in the XML Schema [SKJ04, LWW05a]. Edges are the relationships defined by the types of the elements, which means that an edge implies that the type of one element includes another element (implicit edges [SKJ04]). In the code example in the introduction, the element Person has a type including the element City. Thus Person and City are vertices connected by an edge. The vertices are uniquely labeled [SKJ04] in an arbitrary fashion, e.g. they can be numbered or have the names given to the elements in de XML Schema. A word of caution is required though, because elements can be reused throughout the XML Schema, every occurrence of an element denotes a new vertex and necessitates a unique label. For example in figure 4, a Person can work for a Company which has its own Address (different context), XML Schema allows the address element to be reused for the company, but the schema tree will represent it as a new vertex with a unique label.

Another word of caution is for an XML Schema representing a document. A document can have sections and sections can have sections themselves (recursive [LWW05a]), as in figure 5. A schema tree would then have a vertex for a section and another one for a section that can (recursively) be contained by a section.

![Figure 5: Violation of schema tree properties due to recursion: cycle and no unique labels.](image)

A last word of caution is required when looking at the edges. Because XML Schemas are represented using schema trees, cycles and in degree greater than 1 are not allowed. This means a loss of structural precision. Recursion is lost, because a loop back (for example with sections in the example above) would violate the properties of a schema tree, as in figure 5. This also excludes referential constraints (key, unique) from being part of the structure, because it requires an additional edge between vertices next to the edge connecting the vertex to its parent.

![Figure 6: Violation of schema tree due to referential constraint: in degree is 2 for BankAccount.](image)

In figure 6 a Person has a BankAccount, at the same time a referential constraint is forced upon the BankAccount (unique). This necessitates an additional edge violating the in degree property of schema trees (a BankAccount with only a referential constraint is of little use as it cannot belong to a Person anymore). The transformation of an XML Schema to a tree structure, resembling the way described here, is used in another method too [LWW05]. A lossless representation is a schema graph [SKJ04].

3.2 Matcher

The actual matching is performed by the matcher and is described in Appendix A. It takes a source (1) and target schema tree (2) for input, each having unique vertex labels within and between them ($\Sigma$ has no duplicates and $\Sigma_1 \cap \Sigma_2$ is an empty set). Basically the matcher matches the vicinity of each vertex, representing an XML Schema element, in one tree with the vicinity of each vertex in another tree. This is assumed to be an indication of how well two elements match. An initial mapping stores the costs of the edit path between the vicinities of both vertices in a mapping table. The mapping table is a 2-dimensional array where each row represents a vertex of one tree and each column a vertex of another tree. A cell denotes a mapping. After the initial mapping is completed, only the lowest value mappings per row are preserved. Note that a row can have multiple mappings with the same (lowest) value and all of them are kept as a solution (match) by the matcher. The matcher cannot differentiate between mappings with equal costs.

The matcher reduces the schema matching problem to $|V_1| \times |V_2|$ edit distance computations for vicinities (sub schema trees) of smaller size than the input schema trees. With sufficiently small size of the vicinities, the matcher can determine the mappings.
3.3 Edit Distance using A*

The main operation the matcher relies upon, is the minimum cost edit path or edit distance computation between the vicinities of two vertices (line 5, Appendix A). Appendix B describes the A* algorithm to find the minimum-cost edit path between two schema trees. As stated in section 2.3, edit distance is inherently complex and thus a search method is needed that limits the search space under consideration (as opposed to a brute force search). A widely-known and used search method is based on A* [RN03, NRB06]. It makes use of an informed search strategy by using additional information to guide its search direction, a heuristic, and finds solutions more efficiently [RN03]. Intuitively, if one can guess the best direction to search in, one can search more efficiently. The closer the heuristic is to the actual remaining cost to a goal, the more efficient A* searches [RN03]. For A* to return an optimal solution, the heuristic must be admissible (never overestimate the cost to reach the goal) [RN03, NRB06]. The complexity of A* depends on the heuristic and is hard to characterize, but is exponential in the worst case [RN03]. Dropping admissibility yields suboptimal results, but allows more heuristics and can enable faster searching [RN03]. A* works by constructing a search tree (this has no relation with schema trees or matching in general) in which each vertex contains a (partial) solution [NRB06]. The f-function consists of two components: the costs made so far g(p) and a heuristic h(p) which is a lower bound estimate on the remaining costs to reach the goal [RN03, NRB06]. The resulting function is f(p) = g(p) + h(p) [RN03, NRB06]. The costs made so far are the costs of the partial solution.

To get an idea of how A* in Appendix B works, an example is given for a source and target (simple) schema tree in figure 7.

![Figure 7: Schema trees, source (left) and target (right).](image)

The algorithm gets the best partial solution \( p_{\text{min}} = \{u_1 \rightarrow v_1\} \) from OPEN and determines whether a goal is reached (line 5-6). This can be done by applying the solution (edit path) to the source and comparing the result to the target. In this case, no goal is reached yet. The best partial solution is now expanded with a substitution and a deletion (case 1, line 11-13). The algorithm now repeats itself. Then \( u_1 \rightarrow v_1 \) is expanded. The result of applying the edit paths \( \{u_1 \rightarrow v_1, u_2 \rightarrow v_2\} \) and \( \{u_1 \rightarrow v_1, u_2 \rightarrow \lambda\} \) is depicted in figure 8.

![Figure 8: Three schema trees when applying the edit paths (above) and search tree (below).](image)

Again after repeat, the algorithm expands the current best solution \( u_2 \rightarrow v_3 \). This yields a goal and a final solution. The result of applying the edit path \( \{u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow \lambda\} \) is the transformation of the source into the target, depicted in figure 9.

![Figure 9: Schema trees when applying the edit paths (left) and the search tree (right).](image)

These are the steps the matcher takes to find an optimal solution. It uses the A* algorithm to find the minimum-cost edit path between two schema trees.

![Figure 10: Solution (left) and the search tree (right).](image)
The figures show only edit operations on vertices, not on edges. The reason is that schema trees have an advantage over graphs in general. Substitution applies only to vertices (change of label). Insertion of a vertex implies the insertion of an edge connecting the vertex to its parent. An exception is the insertion of a new root (no parent), but for simplicity the matcher ignores it as a special case. Deletion of a vertex implies the deletion of the edge connecting the vertex to its parent. For simplicity the root cannot be deleted, because then one has to deal with forests (a set of trees, like the top right schema tree in figure 8). This reduces the number of possible (partial) solutions and allows A* to find a solution faster. Further, uniform cost (1) for the edit operations on vertices and edges is assumed: substitution costs 1, insertion costs 2 (vertex + edge) and deletion costs 2 (vertex + edge). Another way to reduce the number of possible (partial) solutions is at line 10, by only selecting unprocessed vertices. If this is not done, A* adds useless solutions to OPEN, like edit paths with both substitutions and insertions on the same vertex.

4. EVALUATION

4.1 Quality quantification

Having specified the method for matching, it is not straightforward to determine its quality. Structure is only part of the information one can use for matching XML Schemas [Do05]. Yet to obtain some indication of the quality, a method used for schema matching systems [DMR03] is used. A match is first performed manually (M_{human} [SKJ04]) and referred to as the real or correct match result [DMR03]. It is then compared to the automatically derived match [DMR03] (M_{system} [SKJ04]). The attempt of the matcher is to increase the area where M_{human} and M_{system} intersect (that area contains the correct mappings of the matcher from the human point of view), see figure 11.

![Figure 11: Match results of human and system.](image)

The following quality measures are used: precision, recall, f-measure and overall [SKJ04, DMR03].

- **Precision** is the ratio of the correct mappings by the match of the system and the total number of mappings of the system [SKJ04, DMR03] in the [0,1] interval.

  \[
  \text{precision} = \frac{|M\text{_{system}} \cap M\text{_{human}}|}{|M\text{_{system}}|}
  \]

- **Recall** is the ratio of the correct mappings by the system and the total number of correct (human) mappings [SKJ04, DMR03] in the [0,1] interval.

  \[
  \text{recall} = \frac{|M\text{_{system}} \cap M\text{_{human}}|}{|M\text{_{human}}|}
  \]

- **F-measure** combines precision and recall as neither alone can accurately measure the quality [DRM03]. Returning as many mappings as possible maximizes recall and returning as few (correct) mappings as possible maximizes precision [DRM03], both at the expense of the other. The combined measure gives equal importance to both precision and recall and is a variant of f-measure(α) [SKJ04, DMR03].

  \[
  f\text{-measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
  \]

- Overall was developed specifically for schema matching [DRM03] and quantifies the post-match effort needed to modify the match result into the correct match result [SKJ04, DRM03]. F-measure is more optimistic than overall [DMR03]. Overall becomes negative if precision < 0.5 [DMR03]. High values mean little effort.

  \[
  \text{overall} = \text{recall} \cdot \left(2 - \frac{1}{\text{precision}}\right)
  \]

4.2 Simulation

Unfortunately, there seems to be no large repository available with XML Schemas transformed into schema trees and matched by humans. It requires a significant amount of time and human resources to perform a large amount of schema tree matches. In order to perform quality quantification for the method described in this paper, a simulated (human) authority is used. The simulated authority consists of a generator and a transformer.

The generator is capable of generating random schema trees. Each generated vertex is labeled with a character followed by a number (e.g. u_0, u_4, v_3) and has a 50% chance of having either one or two children. Each generated vertex is added to a list of expandable vertices and while no given maximum number of iterations is reached, a vertex is removed from the list and expanded with children. Furthermore, the generator can generate the subtree representing the vicinity around a given vertex in a schema tree with k = 1 or k = 2 (the current method is not capable of processing larger vicinity subtrees due to their size). Finally, the generator creates the simulated human mapping M_{human} of two given schema trees. In order to do so, it assumes that a pair of vertices from two schema trees match if their labels have the same number (e.g. u_5 matches with v_5, a_3 matches with x_3).

The transformer is capable of distorting a given schema tree with a given number of transformations. It randomly inserts vertices (as a child), deletes vertices (except from the root) and substitutes swaps labels of one vertex with another vertex within a schema tree. Each has about 33% chance of being chosen. A special case is insertion at the root. There is a 50% chance the vertex is inserted as the new root or as a child of the root.

4.3 Results

Schema trees are randomly generated by the generator. For each generated schema tree a copy of it is transformed by the transformer and a mapping M_{human} generated by the generator. Each pair of generated and transformed schema trees is fed to the matcher, which determines the system mapping M_{system}. For each mapping in M_{human} a corresponding mapping in M_{system} is searched and, if found, put in M_{human} \cap M_{system}. Then the values for the measures are determined. The averaged results are presented in table 1 in Appendix C. Potentially the matcher can return mappings completely different from the human simulated mappings (both = 0). These cases occur infrequently and are left out, because the F-measure and overall then divide by 0.

The first partition of 4 rows are results for 200 matches of small trees with 0 – 7 transformations. The second partition of 4 rows are results for 200 matches of larger size trees with 0 – 25 transformations. Both partitions employ k = 1 for the vicinity. The third partition of 4 rows are results for 200 matches of small trees with 0 – 7 transformations, employing k = 2 for the vicinity. Due to the long processing time of larger vicinities, the test could not directly scale up to larger trees for k = 2. A minor modification to the matcher, is testing the sum of the sizes of both vicinities before matching. If the sum of the sizes exceeds
the mapping is given the maximum possible value (more or less discarded). This enables matching larger trees for k = 2. The results of this modified matcher are represented in the last two partitions for k = 1 and k = 2 respectively. This modification is used for the last two partitions.

From the first and second partition it can be seen that matching direct vicinity only becomes rather ineffective due to the large number of system mappings found. A very large number of mappings is produced for larger trees by the matcher, yielding low precision (though recall is high because good matches are found too). The low values for precision result in low values for the f-measure and overall. This is most likely caused by the inability of the matcher to differentiate between the mappings. Two explanations can be given:

1. Only direct vicinity, parent and children, is matched and intuitively has a larger chance looking like another vicinity (the same edit distance).
2. The uniform costs might not represent a good distribution of the costs. The importance and difficulty of finding a good cost distribution is a known problem for edit distance [NB05, NB06].

The results of the third partition with larger vicinity have better f-measure and overall than the first partition with smaller vicinity. The number of mappings by the matcher is smaller, leading to better precision, but the lower values for $M_{human} \cap M_{system}$ (both) lower the recall. Still, it seems that using a larger vicinity (more differentiation, explanation 1) produces better mappings. Results from the last two partitions (modified matcher) show medium size trees with 0–5 transformations and k = 1 and k = 2 respectively. It clearly demonstrates that a larger vicinity is more effective, as can be seem from the more positive values of the measures.

In general, the more transformations, the lower the measures (quality). This seems logical, because the structure is less likely to be the same. This seems to affect the larger vicinity matches more, because they match larger structures and become more sensitive to changes in the structure. There is also an indication that larger vicinities produce better matches. The results should not be accepted to strong and seen as an indication only, due to the fact that simulation is used instead of humans and no schema trees based on real life XML Schemas are matched.

A result that stems from observations, not apparent from the table in Appendix C is the computation time for schema trees with very different structures. If for example a schema tree with vertices having at most one child is matched with a schema tree with vertices having many children, computation time skyrocket. This can be explained by the heuristic used. Basically the heuristic counts the costs of the remaining edit operations left for the unprocessed nodes of both schema trees. If the schema trees just described have an equal number of vertices, the edit distance algorithm might determine substitutions for each vertex, but after all vertices have been processed the heuristic becomes useless (will always return 0). This is rather inconvenient, because a certain number of delete and insert operations are still necessary, but there is no guidance by the heuristic anymore (reducing efficiency).

A last result from observations, also not apparent from the table in Appendix C is the value for $\theta$ in the heuristic. Large values seem dangerous, because the algorithm then favors longer edit paths over shorter ones too much. If a long but wrong edit path is chosen, the other edit paths are not used and the algorithm gets stuck. Some experimentation lead to $\theta = 1.092$ for the tests.

5. RELATED WORK

Various matchers and schema matching systems have yet been developed [RB01]. Work on graph and string matching and isomorphism is also available [RB01, RFB02, SZ97]. Edit distance has known application in string matching [CRF03, RB01, SE05] and ordered trees [SZ97].

COMA++, build upon the COMA prototype, is a schema matching system with a matcher exploiting structural properties [DR07]. The matcher determines the (Euclidean) distance between structural statistics determined for single nodes (number of children, parents, leaves) [DR07]. Combined matchers incorporate bottom-up matches by deriving similarity of elements by taking into account the similarity of their children and the leaves [DR07]. A top-down approach is taken by deriving similarity of elements by taking into account the similarity of parents and siblings (though leaves are again used) [DR07].

Artemis is another tool making use of matching. It matches elements by their affinities of name, structure and dependencies [CDV01]. The structural affinity of elements is measured by domain compatibility [CDV01], that is the extent of similarity between the sub elements of the elements [CDV01] (similarity of the relationships of the elements [RB01]).

The Cupid matching system has support for structural matching by looking at the leaves and tree structures [BMR01]. Cupid has a bias towards leaves [BMR01]. Non-leave similarity is based upon linguistic similarity, similarity of the sub trees rooted at the non-leaves and the similarity of their leave sets. The latter stems from the assumption that leaves between schema vary less than between structural content and structure described by the schema is mainly represented in the leaves [BMR01, RB01, SE05].

Similarity Flooding uses a so called fix-point computation on graph structures [MGR01]. An initial similarity mapping between elements is based upon string-based comparison [MGR01, SE05]. Then similarity spreads from similar elements to adjacent neighbors and iteratively spreading depth increases and similarity changes, until a fix-point is reached [SE05].

Tree matching is a known problem and involves algorithms for edit distance computation [SZ97]. For ordered trees, efficient algorithms are available for approximate computation [SZ97, RB01]. Unordered trees tend to be more difficult to deal with [SZ97]. Tree matching is purely structural [RB01].

A method related to the one described in this paper uses tree structures and approximate tree matching [LWW05a]. It was observed that similar XML Schemas have similar elements and similar parent-child and sibling relations [LWW05a]. The method maps elements of a schema by looking at name similarity, node similarity (e.g. data type and cardinality) and structural similarity between the subtrees rooted at the nodes under consideration [LWW05a]. It is concerned with approximate common substructures [LWW05a, LWW05b].

The research considered purely structural (tree) matching of schemas. The method, using schema trees where vicinities around elements are matched using edit distance, is new to the best of the author’s knowledge. Additionally the research demonstrated the role structure alone can play in matching.

6. CONCLUSIONS

A method for purely structural matching of XML Schemas by means of schema trees and edit distance is described. An XML Schema must first be transformed into a schema tree, by
transforming elements into vertices (of which one is the root), types describing relations between elements into edges and label the vertices uniquely. A matcher can then match schema trees by determining the vicinities for each pair of vertices from the input schema trees and mapping their edit distance. The edit distance can be found using a suboptimal version of A*. A final elimination of ‘bad’ mappings returns a final mapping. Using the measures precision, recall, f-measure and overall the quality of the matcher is determined. Instead of human matching a simulated authority is used to generate schema trees, transforming schema trees and approximate human mappings.

As stated in section 1.4 the method is of use if it is both efficient and effective (qualitative). The current method is strongly dependant on the edit distance computation of the A* method, which has a rather weak heuristic and takes a long time to compute for larger vicinities. A heuristic more suited for trees can reduce computation time and increase efficiency. If vicinities are of limited size, the method can be relatively efficient, delivering results within several seconds. From the tests it seems that using larger vicinities produces better results for the quality measures, that is, the matcher is more effective but at the expense of computation time (and thus efficiency).

Concerning effectiveness, the method shows potential. A larger vicinity seems more effective, increasing precision at the cost of recall and increasing f-measure and overall. The increase of precision stems largely from the fact that the number of mappings by the matcher is significantly lower with the larger vicinity. The method should not be used as a matching system itself or in situations requiring fast mappings. The true potential of (purely) structural matching using this method requires additional research and comparison with other methods.

7. FUTURE WORK

The method described requires further research. First of all the heuristic being used should be replaced by one that is more efficient for tree structures. The main reason for a better heuristic is faster computation and larger vicinities to be matched. A possible heuristic is given in [RFB02], which is based on bipartite graphs to which trees belong. Second, a better cost distribution for edit operations is necessary to differentiate more between vicinities (and can increase precision). In the current method uniform cost is used, yielding a lot of system generated mappings with equal costs. A method for learning the distributions of the costs based on machine learning [NB05, NB06] is expectation maximization [NB06]. Another way to allow for more differentiation between vicinities is to encode more information in the schema trees. Allowing vertices and edges to have properties, just like schema graphs [SKJ04], and edit operations on properties. By being able to match larger vicinities, more realistic matches can be performed. Real schemas, transformed to schema trees, with a lot of vertices can then be matched by humans and the matcher. This eliminates the need for simulation and enables the matcher to be tested in real life scenarios. The method described in this paper might even be extended to schema other than XML. Schema or the method can be adopted for use in known matching systems (as a composite or hybrid matcher) to see its use in matching systems. Comparison to other structural approaches might be interesting.

ACKNOWLEDGMENTS

The A* algorithm is a slightly adapted version of the algorithm given in [NRB06]. Support and guidance for the research was provided by the supervisor of the project.

REFERENCES


**APPENDIX A: MATCHER**
The following method shows the structural matcher of elements for two schema trees. EDIT-DISTANCE refers to Appendix B.

```
STRUCTURAL-MATCH(Tree t1, Tree t2, Length k)
Input: Unequal non-empty schema trees T1 = (V1, E1, R1, \Sigma1) and T2 = (V2, E2, R2, \Sigma2)
where V1 = \{v1, v2, v3, ... , u1|V1|\} and V2 = \{v1, v2, v3, ... , v|V2|\}
Length k of the path, limiting the vicinity around vertices in the trees.
Output: A mapping of vertices in V1 onto V2.
1: Initialize MAPPING as a 2-dimensional array, with dimensions |V1| x |V2|
2: Initialize MATCHING as an empty set, containing the best mappings found for V1 onto V2
3: For each vertex v in t1 generate the local vicinity (sub) schema tree s_v
4: For each vertex w in t2 generate the local vicinity (sub) schema tree s_w
5: MAPPING[v][w] = EDIT-DISTANCE(s_v, s_w)
6: For each row r in MAPPING
7: Let BEST denote the best mapping (lowest distance) found for r
8: Let ROWMATCHING be the empty set, containing the best mappings found for r
9: For each column c in MAPPING
10: If MAPPING[r][c] < BEST
11: BEST = MAPPING[r][c]
12: Make ROWMATCHING empty (previous matches found in r are invalid)
13: Add (r, c) to ROWMATCHING
14: Else if MAPPING[r][c] == best
15: Add (r, c) to ROWMATCHING
16: Merge ROWMATCHING with MATCHING
17: Return MATCHING
```

**APPENDIX B: EDIT DISTANCE IN A**
The following method shows how to compute the edit distance for two schema trees using A*, this is a slightly adapted version from the algorithm given [NRB06]. F-FUNCTION is the evaluation of f(p). GOAL-TEST determines whether p_min transforms T1 into T2.

```
EDIT-DISTANCE(Tree T1, Tree T2)
Input: Unequal non-empty schema trees T1 = (V1, E1, R1, \Sigma1) and T2 = (V2, E2, R2, \Sigma2)
where V1 = \{v1, v2, v3, ... , u1|V1|\} and V2 = \{v1, v2, v3, ... , v|V2|\}
Output: A minimum cost edit path, set/sequence of edit operations, p_min
to transform T1 into T2
1: Initialize OPEN to the empty queue of edit paths sorted by F-FUNCTION
2: For each vertex \omega in V2 insert edit path \(v_1 \rightarrow \omega\) into OPEN
3: Insert edit path \(v_1 \rightarrow \lambda\) into OPEN
4: Repeat
5: Let p_min = remove minimum cost edit path from OPEN
6: If(GOAL-TEST(p_min))
7: Return p_min as the solution
8: Else
9: Let p_min = e.g. \{v_1 \rightarrow v_{i1}, v_2 \rightarrow v_{i2}, ... , v_k \rightarrow v_{ik}\}
10: Let V2\_un = V2 \setminus {v_{i1}, v_{i2}, ... , v_{ik}}, all unprocessed vertices of V2
11: If k < |V1|, not all nodes of V1 are processed yet
12: For each \omega in V2\_un insert edit path \(p_{\text{min}} \cup \{v_{i1} \rightarrow \omega\}\) into OPEN
13: Insert edit path \(p_{\text{min}} \cup \{v_{i1} \rightarrow \lambda\}\) into OPEN
14: Else
15: Insert edit path \(p_{\text{min}} \cup \bigcup_{\omega \in V2\_un} \{\lambda \rightarrow \omega\}\) into OPEN
```
APPENDIX C: RESULTS

Results of the matcher using a simulated authority (approximating human matching). Size is the number of matches performed, trees is the average number of vertices of the source and target schema trees, depth concerns the vicinity and trans. is the number of transformations applied. The other columns show averages over the number of matches (size). Note that F-measure and Overall are not values calculated from the average precision and recall, but are averages themselves. The column both is the average number of mappings of $M_{human} \cap M_{system}$. Values closer to 1 for precision, recall, F-measure and overall are better. Overall becomes negative due to values of precision below 0.5, though higher values are better.

<table>
<thead>
<tr>
<th>Size</th>
<th>Trees</th>
<th>Human</th>
<th>System</th>
<th>Both</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
<th>Overall</th>
<th>Depth</th>
<th>Trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>8,52</td>
<td>8,52</td>
<td>26,60</td>
<td>8,52</td>
<td>0,336</td>
<td>1,000</td>
<td>0,500</td>
<td>-1,08</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>8,98</td>
<td>7,98</td>
<td>31,24</td>
<td>6,10</td>
<td>0,204</td>
<td>0,759</td>
<td>0,318</td>
<td>-2,321</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>8,54</td>
<td>7,16</td>
<td>26,34</td>
<td>4,34</td>
<td>0,173</td>
<td>0,607</td>
<td>0,267</td>
<td>-2,418</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>8,70</td>
<td>6,88</td>
<td>28,80</td>
<td>3,88</td>
<td>0,137</td>
<td>0,551</td>
<td>0,218</td>
<td>-3,005</td>
<td>1</td>
</tr>
</tbody>
</table>

| 5    | 50    | 46,10 | 46,10  | 721,42| 46,10     | 0,065  | 1,000     | 0,121   | -13,574| 1      |
| 6    | 50    | 46,44 | 44,72  | 725,32| 41,06     | 0,057  | 0,918     | 0,107   | -14,320| 1      |
| 7    | 50    | 46,26 | 43,18  | 726,62| 36,40     | 0,051  | 0,843     | 0,096   | -15,038| 1      |
| 8    | 50    | 46,44 | 38,58  | 731,22| 26,44     | 0,037  | 0,686     | 0,069   | -17,498| 1      |

| 9    | 50    | 8,90  | 8,98   | 20,32 | 8,88      | 0,512  | 0,991     | 0,655   | -0,221 | 2      |
| 10   | 50    | 8,68  | 7,76   | 18,66 | 5,08      | 0,294  | 0,646     | 0,395   | -1,094 | 2      |
| 11   | 50    | 8,57  | 7,16   | 17,00 | 3,74      | 0,233  | 0,520     | 0,315   | -1,316 | 2      |
| 12   | 50    | 8,95  | 7,20   | 20,16 | 3,12      | 0,165  | 0,432     | 0,234   | -1,920 | 2      |

| 13   | 50    | 23,24 | 23,24  | 186,68| 23,24     | 0,127  | 1,000     | 0,225   | -5,982 | 1      |
| 14   | 50    | 23,86 | 22,16  | 198,44| 19,08     | 0,098  | 0,861     | 0,176   | -7,161 | 1      |

| 15   | 50    | 23,06 | 23,06  | 78,50 | 21,20     | 0,304  | 0,923     | 0,451   | -1,479 | 2      |
| 16   | 50    | 23,32 | 21,74  | 84,34 | 14,32     | 0,184  | 0,658     | 0,284   | -2,494 | 2      |