A Selftuning Approach for Improving Composite Schema Matchers

Fabien Duchateau, Remi Coletta, Zohra Bellahsene

A Selftuning Approach for Improving Composite Schema Matchers*

Fabien Duchateau and Zohra Bellahsene and Remi Coletta

LIRMM - Université Montpellier 2
161 rue Ada 34000 Montpellier
{firstname.name}@lirmm.fr

Abstract. Most of the schema matching tools are assembled from multiple match algorithms, each employing a particular technique to improve matching accuracy and making matching systems extensible and customizable to a specific domain. Recently, it has been pointed out that the main issue is how to select the most suitable match algorithms to execute for a given domain and how to adjust the multiple parameters. The solutions provided by current schema matching tools consist in aggregating the results obtained by several match algorithms to improve the quality of the discovered matches. In this article, we present a novel method to replace this aggregation function and its drawbacks. Unlike other composite matchers, our matching engine makes use of a decision tree to combine the most appropriate match algorithms. As a first consequence, the performance of the system is improved since only a subset of match algorithms from a large library is used. The second advantage is the improvement of the quality of matches. Indeed, for a given domain, only the most suitable match algorithms are used. Our approach is also able to learn the most appropriate match algorithms for a given domain by relying on the expert feedback. It can also selftune some parameters like thresholds and the performance versus quality ratio.
Keywords: schema matching, machine learning, MatchPlanner, decision tree.

1 Introduction

Schema matching is the task of discovering correspondences between semantically similar elements of two schemas or ontologies [17, 19]. Syntax based schema matching has been formally discussed in the survey by Rahm and Bernstein [24], extended by Shvaiko and Euzenat in [25] with respect to semantic aspect. Many tools have been designed to fulfill this task, each employing several techniques to improve matching accuracy and extensibility for specific domains. This generation of schema matching tools often share two features: an aggregation function to combine the match algorithms, and some parameters which need to be tuned. Yet, these features suffer from several drawbacks. Due to the aggregation, the tools apply all their match algorithms between all couples of elements. Yet, a match algorithm may be very efficient for a certain set of schema but really unsuitable with another set, implying a time and resource consuming process. The aggregation function itself can lead to a poor matching quality, for example

* Supported by ANR Research Grant ANR-05-MMSA-0007

by favouring closely related match algorithms. Besides, the manual tuning can reveal difficult for the user, who does not know the impact of her changes. Thus setting up some thresholds and weights or editing a list of synonyms might not produce the expected results.

In this article, we present a novel method for combining schema matching algorithm, which enables to avoid the previously mentioned drawbacks: the aggregation function of the matching tools is replaced by a decision tree, which enables to match two elements by executing an appropriate plan of match algorithms instead of applying all match algorithms from a library. This results in better **performance**. Besides, we can learn the most appropriate match algorithms for a given domain thanks to the expert feedback. Thus, the second advantage is the improvement of the **quality of matches**. Indeed, for a given domain, only the most suitable match algorithms are used. Finally, our approach is also able to **tune automatically** the system for providing the optimal configuration for a given matching scenario.

Contributions. We designed a flexible and efficient method for the schema matching problem. The main interesting features of our approach are:

- Introducing the notion of planning in the schema matching process by using a decision tree.
- Learning the best strategy for a given domain.
- A tool has been designed based on the planning approach with selftuning capability.
- Experiments demonstrate that our tool provides good performance and quality of matches regarding the main matching tools.

Outline. The rest of the paper is organised as follows. Section 2 deals with drawbacks of current matching tools. In section 3, we present our idea for a matching plan based on a decision tree. Section 4 describes the capacity of our approach to learn the best decision tree for a given scenario. Section 5 contains an overview of our prototype. The results of experiments for showing the effectiveness of our approach are presented in section 6. Section 7 covers the related work. Finally, we conclude in section 8.

2 Motivations

Most matching tools are assembled from multiple match algorithms, which are then aggregated to improve matching accuracy and making matching systems extensible and customizable to a particular domain. Thus the aggregation function can be seen as the kernel of a matching tool. However, as pointed out in [16], the main issues are how to select and combine the most suitable match algorithms to execute for a given domain and how to adjust the multiple knobs (e.g. threshold, performance, quality, etc.). Two other important issues can be added: how to take into account the user expertise and the precision versus recall problem.

2.1 A Brutal Aggregation Function

Lots of semantic similarity measures have been proposed in the context of schema matching (refer to [24] for a survey of the different measures). And none of these measures outperforms all the others on all existing benchmarks. Therefore, most matching tools [1, 8, 9] aggregate the results obtained by several similarity measures to improve the quality of discovered matches. However, the aggregation function entails major drawbacks on three aspects.

Performance. A first drawback is to apply useless measures, involving a costly time and resource consumption. Indeed, let consider matching two schemas with n and m elements thanks to a matcher which uses k measures. Then $n \times m \times k$ similarities will be computed and aggregated. Yet, there are many cases for which applying the k measures is not necessary. The following example shows that even a reliable match algorithm, like the use of a dictionary, may fail to discover even simple matches. Consider the two elements *name* and *named*. Applying a string matching technique like 3-grams between them provides a similarity value equal to 0.5. On the contrary, a dictionary technique (based on Wordnet[1] for example) would result in a very low similarity value since no relationship between the two elements can be inferred. Thus, some techniques can either be appropriate in some cases or they can reveal totally useless. Applying **all measures** between **every couple of elements** involves a costly time and resource consumption.

Quality. The aggregation function may negatively influence the quality. First, it might give more weight to closely-related match algorithms: using several string matching techniques between the polysemous labels *mouse* and *mouse* leads to a high similarity value, in spite of other techniques, like context-based, which could have identified that one label represents a *computer device* and the other an *animal*. Besides, the quality due to the aggregation does not necessarily increase when the number of similarity measures grows. Matching *mouse* and *mouse* with one or two string matching algorithms already results in a high similarity value. Thus using more string matching algorithms would not have an interesting impact.

Flexibility. The aggregation function often requires manual tuning (thresholds, weights, etc.) in the way of combining the measures. This does not make it really flexible w.r.t. new similarity measures contributions. For instance, if a new measure is considered as reliable for a specific domain (based on an ontology for example), how would it be aggregated easily by an expert ?

2.2 Too Many Parameters for the User

The user often has to manually tune the matching tool: edit a list of synonyms, set up some thresholds or weights, etc. This task can reveal tiresome or difficult, and the less the expert has to tune, the easier the matching system is. Some tools like eTuner [16] have been designed to automatically tune schema matching tools: a given matching

[1] http://wordnet.princeton.edu

tool (e.g. COMA++ [1] or Similarity Flooding [18]) is applied against a set of expert matches in several configurations until an optimal one is discovered. However, eTuner heavily relies on the capabilities of the matching tool, especially for the available match algorithms and its aggregation function. Besides, it does not improve the performance since all match algorithms are computed for every couple of elements.

2.3 Is Expert Feedback Useless ?

Obviously not. Yet, many schema matching tools do not take advantage of this expert feedback. The expert is often asked to validate the matches through a GUI. Unfortunately, this input is rarely used later on. We believe that it should be re-injected in the matching process to improve the results.

2.4 Recall vs Precision

In [14], the author underlines the problem of recall vs precision. Matching tools like COMA++ focus on a better precision, but this does not seem to be the best choice for an end-user in terms of post-effort: consider two schemas containing 100 elements each, there is potentially 10,000 matching possibilities (considering only 1:1 matchings), and the number of relevant matches is 25. Let us assume a first matcher discovers 10 relevant matches and achieves 100% precision, then the expert would have to find manually the 15 missing matches among 81,000 possibilities. On the contrary, another matcher returns a set of 300 matches, and it achieves a 100% recall. As all the relevant matches have been discovered, the expert has to remove the 275 irrelevant matches among the 300 ones. Thus favouring the recall seems a most appropriate choice. And note that technically speaking, it is easier to validate (or not) a discovered mapping than to manually browse two large schemas for adding new matches.

2.5 Another Way to Design Matching Tools ?

To solve previously mentioned drawbacks, our approach aims at replacing the current kernel of matching tools by a decision tree, and it favours the recall to reduce user post-match effort. It also relies on the expert feedback which is re-injected in a machine learning process to improve the decision tree. Finally, we avoid a tiresome tuning task by automatically setting up parameter's values during the machine learning process.

3 A Decision Tree Based Kernel

This section covers the use of a decision tree as the kernel of matching tools, in replacement of the aggregation function. We first explain the notion of decision tree, and give its interesting features for the matching context.

3.1 Decision Trees

The idea is to determine and apply, for a matching scenario, the most suitable matching techniques, by means of a decision tree [22]. In our context, a decision tree is a tree whose internal nodes are the similarity measures, and the edges stand for conditions on the result of the similarity measure. Thus the decision tree contains plans (i.e ordered sequences) of match algorithms. We use well-known match algorithms from Second String[2], and we add the neighbour context from [9], an annotation-based measure, a restriction measure and some dictionary-based techniques. The similarity value computed by a measure must satisfy the condition (continuous or discrete) on the edges to access a next node. Thus, when matching two schema elements with our decision tree, the first similarity measure, at the root node, is computed and returns a similarity value. According to this value, the edge for which its condition is satisfied leads to the next tree node. This process will iterate until a leaf node is reached, indicating whether the two elements should match or not. The final similarity value between two elements is the last one which has been computed, since we consider that the previous similarity values have only been computed to find the most appropriate measures.

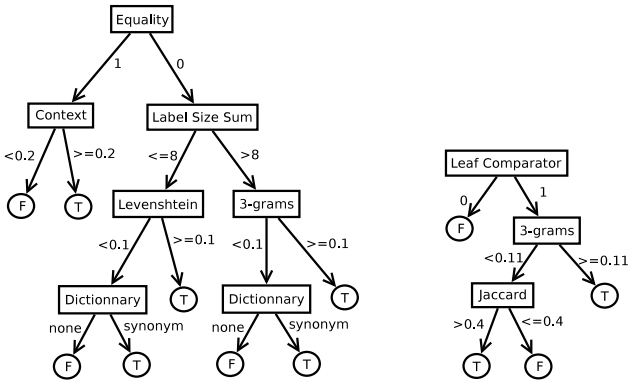
Figure 1 illustrates two examples of decision tree. The first one (1(a)) focuses on the quality, it includes some costly measure (context, dictionary). The tree depicted by figure 1(b) aims at discovering some matches quickly, by using mainly string matching measures. Now let us see how the two couples of elements *(author, writer)* and *(name, named)* are matched with the quality-based decision tree (figure 1(a)): *(author, writer)* is first matched by *equality* which returns 0, then the *label sum size* is computed (value of 12), followed by the *3-grams* algorithm. The similarity value obtained with *3-grams* is low (0.11), implying the *dictionary* technique to be finally used to discover a synonym relationship. On the contrary, *(name, named)* is matched using *equality*, then *label sum size*, and finally *3-grams* which provides a sufficient similarity value (0.5) to stop the process. Thus, only 7 match algorithms have been computed (4 for *(author, writer)* and 3 for *(name, named)*) instead of 12 (if all distinct match algorithms from the decision tree would have been used).

3.2 Advantages of this New Kernel

Decision trees appear especially well-suited for the task of combining similarity measures. Indeed, it is able to handle both numerical (3-gram, Levenhstein,..) and categorical (data restriction, being synonyms, ...) attributes (i.e. measures in our context). In addition, decision trees are robust to irrelevant attributes. Thus the quality of the similarity measure produced by a decision tree is strictly growing up with the number of similarity measure taken as input. We can also point out that the tree structure avoids testing some potentially costly similarity measures. Moreover, a learned decision tree is computed only once, and the learning process is performed in less than 1 second.

Another advantage of our approach deals with the **performance versus quality** aspect. With traditional matching tools, it is very difficult to favour one aspect by tuning

[2] http://secondstring.sourceforge.net/



(a) quality-based (b) performance-based

Fig. 1. Examples of decision tree

some parameters or the aggregation function. In our approach, a decision tree is linked to a performance/quality ratio. A user is able to quickly discover some matches or to emphasize on the matching quality, simply by selecting the decision tree which fulfills her needs.

We also note that the **threshold** is specific to each similarity measure. Thus, it is easy and meaningful to adjust the thresholds for a given similarity measure. On the contrary, the other approaches often have a global threshold, and tuning its value implies to favour the similarity measures with a homogeneous distribution of their values.

Such a kernel suffers from the fact that a decision tree might not be appropriate for all scenarios. However, we show in the next section that building a decision tree for a given scenario is an automatic process thanks to machine learning methods.

4 Learning a Decision Tree in the Schema Matching Context

Although the user validates some of the discovered matches with most matching tools, this expertize is rarely used then. Our approach enables the expert to validate some matches, which results in the learning of new decision trees thanks to the C4.5 algorithm. Another motivation is the number of parameters to be tuned for an expert. We aim at reducing this by providing several Pareto optimal trees. The expert can then easily select the most appropriate one according to her needs (performance or quality).

4.1 Learning New Decision Trees with C4.5

We propose to formulate the problem of determining, for a given schema matching scenario, the most appropriate decision tree as a machine learning task. Our approach is based on C4.5 [23], since it is able to combine both continuous and discrete attributes. It ensures other good properties: a high classification score, which results in a good matching quality, and it minimizes the height of the generated decision tree, implying

better performance. The machine learning classification problem consists in predicting the class of an object from a set of its attributes. In the context of schema matching, an object is a couple of elements and its class represents its validity in terms of mapping relevance. The match algorithms are the attributes of this couple. And as training data, we use the matches validated or rejected by the expert. The C4.5 algorithm is briefly described by algorithm 1.

Algorithm 1: Learning decision tree with C4.5 algorithm

$C \leftarrow$ all couples classified by the expert
foreach *similarity measure m* **do**
  find the condition *cond* over m that maximizes the information gain g (i.e allows to correctly classify a maximum of couples of C with condition *cond*)
$m_{best} \leftarrow m$ that maximize g
create a decision node n that splits on m_{best}
recur the subsets of C obtained by splitting on m_{best} and add those nodes as children of node n

Machine Learning techniques have already been used in the context of reconciling schemas. In [5], the authors propose a full machine learning based approach called LSD, in which most of the computational effort is spent on the classifiers discovery. As a difference, our approach enables to reuse any existing similarity measures and it focuses on combining them. As a consequence, we avoid spending too much time into the learning process and we easily capture any previous and future work on similarity measure definition. This C4.5 technique is flexible, robust and self-tuned to combine several similarity measures.

4.2 Pareto

As the learner generates many different decision trees, only those who provide an advantage (either on the quality or on the performance) are kept. To select them, we apply the Pareto optimality [13].

Pareto optimality: Given a set of learned decision trees, a movement from one tuning to another that can make at least one decision tree better off without making any other decision tree worse off is called a Pareto improvement. A tuning is Pareto optimal when no further Pareto improvements can be made.

Pareto frontier: For a given system, the Pareto frontier is the set of all tunings which are all Pareto optimal. Figure 2 shows some learned decision trees (labelled points from A to G) and the line represents the Pareto frontier. The optimal decision trees (A, B and C) are kept while the other trees (D, E, F, and G) are discarded since they are dominated by at least another point. For instance, the point E is dominated by the point A on the performance and by the point B on the quality. We also notice that

tree A ensures good performance, but a low quality, since this tree mainly uses string matching measures. It could be the tree depicted by figure 1(b). On the contrary, the tree E emphasizes on the quality to the detriment of performance. Indeed, it includes some costly measures like dictionnary-based or context. An example of such tree is depicted by figure 1(a).

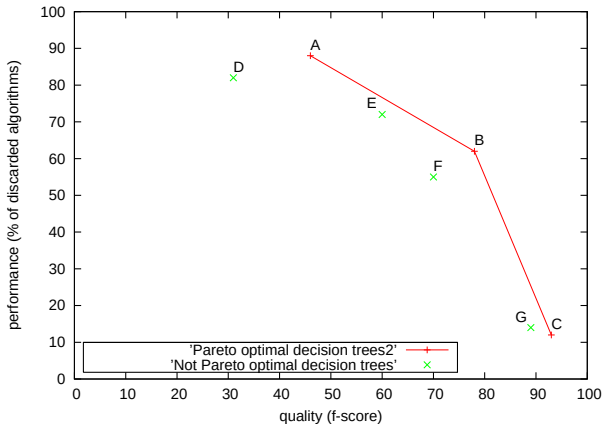


Fig. 2. Pareto frontier to select optimal decision trees

4.3 Discussion

A constraint of the learning approach is the misclassification rate. Indeed, the learning process tends to decrease the number of errors, i.e. it aims at classifying correctly. However, when the number of relevant matches is not significant w.r.t. the total number of couples, the learning process generates a decision tree which always returns false, thus minimizing the misclassification rate. To solve this problem, we use the cost sensitive learning [4] which enables to assign a greater penalty to the irrelevant matches to the benefit of the relevant ones.

Contrary to other matching tools, our approach is flexible and it enables to easily integrate new measures: the learner takes it into account by computing all similarities between all couples of elements. Then the measure will be put (or not) in the decision tree according to its utility. To be short, there is no need to manually update the weights of an aggregation function or to set up thresholds.

5 Prototype

In order to demonstrate the benefits of our decision tree approach, a prototype named MatchPlanner has been implemented in Java. This section briefly describes its architecture and how it works.

5.1 Overview of the Architecture

Figure 3 shows the architecture of our prototype, with two main parts: (i) the schema matcher and (ii) the learner. Both parts share a main component, the decision trees, since the learner is in charge of generating them while the matcher uses them as a kernel.

As a schema matcher, it takes as input schemas and a decision tree. Note that the schemas are also stored in the knowledge base (KB). This KB can be seen as a repository for schemas and expert matches. The matching process mainly relies on the decision tree (see section 3) to generate a list of matches. The expert might decide to validate some discovered matches, which are then stored in the KB and used by the learner.

The learner part aims at generating one or more decision trees from information stored in the KB (schemas and some expert matches). It is based on the C4.5 machine learning algorithm as described in section 4. The learned decision trees can then be used by the matching process.

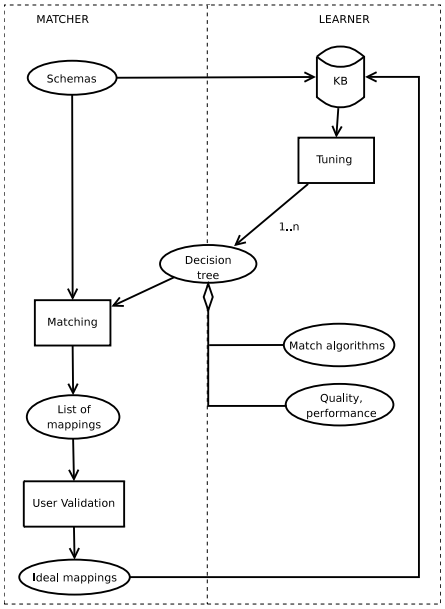


Fig. 3. Architecture of MatchPlanner

5.2 How MatchPlanner works

Here we briefly describe how the whole process is performed in our prototype. At first, the user selects a decision tree (some default trees are provided) and she quickly discovers a few matches. An expert then validates some of these matches, implying them to be stored in the KB. Thanks to this expert knowledge, new decision trees with different performance and quality ratios are generated by the learner. One of these trees can be

	MatchPlanner			COMA++			SF		
	P	R	F	P	R	F	P	R	F
book	1.0	1.0	1.0	1.0	0.25	0.40	0.60	0.75	0.67
university	0.86	0.67	0.75	1.0	0.56	0.72	0.80	0.45	0.58
thalia	0.83	0.67	0.74	1.0	0.53	0.69	0.6	0.6	0.6
travel	0.57	0.8	0.67	0.0	0.0	0.0	0.75	0.6	0.67
person	0.58	1.0	0.73	0.86	0.86	0.86	1.0	0.57	0.73

Table 1. Matching quality of MatchPlanner, COMA++ and Similarity Flooding on the different domains.

used to improve the matching quality and performance, and the discovered matches can be validated again. This process can iterate until the user is satisfied with the results.

6 Experiments

In this section, we demonstrate that MatchPlanner provides good results when compared to other schema matching tools, reputed to provide an acceptable matching quality: COMA++ and Similarity Flooding. COMA++ uses 17 similarity measures to build a matrix between every couple of elements to finally aggregate the similarity values and extract matches. As for Similarity Flooding, it converts the input schemas into graphs, then it discovers some initial matchings between graph elements thanks to string matching measures. And a propagation process enables to refine the matchings. To evaluate and compare MatchPlanner with these two schema matching tools, we first emphasize on the quality aspect, which is crucial in schema matching. Then, we show that MatchPlanner ensures good performance, an important aspect when dealing with large and/or numerous schemas.

6.1 Comparison with Other Matching Tools

This part shows the quality provided by our approach. To compare the matching quality of the three matching tools, we use common measures in the literature, namely precision, recall and F-measure. Precision calculates the proportion of relevant extracted matches among extracted matches. On the contrary, recall computes the proportion of relevant extracted matches among relevant matches. F-measure is a tradeoff between precision and recall. The matching tools were executed on 5 domains: **book** and **university**, widely used in the literature, **courses** from Thalia [15], **travel**, extracted from air-company web forms, and the last one is about **person**. Table 1 shows the resulting matching quality, P, R and F respectively standing for precision, recall and F-measure.

With the **book** domain, MatchPlanner achieves a F-measure equal to 1.0. COMA++ discovers only one relevant mapping, resulting in a low F-measure. This is explained by the fact that COMA++ algorithms are mainly based on the aggregation of string matching measures and a list of synonyms. Similarity Flooding obtains an average quality, but it was able to discover most of the relevant matches (recall equal to 0.75), although

there were also many irrelevant ones (precision equal to 0.6).

The **university** domain has been widely used in the literature [7]. Its two schemas describe an Australian and a US university. They do not contain any data type or annotations. However, we demonstrate that this lack of information does not influence on MatchPlanner's matching quality. We notice that MatchPlanner obtains the best results with a F-measure equal to 0.75. COMA++ has also an acceptable 0.72 F-measure. Similarity Flooding performed a little better, reaching a 0.58 F-measure, probably because of the small schema size, which limits the propagation efficiency. Note that it is possible that MatchPlanner obtains a higher recall with a different learned decision tree, but this heavily decreases the precision.

Thalia domain is difficult to match: although half of the expert matches are similar labels, it is very difficult to discover the other matches. COMA++ only discovers all similar matches thanks to string matching measures, but the 1.0 precision finally ensures a F-measure equal to 0.69. On the contrary, SF succeeds in discovering more relevant matches thanks to the propagation, but it obtains a lower F-measure. MatchPlanner does not achieve a high recall on this domain because most similarity measures does not enable to discover the other relevant matches. Thus, the learned decision tree does not change when the expert validates more matches. However, we obtain the highest F-measure (0.74).

Travel domain is also a difficult challenge for matching: the schemas have been extracted from various air-booking websites. COMA++ is not able to discover any mapping. Both SF and our prototype achieves a 0.67 F-measure. Contrary to SF, MatchPlanner emphasizes on the recall (0.8) to the detriment of precision (0.57). Indeed, as explained in section 2, favouring the recall seems the most appropriate choice to reduce the user post-match effort.

The **person** domain is composed of two schemas with strongly heterogeneous labels. Although COMA++ obtains the best F-measure (0.86), MatchPlanner is the only matching tool to discover all the relevant matches (recall to 1.0). As for SF, it achieves the best precision but it misses half of the relevant matches. With the results of COMA++ or SF, the user has to manually browse the input schemas to discover the rest of the matches. But her effort is reduced with MatchPlanner since she only removes some irrelevant discovered matches.

6.2 Performance Aspect

Finally, we emphasize on the time performance. Indeed, it is important to point out that MatchPlanner performs as quick as the other matching tools. Table 2 shows the performance aspect of MatchPlanner, COMA++ and Similarity Flooding. Note that for COMA++, we added both the time spent to parse the schemas and store their information in the database, and the matching time. Indeed, Similarity Flooding does not dissociate these two phases. As for MatchPlanner, we added both the time to generate a new decision tree and to perform the matching phase. On the 5 domains, the three matching tools perform well: they need less than 2 seconds to provide the set of matches. SF is fast but with detriment to the quality, as seen in section 6.1.

	MP	COMA++	SF
book	≤1s (60/280)	≤1s	≤1s
university	≤1s (214/2008)	2s	≤1s
thalia	2s (847/867)	2s	≤1s
travel	≤1s (222/520)	≤1s	≤1s
person	2s (191/360)	2s	≤1s

Table 2. Time performance of MatchPlanner, COMA++ and Similarity Flooding

Table 2 gives another detail for MatchPlanner. The ratio in parenthesis represents the resource sparing due to the decision tree. The first number stands for the number of calls to match algorithms that has been performed to match the schemas, while the second is the total number of calls if all distinct match algorithms from the decision tree would have been used. For example, on the **book** domain, MatchPlanner computed 60 similarity values. If our tool used all match algorithms from the tree, 280 calls would have been executed. We notice that some cases enable to execute less than half of the match algorithms (**book** or **university** domains for instance). But it also occurs that the gain is minimal (**thalia** domain), because of the structure of the learned decision tree. This shows that MatchPlanner is able to spare some resource by computing only the appropriate match algorithms for a given couple of schema elements.

7 Related Work

This section covers the related work in schema/ontology matching. As many approaches have been proposed in these domains [2, 12, 17, 20, 21], we only detail the works which are closer to MatchPlanner.

Similarity Flooding [18] have been used with Relational, RDF and XML schemas. These schemas are initially converted into labeled graphs and SF approach uses fix-point computation to determine correspondences of 1:1 local and m:n global cardinality between corresponding nodes of the graphs. The algorithm has been implemented as a hybrid matcher, in combination with a name matcher based on string comparisons. First, the prototype does an initial element-level name mapping, and then feeds these matches to the structural SF matcher. The weight of similarity between two elements is increased, if the algorithm finds some similarity between the related elements of the pair of elements.

COMA++ [1] is a generic, composite matcher with very effective match results. It can process the relational, XML, RDF schemas as well as ontologies. Internally it converts the input schemas as trees for structural matching. Similarity of pairs of elements is calculated into a similarity matrix. At present, it uses 17 element level match algorithms with an user defined synonym and abbreviation table. For each source element, elements with similarity higher then than a threshold are displayed to the user for final selection. MatchPlanner does not use the whole set of match algorithms, but it is able to learn the best combination. And the expert feedback is re-injected in the process.

AUTOMATCH [3] is the predecessor of AUTOPLEX, and it uses schema instance data and machine learning techniques to find possible matches between two schemas.

It explicitly uses Naive Bayesian algorithm to analyse the input instances of relational schemas fields against previously built global schema. The match result consists of 1:1 correspondences and global cardinality. The major drawback of this work is the importance of the data instances. In most cases, there are not available for different reasons (security, lack, etc). The data instances can contain errors or some information might be missing, leading to a quality decrease. Although this approach is interesting on the machine learning aspect, it seems risky to have only one matching technique.

eTuner [16] aims at tuning schema matching tools. It proceeds as follows: a given matching tool (e.g., COMA++ or Similarity Flooding) is applied against a set of expert matches until an optimal configuration is found for the matching tool. However, eTuner heavily relies on the capabilities of the matching tool, especially for the available match algorithms and its aggregation function. On the contrary, MatchPlanner is aimed at learning the best combination of a subset of match algorithms (not schema tools). Moreover, it is able to self tune important features like the performance and quality.

GLUE [6] is the extended version of LSD, which creates ontology/taxonomy mapping using machine learning techniques. The system takes as input a set of instances along with the taxonomies. Glue classifies and associates the classes of instances from source to target taxonomies and vice versa. It uses a composite approach, as in LSD, to do so. the computational effort is spent on the classifiers discovery. As a difference, our approach enables to reuse any existing similarity measures and it focuses on combining them. Furthermore, we avoid wasting time into the learning process and our planner tool can easily integrate additional matching techniques.

Decision trees have been used in ontology matching for discovering hidden matches among entities [11]. Their approach is based on learning rules for matching terms in Wordnet. In another work [10], decision trees have been used for learning parameters for semi-automatic ontology alignment method. This approach is aimed at optimizing the process of ontology alignment and supporting the user in creating the training examples. However, the decision trees were not used for choosing the best match algorithms. Moreover, our method is fully automated and provides self tuning capability.

8 Conclusion and Future Work

In this paper, we presented a flexible and efficient approach for the next generation of schema matching tools. To replace the aggregation function, we propose to use a decision tree as the new kernel of schema matching tools: the main idea is to build a matching plan based on learning techniques. Thus our approach enables to combine the most appropriate match algorithms for a given domain, instead of applying the whole set. This results in a performance improvement. Another advantage deals with the matching quality. Finally, MatchPlanner is enhanced with selftuning capability since the learned decision trees focus either on the quality, performance or a tradeoff between these two aspects. As future work, we plan to enrich the KB by adding more schemas and expert mappings, resulting in a more robust tool. We also intend to enrich the discovered matches with their relationship or to focus on complex mappings by relying on the reliable measures.

References

1. D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with coma++. In *SIGMOD Conference, Demo paper*, pages 906–908, 2005.
2. P. Avesani, F. Giunchiglia, and M. Yatskevich. A large scale taxonomy mapping evaluation. In *International Semantic Web Conference*, pages 67–81, 2005.
3. J. Berlin and A. Motro. Database schema matching using machine learning with feature selection. In *CAiSE*, 2002.
4. J. Bradford, C. Kunz, R. Kohavi, C. Brunk, and C. Brodley. Pruning decision trees with misclassification costs. In *Proceedings of ECML*, pages 131–136, 1998.
5. A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *SIGMOD*, pages 509–520, 2001.
6. A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Y. Halevy. Learning to match ontologies on the semantic web. *VLDB J.*, 12(4):303–319, 2003.
7. A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies, International Handbooks on IS*, 2004.
8. C. Drumm, M. Schmitt, H. H. Do, and E. Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM*, pages 107–116. ACM, 2007.
9. F. Duchateau, Z. Bellahsene, and M. Roche. A context-based measure for discovering approximate semantic matching between schema elements. In *RCIS*, 2007.
10. M. Ehrig, S. Staab, and Y. Sure. Bootstrapping ontology alignment methods with apfel. In *ISWC*, 2005.
11. D. W. Embley, L. Xu, and Y. Ding. Automatic direct and indirect schema mapping: Experiences and lessons learned. *SIGMOD Record journal*, 33(4):14–19, 2004.
12. J. Euzenat and P. Valtchev. Similarity-based ontology alignment in owl-lite. In *ECAI*, pages 333–337, 2004.
13. D. Fudenberg and J. Tirole. *Game Theory*. MIT Press, 1991.
14. A. Gal. The generation y of xml schema matching (panel description). In *XSym*, pages 137–139, 2007.
15. J. Hammer, M. Stonebraker, , and O. Topsakal. Thalia: Test harness for the assessment of legacy information integration approaches. In *Proceedings of ICDE*, pages 485–486, 2005.
16. Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal. etuner: tuning schema matching software using synthetic scenarios. *VLDB J.*, 16(1):97–122, 2007.
17. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.
18. S. Melnik, H. G. Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering*, pages 117–128, 2002.
19. T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, pages 122–133, 1998.
20. P. Mitra, G. Wiederhold, and M. L. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *EDBT*, pages 86–100, 2000.
21. N. F. Noy and M. A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI/IAAI*, pages 450–455, 2000.
22. J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1987.
23. J. R. Quinlan. Improved use of continuous attributes in c4.5. In *Journal of Artificial Intelligence Research*, volume 4, pages 77–90, 1996.
24. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.
25. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *J. Data Semantics IV*, pages 146–171, 2005.