Discovering Missing Background Knowledge in Ontology Matching

Fausto Giunchiglia and **Pavel Shvaiko** and **Mikalai Yatskevich** [1]

Abstract. *Semantic matching* determines the mappings between the nodes of two graphs (e.g., ontologies) by computing *logical relations* (e.g., subsumption) holding among the nodes that correspond semantically to each other. We present an approach to deal with the lack of background knowledge in matching tasks by using semantic matching *iteratively*. Unlike previous approaches, where the missing axioms are manually declared before the matching starts, we propose a fully automated solution. The benefits of our approach are: (i) saving some of the pre-match efforts, (ii) improving the quality of match via iterations, and (iii) enabling the future reuse of the newly discovered knowledge. We evaluate the implemented system on large real-world test cases, thus, proving empirically the benefits of our approach.

1 INTRODUCTION

Match is a critical operator in many applications, e.g., AI, Semantic Web, WWW, e-commerce. It takes two graph-like structures, e.g., lightweight ontologies, such as Google and Looksmart[2], or business catalogs, such as UNSPSC and eCl@ss[3], and produces a mapping between the nodes that correspond semantically to each other.

Many various solutions of match have been proposed so far, see for recent surveys [17, 3, 13, 9, 16][4]. We focus on a schema-based solution, namely a matching system exploiting only the schema information, thus not considering instances. We follow the so-called *semantic matching* approach [6]. This approach is based on two key ideas. The first is that mappings are calculated between ontology entities by computing *logical relations* (e.g., equivalence, subsumption), instead of computing coefficients rating match quality in the [0,1] range, as it is the case in the other approaches, e.g., [14, 5, 15]. The second idea is that the relations are determined by analyzing the *meaning* which is codified in the elements and the structures of ontologies. In particular, labels at nodes, written in natural language, are translated into propositional formulas which explicitly codify the labels' intended meaning. This allows the translation of the matching problem into a propositional unsatisfiability problem, which can then be efficiently resolved using (sound and complete) state of the art propositional satisfiability (SAT) deciders, e.g., [2].

Recent industrial-strength evaluations of matching systems, see, e.g., [4, 1], show that lack of background knowledge, most often domain specific knowledge, is one of the key problems of matching systems these days. In fact, most state of the art systems, for the tasks of matching thousands of nodes show low values of *recall* ($\sim$30%), while with *toy* examples, the recall they demonstrated was most often around 90%. This paper addresses the problem of the missing background knowledge by using semantic matching iteratively. The contributions of the paper are: (i) the new automatic *iterative* semantic matching algorithm, which provides such benefits as a better quality of match (recall), saving some of the pre-match efforts, enabling the future reuse of the newly discovered knowledge; (ii) the *quality evaluation* of the implemented system on large real-world test cases.

The rest of the paper is organized as follows. The semantic matching algorithm is briefly summarized in Section 2. Section 3 introduces the problem of the lack of background knowledge in matching and its possible solutions. Section 4 presents the iterative semantic matching algorithm and its details. Section 5 discusses experiments with matching lightweight ontologies. Section 6 reports some conclusions and outlines the future work.

2 SEMANTIC MATCHING

We focus on tree-like structures (e.g., Google, Looksmart, Yahoo!). *Concept of a label* is the propositional formula which stands for the set of documents that one would classify under a label it encodes. *Concept at a node* is the propositional formula which represents the set of documents which one would classify under a node, given that it has a certain label and that it is in a certain position in a tree.

The following relations can be discovered among the concepts at nodes of two ontologies: *equivalence* (=); *more/less general* ($\sqsupseteq$, $\sqsubseteq$); *disjointness* ($\perp$). When none of the relations holds, the special *idk* (I don't know) relation is returned. The relations are ordered according to decreasing binding strength, i.e., from the strongest (=) to the weakest (*idk*). *Semantic matching* is defined as follows: given two trees T1, T2 compute the N1 $\times$ N2 *mapping elements*, $\langle ID_{i,j}, C1_i, C2_j, R' \rangle$, where ID_{ij} is a unique identifier of the given mapping element; $C1_i \in$ T1; i=1,...,N1; $C2_j \in$ T2; j=1,...,N2; R' is the strongest relation holding between the concepts at nodes $C1_i, C2_j$.

Let us summarize the semantic matching algorithm via a running example. We consider ontologies O1 and O2 shown in Figure 1, which are small parts of Google and Looksmart. The algorithm inputs two ontologies and outputs a set of mapping elements in four macro steps. The first two steps represent the pre-processing phase, while the third and the fourth steps are the element level and structure level matching respectively.

Step 1. For all labels *L* in two trees, compute *concepts of labels*. *Labels* at nodes are viewed as concise descriptions of the documents that are stored under the nodes. The meaning of a label at a node is computed by inputting a label, by analyzing its real-world semantics, and by outputting a concept of the label, C_L. For example, by writing $C_{Hobbies\_and\_Interests}$ we move from the natural language ambiguous label $Hobbies\_and\_Interests$ to the concept $C_{Hobbies\_and\_Interests}$, which codifies explicitly its intended meaning, namely the documents which are about hobbies and interests. Technically, based on WordNet (WN) [12] *senses*, concepts of labels are codified as propositional logical formulas, see [10] for details.

From now on, it is assumed that the propositional formula encoding the concept of label is the label itself. Numbers "1" and "2" are used as subscripts to distinguish between trees in which the given concept of label occurs, e.g., TOP_1 (belongs to O1) and TOP_2 (belongs to O2).

[1] Department of Information and Communication Technology, University of Trento, Povo, Trento, Italy. email: {fausto, pavel, yatskevi }@dit.unitn.it

[2] See, http://www.google.com/Top/ and http://www.looksmart.com/

[3] See, http://www.unspsc.org/ and http://www.eclass.de/

[4] See, www.OntologyMatching.org for a complete information on the topic.

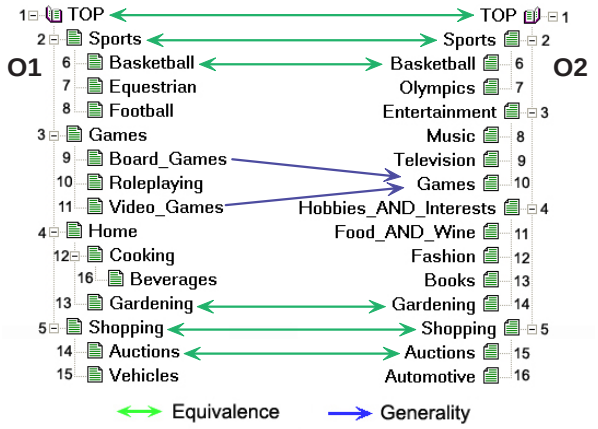

Figure 1. Parts of Google and Looksmart and some of the mappings

Step 2. For all nodes *N* in two trees, compute *concepts at nodes*. We analyze the meaning of the *positions* of labels at nodes in a tree. By doing this concepts of labels are extended to concepts at nodes, C_N. This is required to capture the knowledge residing in the structure of a tree, namely the context in which the given concept at label occurs. Technically, concepts at nodes are written in the same propositional logical language as concepts of labels. For example, $C2_8 = TOP_2 \sqcap Entertainment_2 \sqcap Music_2$ stands for the concept describing all the documents about a particular kind of entertainment which is music.

Step 3. For all pairs of labels in two trees, compute *relations* among *concepts of labels*. Relations between concepts of labels are computed by using a library of element level matchers, see Table 1. The first column contains the names of the matchers. The second column lists the order in which they are executed. The third column introduces the matchers' approximation level. The relations produced by a matcher with the first approximation level are always correct. For example, $Beverages_1$ can be found less general than $Food_2$. In fact, according to WordNet, *beverages* is hyponym (subordinate word) to *food*. Notice, in WordNet *beverages* has 1 sense, while *food* has 3 senses. Some sense filtering techniques are used to discard the irrelevant senses for the given context, see [10] for details. The relations produced by a matcher with the second approximation level are likely to be correct (e.g., *net* = *network*, but *hot* = *hotel* by *Prefix*). The *WordNet* matcher has two WordNet senses in input and computes equivalence, generality, and disjointness relations. *String-based* matchers have two labels as input. These compute only equivalence relations (e.g., equivalence holds if the weighted *distance* between the input strings is lower than a threshold), see [7]. String-based matchers are used iff *WordNet* fails to find a relation. The result of step 3 is a matrix of relations holding between *atomic* concepts of labels. A part of it, for the example of Figure 1, is shown in Table 2.

Table 1. Element level semantic matchers (Part 1)

Matcher name	Execution order	Approximation level	Matcher type	Schema info
WordNet	1	1	Sense-based	WordNet senses
Prefix	2	2	String-based	Labels
Suffix	3	2	String-based	Labels
Edit distance	4	2	String-based	Labels
Ngram	5	2	String-based	Labels

Table 2. cLabsMatrix: relations holding among atomic concepts of labels

	$Games_1$	$Board\_Games_1$	$Beverages_1$
$Games_2$	$=$	$\sqsupseteq$	idk
$Food_2$	idk	idk	$\sqsupseteq$
$Wine_2$	idk	idk	$\sqsubseteq$

Step 4. For all pairs of nodes in two trees, compute *relations* among *concepts at nodes*. The tree matching problem is reformulated into a set of node matching problems, see Algorithm 1.

Algorithm 1 The tree matching algorithm

```
1: Node: struct of
2:         int nodeId;
3:         String label;
4:         String cLabel;
5:         String cNode;
6:         Node parent;
7:         AtomicConceptOfLabel[ ] ACOL;
8: AtomicConceptOfLabel: struct of
9:                                 int id;
10:                                String token;
11:                                String[ ] wnSenses;
12: String[ ][ ] cLabsMatrix, cNodesMatrix;

30: String[ ][ ] treeMatch(Tree of Nodes source, target)
31: Node sourceNode, targetNode;
32: int i, j;
33: String[ ][ ] relMatrix;
34: String axioms, context1, context2;
35: cLabsMatrix = fillCLabMatrix(source, target);

50: for each sourceNode ∈ source do
51:    i = getNodeId(sourceNode);
52:    context1 = getCnodeFormula(sourceNode);
53:    for each targetNode ∈ target do
54:       j = getNodeId(targetNode);
55:       context2 = getCnodeFormula(targetNode);

80:       relMatrix = extractRelMatrix(cLabsMatrix, sourceNode, targetNode);
81:       axioms = mkAxioms(relMatrix);
82:       cNodesMatrix[i][j] = nodeMatch(axioms, context1, context2);

110:   end for
111: end for
150: return cNodesMatrix;
```

In line 30, the treeMatch function inputs two trees of Nodes (source and target). Two loops are run over all the nodes of source and target trees in lines 50-111 and 53-110 in order to formulate all the node matching problems. Then, for each node matching problem, a pair of propositional formulas encoding concepts at nodes and relevant relations holding between concepts of labels are taken by using the getCnodeFormula and extractRelMatrix functions respectively. The former are memorized as context$_1$ and context$_2$ in lines 52 and 55. The latter are memorized in relMatrix in line 80. In order to reason about relations between concepts at nodes, the premises (axioms) are built in line 81. These are a conjunction of atomic concepts of labels which are related in relMatrix. Finally, in line 82, the relations holding between the concepts at nodes are calculated by nodeMatch and are reported in line 150 (cNodesMatrix). A part of this matrix for the example of Figure 1 is shown in Table 3.

Table 3. cNodesMatrix: relations holding among concepts at nodes[4]

	$C1_3$	$C1_4$	$C1_5$	$C1_9$	$C1_{10}$	$C1_{11}$
$C2_{10}$	$=$	idk	idk	$\sqsupseteq$	$=$	$\sqsupseteq$

nodeMatch translates each node matching problem into a propositional validity problem. Thus, we have to prove that $axioms \longrightarrow rel(context_1, context_2)$ is valid. *axioms*, $context_1$, and $context_2$ are as defined in the tree matching algorithm. *rel* is the logical relation that we want to prove holding between $context_1$ and $context_2$. nodeMatch checks for sentence validity by proving that its negation is unsatisfiable. Thus, the algorithm uses, depending on a matching task, either ad hoc reasoning techniques, see [8], or standard DPLL-based SAT solvers, e.g., [2]. From the example in Figure 1, trying to prove that $C1_9$ is less general than $C2_{10}$, requires constructing the following formula: $((TOP_1 \leftrightarrow TOP_2) \wedge (Games_1 \leftrightarrow Games_2) \wedge (Games_1 \leftrightarrow Entertainment_2)^5 \wedge (Board\_Games_1 \rightarrow Games_2)) \rightarrow ((TOP_1 \wedge Games_1 \wedge Board\_Games_1) \rightarrow (TOP_2 \wedge Entertainment_2 \wedge Games_2))$. As it turns out, this formula is unsatisfiable, hence, the less generality holds.

[5] Notice, by applying element level matchers of Table 1 we can only determine the *idk* relation between *games* and *entertainment*. For example, in WordNet there is no direct lexical relation between *games* and *entertainment*. However, to simplify the presentation, we assume that it has been already determined that $Games_1 = Entertainment_2$. See Section 4 for the details of how the equivalence between the given concepts can be discovered.

3 LACK OF KNOWLEDGE

Recent industrial-strength evaluations of matching systems, see, e.g., [4, 1], show that lack of background knowledge, most often domain specific knowledge, is one of the key problems of matching systems these days. In fact, for example, should PO match Post Office, Purchase Order, or Project Officer? At present, most state of the art systems, for the tasks of matching thousands of nodes, perform not with such high values of *recall* (∼30%) as in cases of toy examples, where the recall was most often around 90%. Also, contributing to this problem, [11] shows that complex matching solutions (requiring months of algorithms design and development) on big tasks may perform as badly as a baseline matcher (requiring one hour burden).

In order to understand better the above observations, let us consider a preliminary analytical comparative evaluation of some state of the art matching systems together with a baseline solution[6] on three large real-world test cases. Table 4 provides some indicators of the test cases complexity.

Table 4. Some indicators of the complexity of the test cases

	#nodes	max depth	#labels per tree
Google vs. Looksmart	706/1081	11/16	1048/1715
Google vs. Yahoo	561/665	11/11	722/945
Yahoo vs. Looksmart	74/140	8/10	101/222

These test cases were taken from the OAEI-2005 ontology matching contest[7]. As match quality measures we focus here on recall which is a completeness measure. It varies in the [0,1] range; the higher the value, the smaller is the set of correct mappings (true positives) which have not been found. The summarized evaluation results for all the three matching tasks are shown in Figure 2. Notice, the results for such matching systems as OMAP, CMS, Dublin20, Falcon, FOAM, OLA, and ctxMatch2, were taken from OAEI-2005, see [4], while evaluation results for the baseline matcher and S-Match were taken from [1]. As Figure 2 shows, none of the considered matching systems performs with a value of recall which is higher than 32%.

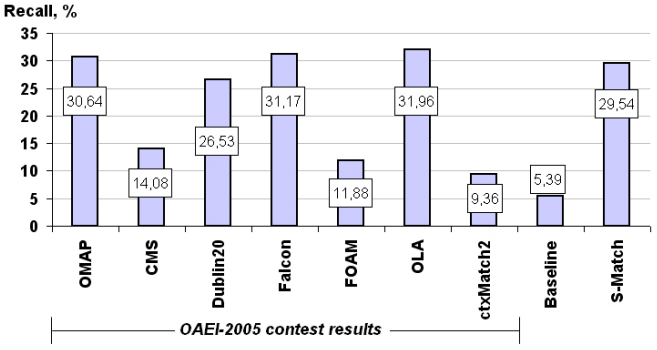

Figure 2. Analytical comparative evaluation

There are multiple strategies to attack the problem of the lack of background knowledge. One of the most often used methods so far is to declare the missing axioms manually as a pre-match effort. Some other plausible strategies include (i) extending stop word lists, (ii) expanding acronyms, (iii) reusing the previous match results, (iv) querying the web, (v) using (if available) domain specific sources of knowledge, and so on.

In this paper, we propose a fully automated solution to address the problem of the lack of knowledge by using semantic matching iteratively. The idea is to repeat *Step 3* and *Step 4* of the matching algorithm for some critical (hard) matching tasks. In particular, the result of SAT is analyzed. We identify critical points in the matching process, namely mapping elements with the *idk* relation where a stronger relation (e.g., generality) should have taken place. We attack critical points by exploiting sophisticated element level matchers which use the *deep* information encoded in WordNet, e.g., its internal structure. Then, taking into account the newly discovered information as additional axioms, we re-run SAT solver on a critical task. Finally, if SAT returns false, we save the newly discovered knowledge, thereby enabling its future reuse.

[6] This matcher does simple string comparison among sets of labels on the paths from nodes under consideration to the roots of the input trees, see [1].

[7] See for details, http://oaei.ontologymatching.org/2005/

4 ITERATIVE SEMANTIC MATCHING

We first discuss how the tree matching algorithm should be modified in order to suitably enable iterations. Then, we present the main building blocks of the iterative tree matching algorithm, namely, algorithms for critical points discovery and critical points resolution.

4.1 Iterative Tree Matching Algorithm

The iterative tree matching algorithm is shown as Algorithm 2. The numbers on the left indicate where the new code must be positioned in Algorithm 1.

Algorithm 2 A vanilla iterative tree matching algorithm

```
13: Boolean[ ][ ] cPointsMatrix;

100:  if (cPointsDiscovery(sourceNode, targetNode) == true) then
101:    cPointsMatrix[i][j] = true;
102:    ResolveCpoint(sourceNode, targetNode, context_1, context_2);
103:  end if
```

In line 13, we introduce cPointsMatrix which memorizes critical points. Semantic matching algorithm works in a top-down manner, and hence, mismatches among the top level classes of ontologies imply further mismatches between their descendants. Thus, the descendants should be processed only after the critical point at those top level nodes has been resolved. This is ensured by suitably positioning the new functions (enabling iterations) in a double loop of Algorithm 1. Hence, in line 100, we check with the help of cPointsDiscovery function if the nodes under consideration are the critical point. If they indeed represent the critical point, they are (memorized and) resolved by using the ResolveCpoint function (line 102). In the example of Figure 1, critical points which are determined are, for example, $\langle C1_2, C2_3\rangle$, $\langle C1_3, C2_3\rangle$, $\langle C1_4, C2_4\rangle$.

An updated cNodesMatrix, after running the iterative tree matching algorithm, is presented in Table 5. Comparing it with the non-iterative matching algorithm result, which is further reported in Table 7, we can see that having identified and resolved the $\langle C1_3, C2_3\rangle$ critical point, we also managed to discover the new correspondences, namely between $C2_3$ and $C1_9$, $C1_{10}$, $C1_{11}$.

Table 5. Recomputed cNodesMatrix: relations among concepts at nodes

	$C1_1$	$C1_2$	$C1_3$	$C1_4$	$C1_5$	$C1_9$	$C1_{10}$	$C1_{11}$
$C2_1$	$=$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$
$C2_3$	$\sqsubseteq$	$=$	$=$	idk	idk	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$

Having computed all the mapping elements for a given pair of ontologies, the identified critical relations are validated by a human user. In particular, user decides if the type of relation determined automatically is appropriate for the given pair of ontologies (e.g., is it appropriate that $Games_1 \leftrightarrow Entertainment_2$ or a weaker relation, namely $Games_1 \rightarrow Entertainment_2$, should have taken place ?), (s)he decides either to use this relation once (only for this pair of ontologies) or to save it in a domain specific oracle in order to enable its future reuse.

Finally, it is worth noting that iterative semantic matching algorithm amounts to *robustness* of the semantic matching. In fact, even if non-iterative semantic matching determines a (false) top level mismatch, this can be discovered and resolved by applying Algorithm 2. Thus, avoiding a further propagation of possible mismatches between the descendants of the initially mismatched top level nodes.

4.2 The Critical Points Discovery Algorithm

The algorithm for discovering critical points is based on the following intuitions:

- each *idk* relation in cNodesMatrix is potentially a critical point, but it is not always the case;
- since critical points arise due to lack of background knowledge, the clue is to check whether some other nodes located below the critical nodes (those representing a critical point) are related somehow. In case of a positive result the actual nodes are indeed the critical point; they represent a false alarm otherwise.

Algorithm 3 Critical points discovery algorithm

```
1: Boolean cPointsDiscovery(Node sourceNode, targetNode)
2: Node[ ] sDescendant, tDescendant;
3: ACOL sACOL, tACOL;
4: if (cNodesMatrix[sourceNode.nodeID][targetNode.nodeID]=="idk")
   then
5:     sDescendant = getSubTree(sourceNode);
6:     tDescendant = getSubTree(targetNode);
7:     for each sACOL ∈ sDescendant.ACOL do
8:         for each tACOL ∈ tDescendant.ACOL do
9:             if cLabsMatrix[sACOL.id][tACOL.id] != "idk" then
10:                return true;
11:            end if
12:        end for
13:    end for
14: else
15:    return false;
16: end if
```

Algorithm 3 formalizes these intuitions. In particular, the first condition mentioned above is checked in line 4. Verifying the second condition is more complicated. We call a relation holding between descendants of the potentially critical nodes a *support relation*. The support relation holds if there exists atomic concept of label (sACOL) in the descendants of sourceNode which is related in cLabsMatrix (by any semantic relation, except *idk*) to any atomic concept of label (tACOL) in the descendants of targetNode. This condition is checked in a double loop in lines 7-13. Finally, if both conditions are satisfied, the cPointsDiscovery function concludes that the nodes under consideration are the critical point (line 10). Under the given critical points discovery strategy, performing such a look up over the cLabsMatrix makes sense, obviously, only when sourceNode and targetNode are non-leaf nodes.

For example, suppose we want to match $C1_3$ and $C2_3$. Parts of cLabsMatrix (notice, the relations in this matrix were computed by applying element level matchers of Table 1) and cNodesMatrix with respect to the given matching task are shown in Table 6 and Table 7.

Table 6. cLabsMatrix: relations holding among atomic concepts of labels

	TOP_1	$Games_1$	$Board\_Games_1$
TOP_2	$=$	idk	idk
$Entertainment_2$	idk	idk	idk
$Games_2$	idk	$=$	$\sqsupseteq$

Table 7. cNodesMatrix: relations holding among concepts of nodes

	$C1_1$	$C1_2$	$C1_3$	$C1_4$	$C1_5$	$C1_9$	$C1_{10}$	$C1_{11}$
$C2_1$	$=$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$
$C2_3$	$\sqsubseteq$	idk	idk	idk	idk	idk	idk	idk

In cNodesMatrix (Table 7) the relation between $C1_3$ and $C2_3$ is *idk*. In cLabsMatrix (Table 6) there is a support relation for the given matching problem, e.g., $Board\_Games_1 \sqsubseteq Games_2$. Therefore, relation between $C1_3$ and $C2_3$ represents the critical point and we should reconsider the relation holding between $Games_1$ and $Entertainment_2$ in cLabsMatrix.

Finally, it is worth noting that this algorithm also properly handles nodes, which are indeed dissimilar, e.g., $C1_5$ and $C2_2$ are determined not to be the critical point.

4.3 The Critical Points Resolution Algorithm

Let us discuss how the critical points are resolved, see Algorithm 4.

Algorithm 4 Critical points resolution algorithm

```
1: ResolveCpoint(Node sourceNode, targetNode, String context_1, context_2)
2: String cRel;
3: String[ ] ExecutionList;
4: ACOL sACOL, tACOL;
5: for each sACOL ∈ sourceNode.ACOL do
6:     for each tACOL ∈ targetNode.ACOL do
7:         cRel = GetMLibRel(ExecutionList, sACOL.wnSenses, tACOL.wnSenses);
8:         cLabsMatrix[sACOL.id][tACOL.id] = cRel;
9:     end for
10: end for
11: cNodesMatrixUpdate(sourceNode, targetNode, context_1, context_2);
```

The ResolveCpoint function determines relations (cRel) for the critical points. Also, by exploiting the cNodesMatrixUpdate procedure, it updates accordingly cNodesMatrix. In particular, ResolveCpoint executes sophisticated element level matchers, see Table 8, over the atomic concepts of labels by using the GetMLibRel function (line 7). Matchers are applied following the order (ExecutionList) given in the second column of Table 8. These matchers have the third approximation level, which means that the relations they produce depend heavily on the context of the matching task. Also, execution times of them are much longer than of those of Table 1. Thus, they can not be applied in all the cases.

Table 8. Element level semantic matchers (Part 2)

Matcher name	Exec. order	Approx. level	Matcher type	Schema info
Hierarchy Distance	1	3	Sense-based	WN senses
WN Gloss	2	3	Gloss-based	WN senses
Extended WN Gloss	3	3	Gloss-based	WN senses
Gloss Comparison	4	3	Gloss-based	WN senses
Extended Gloss Comparison	5	3	Gloss-based	WN senses

For example, a *Hierarchy Distance (HD)* matcher computes the equivalence relation if the distance between two input senses in the WordNet hierarchy is less than a given threshold value (e.g., 3) and returns *idk* otherwise. According to WordNet, *games* and *entertainment* have a common ancestor, which is *diversion*. The distance between these concepts is 2 (1 more general link and 1 less general). Therefore, the *HD* matcher concludes that $Games_1$ is equivalent to $Entertainment_2$. If the *HD* matcher fails, which is not the case in our example, we apply a set of *gloss-based* matchers. These also have two WordNet senses in input and exploit techniques based on comparison of textual definitions (*glosses*) of the words whose senses are taken in input. They compute, depending on a particular matcher, the equivalence, more/less general relations. Due to lack of space, we give here only hints on how some gloss-based matchers work. For example, *WN Gloss (WNG)* counts the number of occurrences of the label of the source input sense in the gloss of the target input sense. If this number is lower than (equal to) a threshold, the less generality (due to a common pattern of defining terms in glosses through a more general term) is returned. *Gloss Comparison (GC)* counts the number of the same words in the glosses of the source and target input senses. If this number (of shared words) exceeds a threshold, the equivalence is returned. *Extended* gloss matchers are build in a straightforward way, by also considering glosses of the parent (children) nodes of the input senses in the WordNet *is-a (part-of)* hierarchy, see for details [7].

In line 8, we update cLabsMatrix with the critical relation, cRel, such that in all the further computations and for the current pair of nodes this relation is available. Finally, given the new axiom ($Games_1 \leftrightarrow Entertainment_2$) we recompute (line 11)[8], by rerunning SAT, the relation holding between the pair of critical nodes, thus determining that $C1_3 = C2_3$.

[8] cNodesMatrixUpdate performs functionalities identical to those of lines 80-82 in Algorithm 1.

5 EVALUATION

In this section we present the quality evaluation of the iterative semantic matching algorithm. Due to lack of space we report here only what we have realized to be the most important findings.

Evaluation set-up. In our evaluation we have used three large real-world test cases, which were introduced in Table 4. As expert mappings for these test cases we used 2265 mappings acquired in [1]. By construction those expert mappings represent only true positives, thereby allowing us to estimate only the recall with them. To the best of our knowledge, at the moment, there are no large datasets as, for example, that one of Table 4, where available expert mappings allow measuring both precision[9] and recall. Thus, in the following we focus mostly on analyzing the recall.

Two further observations. First is that higher values of recall can be obtained at the expense (lower values) of precision. Thus, in order to ensure a *fair* recall evaluation, before running tests on the matching tasks of Table 4, we have analyzed behavior of the iterative semantic matching on a number of test cases, e.g., course university catalogs[10], where expert mappings allowed measuring both precision and recall. Matchers decreasing precision substantially in these tests were discarded from the further evaluation. In fact, for this reason we exclude from the further considerations the *Extended Gloss Comparison* matcher. The second observation is that using matchers of Table 8 exhaustively for all the tasks, hence, omitting the critical points discovery algorithm, also leads to a significant precision decrease, thus justifying usefulness of the cPointsDiscovery algorithm.

Evaluation results. The summarized evaluation results for all the three matching tasks of Table 4 are shown in Figure 3. In particular, it demonstrates contributions to the recall of matchers of Table 8 as well as of their combinations. The *Extended WN Gloss* matcher performed very poorly, i.e., contributing less than 1% to the recall, hence, we do not report its results in Figure 3. By using a combination of the *HD*, *WNG*, *GC* matchers we have improved S-Match recall results (29,5%) up to 46,1% within the iterative S-Match[11].

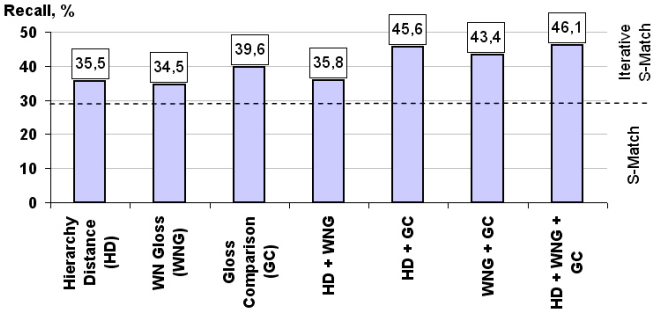

Figure 3. Evaluation results (absolute values)

Let us now consider *relative* characteristics of the iterative S-Match with respect to the non-iterative version, see the first row of Table 9 for a summary. The highest recall increase by using only a single matcher of Table 8 within the iterative S-Match was achieved by the *GC* matcher, namely 34% over the non-iterative S-Match. The best, in this sense, combination of two matchers is being that of the *HD* and *GC* matchers: recall increased by 54%. Finally, a combination of the *HD*, *WNG* and *GC* matchers resulted in the 56% recall increase with respect to the non-iterative S-Match.

Table 9 also reports values of thresholds used within the evaluation. These values were obtained based on the rationale behind designing matchers of Table 8 and their evaluation results.

[9] Precision is a correctness measure; the higher the value, the smaller is the set of false positives which has been computed, see for details, e.g.,[4].

[10] See, http://dit.unitn.it/∼accord/Experimentaldesign.html for a repository of test cases, expert mappings we have used.

[11] Note, this result should be considered as a complimentary one to the results of S-Match++ reported in [1], since they address separate problem spaces.

Table 9. Some element level matchers and their evaluation results

	HD	*GC*	*HD + GC*	*HD + WNG + GC*
Recall increase (relative), %	20	34	54	56
Threshold value	4	2	4 \ 2	4 \ 1 \ 2

Evaluation summary. The evaluation we have conducted shows that the problem of the lack of background knowledge is a hard one. In fact, as it turns out, not all the designed element level matchers can perform always well in real-world applications, as it might (mistakenly) seem from the toy evaluations. Also, new matchers are still needed, since, for example, we could discover that $\langle C1_4, C2_4 \rangle$ is the critical point, however, we were unable to resolve it with the matchers of Table 8, namely to match $Home_1$ and $Hobbies\_AND\_Interests_2$.

6 CONCLUSIONS

We have presented an automated approach to attack the problem of the lack of background knowledge by applying semantic matching iteratively. We implemented the proposed approach and evaluated it on large real-world (lightweight) ontology matching tasks with encouraging results. Future work proceeds at least in the following directions: (i) adapting the iterative semantic matching algorithm for handling leaf-node critical points, (ii) design and development of the new element level matchers, (iii) involving user within the matching process, where his/her input is maximally useful, (iv) conducting further large and extensive real-world evaluations.

REFERENCES

[1] P. Avesani, F. Giunchiglia, and M. Yatskevich, 'A large scale taxonomy mapping evaluation', in *Proceedings of ISWC*, pp. 67 – 81, (2005).

[2] D. Le Berre. A satisfiability library for Java. http://www.sat4j.org/.

[3] A. Doan and A. Halevy, 'Semantic integration research in the database community: A brief survey', *AI Magazine*, (2005).

[4] J. Euzenat, H. Stuckenschmidt, and M. Yatskevich, 'Introduction to the ontology alignment evaluation 2005', in *Proceedings of Integrating Ontologies workshop at K-CAP*, (2005).

[5] J. Euzenat and P. Valtchev, 'Similarity-based ontology alignment in OWL-lite', in *Proceedings of ECAI*, pp. 333–337, (2004).

[6] F. Giunchiglia and P. Shvaiko, 'Semantic matching', *Knowledge Engineering Review Journal*, (18(3)), 265–280, (2003).

[7] F. Giunchiglia and M. Yatskevich, 'Element level semantic matching', in *Proceedings of the Meaning Coordination and Negotiation workshop at ISWC*, (2004).

[8] F. Giunchiglia, M. Yatskevich, and E. Giunchiglia, 'Efficient semantic matching', in *Proceedings of ESWC*, pp. 272–289, (2005).

[9] Y. Kalfoglou and M. Schorlemmer, 'Ontology mapping: the state of the art', *Knowledge Engineering Review Journal*, (18(1)), 1–31, (2003).

[10] B. Magnini, L. Serafini, and M. Speranza, 'Making explicit the semantics hidden in schema models', in *Proceedings of the workshop on Human Language Technology for the Semantic Web and Web Services at ISWC*, (2003).

[11] B. Magnini, M. Speranza, and C. Girardi, 'A semantic-based approach to interoperability of classification hierarchies: Evaluation of linguistic techniques', in *Proceedings of COLING*, (2004).

[12] A.G. Miller, 'WordNet: A lexical database for English', *Communications of the ACM*, (38(11)), 39–41, (1995).

[13] N. Noy, 'Semantic Integration: A survey of ontology-based approaches', *SIGMOD Record*, **33**(4), 65–70, (2004).

[14] N. Noy and M. Musen, 'The PROMPT Suite: Interactive tools for ontology merging and mapping', *International Journal of Human-Computer Studies*, (59(6)), 983–1024, (2003).

[15] L. Palopoli, G. Terracina, and D. Ursino, 'DIKE: a system supporting the semi-automatic construction of cooperative information systems from heterogeneous databases', *SPE*, **33**(9), 847–884, (2003).

[16] E. Rahm and P. Bernstein, 'A survey of approaches to automatic schema matching', *VLDB Journal*, (10(4)), 334–350, (2001).

[17] P. Shvaiko and J. Euzenat, 'A survey of schema-based matching approaches', *Journal on Data Semantics*, (IV), 146–171, (2005).