Discovering Missing Background Knowledge in Ontology Matching

Fausto Giunchiglia and Pavel Shvaiko and Mikalai Yatskevich

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\textsuperscript{2}, or business catalogs, such as UNSPSC and eClass\textsuperscript{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]\textsuperscript{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 (∼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 (⊇, ⊆); disjointness (⊥). 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 × N2 mapping elements, \( \langle ID_1, C_1, C_2, R \rangle \), where \( ID_1 \) is a unique identifier of the given mapping element; \( C_1 \in T_1; i=1,\ldots,N1; C_2 \in T_2; j=1,\ldots,N2; R \) is the strongest relation holding between the concepts at nodes \( C_1, C_2 \).

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, \( L \). For example, by writing \( \text{Hobbies and interests} \) we move from the natural language ambiguous label \( \text{Hobbies and interests} \) to the concept \( \text{Hobbies and interests} \), 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., \( \text{TOP}_1 \) (belongs to O1) and \( \text{TOP}_2 \) (belongs to O2).
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, CN. 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 pros- positional logical language as concepts of labels. For example, C = TOP2 ∩ Entertainment2 ∩ Music2 stands for the concept de- scribing 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 col-

Table 1. Element level semantic matchers (Part 1)

<table>
<thead>
<tr>
<th>Matcher name</th>
<th>Execution order</th>
<th>Approximation level</th>
<th>Matcher type</th>
<th>Schema info</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordNet</td>
<td>1</td>
<td>1</td>
<td>Sense-based</td>
<td>WordNet senses</td>
</tr>
<tr>
<td>Prefix</td>
<td>2</td>
<td>2</td>
<td>String-based</td>
<td>Labels</td>
</tr>
<tr>
<td>Suffixes</td>
<td>3</td>
<td>2</td>
<td>String-based</td>
<td>Labels</td>
</tr>
<tr>
<td>Edit distance</td>
<td>4</td>
<td>2</td>
<td>String-based</td>
<td>Labels</td>
</tr>
<tr>
<td>Ngram</td>
<td>5</td>
<td>2</td>
<td>String-based</td>
<td>Labels</td>
</tr>
</tbody>
</table>

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

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 getGnodeFormula and extractRelMatrix functions respectiv- ely. The former are memorized as context1 and context2 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 con- cepts 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

<table>
<thead>
<tr>
<th>nodeMatch</th>
<th>C13</th>
<th>C14</th>
<th>C15</th>
<th>C16</th>
<th>C110</th>
<th>C111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Games2</td>
<td>sdk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Food2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Winces1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

deNodeMatch translates each node matching problem into a propositional validity problem. Thus, we have to prove that axioms ➔ rel(context1, context2) is valid. axioms, context1, and context2 are as defined in the tree matching algorithm. rel is the logical rela-
tion that we want to prove holding between context1 and context2. 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- 

Algorithm 1 The tree matching algorithm

1: Node: struct of
2: int nodeId;
3: String label;
4: String cLabel;
5: String eNode;
6: Node parent;
7: AtomicConceptOfTypeLabel | ACOL;
8: AtomicConceptOfTypeLabel: struct of
9: int id;
10: String token;
11: String[ ] wSenses;
12: String[ ] relMatrix;
13: String axioms, context1, context2;
14: cLabsMatrix = fillCLabelMatrix(source, target);
15: for each sourceNode ∈ source do
16: i = getNodeId(sourceNode);
17: context1 = getGnodeFormula(sourceNode);
18: for each targetNode ∈ target do
19: j = getNodeId(targetNode);
20: context2 = getGnodeFormula(targetNode);
21: relMatrix = extractRelMatrix(cLabsMatrix, sourceNode, targetNode);
22: axioms = mkAxioms(relMatrix);
23: cNodesMatrix[i][j] = nodeMatch(axioms, context1, context2);
24: end for
25: end for
26: return cNodesMatrix;

Notice, by applying element level matchers of Table 1 we can only deter-
mine 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 Games2 = Entertainment2. 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$^7$ on three large real-world test cases. Table 4 provides some indicators of the test cases complexity.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>#nodes</th>
<th>max depth</th>
<th>#labels per tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google vs. Looksmart</td>
<td>706/1081</td>
<td>11/16</td>
<td>1048/7175</td>
</tr>
<tr>
<td>Google vs. Yahoo</td>
<td>561/665</td>
<td>11/11</td>
<td>722/945</td>
</tr>
<tr>
<td>Yahoo vs. Looksmart</td>
<td>174/140</td>
<td>8/10</td>
<td>1072/222</td>
</tr>
</tbody>
</table>

These test cases were taken from the OAEI-2005 ontology matching contest$^6$. 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](image)

<table>
<thead>
<tr>
<th>Recall, %</th>
<th>OMAP</th>
<th>CMS</th>
<th>Dublin20</th>
<th>Falcon</th>
<th>FOAM</th>
<th>OLA</th>
<th>ctxMatch2</th>
<th>Baseline</th>
<th>S-Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.64</td>
<td>44.88</td>
<td>26.52</td>
<td>31.17</td>
<td>31.90</td>
<td>9.36</td>
<td>5.90</td>
<td>26.54</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```plaintext
13: Boolean cPointsMatrix;
100: if cPointsDiscovery(sourceNode, targetNode) == true then
101: cPointsMatrix[i][j] = true;
102: ResolveCpoint(sourceNode, targetNode, context1, context2);
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).

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 \( C_{12}, C_{23} \) critical point, we also managed to discover the new correspondences, namely between \( C_{23} \) and \( C_{12}, C_{11} \).

<table>
<thead>
<tr>
<th>cNodesMatrix: relations among concepts at nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{12} )</td>
</tr>
<tr>
<td>( C_{23} )</td>
</tr>
</tbody>
</table>

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 Games1 \( \rightarrow \) Entertainment2 or a weaker relation, namely Games1 \( \rightarrow \) Entertainment2, 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 | dDescendant, dDescendant;
3: ACOL | ACOL;
4: If cNodesMatrix[sourceNode.nodeID][targetNode.nodeID] == "idk" then
5: dDescendant = getSubTree(sourceNode);
6: tDescendant = getSubTree(targetNode);
7: for each sACOL ∈ dDescendant.ACOL do
8: for each tACOL ∈ tDescendant.ACOL do
9: if cNodesMatrix[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 C13 and C23. Parts of cLabsMatrix (notice, the relations in this matrix were computed by applying extended 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

<table>
<thead>
<tr>
<th>TOP2</th>
<th>GAMES3</th>
<th>BOARDGAMES3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOL3</td>
<td>idk</td>
<td>idk</td>
</tr>
<tr>
<td>games2</td>
<td>idk</td>
<td>idk</td>
</tr>
</tbody>
</table>

Table 7. cNodesMatrix: relations holding among concepts of nodes

<table>
<thead>
<tr>
<th>C13</th>
<th>C12</th>
<th>C14</th>
<th>C1x</th>
<th>C1q</th>
<th>C110</th>
<th>C111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In cNodesMatrix (Table 7) the relation between C13 and C23 is idk. In cLabsMatrix (Table 6) there is a support relation for the given matching problem, e.g., BoardGames3 ⊆ Games2. Therefore, relation between C13 and C23 represents the critical point and we should reconsider the relation holding between Games3 and Entertainment2 in cLabsMatrix.

Finally, it is worth noting that this algorithm also properly handles nodes, which are indeed dissimilar, e.g., C13 and C23 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 context1, context2)
2: String sRel;
3: String | ExecutionList;
4: ACOL | ACOL;
5: for each sACOL ∈ sNodesMatrix ACOL do
6: for each tACOL ∈ tNodesMatrix ACOL do
7: if GetMLibRel(ExecutionList, sACOL) == idk then
8: ACOLMatrix[sACOL.id][tACOL.id] == sRel;
9: end for
10: end for
11: cNodesMatrixUpdate(sourceNode, targetNode, context1, context2);

The ResolveCpoint function determines relations (sRel) 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 those of Table 1. Thus, they can not be applied in all the cases.

Table 8. Element level semantic matchers (Part 2)

<table>
<thead>
<tr>
<th>Matcher name</th>
<th>Exec. order</th>
<th>Approx. level</th>
<th>Matcher type</th>
<th>Schema info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchy Distance</td>
<td>1</td>
<td>3</td>
<td>Sense-based</td>
<td>WN senses</td>
</tr>
<tr>
<td>Extended WN Gloss</td>
<td>3</td>
<td>3</td>
<td>Gloss-based</td>
<td>WN senses</td>
</tr>
<tr>
<td>Gloss Comparison</td>
<td>4</td>
<td>3</td>
<td>Gloss-based</td>
<td>WN senses</td>
</tr>
<tr>
<td>Extended Gloss Comparison</td>
<td>5</td>
<td>3</td>
<td>Gloss-based</td>
<td>WN senses</td>
</tr>
</tbody>
</table>

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 Games3 is equivalent to Entertainment2. 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 (Games3 ⇔ Entertainment2) we recompute (line 11)6, by re-running SAT, the relation holding between the pair of critical nodes, thus determining that C13 = C23.

6 cNodesMatrixUpdate performs functionalities identical to those of lines 80-82 in Algorithm 1.
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 setup. 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 ing tasks of Table 4, where available expert mappings allow measuring both precision\(^9\) 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)](image)

<table>
<thead>
<tr>
<th>Recall increase (relative), %</th>
<th>HD</th>
<th>GC</th>
<th>HD + GC</th>
<th>HD + WNG + GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold value</td>
<td>4</td>
<td>2</td>
<td>4 (\backslash) 2</td>
<td>4 (\backslash) 1 2</td>
</tr>
</tbody>
</table>

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 \((C1_1, C2_2)\) is the critical point, however, we were unable to resolve it with the matchers of Table 8, namely to match Home\(_1\) and Hobbies\(_1\) AND Interests\(_2\).

REFERENCES


