UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

COMPUTING MINIMAL AND REDUNDANT MAPPINGS BETWEEN LIGHTWEIGHT ONTOLOGIES

Vincenzo Maltese and Aliaksandr Autayeu

December 2008

Technical Report # DISI-08-079

Also: Presented at the AISB Workshop on Matching and Meaning 2009, 9th April 2009, Edinburgh, UK

Computing minimal and redundant mappings between lightweight ontologies [1]

Vincenzo Maltese, Aliaksandr Autayeu
{maltese, autayeu}@disi.unitn.it

Dipartimento di Ingegneria e Scienza dell'Informazione (DISI)
Università di Trento

Abstract. The minimal mapping between two lightweight ontologies contains that minimal subset of mapping elements such that all the others can be efficiently computed from them. They have clear advantages in visualization and user interaction since they are the minimal amount of information that needs to be dealt with. They make the work of the user much easier, faster and less error prone. Experiments on our proposed algorithm to compute them demonstrate a substantial improvement both in run-time and number of elements found.

1 Introduction

As a possible answer to the semantic heterogeneity problem, in the recent years many matching solutions have been proposed. However, despite the progresses made, a lot of work still has to be done [4, 20]. In this paper, we focus on semantic matching techniques (see for instance [3, 5, 6, 7, 18, 19]), namely, techniques that find all semantically related nodes between any two graph-like structures in input (e.g., database or XML schemas and ontologies). Any such pair of nodes, along with the semantic relationship holding between the two, is what we informally call a *mapping element*.

In particular, exploiting the semantics encoded in the structure, our proposed algorithm takes two lightweight ontologies, as formally defined in [8], and computes the *minimal mapping*, namely the subset of all possible mapping elements between them such that: i) all the other can be computed from them in time linear in the size of the input graphs, and ii) none of them can be dropped without losing property i). Lightweight ontologies are DAG-like graph structures where each node is labelled by a natural language sentence which can be translated into a propositional Description Logic (DL) formula codifying the meaning of the node and where, following the get-specific principle [17], each node is subsumed by the formula of the node above. Each mapping element is attached with one of the following semantic relations: disjointness ('$\perp$'), equivalence ('$\equiv$'), more specific ('$\sqsubseteq$') and less specific ('$\sqsupseteq$').

Minimal mappings provide clear usability advantages. Consider for example the two fragments of lightweight ontologies depicted in Fig. 1. They represent two small classifications designed to arrange more or less the same content, but from different perspectives. The second ontology is a fragment taken from the Yahoo web directory[2]. Following the approach described in [8] and exploiting dedicated NLP techniques tuned to short phrases (see for instance [14]), each node label can be translated in an unambiguous, propositional DL expression. The resulting formulas are reported under the corresponding label. Here each string denotes a concept (e.g., journals#1) and the numbers at the end of the strings denote a specific concept constructed from a WordNet sense.

If we compute the whole set of mapping elements between the two ontologies, we obtain the set of arrows depicted in Fig. 1. However, only the two solid ones are in the minimal set, since all the others can be entailed from them. They are therefore said to be *redundant*. For instance, B$\sqsubseteq$D is a trivial consequence of B$\sqsubseteq$E and E$\sqsubseteq$D. Since the number of possible mapping elements can grow up to *n*m* with *n* and *m* the size of the two input ontologies, it is clear that visualization and manageability problems become crucial with large ontologies with potentially millions of mapping elements between them. Current interfaces have clear scalability problems in visualizing even small sets [4]. Minimal mappings are clearly more manageable, since they are the minimal amount of information that needs to be dealt with. They make the work of the user much easier, faster and less error prone [12].

[1] This paper is a short version of [1].
[2] http://dir.yahoo.com/

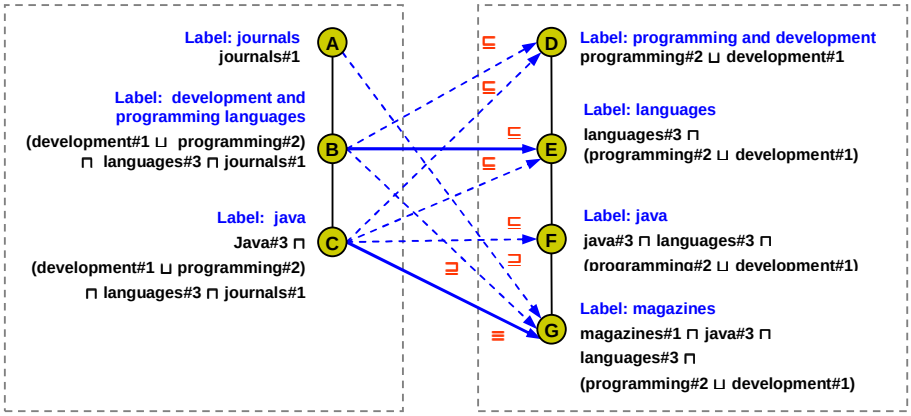

Fig. 1. The minimal mapping between two lightweight ontologies

In this paper we provide a definition of *minimal* and, dually, *redundant mappings* (where a redundant mapping is a mapping which is not minimal, namely contains redundant elements) and an algorithm for computing them. It makes use of a few node matching functions which take two nodes in input and return a positive answer in case a given semantic relation holds between them. The algorithm can be proved to be correct and complete, in the sense that it always computes the minimal set; it is very efficient as it minimizes the number of calls to the node matching functions; it computes the maximum number of mapping elements (i.e., including all the redundant ones) by maximally exploiting the information codified in the ontologies.

To the best of our knowledge, a few amount of work has been done on the problem of minimal mappings. In [10, 11, 12] Distributed Description Logics (DDL) [13] is used to represent and reason about existing ontology mappings. They introduce a few debugging heuristics which remove mapping elements which are redundant or generate inconsistencies from a given set [11]. The main problem of this approach, as also recognized by the authors, is the complexity of DDL reasoning [12]. Conversely, instead of pruning redundant elements, we directly compute the minimal set. Among other things, our approach allows us to be as fast as possible by minimizing the number of calls to the node matching functions.

The rest of the paper is organized as follows. Section 2 informally provides the definition for redundant and minimal mappings. Section 3 and 4 describe proposed algorithms. Finally, Section 5 draws some conclusions and outlines the future work.

2 Redundant and minimal mappings

Before applying any form of automated reasoning it is fundamental to convert natural language node labels of the DAG-like graph structures in input into a formal language. Lightweight ontologies [8] are formal representations where each node label is translated into a propositional DL formula and each node formula subsumes the one of the parent node.

A mapping element between two lightweight ontologies is a triple <n1, n2, R>, where n1 is the source node on the first ontology and n2 is the target node on the second one. R is a semantic relation in the set {⊥, ≡, ⊑, ⊒}. They are ordered such that disjointness precedes equivalence which is stronger than subsumption (in both directions), and such that the two subsumption symbols are unordered. This in order to return subsumption only when it is not a consequence of an equivalence relation or of one of the two nodes being inconsistent (this latter case generating at the same time both a disjointness and a subsumption relation). Under this ordering there can be at most one mapping element between two nodes.

A mapping element *m'* is redundant w.r.t. another mapping element *m* if the existence of *m'* can be asserted simply by looking at the positions of its nodes w.r.t. the nodes of *m* in their respective ontologies. In algorithmic terms, this means that the second mapping element can be computed without running SAT. We identified four basic redundancy patterns, one for each semantic relation. For sake of space we focus here on the pattern for ⊑. The interested reader can look at [1] for the other patterns. In Fig. 2, the blue dashed mapping element <C, D, ⊑> is redundant w.r.t. the solid blue one <A, B, ⊑>. In fact, C is more specific than A which is more specific than B which is more specific than D. As a consequence, by transitivity C is more

specific than D. Notice that it still holds in case we substitute subsumption between A and B with equivalence. The red solid curve shows how the semantic relation propagates. This can be codified in the following ***redundancy condition***:

Given two lightweight ontologies O_1 and O_2, a mapping M, and a mapping element $m' \in M$ with $m' = \langle C, D, \sqsubseteq \rangle$ between them, we say that m' is redundant in M iff $\exists\, m \in M$ with $m = \langle A, B, R \rangle$ and $m \neq m'$ such that $R \in \{\sqsubseteq, \equiv\}$, $A \in path(C)$ and $D \in path(B)$;

Here path(n) is the path from the root to the node n. Note that we impose $m \neq m'$ to ensure we are not trivially comparing a mapping element with itself.

Using the definition of redundancy, the minimal mapping is the set of mapping elements with maximum size with no redundant mapping elements between the two ontologies. Note that for any two given lightweight ontologies, the minimal set always exists and it is unique. A proof is provided in [1]. Taking any two paths in the two ontologies, a minimal subsumption mapping element is an element with the highest node in one path whose formula is subsumed by the other node and the lowest node in the other path which is subsumed by the formula in the other node.

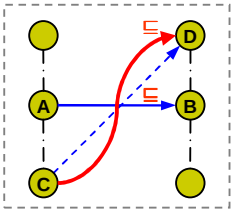

Fig. 2. Redundancy detection pattern for is subsumed by (⊑).

3 Computing minimal mappings

In this paper, we propose the function **TreeSubsumedBy** which computes the minimal set of subsumption mapping elements between two subtrees rooted in n1 and n2. The first call will be on the roots of the two lightweight ontologies T1 and T2 in input. The pseudo-code is given in Fig. 3. The complete pseudo-code to compute the minimal set for all kinds of relations is given in [1].

```
node: struct of {cnode: wff; children: node[];}
T1,T2: tree of (node);
relation in {⊑, ⊒, ≡, ⊥};
element: struct of {source: node; target: node; rel: relation;};
M: list of (element);
10  function boolean TreeSubsumedBy(node n1, node n2)
20   {c1,c2: node; LastNodeFound: boolean;
30    if (<n1,n2,⊥> ∈ M) then return false;
40    if (!NodeSubsumedBy(n1, n2)) then
50      foreach c1 in GetChildren(n1) do TreeSubsumedBy(c1,n2);
60    else
70      {LastNodeFound := false;
80       foreach c2 in GetChildren(n2) do
90         if (TreeSubsumedBy(n1,c2)) then LastNodeFound := true;
100      if (!LastNodeFound) then AddSubsumptionElement(n1,n2);
120         return true;
140     };
150   return false;
160  };
170 function boolean NodeSubsumedBy(node n1, node n2)
180  {if (Unsatisfiable(mkConjunction(n1.cnode,negate(n2.cnode)))) then
        return true;
190   else return false; };
```

Fig. 3. Pseudo-code for the TreeSubsumedBy function.

At each step it makes use of the **NodeSubsumedBy** (lines 170-190) which is the node matching function that, given two nodes n1 and n2, returns a positive answer in case subsumption holds between the two, or a negative answer in case is not or it cannot establish that this is the case. It embeds a call to a SAT solver. It is potentially very expensive, since it takes exponential time in worst case [5]. However, by processing both trees top down and using the described pattern, **TreeSubsumedBy** significantly reduces the amount of calls to the node matcher. In fact, once we identify a subsumption mapping element we know which parts of the trees we can safely avoid to check. At each step it assumes that minimal disjointness map-

ping elements are already computed; as a consequence, at line 30 it checks whether the mapping element between the nodes n1 and n2 is already in the minimal set. If this is the case it stops the recursion. This allows computing the disjointness relation rather than subsumption when both hold (namely with an inconsistent node). Given n2, lines 40-50 implement a depth first recursion in the first tree till a subsumption is found. Lines 60-140 implement what happens after the first subsumption is found. The key idea is that, after finding the first subsumption, **TreeSubsumedBy** keeps recursing down the second tree till it finds the last subsumption. When this happens the resulting mapping element is added to the minimal set (line 100).

Our algorithm, let us call it MinSMatch, has been implemented by taking the node matching routines of the state of the art matcher S-Match [5] and by changing the way the tree structure is matched. The selected four datasets had been already used in previous evaluations; see [5, 15, 16] and the full version of this paper [1] for details (in terms of computed mapping elements, run time and SAT calls). For brevity we give below only the number of subsumption mapping elements of the minimal and the redundant mapping of maximum size (as it is computed using an extended version of the algorithm we present in the next section) identified by the algorithm. We also provide the percentages of reduced elements (1-minimal/total) which turns to be very high, in the range 87-100% (the latter interesting case is because they are all redundant w.r.t. an equivalence mapping element). We observed also an overall (namely for all semantic relations) significant reduction in computation time (in the range 16-59%) and calls to SAT (in the range 43-66%) w.r.t. S-Match. The interested reader can refer to [5, 15] for a detailed qualitative and performance evaluation of SMatch w.r.t. other state of the art matching algorithms.

	S-Match	MinSMatch		
	Total mapping elements	Total mapping elements	Mapping elements in the minimal set	Reduction (%)
⊑	92/541 20587/8295	92/541 20594/8401	5/0 1534/1103	94.56/100.00 92.55/86.87

Table 1. Mapping results (subsumption only) for the selected four datasets.

4 Computing the mapping of maximum size

Computing the mapping of maximum size can be seen as the result of the propagation of the elements in the minimal set according to the given redundancy condition for subsumptions. If new mapping elements are identified, the resulting mapping is clearly redundant. Note that no calls to SAT are needed for that. See pseudo-code above for subsumption.

```
10  function list PropagateSubsumedBy(node n1, node n2)
20   {M': list of (element);
30    foreach c1 in SubTree(n1) do
40       foreach c2 in Path(n2) do AddElement(<c1,c2, ⊑>,M');
50    return M'};
```

Fig. 4. Pseudo-code to propagate a subsumption

Given two lightweight ontologies T1 and T2, the corresponding minimal set M, and two nodes n1 in T1 and n2 in T2 we can verify whether a subsumption mapping element holds between the two given the current available background knowledge. Intuitively, <n1, n2, ⊑> is a suitable mapping element either if it is in the minimal set M or it is redundant w.r.t. some mapping element *m* in M. Test for redundancy can be performed as described in Fig. 5. Note that a subsumption element can be the result of the propagation of a non redundant subsumption or equivalence mapping element (modulo inconsistencies).

```
10  function boolean VerifySubsumedBy(node n1, node n2)
20   {c1,c2: node;
30    foreach c1 in Path(n1) do
40      foreach c2 in SubTree(n2) do
50        if ((<c1,c2,⊑> ∈ M) || (<c1,c2,≡> ∈ M)) then return true;
60    return false;
70   };
```

Fig. 5. Pseudo-code to verify the redundancy of a subsumption mapping element

5 Conclusion

In this paper we have provided a definition and a very fast algorithm for the computation of the minimal mapping and, on user request, the mapping of maximum size between two lightweight ontologies. We have evaluated the resulting system with respect to the state-of-the-art matching system S-Match [5]. The results show a substantial improvement in the (much lower) computation time, in the (much lower) number of mapping elements which need to be stored and handled and in the (higher) total number of mappings which are computed. The latter point because, maximally exploiting the information codified in the tree structure of the two input ontologies, we minimize the impact of the lack of background knowledge [9].

The future work includes the development of a suitable user interface which exploits minimal mapping elements thus avoiding the messy visualizations which are generated whenever the number of elements grows, but also the experimentation with large scale mapping tasks, for instance between large library classification systems (e.g., NALT, AGROVOC, LCSH).

References

1. F. Giunchiglia, V. Maltese, A. Autayeu, 2008. Computing minimal mappings. University of Trento, KnowDive Group Technical Report.
2. F. Giunchiglia, M. Marchese, I. Zaihrayeu, 2006. Encoding Classifications into Lightweight Ontologies. Journal of Data Semantics 8, pp. 57-81.
3. P. Shvaiko, J. Euzenat, 2007. Ontology Matching. Springer-Verlag New York, Inc. Secaucus, NJ, USA.
4. P. Shvaiko, J. Euzenat, 2008. Ten Challenges for Ontology Matching. In Proceedings of the 7th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2008).
5. F.Giunchiglia, M. Yatskevich, P. Shvaiko, 2007. Semantic Matching: algorithms and implementation. Journal on Data Semantics, IX, 2007.
6. F. Giunchiglia, M. Yatskevich, and E. Giunchiglia. Efficient semantic matching. In Proceedings of the 2nd european semantic web conference (ESWC'05), Heraklion, 2005.
7. F. Giunchiglia, P. Shvaiko, and M. Yatskevich, 2005. Semantic schema matching. In Proceedings of CoopIS, pp. 347-365.
8. F.Giunchiglia, I. Zaihrayeu, 2007. Lightweight Ontologies. In The Encyclopedia of Database Systems, to appear. Springer, 2008.
9. F.Giunchiglia, P. Shvaiko, M. Yatskevich, 2006. Discovering missing background knowledge in ontology matching. Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006), pp. 382–386.
10. H. Stuckenschmidt, L. Serafini, H. Wache, 2006. Reasoning about Ontology Mappings. Proceedings of the ECAI-06 Workshop on Contextual Representation and Reasoning, 2006.
11. C. Meilicke, H. Stuckenschmidt, A. Tamilin, 2006. Improving automatically created mappings using logical reasoning. In the proceedings of the 1st International Workshop on Ontology Matching OM-2006, CEUR Workshop Proceedings Vol. 225.
12. C. Meilicke, H. Stuckenschmidt, A. Tamilin, 2008. Reasoning support for mapping revision. Journal of Logic and Computation, 2008.
13. A. Borgida, L. Serafini. Distributed Description Logics: Assimilating Information from Peer Sources. Journal on Data Semantics pp. 153-184.
14. I. Zaihrayeu, L. Sun, F. Giunchiglia, W. Pan, Q. Ju, M. Chi, and X. Huang, 2007. From web directories to ontologies: Natural language processing challenges. In 6th International Semantic Web Conference (ISWC 2007).
15. P. Avesani, F. Giunchiglia and M. Yatskevich, 2005. A Large Scale Taxonomy Mapping Evaluation. In Proceedings of International Semantic Web Conference (ISWC 2005), pp. 67-81.
16. M. Yatskevich, F. Giunchiglia, and P. Avesani, 2007. A Large Scale Dataset for the Evaluation of Matching. In Posters of 4th European Semantic Web Conference (ESWC 2007).
17. F. Giunchiglia, M. Marchese, I. Zaihrayeu, 2006. Encoding Classifications into Lightweight Ontologies. Proceedings of the 3rd European Semantic Web Conference (ESWC 2006) pp. 80–94.
18. P. Shvaiko, J. Euzenat, 2005. A Survey of Schema-based Matching Approaches. Journal on Data Semantics, (IV) pp. 146–171.
19. F. Giunchiglia and M. Yatskevich. Element level semantic matching. In Proceedings of Meaning Coordination and Negotiation workshop at the International Semantic Web Conference (ISWC), 2004.
20. C. Caracciolo, J. Euzenat, L. Hollink, R. Ichise, A. Isaac, V. Malaisé, C. Meilicke, J. Pane, P. Shvaiko, 2008. First results of the Ontology Alignment Evaluation Initiative 2008.