Computing minimal mappings

Fausto Giunchiglia, Vincenzo Maltese, Aliaksandr Autayeu

Dipartimento di Ingegneria e Scienza dell'Informazione (DISI) - Università di Trento
{fausto, maltese, autayeu}@disi.unitn.it

Abstract. Given two classifications, or lightweight ontologies, we compute the minimal mapping, namely the subset of all possible correspondences, called mapping elements, between them such that i) all the others can be computed from them in time linear in the size of the input ontologies, and ii) none of them can be dropped without losing property i). In this paper we provide a formal definition of minimal mappings and define a time efficient computation algorithm which minimizes the number of comparisons between the nodes of the two input ontologies. The experimental results show a substantial improvement both in the computation time and in the number of mapping elements which need to be handled.

Keywords: Ontology matching, lightweight ontologies, minimal mappings

1 Introduction

Given any two graph-like structures, e.g., database and XML schemas, classifications, thesauri and ontologies, matching is usually identified as the problem of finding those nodes in the two structures which semantically correspond to one another. Any such pair of nodes, along with the semantic relationship holding between the two, is what we informally call a *mapping element*. In the last few years a lot of work has been done on this topic both in the digital libraries [15, 16, 17, 21] and the computer science [2, 3, 4, 5, 6, 8, 9] communities. In this paper we concentrate on lightweight ontologies (or formal classifications), as formally defined in [1, 7], and we focus on the problem of finding *minimal mappings,* that is, the subset of all possible correspondences, called *mapping elements*, such that i) all the others 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). This must not be seen as a limitation. There are plenty of schemas in the world which can be translated, with almost no loss of information, into lightweight ontologies. For instance, thesauri, library classifications, file systems, email folder structures, web directories, business catalogues and so on. Lightweight ontologies are well defined and pervasive. The main advantage of minimal mappings is that they are the minimal amount of information that needs to be dealt with. Notice that this is a rather important feature as the number of possible mapping elements can grow up to $n*m$ with n and m being the size of the two input ontologies. Minimal mappings provide clear usability advantages. Many systems and corresponding interfaces, mostly graphical, have been provided for the management of mappings but all of them hardly scale with the increasing number of nodes, and the resulting visualizations are rather messy [3]. Furthermore, the maintenance of smaller sets makes the work of the user much easier, faster and less error prone [11].

The main contributions of this paper are a formal definition of *minimal* and, dually, *redundant mappings*, evidence of the fact that the minimal mapping always exists and it is unique and an algorithm for computing it. This algorithm has the following main features:

1. It can be proved to be correct and complete, in the sense that it always computes the minimal mapping;
2. It minimizes the number of calls to the node matching function which computes the relation between two nodes. Notice that node matching in the general case amounts to logical reasoning [5], and it may require exponential time;
3. It computes the mapping of maximum size (including the maximum number of redundant elements) as it maximally exploits the information codified in the graph of the lightweight ontologies in input. This, in turn, avoids missing mapping elements due to pitfalls in the node matching functions**,** e.g. because of missing background knowledge [8].

As far as we know very little work has been done on the issue of computing minimal mappings. In general the computation of minimal mappings can be seen as a specific instance of the mapping inference problem [4]. Closer to our work, in [9, 10, 11] the authors use Distributed Description Logics (DDL) [12] 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 [10]. The main problem of this approach, as also recognized by the authors, is the complexity of DDL reasoning [11]. In our approach, instead of pruning redundant elements, we directly compute the minimal set. Among other things, our approach allows us to minimize the number of calls to node matching.

The rest of the paper is organized as follows. Section 2 provides a motivating example. Section 3 provides the definition for redundant and minimal mappings, and it shows that the minimal set always exists and it is unique. Section 4 describes the algorithm while Section 5 evaluates it. Finally, Section 6 draws some conclusions and outlines the future work.

2 A motivating example

Classifications are perhaps the most natural tool humans use to organize information content. Information items are hierarchically arranged under topic nodes moving from general ones to more specific ones as long as we go deeper in the hierarchy.

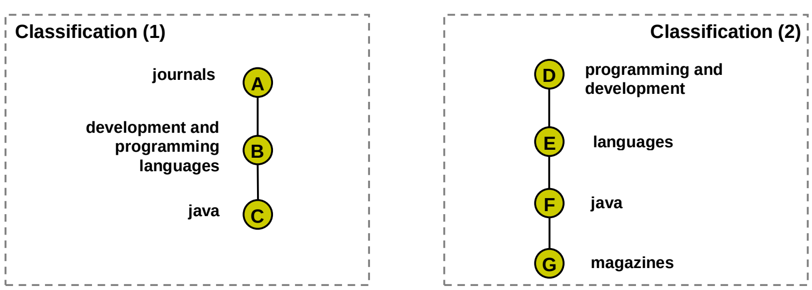
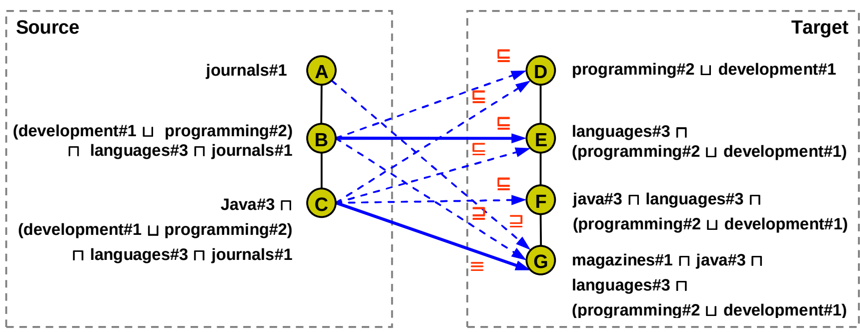


Fig. 1. Two classifications

This attitude is well known in Knowledge Organization as the principle of organizing from the general to the specific [16], called synthetically the *get-specific principle* in [1, 7]. Consider the two fragments of classifications depicted in Fig. 1. They are designed to arrange more or less the same content, but from different perspectives. The second is a fragment taken from the Yahoo web directory[1] (category Computers and Internet).

Following the approach described in [1] and exploiting dedicated NLP techniques tuned to short phrases (for instance, as described in [13]), classifications can be converted, exactly or with a certain degree of approximation, into their formal alter-ego, namely into lightweight ontologies. Lightweight ontologies [1, 7] are acyclic graph structures where each natural language node label is translated into a propositional Description Logic (DL) formula codifying the meaning of the node. Notice that the formula associated to each node contains the formula of the node above to capture the fact that the meaning of each node is contextualized by the meaning of its ancestor nodes. As a consequence, the backbone structure of the resulting lightweight ontologies is represented by subsumption relations between nodes. The resulting formulas are reported in Fig. 2.



M' = {<A, G, ⊒>, <B, D, ⊑>, <B, E, ⊑>, <B, G, ⊒>, <C, D, ⊑>, <C, E, ⊑>, <C, F, ⊑>, <C, G, ≡>}
M = { <B, E, ⊑>, <C, G, ≡>}

Fig. 2. The minimal and redundant mapping between two lightweight ontologies

Here each string denotes a concept (e.g., journals#1) and the number at the end of the strings denote a specific concept constructed from a WordNet sense. Fig. 2 also reports the resulting mapping elements. We assume that each mapping element is associated with one of the following semantic relations: disjointness ($\perp$), equivalence ($\equiv$), more specific ($\sqsubseteq$) and less specific ($\sqsupseteq$), as computed for instance by semantic matching [5]. Notice however that not all the mapping elements have the same semantic valence. For instance, $B \sqsubseteq D$ is a trivial logical consequence of $B \sqsubseteq E$ and $E \sqsubseteq D$, and similarly for $C \sqsubseteq F$ and $C \equiv G$. We represent the elements in the minimal mapping using solid lines and redundant elements using dashed lines. M' is the set of maximum size (including the maximum number of redundant elements) while M is the minimal set. The problem is how to compute the minimal set in the most efficent way.

[1] http://dir.yahoo.com/

3 Redundant and minimal mappings

Adapting the definition in [1] we define a lightweight ontology as follows:

Definition 1 (Lightweight ontology). A lightweight ontology O is a rooted tree $<N,E,L^F>$ where:

a) N is a finite set of nodes;
b) E is a set of edges on N;
c) L^F is a finite set of labels expressed in a Propositional DL language such that for any node $n_i \in N$, there is one and only one label $l_i^F \in L^F$;
d) $l_{i+1}^F \sqsubseteq l_i^F$ with n_i being the parent of n_{i+1}.

The superscript F is used to emphasize that labels are in a formal language. Fig. 2 above provides an example of (a fragment of) two lightweight ontologies.

We then define mapping elements as follows:

Definition 2 (Mapping element). Given two lightweight ontologies O_1 and O_2, a mapping element m between them is a triple $<n_1, n_2, R>$, where:

a) $n_1 \in N_1$ is a node in O_1, called the source node;
b) $n_2 \in N_2$ is a node in O_2, called the target node;
c) $R \in \{\equiv, \sqsubseteq, \sqsupseteq, \perp\}$ is the strongest semantic relation holding between n_1 and n_2.

The partial order is such that disjointness is stronger than equivalence which, in turn, is stronger than subsumption (in both directions), and such that the two subsumption symbols are unordered. This is in order to return subsumption only when equivalence does not hold or one of the two nodes being inconsistent (this latter case generating at the same time both a disjointness and a subsumption relation), and similarly for the order between disjointness and equivalence. Notice that, under this ordering, there can be at most one mapping element between two nodes.

The next step is to define the notion of redundancy. The key idea is that, given a mapping element $<n_1, n_2, R>$, a new mapping element $<n_1', n_2', R'>$ is redundant with respect to the first if the existence of the second can be asserted simply by looking at the relative positions of n_1 with n_1', and n_2 with n_2'. In algorithmic terms, this means that the second can be computed without running the time expensive node matching functions. We have identified four basic redundancy patterns as follows:

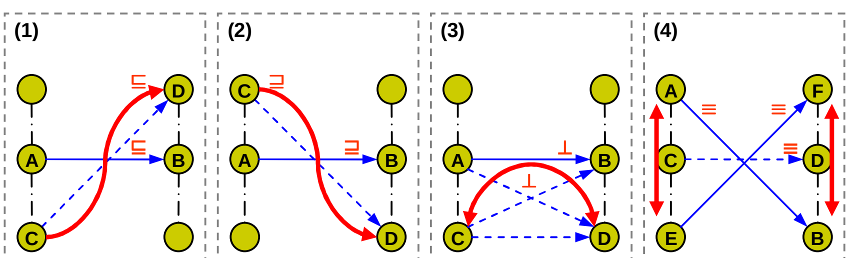


Fig. 3. Redundancy detection patterns

In Fig. 3, the blue dashed mappings are redundant w.r.t. the solid blue ones. The bold red solid lines show how a semantic relation propagates. Let us discuss the rationale for each of the patterns:

- **Pattern (1)**: each mapping element <C, D, $\sqsubseteq$> is redundant w.r.t. <A, B, $\sqsubseteq$>. 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.
- **Pattern (2)**: dual argument as in pattern (1).
- **Pattern (3)**: each mapping element <C, D, $\perp$> is redundant w.r.t. <A, B, $\perp$>. In fact, we know that A and B are disjoint, that C is more specific than A and that D is more specific than B. This implies that C and D are also disjoint.
- **Pattern (4)**: Pattern 4 is the combinations of patterns (1) and (2).

In other words, the patterns are the way to capture logical inference from structural information, namely just by looking at the position of the nodes in the two trees. As we will show, this on turn allows computing the redundant elements in linear time (w.r.t. the size of the two ontologies) from the ones in the minimal set. Notice that patterns (1) and (2) are still valid in case we substitute subsumption with equivalence. However, in this case we cannot exclude the possibility that a stronger relation holds between C and D. A trivial example of where this is not the case is provided in Fig. 4 (a).

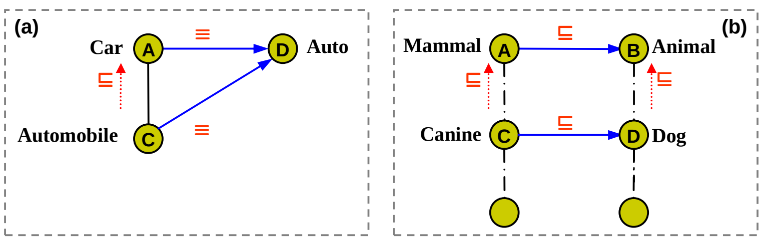


Fig. 4. Examples of non redundant mapping elements

On the basis of the patterns and the considerations above we can define redundant elements as follows. Here path(n) is the path from the root to the node n.

Definition 3 (Redundant mapping element). Given two lightweight ontologies O_1 and O_2, a mapping M and a mapping element m'$\in$M with m' = <C, D, R'> between them, we say that m' is redundant in M iff one of the following holds:

(1) If R' is $\sqsubseteq$, $\exists$m$\in$M with m = <A, B, R> and m $\neq$ m' such that R $\in$ {$\sqsubseteq$, $\equiv$}, A $\in$ path(C) and D $\in$ path(B);

(2) If R' is $\sqsupseteq$, $\exists$m$\in$M with m = <A, B, R> and m $\neq$ m' such that R $\in$ {$\sqsupseteq$, $\equiv$}, C $\in$ path(A) and B $\in$ path(D);

(3) If R' is $\perp$, $\exists$m$\in$M with m = <A, B, $\perp$> and m $\neq$ m' such that A $\in$ path(C) and B $\in$ path(D);

(4) If R' is $\equiv$, conditions (1) and (2) must be satisfied.

See how Definition 3 maps to the four patterns in Fig. 3. Fig. 2 in Section 2 provides examples of redundant elements. Definition 3 can be proved to capture all and only the cases of redundancy.

Theorem 1 (Redundancy, soundness and completeness). Given a mapping M between two lightweight ontologies O_1 and O_2, a mapping element m' $\in$ M is redundant if and only if it satisfies one of the conditions of Definition 3.

The soundness argument is the rationale described for the patterns above. Completeness can be shown by constructing the counterargument that we cannot have redundancy in the remaining cases. We can proceed by enumeration, negating each of the patterns, encoded one by one in the conditions appearing in the Definition 3. The complete proof is given in [22]. Fig. 4 (b) provides an example of non redundancy which is based on pattern (1). It tells us that the existence of a link between two nodes does not necessarily propagate to the two nodes below. For example we cannot derive that Canine $\sqsubseteq$ Dog from the set of axioms {Canine $\sqsubseteq$ Mammal, Mammal $\sqsubseteq$ Animal, Dog $\sqsubseteq$ Animal}, and it would be wrong to do so.

The notion of redundancy allows us to formalize the notion of minimal mapping as follows:

Definition 4 (Minimal mapping). Given two lightweight ontologies O_1 and O_2, we say that a mapping M between them is minimal iff:

a) $\nexists$m$\in$M such that m is redundant (minimality condition);

b) $\nexists$M'$\supset$M satisfying condition a) above (maximality condition).

A mapping element is minimal if it belongs to the minimal mapping.

Note that conditions (a) and (b) ensure that the minimal set is the set of maximum size with no redundant elements. As an example, the set M in Fig. 2 is minimal. Comparing this mapping with M' we can observe that all elements in the set M' - M are redundant and that, therefore, there are no other supersets of M with the same properties. In effect, <A, G, $\sqsupseteq$> and <B, G, $\sqsupseteq$> are redundant w.r.t. <C, G, $\equiv$> for pattern (2); <C, D, $\sqsubseteq$>, <C, E, $\sqsubseteq$> and <C, F, $\sqsubseteq$> are redundant w.r.t. <C, G, $\equiv$> for pattern (1); <B, D, $\sqsubseteq$> is redundant w.r.t. <B, E, $\sqsubseteq$> for pattern (1). Note that M contains far less mapping elements w.r.t. M'.

As last observation, for any two given lightweight ontologies, the minimal mapping always exists and it is unique.

Theorem 2 (Minimal mapping, existence and uniqueness). Given two lightweight ontologies O_1 and O_2, there is always one and only one minimal mapping between them.

A proof is given in [22].

4 Computing minimal and redundant mappings

The patterns described in the previous section suggest how to significantly reduce the amount of calls to the node matchers. By looking for instance at pattern (2) in Fig. 3, given a mapping element m = <A, B, $\sqsupseteq$> we know that it is not necessary to compute the semantic relation holding between A and any descendant C in the sub-tree of B since we know in advance that it is $\sqsupseteq$. At the top level the algorithm is organized as follows:

- **Step 1, computing the minimal mapping modulo equivalence:** compute the set of disjointness and subsumption mapping elements which are *minimal modulo equivalence*. By this we mean that they are minimal modulo collapsing, whenever possible, two subsumption relations of opposite direction into a single equivalence mapping element;
- **Step 2, computing the minimal mapping:** eliminate the redundant subsumption mapping elements. In particular, collapse all the pairs of subsumption elements (of opposite direction) between the same two nodes into a single equivalence element. This will result into the *minimal mapping*;
- **Step 3, computing the mapping of maximum size:** Compute the mapping of maximum size (including minimal and redundant mapping elements). During this step the existence of a (redundant) element is computed as the result of the propagation of the elements in the minimal mapping.

The first two steps are performed at matching time, while the third is activated whenever the user wants to exploit the pre-computed mapping elements, for instance for their visualization. For lack of space in the following we give only the pseudo-code for the first step. The interested reader can look at [22] for the pseudo-code of the other two steps.

The minimal mapping is computed by a function **TreeMatch** whose pseudo-code is given in Fig. 5. M is the minimal set while T1 and T2 are the input lightweight ontologies.

```
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);
boolean direction;

function TreeMatch(tree T1, tree T2)
 {TreeDisjoint(root(T1),root(T2));
  direction := true;
  TreeSubsumedBy(root(T1),root(T2));
  direction := false;
  TreeSubsumedBy(root(T2),root(T1));
  TreeEquiv();
 };
```

Fig. 5. Pseudo-code for the tree matching function

TreeMatch is crucially dependent on the node matching functions **NodeDisjoint** (given in [22]) and **NodeSubsumedBy** (Fig. 6) which take two nodes n1 and n2 and return a positive answer respectively in case of disjointness or subsumption, or a negative answer if it is not the case or they are not able to establish it. Notice that these two functions hide the heaviest computational costs; in particular their computation time is exponential when the relation holds, but possibly much faster, when the relation does not hold. The main motivation for this is that the node matching problem, in the general case, should be translated into disjointness or subsumption problem in propositional DL (see [5] for a detailed description). The goal, therefore, is to compute the minimal mapping by minimizing the calls to the node matching functions and, in particular minimizing the calls where the relation will turn out to hold. We achieve this purpose by processing both trees top down. To maximize the performance of the system, **TreeMatch** has therefore been built as the sequence of three function calls: the first call to **TreeDisjoint** (line 80) computes the minimal set of disjointness mapping elements, while the second and the third call to **TreeSubsumedBy** compute the minimal set of subsumption mapping elements in the two directions modulo equivalence (lines 90-120). Notice that in the second call, **TreeSubsumedBy** is called with the input ontologies with swapped roles. These three calls correspond to Step 1 above. Line 130 in the pseudo code of the **TreeMatch** implements the Step 2.

Given two sub-trees in input, rooted in n1 and n2, the **TreeDisjoint** function searches for the first disjointness elements along any pair of paths in them. Look at [22] for corresponding pseudo-code and the complete description.

TreeSubsumedBy (Fig. 6) recursively finds all minimal mapping elements where the strongest relation between the nodes is $\sqsubseteq$ (or dually, $\sqsupseteq$ in the second call in the **TreeMatch**, line 120. In the following we will concentrate only on the first call).

```
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 AddSubsumptionMappingElement(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; };
200 function AddSubsumptionMappingElement(node n1, node n2)
210  {if (direction) then AddMappingElement(<n1,n2,⊑>);
220   else AddMappingElement(<n2,n1,⊒>); };
```

Fig. 6. Pseudo-code for the **TreeSubsumedBy** function

Notice that **TreeSubsumedBy** assumes that the minimal disjointness 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 stronger disjointness relation rather than subsumption when both hold (namely in presence of an inconsistent node). Given n2, lines 40-50 implement a depth first recursion in the first tree till a subsumption is found. The test for subsumption is performed by the **NodeSubsumedBy** function that checks whether the formula obtained by the conjunction of the formulas associated to the node n1 and the negation of the formula for n2 is unsatisfiable (lines 170-190). 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). Notice that both **NodeDisjoint** and **NodeSubsumedBy** call the function **Unsatisfiable** which embeds a call to a SAT solver.

To fully understand **TreeSubsumedBy,** the reader should check what happens in the four situations in Fig. 7. In case (a) the first iteration of the **TreeSubsumedBy** finds a subsumption between A and C. Since C has no children, it skips lines 80-90 and directly adds the mapping element <A, C, ⊑> to the minimal set (line 100). In case (b), since there is a child D of C the algorithm iterates on the pair A-D (lines 80-90) finding a subsumption between them. Since there are no other nodes under D, it adds the mapping element <A, D, ⊑> to the minimal set and returns true. Therefore LastNodeFound is set to true (line 90) and the mapping element between the pair A-C is recognized as redundant. Case (c) is similar. The difference is that **TreeSubsumedBy** will return false when checking the pair A-D (line 30), thanks to previous computation of minimal disjointness mapping elements, and therefore the mapping element <A, C, ⊑> is recognized as minimal. In case (d) the algorithm iterates after the second subsumption mapping element is identified. It first checks the pair A-C and iterates on A-D concluding that subsumption does not hold between them (line 40). Therefore, it recursively calls **TreeSubsumedBy** between B and D. In fact, since <A, C, ⊑> will be recognized as minimal, it is not worth checking <B, C, ⊑> for pattern (1). As a consequence <B, D, ⊑> is recognized as minimal together with <A, C, ⊑>.

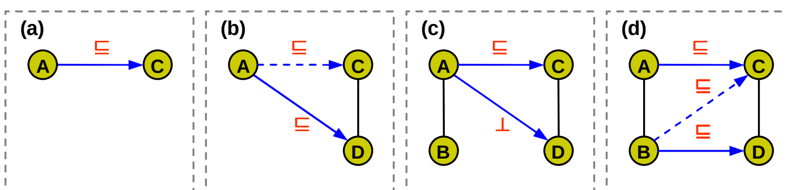


Fig. 7. Examples of applications of the **TreeSubsumedBy**

Five observations. The first is that, even if, overall, **TreeMatch** implements three loops instead of one, the wasted (linear) time is largely counterbalanced by the exponential time saved by avoiding a lot of useless calls to the SAT solver. The second is that, when the input trees T1 and T2 are two nodes, **TreeMatch** behaves as a node matching function which returns the semantic relation holding between the input nodes. The third is that the call to **TreeDisjoint** before the two calls to **TreeSubsumedBy** allows us to implement the partial order on relations defined in the previous section. In particular it allows returning only a disjointness mapping element when both disjointness and subsumption hold. The fourth is the fact that skipping (in the body of the **TreeDisjoint**) the two sub-trees where disjointness holds is what allows not only implementing the partial order (see the previous observation) but also saving a lot of useless calls to the node matching functions. The fifth and last observation is that the implementation of **TreeMatch** crucially depends on the fact that the minimal elements of the two directions of subsumption and disjointness can be computed independently (modulo inconsistencies).

5 Evaluation

The algorithm presented in the previous section, let us call it MinSMatch, has been implemented by taking the node matching routines of the state of the art matcher SMatch [5] and by changing the way the tree structure is matched. The evaluation has been performed by directly comparing the results of MinSMatch and SMatch on several real-world datasets. All tests have been performed on a Pentium D 3.40GHz with 2GB of RAM running Windows XP SP3 operating system with no additional applications running except the matching system. Both systems were limited to allocating no more than 1GB of RAM. The tuning parameters were set to the default values. The selected datasets had been already used in previous evaluations, see [14]. Some of these datasets can be found at OAEI web site[2]. The first two datasets describe courses and will be called **C**ornell and **W**ashington, respectively. The second two come from the arts domain and will be referred to as **T**opia and **I**con, respectively. The third two datasets have been extracted from the Looksmart, Google and Yahoo! directories and will be referred to as **S**ource and **T**arget. The fourth two datasets contain portions of the two business directories eCl@ss[3] and UNSPSC[4] and will be referred to as **E**class and **U**nspsc. Table 1 describes some indicators of the complexity of these datasets.

#	Dataset pair	Node count	Max depth	Average branching factor
1	Cornell/Washington	34/39	3/3	5.50/4.75
2	Topia/Icon	542/999	2/9	8.19/3.66
3	Source/Target	2857/6628	11/15	2.04/1.94
4	Eclass/Unspsc	3358/5293	4/4	3.18/9.09

Table 1. Complexity of the datasets

Consider Table 2. The reduction in the last column is calculated as (1-m/t), where m is the number of elements in the minimal set and t is the total number of elements in the mapping of maximum size, as computed by MinSMatch. As it can be easily noticed, we have a significant reduction, in the range 68-96%.

The second interesting observation is that in Table 2, in the last two experiments, the number of total mapping elements computed by MinSMatch is slightly higher (compare the second and the third column). This is due to the fact that in the presence of one of the patterns, MinSMatch directly infers the existence of a mapping element without testing it. This allows MinSMacth, differently from SMatch, to avoid missing

[2] http://oaei.ontologymatching.org/2006/directory/
[3] http://www.eclass-online.com/
[4] http://www.unspsc.org/

elements because of failures of the node matching functions (because of lack of background knowledge [8]). One such example from our experiments is reported below (directories Source and Target):

\Top\Computers\Internet\Broadcasting\Video Shows

\Top\Computing\Internet\Fun & Games\Audio & Video\Movies

We have a minimal mapping element which states that Video Shows $\sqsupseteq$ Movies. The element generated by this minimal one, which is captured by MinSMatch and missed by SMatch (because of the lack of background knowledge about the relation between 'Broadcasting' and 'Movies') states that Broadcasting $\sqsupseteq$ Movies.

	S-Match	MinSMatch		
#	Total mapping elements (t)	Total mapping elements (t)	Minimal mapping elements (m)	Reduction, %
1	223	223	36	83.86
2	5491	5491	243	95.57
3	282638	282648	30956	89.05
4	39590	39818	12754	67.97

Table 2. Mapping sizes.

To conclude our analysis, Table 3 shows the reduction in computation time and calls to SAT. As it can be noticed, the time reductions are substantial, in the range 16% - 59%, but where the smallest savings are for very small ontologies. In principle, the deeper the ontologies the more we should save. The interested reader can refer to [5, 14] for a detailed qualitative and performance evaluation of SMatch w.r.t. other state of the art matching algorithms.

	Run Time, ms			SAT calls		
#	S-Match	MinSMatch	Reduction, %	S-Match	MinSMatch	Reduction, %
1	472	397	15.88	3978	2273	42.86
2	141040	67125	52.40	1624374	616371	62.05
3	3593058	1847252	48.58	56808588	19246095	66.12
4	6440952	2642064	58.98	53321682	17961866	66.31

Table 3. Run time and SAT problems

6 Conclusions

In this paper we have provided a definition and a fast algorithm for the computation of the minimal mapping between two lightweight ontologies. The evaluation shows a substantial improvement in the (much lower) computation time, in the (much lower) number of elements which need to be stored and handled and in the (higher) total number of mapping elements which are computed.

The future work includes the experimentation with various large Knowledge Organization Systems (e.g., NALT, AGROVOC, LCSH).

References

1. F. Giunchiglia, M. Marchese, I. Zaihrayeu, 2006. Encoding Classifications into Lightweight Ontologies. Journal of Data Semantics 8, pp. 57-81.

2. P. Shvaiko, J. Euzenat, 2007. Ontology Matching. Springer-Verlag New York, Inc. Secaucus, NJ, USA.
3. 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).
4. J. Madhavan, P. A. Bernstein, P. Domingos, A. Y. Halevy, 2002. Representing and Reasoning about Mappings between Domain Models. At the 18th National Conference on Artificial Intelligence (AAAI 2002).
5. F. Giunchiglia, M. Yatskevich, P. Shvaiko, 2007. Semantic Matching: algorithms and implementation. Journal on Data Semantics, IX, 2007.
6. 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.
7. F.Giunchiglia, I. Zaihrayeu, 2007. Lightweight Ontologies. In The Encyclopedia of Database Systems, to appear. Springer, 2008.
8. 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.
9. H. Stuckenschmidt, L. Serafini, H. Wache, 2006. Reasoning about Ontology Mappings. Proceedings of the ECAI-06 Workshop on Contextual Representation and Reasoning.
10. 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.
11. C. Meilicke, H. Stuckenschmidt, A. Tamilin, 2008. Reasoning support for mapping revision. Journal of Logic and Computation, 2008.
12. A. Borgida, L. Serafini. Distributed Description Logics: Assimilating Information from Peer Sources. Journal on Data Semantics pp. 153-184.
13. 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).
14. 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.
15. M. L. Zeng, L. M. Chan, 2004. Trends and Issues in Establishing Interoperability Among Knowledge Organization Systems. Journal of the American Society for Information Science and Technology, 55(5) pp. 377–395.
16. L. Kovács. A. Micsik, 2007. Extending Semantic Matching Towards Digital Library Contexts. Proc.eedings of the 11th European Conference on Digital Libraries (ECDL 2007), pp. 285-296.
17. B. Marshall, T. Madhusudan, 2004. Element matching in concept maps. Proceedings of the 4th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL 2004), pp.186-187.
18. B. Hjørland, 2008. What is Knowledge Organization (KO)?. Knowledge Organization. International Journal devoted to Concept Theory, Classification, Indexing and Knowledge Representation 35(2/3) pp. 86-101.
19. D. Soergel, 1972. A Universal Source Thesaurus as a Classification Generator. Journal of the American Society for Information Science 23(5), pp. 299–305.
20. D. Vizine-Goetz, C. Hickey, A. Houghton, and R. Thompson. 2004. Vocabulary Mapping for Terminology Services. Journal of Digital Information, Volume 4, Issue 4.
21. M. Doerr, 2001. Semantic Problems of Thesaurus Mapping. Journal of Digital Information, Volume 1, Issue 8.
22. F. Giunchiglia, V. Maltese, A. Autayeu, 2008. Computing minimal mappings. University of Trento, DISI Technical Report: http://eprints.biblio.unitn.it/archive/00001525/