# Efficient Semantic Matching

Fausto Giunchiglia[1], Mikalai Yatskevich[1], and Enrico Giunchiglia[2]

[1] Dept. of Information and Communication Technology,
University of Trento,
38050 Povo, Trento, Italy
{fausto, yatskevi}@dit.unitn.it
[2] DIST – Universita di Genova,
Viale Causa 13, 16165, Genova, Italy
enrico@dist.unige.it

**Abstract.** We think of Match as an operator which takes two graph-like structures and produces a mapping between semantically related nodes. We concentrate on classifications with tree structures. In semantic matching, correspondences are discovered by translating the natural language labels of nodes into propositional formulas, and by codifying matching into a propositional unsatisfiability problem. We distinguish between problems with conjunctive formulas and problems with disjunctive formulas, and present various optimizations. For instance, we propose a linear time algorithm which solves the first class of problems. According to the tests we have done so far, the optimizations substantially improve the time performance of the system.

## 1 Introduction

We think of matching as the task of finding semantic correspondences between elements of two graph-like structures (e.g., conceptual hierarchies, classifications, database schemas or ontologies). Matching has been successfully applied in many well-known application domains, such as schema/ontology integration, data warehouses, and XML message mapping. In this paper we concentrate on classifications with tree structures.

Semantic matching, as introduced in [1, 5], is based on the key intuition that labels at nodes, which are written in natural language, are translated into propositional formulas which codify the intended meaning of the labels themselves. This allows us to codify the matching problem into a propositional unsatisfiability problem, which can then be efficiently implemented using state of the art propositional satisfiability (SAT) solvers [8, 9]. We call *concept of a label* the propositional formula which stands for the set of documents that one would classify under a label it encodes. We call *concept at a node* 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 [5]. As from [5], all previous approaches, though implicitly or explicitly exploiting the semantic information codified in graphs, differ substantially from our approach in that they compute a syntactic “similarity” coefficients between labels in the [0,1] range (see for instance [3, 10] ).

A. Gómez-Pérez and J. Euzenat (Eds.): ESWC 2005, LNCS 3532, pp. 272–289, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The system we have developed, called *S-Match* [6], takes two classifications and computes the strongest semantic relation holding between any pair of nodes. The matching problem is articulated into two macro steps, namely element and structure level matching. Element level matchers consider only the information on the atomic level [7] (the labels of nodes), while structure level matchers consider also the structure of the trees. Our goal in this paper is to describe the structure level matching algorithm, as it has been implemented within *S-Match*, and present a set of optimizations. In particular, we distinguish between two main classes of problems. In the first class all the concepts at nodes are *atomic* or *conjunctive* formulas. In the second class the concepts at nodes may also contain *disjunctive* formulas. In the case of conjunctive concepts at nodes we present a modification of the original algorithm which solves the node matching problem in linear time. With disjunctive concepts we present various techniques, which, among the other things, allow us to avoid the exponential space explosion which arises when converting disjunctive formulas into Conjunctive Normal Form (CNF). This modification is required since all state of the art SAT deciders take CNF formulas in input.

We have evaluated the time performance of the optimized algorithm against its basic version and several state of the art matching systems. The optimizations seem to improve substantially the time performance of *S-Match*. In all cases *S-Match* performs better or much better than the unoptimized version and always competes well with the other matching systems. In particular, it outperforms them on trees with hundreds or thousands of nodes.

The rest of the paper is organized as follows. Section 2 provides an overview of the *S-Match* tree matching algorithm. Section 3 discusses the basic node matching algorithm. The next two sections are dedicated to the two classes of node matching problems we have identified. Node matching problems with *conjunctive* concepts at nodes (and their optimizations) are discussed in Section 4, while the node matching problems with *disjunctive* concepts at nodes (and their optimizations) are described in Section 5. We discuss the evaluation results in Section 6. Section 7 concludes the paper.

## 2 The Tree Matching Algorithm

As from [6], the *S-Match* algorithm is organized according the following four macro steps:

- *Step 1*: for all labels in the two trees, compute concepts of labels;
- *Step 2*: for all nodes in the two trees, compute concepts at nodes;
- *Step 3*: for all pairs of labels in the two trees, compute the semantic relations between concepts of labels;
- *Step 4*: for all pairs of nodes in the two trees, compute the semantic relations between concepts at nodes.

The first two steps represent the pre-processing phase, while the third and the fourth steps correspond to the element-level and structure-level matching respectively. The semantic relations we consider are: *equivalence* (=); *more general*

(⊒); *less general* (⊑); *disjointness* (⊥); *overlapping* (⊓). When none of the relations holds, the special *Idk* (I don't know or (?)) relation is returned.

The version of the algorithm defined in this paper assumes that:

- There are no negated atomic concepts of labels (one example of negated concept of label is $C_{except\ apple}=\neg C_{apple}$)
- The information we use, namely the labels of nodes and the knowledge residing in WordNet (see below) is all globally consistent. Under this assumption the only reason why we get an unsatisfiable formula is because we have found a match between two nodes

In order to understand how the algorithm works, consider for instance the two trees depicted in Figure 1a.

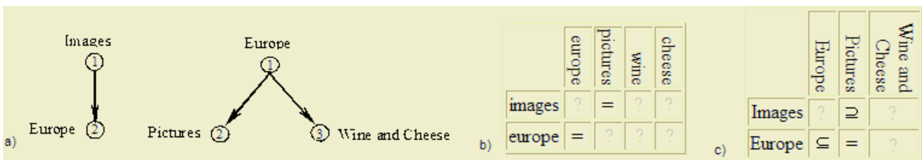



**Fig. 1.** (a): Two trees. (b): The matrix of relations between concepts of labels. (c): The matrix of relations between the concepts at nodes (matching result)

During *Step 1* we first tokenize labels. For instance "*Wine and Cheese*" becomes <*Wine*, *and*, *Cheese*>. Then we lemmatize tokens. Thus for instance "*Images*" becomes "*image*". Then, an Oracle (at the moment we use WordNet 2.0) is queried in order to obtain the senses of the lemmatized tokens. Afterwards, these senses are attached to atomic concepts. Finally, complex concepts are built suitably composing atomic concepts. Thus, the concept of the label *Wine and Cheese is computed as* $C_{Wine\ and\ Cheese}$=<*wine, {senses*$_{WN}$*#4}*>∨<*cheese, {sensesWN#4}*>, where <*cheese, {seneses*$_{WN}$*4}*> is taken to be the union of the four WordNet senses, and similarly for *wine*. Notice that natural language *and* is converted into logical disjunction rather than conjunction.

*Step 2* takes into account the structural schema properties. The logical formula for a concept at a node is constructed most often as the conjunction of the concept of a label formulas in the concept path to the root [5]. For example, the concept $C_2$ for the node *Pictures* in Figure 1a is computed as $C_2=C_{Europe} \wedge C_{Pictures}$.

Element level semantic matchers are applied during *Step 3*. They determine the semantic relations holding between pairs of atomic concepts of labels. For example, from WordNet we can derive that *image* and *picture* are synonyms, and therefore, $C_{Images} = C_{Pictures}$. Notice that *Image* and *Picture* have 8 and 11 senses in WordNet, respectively. In order to determine the senses which are relevant in the current context, sense filtering techniques are applied (see [11] for more details). The relations between the atomic concepts of labels for the trees depicted in Figure 1a are reported in Figure 1b.

Element level semantic matchers provide the input to the structure level matcher, which is applied in *Step 4*. This matcher produces the set of semantic relations between concepts at nodes (see Figure 1c for example). On this step the tree matching problem is reformulated into the set of node matching problems, one for each pair of nodes. Further, each node matching problem is reduced to a propositional validity problem.

The pseudo code of the Steps 3 and 4 of the semantic matching algorithm is reported in Figure 2. **treeMatch** takes 2 trees of Nodes (*source, target*) and returns the matrix of semantic relations between concepts at nodes in both trees (*cNodesMatrix*). First, **fillCLabMatrix** exploit element level semantic matchers library in order to fill the matrix of relations between concepts of labels in both trees (*cLabsMatrix*) (line 11). This action corresponds to the third step of the tree matching algorithm. Afterwards, two loops over all nodes of source and target trees are executed (lines 12-20 and 15-20). Within these loops, the propositional formulas corresponding to the concepts at nodes ($context_A$, $context_B$) are computed by **getCnodeFormula** (lines 14, 17).

```
1. Node: struct of
2.        int nodeId;
3.        String label;
4.        String cLabel;
5.        String cNode;

6.String[][] treeMatch(Tree of Nodes source, target)
7. Node sourceNode,targetNode;
8. String[][] cLabsMatrix, cNodesMatrix, relMatrix;
9. String axioms, context_A, context_B;
10.int i,j;
11.cLabsMatrix=fillCLabMatrix(source,target);
12.For each sourceNode in source
13.  i=getNodeId(sourceNode);
14.  context_A=getCnodeFormula (sourceNode);
15.  For each targetNode in target
16.    j=getNodeId(targetNode);
17.    context_B=getCnodeFormula (targetNode);
18.    relMatrix=extractRelMatrix(cLabsMatrix,
                                  sourceNode, targetNode);
19.    axioms=mkAxioms(relMatrix);
20.    cNodesMatrix[i][j]=nodeMatch(axioms,context_A,
                                                      context_B);
21. return cNodesMatrix;
```

**Fig. 2.** The pseudo code of the tree matching algorithm

*relMatrix* is calculated in the inner loop by **extractRelMatrix** (line 18). It contains the part of the *cLabsMatrix* relevant to the particular node matching problem. *axioms* (line 19) contains the conjunction of the propositional formulas in *relMatrix*. For example, the semantic relations in Figure 1b, which are considered

when we match *Europe* and *Pictures* are $Europe_A = Europe_B$, $Images_A = Pictures_B$. In this case `axioms` is $(Europe_A \leftrightarrow Europe_B) \wedge (Images_A \leftrightarrow Pictures_B)$. Notice that, subscripts designate the *context* (either *A* or *B*) to which a propositional variable (or concept) belongs. The detailed description of **`nodeMatch`** is provided in the next section.

## 3 The Node Matching Algorithm

**`nodeMatch`** input formulas are combined to obtain the following formula:

$$(axioms) \rightarrow rel(context_A, context_B), \quad (1)$$

where *axioms*, $context_A$, $context_B$ are as defined in **`treeMatch`** (Figure 2), while $rel(context_A, context_B)$ is the formula corresponding to the semantic relation being checked, (namely equivalence, less or more generality, or disjointness). As from [5], two nodes match if and only if Eq. 1 is valid, namely if it is *true* for all possible truth assignments to its propositional variables. Given that most of the available propositional solvers are satisfiability checkers, the negation of the matching formula is checked for unsatisfiability. This yields the following formula

$$axioms \wedge \neg rel(context_A, context_B) \quad (2)$$

Table 1 reports the resulting matching formulas as a function of the semantic relation being tested. Notice that the check for equality is omitted. In fact A = B holds iff A⊆B and A⊇B hold.

**Table 1.** The relationship between semantic relations and propositional formulas

| $rel(a,b)$ | Translation of $rel(a, b)$ in propositional logic | CNF translation of Eq. 2 |
|---|---|---|
| $a=b$ | $a \leftrightarrow b$ | *N/A* |
| $a \subseteq b$ | $a \rightarrow b$ | $axioms \wedge context_A \wedge \neg context_B$ |
| $a \supseteq b$ | $b \rightarrow a$ | $axioms \wedge context_B \wedge \neg context_A$ |
| $a \perp b$ | $\neg(a \wedge b)$ | $axioms \wedge context_A \wedge context_B$ |

Consider the pseudo code of the node matching algorithm, as described in Figure 3.

**`nodeMatch`** constructs the formulas needed for testing less generality (line 120) and more generality (line 150), it converts them to CNF (lines 130, 160) and checks for unsatisfiability (lines 140, 170). If both relations hold, then the equivalence relation is returned (line 190). Afterwards, the same procedure is repeated for disjointness test. If all the tests fail *"Idk"* is returned (line 290).

Prior to the discussion of optimizations to our basic solution, let us classify the concepts of labels and concepts at nodes. We distinguish between four categories of concepts of labels:

```
110.String nodeMatch(String axioms, context_A, context_B)
120.   String formula=And(axioms,context_A,Not(context_B));
130.   String formulaInCNF=convertToCNF(formula);
140.   boolean isLG=isUnsatisfiable(formulaInCNF)
150.   formula=And(axioms, Not(context_A), context_B);
160.   formulaInCNF=convertToCNF(formula);
170.   boolean isMG= isUnsatisfiable(formulaInCNF);
180.   if (isMG && isLG)
190.    return "=";
200.  if (isLG)
210.    return "⊆";
220.  if (isMG)
230     return "⊇";
240.  formula= And(axioms, context_A, context_B);
250.  formulaInCNF=convertToCNF(formula);
260.  boolean isOpposite= isUnsatisfiable(formulaInCNF);
270.  if (isOpposite)
280.    return "⊥";
290.  return "Idk";
```

**Fig. 3.** The pseudo code of the node matching algorithm

- **Atomic**: the concept of a label is an atomic proposition. For example, the concept of the label *Europe* is $C_{Europe}$ = *<Europe, {senses*$_{WN}$*#1}>*, where *senses*$_{WN}$*#1* stands for a WordNet sense.
- **Conjunctive**: the concept of a label is a conjunction. For example, the concept of the label *transmission gearbox* is $C_{transmission\ gearbox} = C_{transmission} \wedge C_{gearbox}$.
- **Disjunctive**: the concept of a label is a disjunction. For example, the concept of the label *jet and trains and cars* is $C_{jet\ and\ trains\ and\ cars}=C_{jet} \vee C_{train} \vee C_{car}$.
- **Full proposition at logic**: the concept of a label contains both conjunctions and disjunctions. For example the concept of the label *computers and electrical equipment* is $C_{computers\ and\ electrical\ equipment}=C_{computer} \vee (C_{electrical} \wedge C_{equipment})$.

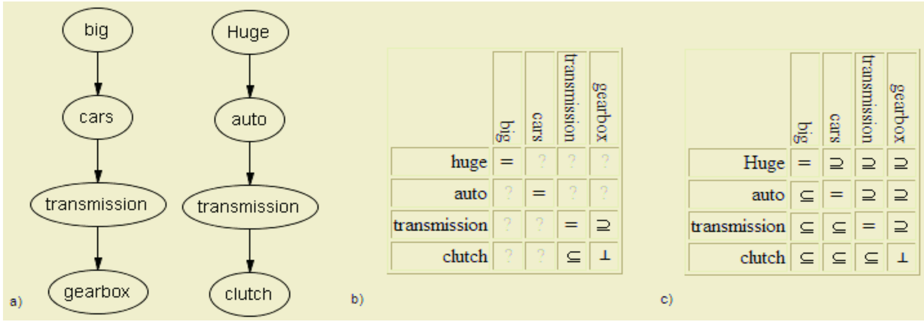
This classification allows us to further distinguish between two classes of *concepts at nodes*, which are at the basis of our optimizations:

- **Conjunctive concepts at nodes:** the concept at a node is a conjunction.
- **Disjunctive concepts at nodes:** the concept at a node contains both conjunctions and disjunctions in any order.

# 4 Conjunctive Concepts at Nodes

## 4.1 Node Matching Problems

Consider the two trees depicted in Figure 4a. Notice that they have only atomic concepts of labels. Let us consider the matching of *gearbox* and *clutch*.




|  | big | cars | transmission | gearbox |
|---|---|---|---|---|
| huge | = | ? | ? | ? |
| auto | ? | = | ? | ? |
| transmission | ? | ? | = | ⊒ |
| clutch | ? | ? | ⊑ | ⊥ |

|  | big | cars | transmission | gearbox |
|---|---|---|---|---|
| Huge | = | ⊒ | ⊒ | ⊒ |
| auto | ⊑ | = | ⊒ | ⊒ |
| transmission | ⊑ | ⊑ | = | ⊒ |
| clutch | ⊑ | ⊑ | ⊑ | ⊥ |

**Fig. 4.** (a): Two trees. (b): The matrix of relations between concepts of labels. (c): The matrix of relations between concepts at nodes (matching result)

The relevant semantic relations between concepts of labels are depicted in Figure 4b. As from Table 1, *axioms* is:

$$(big_A \leftrightarrow huge_B) \wedge (car_A \leftrightarrow auto_B) \wedge (transmission_A \leftrightarrow transmission_B) \wedge (gearbox_A \rightarrow transmission_B) \wedge (clutch_B \rightarrow transmission_A) \wedge \neg(clutch_B \wedge gearbox_A) \quad (3)$$

which, translated in CNF, becomes:

$$(\neg big_A \vee huge_B) \wedge (big_A \vee \neg huge_B) \wedge (\neg car_A \vee auto_B) \wedge (car_A \vee \neg auto_B) \wedge (\neg transmission_A \vee transmission_B) \wedge (transmission_A \vee \neg transmission_B) \wedge (\neg gearbox_A \vee transmission_B) \wedge (\neg clutch_B \vee transmission_A) \wedge (\neg clutch_B \vee \neg gearbox_A) \quad (4)$$

As from Step 2 in Section 2, $context_A$ and $context_B$ are constructed by taking the conjunction of the concepts of labels in the path to root. Therefore, $context_A$ and $context_B$ are:

$$big_A \wedge car_A \wedge transmission_A \wedge gearbox_A \quad (5)$$

$$huge_B \wedge auto_B \wedge transmission_B \wedge clutch_B \quad (6)$$

while their negations are:

$$\neg big_A \vee \neg car_A \vee \neg transmission_A \vee \neg gearbox_A \quad (7)$$

$$\neg huge_B \vee \neg auto_B \vee \neg transmission_B \vee \neg clutch_B \quad (8)$$

Let us consider the formula to be checked for unsatisfiability, as from Table 1. The first observation is that *axioms* remains the same for all the tests, and it contains only clauses with two variables, where a clause is a finite disjunction of literals. In the worst case it contains $2*n_A*n_B$ clauses, where $n_A$ and $n_B$ are the number of atomic concepts of labels in the paths to the root (in our example $n_A$ and $n_B$ are equal to 4). The second observation is that the formulas for less and more generality are very similar and differ only in the context formula which is negated. Thus, for instance, in the less generality test $context_B$ is negated. This means that Eq. 1 contains one clause with $n_B$ variables (Eq. 8) in addition to $n_A$ clauses with one variable derived from $context_A$ (Eq. 5). Finally, again from Table 1, in the case of disjointness test $context_A$

and $context_B$ are not negated. Therefore, Eq. 1 contains $n_A+n_B$ clauses with one variable (Eq. 5 and Eq. 6).

So far we have concentrated on atomic concepts of labels. The propositional formulas remain the same if we move to conjunctive concepts at labels. Consider the trees depicted in Figure 5a. Let us consider the matching between *transmission gearbox* and *transmission clutch*.

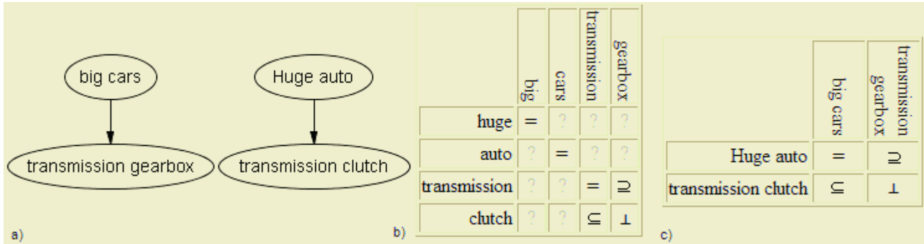



**Fig. 5.** (a): Two trees. (b): The matrix of relations between concepts of labels in the trees. (c): The matrix of relations between concepts at nodes (matching result)

Compare the matrices on the Figure 5b and Figure 4b. They are the same. The matrix of the relations between concepts of labels unambiguously determines *axioms* (see Eq. 3 and 4). Furthermore, as from Step 2 in Section 2, the propositional formulas for $context_A$ and $context_B$ are the same for atomic and for conjunctive concepts of labels as long as they "globally" contain the same formulas. In fact, concepts at nodes are constructed by taking the conjunction of concepts at labels. Splitting a concept of a label with two conjuncts into two atomic concepts has no effect on the resulting matching formula.

### 4.2 Optimizations

Let us consider first more and less generality and then disjointness.

#### 4.2.1 Less and More Generality Tests

As from Section 4.1, formula (Eq. 1) in this case is as follows:

$$\overbrace{\bigwedge_{q=0}^{n*m}(\neg A_s \vee B_t) \wedge \bigwedge_{w=0}^{n*m}(A_k \vee \neg B_l) \wedge \bigwedge_{v=0}^{n*m}(\neg A_p \vee \neg B_r)}^{Axioms} \wedge \overbrace{\bigwedge_{i=1}^{n} A_i}^{Context_A} \wedge \overbrace{\bigvee_{j=1}^{m} \neg B_j}^{\neg Context_B} \quad (9)$$

where $n$ is the number of variables in $context_A$, $m$ is the number of variables in $context_B$. $A_i$'s belong to $context_A$, and $B_j$'s belong to $context_B$. $s$, $k$, $p$ are in the [0..n] range, while $t$, $l$, $r$ are in the [0..m] range. *Axioms* can be empty. Eq. 9 is composed of clauses with 1 or 2 variables plus one clause with possibly more variables (the clause

corresponding to the negated context). The key observation is that the formula in Eq. 9 is Horn: each clause contains at most one positive literal. Therefore, the satisfiability problem can be decided in linear time by the unit resolution rule [2]. Notice, that DPLL-based SAT solvers require quadratic time in this case [15].

In order to understand how the linear time algorithm works, let us prove the unsatisfiability of Eq. 9 in the case of *gearbox* and *clutch*. In this case, Eq. 9 becomes

$$\begin{gathered}((\neg \mathbf{big}_A \vee huge_B) \wedge (\mathbf{big}_A \vee \neg huge_B) \wedge (\neg \mathbf{car}_A \vee auto_B) \wedge (\mathbf{car}_A \vee \neg auto_B) \wedge \\ (\neg transmission_B \vee \mathbf{transmission}_A) \wedge (transmission_B \vee \neg \mathbf{transmission}_A) \wedge \\ (\neg \mathbf{gearbox}_A \vee transmission_B) \wedge (\neg clutch_B \vee \mathbf{transmission}_A) \wedge \\ (\neg clutch_B \vee \neg \mathbf{gearbox}_A)) \wedge \mathbf{big}_A \wedge \mathbf{car}_A \wedge \mathbf{transmission}_A \wedge \mathbf{gearbox}_A \wedge \\ (\neg huge_B \vee \neg auto_B \vee \neg transmission_B \vee \neg clutch_B)\end{gathered} \tag{10}$$

where the variables from $context_A$ are written in bold.

First, we assign *true* to all unit clauses occurring in Eq 10 positively. Notice that these are all and only the clauses in $context_A$. This allows us to discard the clauses where $context_A$ variables occur positively (in this case: $\mathbf{big}_A \vee \neg huge_B$, $\mathbf{car}_A \vee \neg auto_B$, $\neg \mathbf{gearbox}_A \vee transmission_B$ and $\neg clutch_B \vee \mathbf{transmission}_A$). The resulting formula is

$$\begin{gathered}huge_B \wedge auto_B \wedge transmission_B \wedge \neg clutch_B \wedge \\ (\neg huge_B \vee \neg auto_B \vee \neg transmission_B \vee \neg clutch_B)\end{gathered} \tag{11}$$

Notice that this formula does not contain any variable derived from $context_A$. Notice also that, by assigning *true* to $huge_B$, $auto_B$ and $transmission_B$ and *false* to $clutch_B$ we do not derive a contradiction. Therefore, (Eq. 10) is satisfiable. In fact, a (Horn) formula is unsatisfiable if and only if the empty clause is derived (and satisfiable otherwise).

Consider again Eq. 11. For this formula to be unsatisfiable all the variables occurring in the negation of $context_B$ ($\neg huge_B \vee \neg auto_B \vee \neg transmission_B \vee \neg clutch_B$ in our example) should occur positively in the unit clauses obtained after resolving *Axioms* with the unit clauses in $context_A$ ($huge_B$, $auto_B$ and $transmission_B$ in our example). But for this to happen, for any $B_j$ in $context_B$ there must be a clause of form $\neg A_i \vee B_j$ in *axioms*, where $A_i$ is a formula of $context_A$. But formulas of the form $\neg A_i \vee B_j$ occur in Eq. 9 if and only if we have the axioms of the form $A = B_j$ and $A_i \sqsubseteq B_j$. These considerations suggest the following algorithm for testing satisfiability:

- *Step 1.* Create an array of size *m*. Each entry in the array stands for one $B_j$ in Eq. 9.
- *Step 2*. For each axiom of type $A_i = B_j$ and $A_i \sqsubseteq B_j$ mark the corresponding $B_j$.
- *Step 3*. If all the $B_j$*'s* are marked, then the formula is unsatisfiable.

`nodeMatch` can be modified as in Figure 6 (the numbers on the left indicate where the new code must be positioned):

```
111.  if (contextA and contextB are conjunctive)
112.    isLG=fastHornUnsatCheck (contextA, axioms,"⊆");
113.    isMG=fastHornUnsatCheck (contextB, axioms,"⊇");
114.  else

301.boolean fastHornUnsatCheck(String context, axioms,
                                                  rel);
302. int m=getNumOfVar(String context);
303. boolean array[m];
304. for each axiom in axioms
305.  if((getAType(axiom)="=")||(getAType(axiom)=rel))
306.   int j=getNumberOfSecondVariable(axiom);
307.   array[j]=true;
308. for (i=0; i<m; i++)
309.  if (!array[i])
310.   return false;
311. return true;
```

**Fig. 6.** Less and more generality tests optimization pseudo code

**`fastHornUnsatCheck`** implements the three steps above. Step 1 is performed in lines (302-303). Then, a loop on `axioms` (lines 304-307) implements Step 2. The final loop (lines 308-310) implements Step 3.

#### 4.2.2 Disjointness Test

Using the same notation as in Section 4.2.1, formula (Eq. 1) is as follows:

$$\overbrace{\bigwedge_{q=0}^{n*m}(\neg A_s \vee B_t)\wedge \bigwedge_{w=0}^{n*m}(A_k \vee \neg B_l)\wedge \bigwedge_{v=0}^{n*m}(\neg A_p \vee \neg B_r)\wedge}^{Axioms} \overbrace{\bigwedge_{i=1}^{n} A_i}^{Context_A} \wedge \overbrace{\bigwedge_{j=1}^{m} B_j}^{Context_B} \quad (12)$$

For example, the formula for testing disjointness between *gearbox* and *clutch* is

$$\begin{gathered}(\neg \mathbf{big}_A \vee huge_B) \wedge (\mathbf{big}_A \vee \neg huge_B) \wedge (\neg \mathbf{car}_A \vee auto_B) \wedge (\mathbf{car}_A \vee \neg auto_B) \wedge \\ (\neg transmission_B \vee \mathbf{transmission}_A) \wedge (transmission_B \vee \neg \mathbf{transmission}_A) \wedge \\ (\neg \mathbf{gearbox}_A \vee transmission_B) \wedge (\neg clutch_B \vee \mathbf{transmission}_A) \wedge \\ (\neg clutch_B \vee \neg \mathbf{gearbox}_A) \wedge \mathbf{big}_A \wedge \mathbf{car}_A \wedge \mathbf{transmission}_A \wedge \mathbf{gearbox}_A \wedge \\ huge_B \wedge auto_B \wedge transmission_B \wedge clutch_B\end{gathered} \quad (13)$$

Here again, the formula in Eq. 12 is Horn and thus, similarly to Section 4.2.1, the satisfiability of the formula can be decided by unit propagation. After assigning *true* to all the variables in $context_A$ and propagating the results we obtain the following formula:

$$huge_B \wedge auto_B \wedge transmission_B \wedge \neg clutch_B \wedge huge_B \wedge auto_B \wedge transmission_B \wedge clutch_B \quad (14)$$

If we further unit propagate $huge_B$, $auto_B$ and $transmission_B$ (this means that we assign true to them), then get the contradiction $clutch_B \wedge \neg clutch_B$. Therefore, the formula is unsatisfiable. This contradiction arises because $(\neg clutch_B \vee \neg \mathbf{gearbox}_A)$ occurs in Eq. 13, which, in turn, is derived (as from Table 1) from the disjointness axiom ($clutch_B \perp gearbox_A$). In fact, all the clauses in Eq. 12 contain one positive literal except for the clauses in *axioms* corresponding to disjointness relations. Thus, the key intuition here is that if there are no disjointness axioms, then Eq. 12 is satisfiable. On the other hand, if there is a disjointness axiom, atoms occurring there are also ensured to be either in $context_A$ or in $context_B$ and thus Eq. 12 is unsatisfiable. Therefore, the optimization consists of just checking the presence/absence of disjointness axioms in *axioms*.

The pseudo code of **`nodeMath`** can therefore be modified as follows:

```
231. If (contextA and contextB are conjunctive)
232.   If (there is disjointness axiom in the axioms)
233.     isOpposite=true;
234.   else
235.     isOpposite=false;
236. else
```

**Fig. 7.** Disjointness test optimization pseudo code

# 5 Disjunctive Concepts at Nodes

## 5.1 The Node Matching Problem

Consider the trees depicted in Figure 8a. Notice that the concepts at nodes contain disjunctive concepts of labels. Let us consider matching *fifties or sixties or seventies* with *twenties or thirties or forties*.

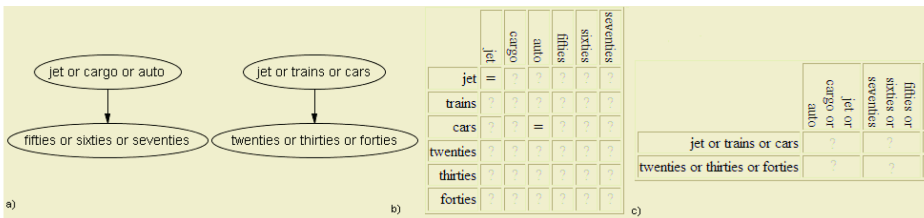



**Fig. 8.** (a): Two trees. (b): The matrix of relations between concepts of labels in the trees. (c): The matrix of relations between concepts at nodes (matching result)

The relations between atomic concepts of labels in both trees are depicted in Figure 8b. As from the second column of Table 1 *axioms* is:

$$(cars_B \leftrightarrow auto_A) \wedge (jet_A \leftrightarrow jet_B) \tag{15}$$

which can be rewritten as:

$$(\neg cars_B \vee auto_A) \wedge (cars_B \vee \neg auto_A) \wedge (\neg jet_A \vee jet_B) \wedge (jet_A \vee \neg jet_B) \tag{16}$$

As from Step 2 in Section 2 $context_A$ and $context_B$ are:

$$(jet_A \vee cargo_A \vee auto_A) \wedge (fifties_A \vee sixties_A \vee seventies_A) \tag{17}$$

$$(jet_B \vee train_B \vee cars_B) \wedge (twenties_B \vee thirties_B \vee forties_B) \tag{18}$$

The negations of $context_A$ and $context_B$ are:

$$(\neg jet_A \wedge \neg cargo_A \wedge \neg auto_A) \vee (\neg fifties_A \wedge \neg sixties_A \wedge \neg seventies_A) \tag{19}$$

$$(\neg jet_B \wedge \neg train_B \wedge \neg cars_B) \vee (\neg twenties_B \wedge \neg thirties_B \wedge \neg forties_B) \tag{20}$$

Let us consider the formula to be tested for unsatisfiability, as from Table 1. Again, *axioms* is the same for all the tests. As from Section 4.1, it consists up to $2*n_A*n_B$ clauses with two variables, where $n_A$ and $n_B$ are the number of atomic concepts of labels in the paths to root. In our example $n_A$ and $n_B$ are both equal to 6. The key observation here is that $context_A$ and $context_B$ may contain any number of disjunctions. Some exist because derived from the labels, while others may be obtained by negating $context_A$ or $context_B$ (as from the above example, in the case of less and more generality tests). Thus, for instance, as from Table 1 in case of less generality test we obtain the formula.

$$\begin{gathered}(\neg cars_B \vee auto_A) \wedge (cars_B \vee \neg auto_A) \wedge (\neg jet_A \vee jet_B) \wedge (jet_A \vee \neg jet_B) \wedge \\ (jet_A \vee cargo_A \vee auto_A) \wedge (fifties_A \vee sixties_A \vee seventies_A) \wedge \\ ((\neg jet_B \wedge \neg train_B \wedge \neg cars_B) \vee (\neg twenties_B \wedge \neg thirties_B \wedge \neg forties_B))\end{gathered} \tag{21}$$

## 5.2 Optimizations

With disjunctive concepts at nodes, Eq. 1 is a full propositional formula and no hypothesis can be made on its structure. As a consequence its satisfiability must be tested using a standard DPLL SAT solver. Thus for instance CNF conversion of Eq. 21 is

$$\begin{gathered}(\neg cars_B \vee auto_A) \wedge (cars_B \vee \neg auto_A) \wedge (\neg jet_A \vee jet_B) \wedge (jet_A \vee \neg jet_B) \wedge \\ (jet_A \vee cargo_A \vee auto_A) \wedge (fifties_A \vee sixties_A \vee seventies_A) \wedge \\ ((\neg jet_B \vee \neg twenties_B) \wedge (\neg jet_B \vee \neg thirties_B) \wedge (\neg jet_B \vee \neg forties_B) \wedge \\ (\neg train_B \vee \neg twenties_B) \wedge (\neg train_B \vee \neg thirties_B) \wedge (\neg train_B \vee \neg forties_B) \wedge \\ (\neg cars_B \vee \neg twenties_B) \wedge (\neg cars_B \vee \neg thirties_B) \wedge (\neg cars_B \vee \neg forties_B))\end{gathered} \tag{22}$$

In order to avoid the space explosion, which may arise when converting a formula to CNF (see for instance Eq. 22), we apply a set of structure preserving transformations [14, 4]. The main idea is to replace disjunctions occurring in the original formula with newly introduced variables and explicitly state that these variables imply the subformulas they substitute. Consider for instance Eq. 21. We obtain:

$$(\neg cars_B \vee auto_A) \wedge (cars_B \vee \neg auto_A) \wedge (\neg jet_A \vee jet_B) \wedge (jet_A \vee \neg jet_B) \wedge (jet_A \vee cargo_A \vee auto_A) \wedge (fifties_A \vee sixties_A \vee seventies_A) \wedge (new_1 \vee new_2) \wedge (\neg new_1 \vee \neg jet_B \vee \neg train_B \vee \neg car_B) \wedge (\neg new_2 \vee \neg twenties_B \vee \neg thirties_B \vee \neg forties_B) \quad (23)$$

Notice that the size of the propositional formula in CNF grows linearly with respect to number of disjunctions in original formula.

To account for this optimization in **nodeMatch** all calls to **convertToCNF** are replaced with calls to **optimizedConvertToCNF**, (see Figure 9):

```
130. formulaInCNF=optimizedConvertToCNF(formula);
...
160. formulaInCNF=optimizedConvertToCNF(formula);
...
250. formulaInCNF=optimizedConvertToCNF(formula);
```

**Fig. 9.** The CNF conversion optimization pseudo code

## 6 Evaluation Results

We have implemented the optimizations described above and evaluated the resulting system *S-Match* against the original system and two state of the art matching systems, namely COMA [3] and Similarity Flooding (SF) [12] as implemented in Rondo system [13]. Let us call $S\text{-}Match_B$ the original version without optimizations. Notice that S-Match, COMA, and SF exploit different matching techniques and differ substantially in the quality of matching results. See [6] for a detailed comparison

**Table 2.** The structural properties of the trees in the matching problems

| | Trees max. depth | # of nodes per tree | # of labels per tree | Average # of labels per node | Concepts at nodes |
|---|---|---|---|---|---|
| Cornell-Washington with atomic concepts of labels | 10/8 | 253/220 | 253/220 | 1/1 | Conjunctive |
| Handmade trees with disjunctive concepts of labels | 10/10 | 10/10 | 30/30 | 3/3 | Disjunctive |
| Looksmart-Yahoo | 10/8 | 140/74 | 222/101 | 1,58/1,36 | Conjunctive Disjunctive |
| Yahoo-Standard | 3/3 | 333/115 | 965/242 | 2,9/2,1 | Conjunctive Disjunctive |
| Google-Yahoo | 11/11 | 561/665 | 722/945 | 1,28/1,42 | Conjunctive Disjunctive |
| Google-Looksmart | 11/16 | 706/1081 | 1048/1715 | 1,48/1,63 | Conjunctive Disjunctive |

among these systems. In this evaluation we have concentrated only on the time performance of the systems. The tests have been performed on a P4 computer with 512 MB of RAM installed. The systems were limited to allocate no more than 512 MB of memory.

The systems have been tested on the six matching problems which can be found at http://dit.unitn.it/~accord/. Table 3 reports the properties of these problems.

### 6.1 Conjunctive Concepts at Nodes

On this problem *S-Match*$_B$ works two times faster than COMA. In fact, in this case the DPLL SAT solver of *S-Match* runs in polynomial time. *S-Match* instead works more than 5 times faster than COMA. However it still runs about 17% slower than SF. This can be explained by noticing that in SF the similarities between the labels of nodes obtained by a simple and fast string matcher, and propagated through a graph structure using a fix point algorithm. This algorithm is very fast and, on these examples, it converges after a few iterations. The drawback of SF, as the last test below shows, is that it requires a much larger amount of memory.

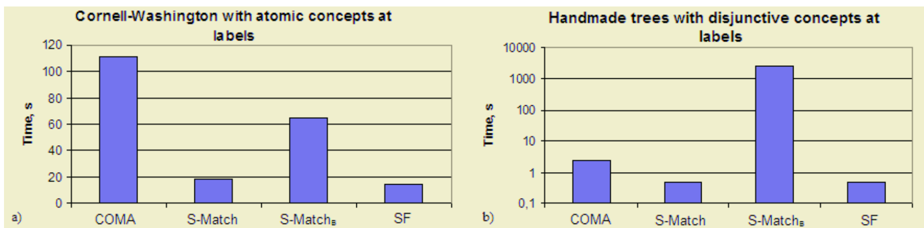



**Fig. 10.** Execution time of the matching systems

### 6.2 Disjunctive Concepts at Nodes

Let us consider the test with handmade trees. As from Figure 10b, *S-Match* works about 4 orders of magnitude faster than *S-Match*$_B$, about 4 times faster than COMA, and as fast as SF. The significant improvement of the optimized algorithm can be explained by considering that *S-Match*$_B$ does not control the exponential space explosion on such trees. In fact, the biggest formula in this case consists of about 118000 clauses. The optimization introduced in the Section 5.2 reduces this number to about 20-30 clauses.

We have then considered 4 matching problems involving real world classifications. Three of them, Looksmart-Yahoo, Google-Yahoo, and Google-Looksmart, involve web directories. The forth involves parts of the Yahoo and the Standard catalogues which describe business activities. The results obtained for the Looksmart-Yahoo matching problem are depicted in Figure 11a. In this case the trees contain about 100 nodes each. *S-Match* works about 18% faster than *S-Match*$_B$ and about 2 % slower than COMA. SF works about 3 times faster. The relatively poor improvement (18%) can be explained by the fact that our optimizations are

implemented in a straightforward way. The higher implementational constants on small trees (like Looksmart-Yahoo) can overcome the order of growth the complexity function.

Figure 11b reports the results obtained for the Yahoo-Standard matching problem. *S-Match* works about 40% faster than *S-Match*$_B$. It performs 1% faster than COMA and about 5 times slower than SF. The relatively small improvement in this case can be explained by noticing that the maximum depth in both trees is 3 and that the average number of labels at node is about 2. The optimizations can not significantly influence on the system performance.

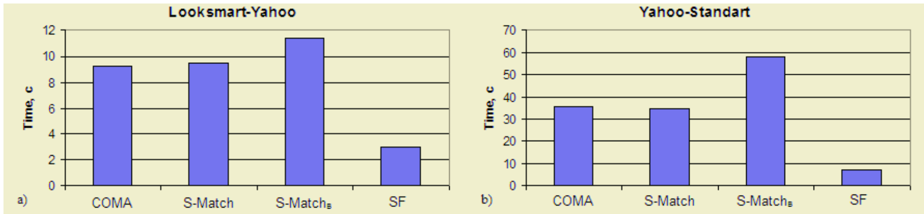



**Fig. 11.** Execution time of the matching systems

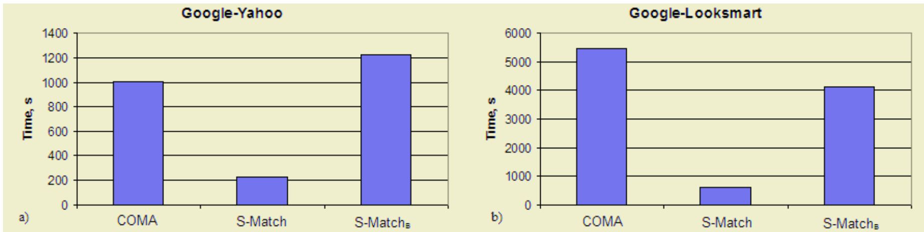



**Fig. 12.** Execution time of the matching systems

The next two matching problems are much bigger than the previous ones. They contain hundreds and thousands of nodes. On these trees SF went out of memory. Therefore, we provide the results only for the other systems. The results are reported in Figure 12a. *S-Match* is more than 6 times faster than *S-Match*$_B$. COMA performs about 5 times slower than the optimized version. These results suggest that the optimizations described in this paper are better suited for big schemas. The results of the biggest matching problem, involving Google-Looksmart, are presented in Figure 12b. In this case *S-Match* performs about 9 times faster than COMA, and about 7 times faster than *S-Match*$_B$.

## 8 Conclusion

We have presented a structure level semantic matching algorithm and proposed several optimizations to its original version. In particular we have distinguished

between two main classes of problems, namely the problems with *conjunctive* and with *disjunctive* concepts at nodes. For the first class of problems we have presented a modification to the original algorithm which solves the node matching problem in linear time. With *disjunctive* concepts we have presented various techniques, which allow us to avoid the exponential space explosion which arises when converting disjunctive formulas into CNF. We have evaluated *S-Match* against several state of the art matching systems and against the original unoptimized version, *S-Match*$_B$. The results thorough preliminary are promising. *S-Match* always performs better than *S-Match*$_B$. Furthermore, in most cases *S-Match* competes well, in terms of time performance, with various state of the art matching systems. Optimizations are most effective on big trees with hundreds and thousands of nodes.

**Acknowledgements.** This work has been partially supported by the European Knowledge Web network of excellence (IST-2004-507482) and by the research grant COFIN 2003 Giunchiglia 40100657.

## References

[1] P. Bouquet, L. Serafini, S. Zanobini. Semantic Coordination: A new approach and an application. In Proceedings of ISWC 2003.

[2] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM*, number 7, pages 201–215, 1960.

[3] H. Do, E. Rahm. COMA - A system for Flexible Combination of Schema Matching Approaches, In Proceedings of VLDB 2002

[4] E. Giunchiglia, R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas . In AIIA'99.

[5] F. Giunchiglia, P. Shvaiko. Semantic Matching. In The Knowledge Engineering Review Journal, 18(3) 2003.

[6] F. Giunchiglia, P. Shvaiko, M. Yatskevich. S-Match: An algorithm and an implementation of semantic matching. In Proceedings of ESWS'04.

[7] F. Giunchiglia and M. Yatskevich. Element level semantic matching. In Proceedings of Meaning Coordination and Negotiation workshop at ISWC, 2004.

[8] D. Le Berre JSAT: The java satisfiability library. http://cafe.newcastle.edu.au/daniel/JSAT/.ï

[9] D. Le Berre SAT4J: A satisfiability library for Java. http://www.sat4j.org/.

[10] J. Madhavan, P. Bernstein, E. Rahm. Generic Schema Matching with Cupid. VLDB 2001

[11] B. Magnini, M. Speranza, C. Girardi. A Semantic-based Approach to Interoperability of classification Hierarchies: Evaluation of Linguistic Techniques. In: Proceedings of COLING-2004, Geneva, Switzerland, August 23-27, 2004.ï

[12] S. Melnik,, H. Garcia-Molina, E. Rahm: Similarity Flooding: A Versatile Graph Matching Algorithm. Proceedings of ICDE, (2002) 117-128.

[13] S. Melnik, E. Rahm, P. Bernstein: Rondo: A programming platform for generic model management. Proceedings of SIGMOD'03, (2003) 193-204.ï

[14] D. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. Journal of Symbolic Computation, 2:293-304, 1986

[15] G. Tsetin. On the complexity proofs in propositional logics. Seminars in Mathematics, 8, 1970

## Appendix A. The Pseudo Code of the Optimized S-Match Algorithm

```
1. Node: struct of
2.        int nodeId;
3.        String label;
4.        String cLabel;
5.        String cNode;

6.String[][] treeMatch(Tree of Nodes source, target)
7. Node sourceNode,targetNode;
8. String[][] cLabsMatrix, cNodesMatrix, relMatrix;
9. String axioms, context_A, context_B;
10.int i,j;
11.cLabsMatrix=fillCLabMatrix(source,target);
12.For each sourceNode in source
13. i=getNodeId(sourceNode);
14. context_A=getCnodeFormula (sourceNode);
15. For each targetNode in target
16.  j=getNodeId(targetNode);
17.  context_B=getCnodeFormula (targetNode);
18.  relMatrix=extractRelMatrix(cLabMatrix,
                         sourceNode, targetNode);
19.  axioms=mkAxioms(relMatrix);
20.  cNodesMatrix[i][j]=nodeMatch(axioms,
                                    context_A, context_B);
21. return cNodesMatrix;


110.String nodeMatch(String axioms, context_A, context_B)
111. if (contextA and contextB are conjunctive)
112.  isLG= fastHornUnsatCheck (context_A, axioms, "⊆")
113.  isMG= fastHornUnsatCheck (context_B, axioms,"⊇")
114. else
120. String formula=And(axioms,context_A,Not(context_B))
130. String formulaInCNF=optimizedConvertToCNF(formula)
140.  boolean isLG=isUnsatisfiable(formula)
150.  formula=And(axioms, Not(context_A), context_B);
160.  formulaInCNF= optimizedConvertToCNF (formula);
170.  boolean isMG= isUnsatisfiable(formula);
180. if (isMG && isLG)
190.  return "=";
200.if (isLG)
210.  return "⊆";
220.if (isMG)
230   return "⊇";
231. If (context_A and context_B are conjunctive)
232.   If (there is disjointness axiom in the axioms)
233.     isOpposite=true;
234.   else
```

```
235.     isOpposite=false;
236. else
240.  formula= And(axioms, context_A, context_B);
250.  formulaInCNF= optimizedConvertToCNF (formula);
260.  boolean isOpposite= isUnsatisfiable(formula);
270.if (isOpposite)
280.  return "⊥";
290.return "Idk";

301.boolean fastHornUnsatCheck(String context, axioms,
                                                    rel)
302. int m=getNumOfVar(String context);
303. boolean array[m];
304. for each axiom in axioms
305.  if((getAType(axiom)="=")||(getAType(axiom)= rel))
306.   int j=getNumberOfSecondVariable(axiom);
307.   array[j]=true;
308.  for (i=0; i<m; i++)
309.   if (!array[i])
310.    return false;
311. return true;
```