Efficient θ-subsumption under Object Identity

Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro and Teresa M.A. Basile
Dipartimento di Informatica
Università di Bari
via E. Orabona, 4 - 70125 Bari - Italia
{ferilli, fanizzi, nicodimauro, basile }@di.uniba.it

Abstract. Efficiency of the first-order logic proof procedure is a major issue when deduction systems are to be used in real environments, both on their own and as a component of other systems (e.g., learning systems). Hence, the need of techniques that can speed up such a process. This paper proposes a proof procedure that works under the Object Identity assumption, and shows experimental results in support of its performance. Imposing the Object Identity bias does not limit expressive power, while turning out to be even more intuitive to the human way of thinking than the classical setting.

1 Introduction

Inductive Logic Programming (ILP) aims at automatically learning logic programs that explain ('*cover*') a given set of positive examples while rejecting other negative ones. In this context, because of the implication relationship being undecidable in general [12], a central role is played by θ-subsumption, used to decide if a rule covers an example as well as to obtain the reduction of clauses. Completeness and consistency checks of new clauses (or clause refinements) against given examples, in particular, require a large amount of subsumption tests. Hence, the availability of an efficient θ-subsumption algorithm heavily affects the overall efficiency of ILP learning systems.

Given C and D clauses, C θ-subsumes D iff there is a substitution θ such that $C\theta \subseteq D$. A substitution is a mapping from variables to terms. It is possible to denote substitutions by $\theta = \{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$. Application of a substitution θ to a clause C, denoted by $C\theta$, rewrites all the occurrences of variables X_i $(i = 1 \ldots n)$ in C by the corresponding term t_i.

A basic algorithm for checking if a clause C θ-subsumes a clause D, having the same predicate in their head, can be obtained in Prolog by skolemizing D, then asserting all the literals in the body of D, and finally querying the head of C. The outcome is computed by Prolog through SLD resolution [10], which can be very inefficient under some conditions.

Various studies were carried out on such a topic, and a number of improvements were proposed. Among the most important we recall those in [7] (based on the concept of *determinate matching*) and in [11] (that uses the *graph context* to extend the scope of determinate matching). A recent algorithm, named *Django* [8], showed the best results. Its authors formalize θ-subsumption as a binary CSP problem and present a new combination of CSP heuristics for

it. The experimental validation of Django outperforms the θ-subsumption algorithms based on determinate matching and graph context in computational cost.

The new matching procedure that we propose in this paper is able to check in an efficient way if a clause θ-subsumes another clause under Object Identity (*OI* for short), and in such a case it outputs the set of all possible substitutions by which it happens. A description of the OI framework can be found in [5]. Here, we just recall some basic definitions.

> **Object Identity assumption**. Within a clause, terms (even variables) denoted with different symbols must be distinct (i.e., they must refer to different objects).

Example 1.1. Under OI assumption, the Datalog clause[1]

$$C = p(X) :- q(X, X), q(Y, a).$$

is an abbreviation for the *Datalog*^{OI} (a logic language resulting from the application of OI to Datalog) clause

$$C_{OI} = p(X) :- q(X, X), q(Y, a) \parallel [X \neq Y], [X \neq a], [Y \neq a].$$

where '$\parallel$' means *and* just like ',' but is used for the sake of readability to put in evidence the inequalities between terms, that are explicitly added to each clause in order to embed the Object Identity assumption into the normal proof procedure.

Definition 1.1 (θ-subsumption under OI). Let C, D be two Datalog clauses. D θ-subsumes C under OI (D θ_{OI}-subsumes C) iff $\exists\sigma$ substitution s.t. $D_{OI}\sigma \subseteq C_{OI}$.

This paper is organized as follows. The next Section argues in support of Object Identity; then, Section 3 presents a proof-procedure for the space of Datalog Horn clauses under such an assumption, while Section 4 shows experimental results concerning its performance. Lastly, Section 5 draws some conclusions and outlines future works.

2 Why Object Identity

It is our strong belief that the Object Identity assumption is built-in the human way of thinking. Put it another way, people think under Object Identity: Whenever they talk about generic objects giving them different dummy names (in fact, what they do is using placeholders - or, more formally, *variables* - for such objects), they implicitly intend to refer to objects that are *different*. Many examples drawn from everyday life clearly support such a claim. For instance, when one says that "X and Y are brothers if they share a common parent", he means (and everybody understands without any difficulty) that X and Y are two different persons, even if it is not explicitly stated. Analogously, when we define a bicycle as "an object made up of, among other components, a wheel X and a wheel Y", anybody listening to us automatically avoids thinking of a mono-cycle as being a bicycle, even if X and Y could well be associated to the very same wheel of the mono-cycle. More and more examples could be made at demand.

Logics should hence reproduce as closely as possible, among others, such a characterizing and fundamental feature. Unfortunately, this is not the case as, again, many examples

[1] Datalog is a logic language that can be seen as the function-free fragment of Prolog [1].

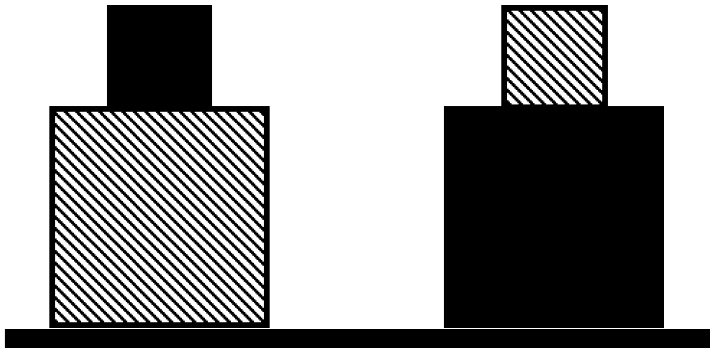

Figure 1: Two structures in a blocks world.

can demonstrate. Let us refer to the First-Order Logics formalism, and in particular to the space of clausal representations ordered by implication or, more specifically, by the widely adopted ordering relationship induced by the generalization model of θ-subsumption. The clause $p(X, X) \vee p(X, Y) \vee p(Y, Z)$ clearly involves more objects than the clause $p(X, X)$, nevertheless they are equivalent under θ-subsumption, since the latter is clearly a subset of the former, and the former may become a subset of (in fact, equal to) the latter through the substitution $\{X/Y, X/Z\}$. Let us show a more practical example. Given the two structures in a blocks world reported in Figure 1, we can represent them as follows:

E_1: `blocks(obj1)` ← `part_of(obj1,p1), part_of(obj1,p2), on(p1,p2), cube(p1), cube(p2), small(p1),big(p2), black(p1), stripes(p2).`

E_2: `blocks(obj2)` ← `part_of(obj2,p3),part_of(obj2,p4), on(p3,p4), cube(p3), cube(p4), small(p3), big(p4), black(p4), stripes(p3).`

Now, according to the algorithm given by Plotkin [9], the *least general generalization* (meaning that no generalization more specific than this one can be found) under θ-subsumption between these two structures is as follows:

G: `blocks(X)` ← `part_of(X,X1), part_of(X,X2), part_of(X,X3), part_of(X,X4), on(X1,X2), cube(X1), cube(X2), cube(X3), cube(X4), small(X1), big(X2), black(X3), stripes(X4).`

As anybody straightforwardly notes, the most specific *generalization* of two structures, involving two objects each, involves *four* objects, which is obviously almost counterintuitive! Any person asked to tell what the two structures have in common would give two alternative answers, that are 'a small cube on a big cube *or* a black cube and a stripes cube', each of which captures only a portion of the correspondences, and specifically those that are consistent with each other. In fact, this last answer is exactly what the least general generalization under Object Identity [13] would yield:

G_1: `blocks(X)` ← `part_of(X,X1), part_of(X,X2), on(X1,X2), cube(X1), cube(X2), small(X1), big(X2).`

G_2: `blocks(X)` ← `part_of(X,X1), part_of(X,X2), cube(X1), cube(X2), black(X1), stripes(X2).`

Such tricky features carry on even when practical issues are taken into account. It has been proved that in the space of clauses ordered by θ-subsumption infinite unbounded strictly ascending/descending chains exist, that avoid the possibility of having ideal refinement operators which in turn are fundamental for the efficiency and effectiveness of the theory revision process [15]. On the contrary, this does not happen when the Object Identity assumption is applied to the same space. A study on the well-known learning system FOIL found that, because of its not fulfilling the Object Identity assumption, problems may arise when trying to infer definitions for particular settings [3].

As a concluding remark, it is worth pointing out that studies on Object Identity revealed many nice properties and features that hold when such a bias is applied [5]. In particular, it is important to underline that such a bias does not limit the expressive power of the representation language, since for any clause/program it is possible to find a set of clauses under Object Identity that are equivalent to it [13].

3 The Matching Procedure

Before discussing our new procedure for computing θ_{OI}-subsumption, it is necessary to preliminarily give some definitions on which the algorithm is based. In the following we will assume C and D to be clauses, such that C is constant-free and D is ground.

Definition 3.1 (Matching substitution [11]). A *matching substitution* from a literal l_1 to a literal l_2 is a substitution μ, such that $l_1\mu = l_2$.

The set of all matching substitutions from a literal $l_i \in C$ to some literal in D is denoted by [2]:

$$uni(C, l_i, D) = \{\mu \mid l_i \in C, l_i\mu \in D\}$$

Let us now define a structure to compactly represent sets of substitutions.

Definition 3.2 (Multibind substitutions).
A *multibind* is denoted by $X \rightarrow T$, where X is a variable and $T \neq \emptyset$ is a set of constants. A *multibind substitution* is a set of multibinds $\Theta = \{X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\} \neq \emptyset$, where $\forall i \neq j : X_i \neq X_j$.

Informally, a multibind identifies a set of constants that can be associated to a variable, while a multibind substitution represents in a compact way a set of possible substitutions for a tuple of variables. In particular, a single substitution is represented by a multibind substitution in which each constants set is a singleton ($\forall i : \mid T_i \mid = 1$).

Definition 3.3 (split). Given a multibind substitution $\Theta = \{X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\}$, *split($\Theta$)* is the set of all substitutions represented by Θ:

$$split(\Theta) = \{\ \{X_1 \rightarrow c_{i_1}, \ldots, X_n \rightarrow c_{i_n}\} \mid \forall k = 1 \ldots n : c_{i_k} \in T_k \wedge i = 1 \ldots |T_k|\}.$$

Example 3.1. $split(\{X \rightarrow \{1,3,4\}, Y \rightarrow \{7\}, Z \rightarrow \{2,9\}\}) =$
$= \{\{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 2\}, \{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 9\}, \{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 2\},$
$\{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 9\}, \{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 2\}, \{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 9\}\}$

Definition 3.4 (Union of multibind substitutions). The union of two multibind substitutions $\Theta' = \{\overline{X} \to T', X_1 \to T_1, \ldots, X_n \to T_n\}$ and $\Theta'' = \{\overline{X} \to T'', X_1 \to T_1, \ldots, X_n \to T_n\}$ is the multibind substitution defined as

$$\Theta' \sqcup \Theta'' = \{\overline{X} \to T' \cup T''\} \cup \{X_i \to T_i\}_{1 \leq i \leq n}$$

Note that the two input multibind substitutions must be defined on the same set of variables and must differ in at most one multibind.

Definition 3.5 (merge). Given a set $\mathcal{S}$ of substitutions on the same variables, *merge($\mathcal{S}$)* is the set of multibind substitutions obtained according to Algorithm 1.

Algorithm 1 merge($\mathcal{S}$)

Require: $\mathcal{S}$: set of substitutions (each represented as a multibind substitution)
  while $\exists u, v \in \mathcal{S}$ such that $u \neq v$ and $u \sqcup v = t$ **do**
    $\mathcal{S} := (\mathcal{S} \setminus \{u, v\}) \cup \{t\}$
  end while
  return $\mathcal{S}$

Example 3.2.
merge($\{\{X \to 1, Y \to 2, Z \to 3\}, \{X \to 1, Y \to 2, Z \to 4\}, (X \to 1, Y \to 2, Z \to 5\}\}$)
= merge($\{\{X \to \{1\}, Y \to \{2\}, Z \to \{3, 4\}\}, \{X \to \{1\}, Y \to \{2\}, Z \to \{5\}\}\}$)
= $\{\{X \to \{1\}, Y \to \{2\}, Z \to \{3, 4, 5\}\}\}$.
This way we can represent 3 substitutions with only one tree of substitutions.

Definition 3.6 (Intersection of multibind substitutions). The intersection of two multibind substitutions $\Sigma = \{X_1 \to S_1, \ldots, X_n \to S_n, Y_1 \to S_{n+1}, \ldots, Y_m \to S_{n+m}\}$ and $\Theta = \{X_1 \to T_1, \ldots, X_n \to T_n, Z_1 \to T_{n+1}, \ldots, Z_l \to T_{n+l}\}$, where $n, m, l \geq 0$ and $\forall j, k : Y_j \neq Z_k$, is the multibind substitution defined as:

$$\Sigma \sqcap \Theta = \{X_i \to S_i \cap T_i\}_{i=1\ldots n} \cup \{Y_j \to S_{n+j}\}_{j=1\ldots m} \cup \{Z_k \to T_{n+k}\}_{k=1\ldots l}$$

iff $\forall i = 1 \ldots n : S_i \cap T_i \neq \emptyset$; otherwise it is undefined.

Lemma 3.1. *The $\sqcap$ operator is monotonic in the set of variables. Specifically, it holds*

$$|\Sigma|, |\Theta| \leq |\Sigma \sqcap \Theta| = n + m + l$$

Proof. The $\sqcap$ operator transposes in the result all the multibinds concerning $Y_j, j = 1 \ldots m$ variables from Σ, and all the multibinds concerning $Z_k, k = 1 \ldots l$ variables from Θ, whose constant sets are all nonempty by definition. Moreover, it preserves all the multibinds concerning $X_i, i = 1 \ldots n$ variables common to Σ and Θ, since all intersections of the corresponding constants sets must be nonempty for the result to be defined. Hence: $n, m, l \geq 0$ and $\forall j, k : Y_j \neq Z_k$ implies that $|\Sigma \sqcap \Theta| = n + m + l$ and both $|\Sigma| = n + m \leq |\Sigma \sqcap \Theta|$ and $|\Theta| = n + l \leq |\Sigma \sqcap \Theta|$. □

The above $\sqcap$ operator is able to check if two multibind substitutions are compatible (i.e., if they share at least one of the substitutions they represent). Indeed, given two multibind substitutions Σ and Θ, if $\Sigma \sqcap \Theta$ is undefined, then there must be at least one variable X, common

to Σ and Θ, to which the corresponding multibinds associate disjoint sets of constants, which means that it does not exist a constant to be associated to X by both Σ and Θ, and hence a common substitution cannot exist as well.

The $\sqcap$ operator can be extended to the case of sets of multibind substitutions. Specifically, given two sets of multibind substitutions $\mathcal{S}$ and $\mathcal{T}$, their intersection is defined as the set of multibind substitutions obtained as follows:

$$\mathcal{S} \sqcap \mathcal{T} = \{\Sigma \sqcap \Theta \mid \Sigma \in \mathcal{S}, \Theta \in \mathcal{T}\}$$

Note that, whereas a multibind substitution (and hence an intersection of multibind substitutions) is or is not defined, but cannot be empty, a set of multibind substitutions can be empty. Hence, an intersection of sets multibind substitutions, in particular, can be empty (which happens when all of its composing intersections are undefined).

Given a clause, multibind, multibind substitution or set of multibind substitutions O, in the following we will denote by $vars(O)$ the set of variables that are present in O.

Proposition 3.2. *Let $C = \{l_1, \ldots, l_n\}$ and $\forall i = 1 \ldots n : \mathcal{T}_i = merge(uni(C, l_i, D))$; let $\mathcal{S}_1 = \mathcal{T}_1$ and $\forall i = 2 \ldots n : \mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i$. C θ-subsumes D iff $\mathcal{S}_n \neq \emptyset$.*

Proof.

($\Rightarrow$) Suppose (ad absurdum) that $\mathcal{S}_n = \emptyset$. Then $\exists \bar{i} \nexists \forall i, j, 1 \leq i < \bar{i} \leq j \leq n : \mathcal{S}_i \neq \emptyset \wedge \mathcal{S}_j = \emptyset$. But then $\mathcal{S}_{\bar{i}} = \mathcal{S}_{\bar{i}-1} \sqcap \mathcal{T}_{\bar{i}} = \emptyset \Rightarrow \mathcal{T}_{\bar{i}} = \emptyset$, i.e. there is no matching substitution for the literal $l_{\bar{i}}$. Thus, C cannot θ-subsume D, which is an absurd since C θ-subsumes D by hypothesis.

($\Leftarrow$) $\mathcal{S}_n \neq \emptyset \Rightarrow \forall i = 1 \ldots n : \mathcal{T}_i \neq \emptyset \wedge \exists \Theta \in \mathcal{S}_n$, Θ multibind substitution.
Now, $\forall \theta \in split(\Theta) : vars(\theta) = vars(\Theta) =$ (being $\sqcap$ monotonic) $\bigcup vars(\mathcal{T}_i) = \bigcup vars(l_i) = vars(\bigcup l_i) = vars(C)$. Each constant c such that $X \rightarrow c \in \theta$, is drawn from a constant set K such that $X \rightarrow K \in \Theta$. K is obtained by construction as the intersection of all constant sets K_i such that $X \rightarrow K_i$ belongs to some $\Sigma_i \in \mathcal{T}_i$, for each $\mathcal{T}_i$ including variable X. Then, it holds $C\theta \subseteq D$, i.e. C θ-subsumes D.

□

This leads to the θ-subsumption procedure reported in Algorithm 2.

Algorithm 2 matching(C, D)

Require: $C : c_0 \leftarrow c_1, c_2, \ldots, c_n$, $D : d_0 \leftarrow d_1, d_2, \ldots, d_m$: clauses
  if $\exists \theta_0$ substitution such that $c_0\theta_0 = d_0$ **then**
    $S_0 := \{\theta_0\}$;
    for i := 1 to n **do**
      S_i := $S_{i-1} \sqcap merge(uni(C, c_i, D))$
    end for
  end if
  return ($S_n \neq \emptyset$)

We now extend the above concepts to obtain substitutions such that C θ-subsumes D under Object Identity (C θ_{OI}-subsumes D).

Definition 3.7 (Injectivity).
A substitution $\{X_1 \rightarrow c_1, \ldots, X_n \rightarrow c_n\}$ is *injective* iff $\forall i \neq j : c_i \neq c_j$.
A multibind substitution Θ is *injective* iff $\exists \theta \in split(\Theta)$, and θ is injective.

Definition 3.8 (Intersection of multibind substitutions under Object Identity). The intersection of two multibind substitutions under Object Identity (denoted by $\sqcap_{OI}$) is defined as in Definition 3.6, but with the additional requirement that the result must be an injective multibind substitution; otherwise, it is undefined.

A result analogous to Proposition 3.2 holds under Object Identity as well:

Proposition 3.3. *Let* $C = \{l_1, \ldots, l_n\}$. *Let us define* $\forall i = 1 \ldots n : \mathcal{T}_i = merge(uni(C, l_i, D))$; $\mathcal{S}_1 = \mathcal{T}_1$ *and* $\forall i = 2 \ldots n : \mathcal{S}_i = \mathcal{S}_{i-1} \sqcap_{OI} \mathcal{T}_i$. C θ_{OI}*-subsumes* D *iff* $\mathcal{S}_n \neq \emptyset$.

Proof. Analogous to Proposition 3.2, but choosing for θ_{OI}-subsumption exactly the injective substitution (that exists by hypothesis). □

Replacing in Algorithm 2 the $\sqcap$ operator with the $\sqcap_{OI}$ operator yields an algorithm that computes the θ_{OI}-subsumption between two clauses.

4 Empirical results

In order to verify its efficiency, our algorithm was compared to Django, that, as already stated, had in turn been compared to the other algorithms by its authors. However, the comparison was not straightforward since Django performs a θ-subsumption test, whereas our procedure is designed to work under θ_{OI}-subsumption. To overcome such difficulty, we modified the body of Django input clauses by adding literals that force it to fulfill the OI assumption. In particular, given a clause C, we added to its body a literals $diff(t, t')$ (meaning that the two terms must be different) and a literal $diff(t', t)$ (expressing that difference is a symmetric relationship)[2] for all possible pairs of terms t and t' in C. Note that the above inequalities have to be stated also among constants (even if Django already ensures them), in order to allow subsumption. In fact, Django (as well as our procedure) requires one of the two clauses to be a rule (and hence to contain variables only), and the other to be an example (and hence to contain constants only).

Example 4.1. The clause

```
atom(X) ← bond(X,Y,Z), bond(X,Z,W), bond(X,W,Y).
```

is input to Django augmented by the Object Identity constraints
`diff(X,Y),diff(X,Z),diff(X,W),diff(Y,Z),diff(Y,W),diff(Z,W)`
while the example

```
atom(a) ← bond(a,b,c), bond(a,c,d).
```

is input to Django augmented by the Object Identity constraints
`diff(a,b),diff(b,a),diff(a,c),diff(c,a),diff(a,d),diff(d,a),`
`diff(b,c),diff(c,b),diff(b,d),diff(d,b),diff(c,d),diff(d,c).`

[2]Literals expressing these symmetric inequalities were introduced exclusvely in the representation of the examples. In rule clauses, it suffices to include one of them since this does not affect (the correctness of) the matching algorithm.

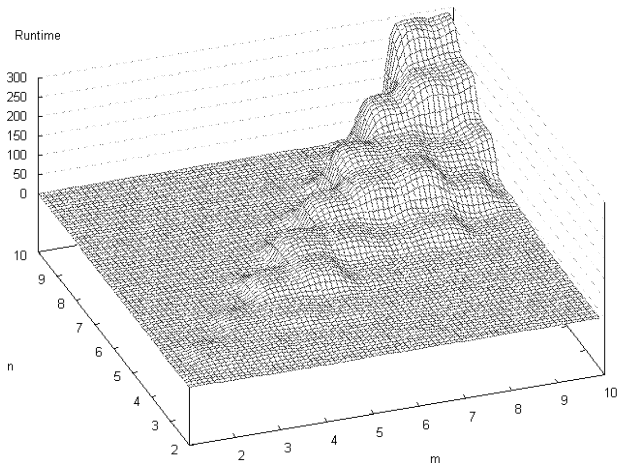

Figure 2: θ_{OI}-subsumption performance for Django (msecs).

The experiment for testing the algorithm efficiency is derived from the standard dataset concerning the real-world problem of Mutagenesis [14]. Artificial hypotheses were generated according to the procedure reported in [8]. For given m and n, such a procedure returns an hypothesis made up of m literals $bond(X_i, X_j)$ and involving n variables, where the variables X_i and X_j in each literal are randomly selected among n variables $\{X_1, \ldots, X_n\}$ in such a way that $X_i \neq X_j$ and the overall hypothesis is linked [6]. The cases in which $n > m+1$ were not considered, since it is not possible to build a clause with m binary literals that contains more than $m+1$ variables and that fulfills the linkedness constraint imposed by the previously described construction method.

Specifically, for each (m, n) pair ($1 \leq m \leq 10$, $2 \leq n \leq 10$), 10 artificial hypotheses were generated and each was checked against all 229 examples provided in the Mutagenesis dataset. Then, the mean performance of each hypothesis on the 229 examples was computed, and finally the computational cost for each (m, n) pair was obtained as the average θ_{OI}-subsumption cost over all the times of the corresponding 10 hypotheses. The experiments were performed on a PC platform equipped with an Intel PentiumIII 800 MHz processor and running the Linux operating system. Figures 2 and 3 report the performance obtained by Django and by our algorithm (respectively) on the θ_{OI}-subsumption tests for the Mutagenesis dataset. Figure 4 shows the difference between the two performances. All timings are measured in milliseconds.

It is straightforward to note that both algorithms show the worst performance in the region around the diagonal, corresponding to hypotheses with i literals and $i + 1$ variables. Such hypotheses are particularly challenging for the θ_{OI}-subsumption test since their literals form a chain of variables (because of linkedness). However, while Django has a steady and continuous rise of the computational cost as long as n and m grow, our matching algorithm, although performing slightly worse on the easiest problems (the region on the right-hand side of the diagonal), shows a neat improvement in computational times (as the smoother shape and lower peaks of the plot in Figure 3 demonstrate). It should also be noted that the critical region (identified by higher peaks) is located at higher (m, n) pairs than Django, and that the increase rate is lower than Django (actually, it even shows a decrease for $m = n = 10$).

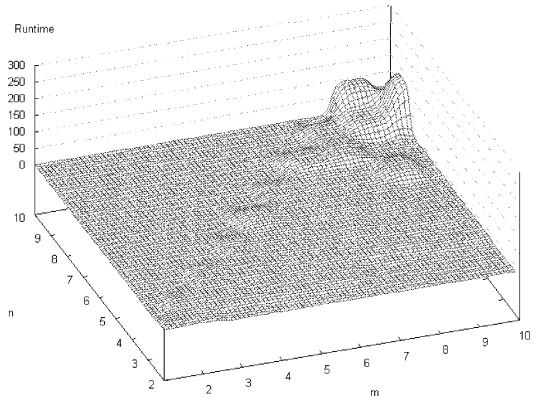

Figure 3: θ_{OI}-subsumption performance for Matching (msecs).

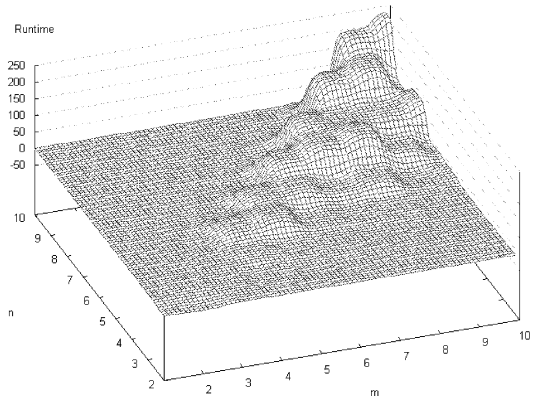

Figure 4: Difference in performance between Django and Matching (msecs).

5 Conclusions and Future Work

We proposed a new algorithm for checking θ_{OI}-subsumption. Preliminary results suggest that it is able to improve the time performance with respect to other state-of-the-art systems. The first prototype of the algorithm, implemented in Prolog, is currently used in *INTHELEX* [4], a system for inductive learning from examples based on the Object Identity assumption, that is employed in the EU project COLLATE to learn rules for classification and interpretation of historical archive material. The current implementation uses the same search strategy as SLD resolution. Efficiency of the algorithm could be improved implementing it in a procedural language, such as C, and using heuristics that help in choosing the best literal to take into account at any step.

Future work will concern a more extensive experimentation, using other and more challenging datasets, aimed at carrying out a better analysis of the complexity of the presented algorithm.

Acknowledgements.

This work was partially funded by the EU project IST-1999-20882 COLLATE "Collaboratory for Annotation, Indexing and Retrieval of Digitized Historical Archive Material".

References

[1] S. Ceri, G. Gottlöb, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Heidelberg, Germany, 1990.

[2] N. Eisinger. Subsumption and connection graphs. In J. H. Siekmann, editor, *GWAI-81, German Workshop on Artificial Intelligence, Bad Honnef, January 1981*, pages 188–198. Springer, Berlin, Heidelberg, 1981.

[3] F. Esposito, D. Malerba, G. Semeraro, C. Brunk, and M. Pazzani. Traps and pitfalls when learning logical definitions from relations. In Z. W. Raś and M. Zemankova, editors, *Methodologies for Intelligent Systems*, number 869 in Lecture Notes in Artificial Intelligence, pages 376–385. Springer-Verlag, 1994.

[4] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy Theory Revision: Induction and abduction in INTHELEX. *Machine Learning Journal*, 38(1/2):133–156, 2000.

[5] S. Ferilli. *A Framework for Incremental Synthesis of Logic Theories: An Application to Document Processing*. Ph.D. thesis, Dipartimento di Informatica, Università di Bari, Bari, Italy, November 2000.

[6] N. Helft. Inductive generalization: A logical framework. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning*, pages 149–157, Wilmslow, UK, 1987. Sigma Press.

[7] Jörg-Uwe Kietz and Marcus Lübbe. An efficient subsumption algorithm for inductive logic programming. In W. Cohen and H. Hirsh, editors, *Proc. Eleventh International Conference on Machine Learning (ML-94)*, pages 130–138, 1994.

[8] J. Maloberti and M. Sebag. θ-subsumption in a constraint satisfaction perspective. In Céline Rouveirol and Michèle Sebag, editors, *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 164–178. Springer, September 2001.

[9] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[10] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.

[11] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient θ-subsumption based on graph algorithms. In Stephen Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming (ILP-96)*, volume 1314 of *LNAI*, pages 212–228, Berlin, August 26–28 1997. Springer.

[12] M. Schmidt-Schauss. Implication of clauses is undecidable. *Theoretical Computer Science*, 59:287–296, 1988.

[13] G. Semeraro, F. Esposito, D. Malerba, N. Fanizzi, and S. Ferilli. A logic framework for the incremental inductive synthesis of datalog theories. In N. E. Fuchs, editor, *Logic Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 300–321. Springer-Verlag, 1998.

[14] Ashwin Srinivasan, Stephen Muggleton, Michael J. E. Sternberg, and Ross D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

[15] P. R. J. van der Laag. *An Analysis of Refinement Operators in Inductive Logic Programming*. PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1995.