An Exhaustive Matching Procedure for the Improvement of Learning Efficiency

Nicola Di Mauro, Teresa Maria Altomare Basile, Stefano Ferilli, Floriana Esposito, and Nicola Fanizzi

Dipartimento di Informatica, Università di Bari
via E. Orabona, 4 - 70125 Bari - Italia
{nicodimauro,asile,ferilli,esposito,fanizzi}@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 larger systems (e.g., learning systems). Hence, the need of techniques that can speed up such a process. This paper proposes a new algorithm for matching first-order logic descriptions under θ-subsumption that is able to return the set of all substitutions by which such a relation holds between two clauses, and shows experimental results in support of its performance.

1 Introduction

The induction of logic theories intensively relies on the use of a covering procedure that essentially computes whether a candidate concept definition (a *hypothesis*) explains a given example. This is the reason why the covering procedure should be carefully designed for an efficient hypothesis evaluation, which in turn means that the underlying matching procedure according to which two descriptions are compared must *a fortiori* fulfill the very same requirement. When the underlying generalization model is θ-subsumption, which is common for the standard ILP setting [11,12], the complexity of the evaluation is known to be NP-hard [4], unless biases are applied to make the problem tractable.

If ILP algorithms have to be applied to real-world problems the efficiency of the matching procedure is imperative, especially when it has to return (some of) these substitutions in case of success (e.g., when they are to be used for performing a resolution step). A prototypical case is represented by the domain of document understanding, in which the learning task concerns the induction of rules for the detection of the roles and relationships between the numerous logic parts that make up a structured document layout. By testing the performance of ILP algorithms in this domain, it can be proven that in this and in similar tasks, where many objects and relations are involved, the ILP systems easily show their poor efficiency since the computational time grows exponentially, as expected for the worst case. Indeed, when profiling the execution of our learning system INTHELEX [3] on instances of that task, it was clear that the embedded matching procedure was to blame as the main source of inefficiency.

T. Horváth and A. Yamamoto (Eds.): ILP 2003, LNAI 2835, pp. 112–129, 2003.

Specifically, the matching procedure embedded in INTHELEX simply exploited the Prolog built-in SLD resolution: indeed, the θ-subsumption test can be cast as a refutation of the hypothesis plus the example description added with the negation of the target classification. The proof is computed through various SLD resolution steps [9], which can be very inefficient in some cases. For instance, given the following two clauses:

$\mathtt{h}(X)\ \mathtt{:-}\ \mathtt{p}(X,X_1),\mathtt{p}(X,X_2),\ldots,\ \mathtt{p}(X,X_n),\mathtt{q}(X_n).$

$\mathtt{h}(c)\ \mathtt{:-}\ \mathtt{p}(c,c_1),\mathtt{p}(c,c_2),\ldots,\mathtt{p}(c,c_m).$

SLD-resolution will have to try all m^n possible mappings (backtrackings) before realizing that the former does not match the latter because of the lack of property q. Thus, the greater n and m, the sooner it will not be able to compute subsumption between the two clauses within acceptable time[1]. It is clear that, in real-world settings, situations like this are likely to happen, preventing a solution to the problem at hand from being found. Needless to say, this would affect the whole learning task. Indeed, the absence of just one matching solution could stop a whole deduction, that in turn might be needed to reach the overall solution and complete the experimental results on which drawing the researcher conclusions.

The above consideration motivated the decision to find a more efficient algorithm for matching, so to cope with difficult learning problems both in artificial and in real domains. It should be highlighted that, for our system purposes, the procedure must solve a search problem, that is more complex than a mere θ-subsumption test, since it is mandatory that every possible substitution is computed and returned. There are two unnegligible motivation for this. The former is that the specializing operator relies on this information to find refinements that are able to rule out a problematic negative example while still covering all the past positive ones [2]. The latter is that the saturation and abstraction operators embedded in INTHELEX require resolution to find all possible instances of sub-concepts hidden in the given observations, in order to explicitly add them to the example descriptions. Further reasons are related to the fact that a clause covering an example in many ways could give a higher confidence in the correct classification of the example itself. Besides, most ILP learning algorithms require a more complex matching procedure that solves a search problem rather than a simple decision problem. Indeed, while the mere covering would suffice for a naive *generate-and-test* algorithm, in general, the induction of new candidate hypotheses may require the calculation of one or all the substitutions that lead a concept definition to explain an example. Finding all substitutions for matching has been investigated in other fields, such as production systems, theorem proving, and concurrent/parallel implementations of declarative languages. For instance, in functional programming search procedures can be often formalized as functions yielding lists as values. As Wadler [17] points out, if we make the result of a function a list, and regard the items of this list as multiple results, lazy evaluation provides the counterpart of backtracking. In automated theorem

[1] In some complex cases of document understanding the Prolog SLD resolutions required by a single refutation have been observed to last for over 20 days!

proving, *tableaux calculi* have been introduced such as the Confluent Connection Calculus where backtracking is avoided.

We looked at the available literature on the subject, and found out that a recent system available was Django, kindly provided to us by its Authors, but unfortunately it did not fit our requirements of returning the matching solutions. As to the previous algorithms, they were not suitable as well, since they return just one solution at a time, thus requiring backtracking for collecting all solutions. But blind backtracking is just the source of the above inefficiency, thus our determination has been to definitely avoid it, and having a matching procedure that always goes forward without loosing information. No such thing was available in the literature, and this forced us to build one ourselves.

This paper is organized as follows. The next section presents related work in this field; then, Section 3 presents a new matching algorithm, while Section 4 shows experimental results concerning its performance. Lastly, Section 5 draws some conclusions and outlines future work.

2 Related Work

The great importance of finding efficient algorithms for matching descriptions under θ-subsumption is reflected by the amount of work carried out so far in this direction in the literature. The latest results have been worked out by Maloberti and Sebag in [10], where the problem of θ-subsumption is faced by means of a Constraint Satisfaction Problem (CSP) approach.

Briefly, a CSP involves a set of variables $X_1, \ldots, X_n$, where each X_i has domain $dom(X_i)$, and a set of constraints, specifying the simultaneously admissible values of the variables. A CSP solution assigns to each X_i a value $a_i \in dom(X_i)$ such that all constraints are satisfied. CSP algorithms exploit two kinds of heuristics. *Reduction* heuristics aim at transforming a CSP into an equivalent CSP of lesser complexity by pruning the candidate values for each variable (local consistency). *Search* heuristics are concerned with the backtracking procedure (*look-back* heuristics) and with the choice of the next variable and candidate value to consider (*look-ahead* heuristics): a value is chosen for each variable and then consistency is checked. Since there is no universally efficient heuristic in such a setting, different combinations thereof may be suited to different situations.

θ-subsumption is mapped onto a CSP by transforming each literal involved in the hypothesis into a CSP variable with proper constraints encoding the θ-subsumption structure. Specifically, each CSP variable X_p corresponds to a literal in clause C built on predicate symbol p, and $dom(X_p)$ is taken as the set of all literals in the clause D built on the same predicate symbol. A constraint $r(X_p, X_q)$ is set on a variable pair (X_p, X_q) iff the corresponding literals in C share a variable. Given such a representation, different versions of a correct and complete θ-subsumption algorithm, named *Django*, were built, each implementing different (combinations of) CSP heuristics. Some exploited look-ahead heuristics to obtain a failure as soon as possible. Others were based on forward checking (propagation of forced assignments — i.e., variables with a single candidate value) and arc-consistency (based on the association, to each pair variable-candidate value, of a *signature* encoding the literal links — i.e., shared variables — with all other literals).

Experiments in hard artificial domains are reported, proving a difference in performance of several orders of magnitude in favor of Django compared to previous algorithms. In sum, all the work carried out before Django (namely, by Gottlob and Leitsch [6], by Kiets and Lübbe [8] and by Scheffer, Herbrich and Wysotzki [13]) aimed at separating the part of the clause for which finding a matching is straightforward or computationally efficient, limiting as much as possible the complexity of the procedure. All the corresponding techniques rely on backtracking, and try to limit its effect by properly choosing the candidates in each tentative step. Hence, all of them return only the first matching substitution found, even if many exist. On the contrary, it is important to note that Django only gives a binary (*yes* or *no*) answer to the subsumption test, without providing any matching substitution in case of positive outcome.

3 A New Matching Algorithm

Ideas presented in related work aimed, in part, at leveraging on particular situations in which the θ-subsumption test can be computed with reduced complexity. However, after treating efficiently the subparts of the given clauses for which it is possible, the only way out is applying classical, complex algorithms, possibly exploiting heuristics to choose the next literal to be unified. In those cases, the CSP approach proves very efficient, but at the cost of not returning (all the) possible substitutions by which the matching holds. Actually, there are cases in which at least one such substitution is needed by the experimenter. Moreover, if *all* such substitutions are needed (e.g., for performing successive resolution steps), the feeling is that the CSP approach has to necessarily explore the whole search space, thus loosing all the advantages on which it bases its efficiency. The proposed algorithm, on the contrary, returns *all* possible matching substitutions, without performing any backtracking in their computation. Such a feature is important, since the found substitutions can be made available to further matching problems, thus allowing to perform resolution.

Before discussing the new procedure proposed in this paper, it is necessary to preliminarily give some definitions on which the algorithm is based. In the following, we will assume that C and D are Horn clauses having the same predicate in their head, and that the aim is checking whether a matching exists between C and D, i.e. if C θ-subsumes D, in which case we are interested in *all* possible ways (substitutions) by which it happens. Note that D can always be considered ground (i.e., variable-free) without loss of generality. Indeed, in case it is not, a new corresponding clause D' can be obtained by replacing each of its variables by a new constant not appearing in C nor in D, and it can be proven that C θ-subsumes D iff C θ-subsumes D'.

Definition 1 (Matching Substitution). *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 $l \in C$ to some literal in D is denoted by [1]:

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

Let us start by defining a structure to compactly represent sets of substitutions.

Definition 2 (Multi-substitutions). *A* multibind *is denoted by $X \rightarrow T$, where X is a variable and $T \neq \emptyset$ is a set of constants. A* multi-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 *multi-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 multi-substitution in which each constants set is a singleton.

Example 1. $\Theta = \{X \rightarrow \{1, 3, 4\}, Y \rightarrow \{7\}, Z \rightarrow \{2, 9\}\}$ is a multi-substitution. It contains 3 multibinds, namely: $X \rightarrow \{1, 3, 4\}$, $Y \rightarrow \{7\}$ and $Z \rightarrow \{2, 9\}$.

Given a multi-substitution, the set of all substitutions it represents can be obtained by choosing in all possible ways one constant for each variable among those in the corresponding multibind.

Definition 3 (Split). *Given a multi-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 2. $split(\{X \rightarrow \{1, 3, 4\}, Y \rightarrow \{7\}, Z \rightarrow \{2, 9\}\}) =$
$\{\{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 2\}$, $\{X \rightarrow 1, Y \rightarrow 7, Z \rightarrow 9\}$, $\{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 2\}$, $\{X \rightarrow 3, Y \rightarrow 7, Z \rightarrow 9\}$, $\{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 2\}$, $\{X \rightarrow 4, Y \rightarrow 7, Z \rightarrow 9\}\}$.

Definition 4 (Union of Multi-substitutions). *The union of two multi-substitutions $\Theta' = \{\overline{X} \rightarrow T', X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\}$ and $\Theta'' = \{\overline{X} \rightarrow T'', X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\}$ is the multi-substitution defined as*

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

Informally, the *union* $\sqcup$ of two multi-substitutions that are identical but for the multibind referred to one variable is a multi-substitution that inherits the common multibinds and associates to the remaining variable the union of the corresponding sets of constants in the input multi-substitutions. Note that the two input multi-substitutions must be defined on the same set of variables and must differ in at most one multibind.

Example 3. The union of two multi-substitutions
$\Sigma = \{X \rightarrow \{1, 3\}, Y \rightarrow \{7\}, Z \rightarrow \{2, 9\}\}$
and $\Theta = \{X \rightarrow \{1, 4\}, Y \rightarrow \{7\}, Z \rightarrow \{2, 9\}\}$,
is: $\Sigma \sqcup \Theta = \{X \rightarrow \{1, 3, 4\}, Y \rightarrow \{7\}, Z \rightarrow \{2, 9\}\}$
(the only different multibinds being those referring to variable X).

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

```
Require: $\mathcal{S}$: set of substitutions (each represented as a multi-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 $\mathcal{S}$
```

Definition 5 (Merge). *Given a set $\mathcal{S}$ of substitutions on the same variables,* merge($\mathcal{S}$) *is the set of multi-substitutions obtained according to Algorithm 1.*

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

The merge procedure is in charge of compressing many substitutions into a smaller number of multi-substitutions. It should be noted that there are cases in which a set of substitutions cannot be merged at all; moreover, the set of multi-substitutions resulting from the merging phase could be not unique. In fact, it may depend on the order in which the two multi-substitutions to be merged are chosen at each step.

Example 5. Let us consider the following substitutions:
$\theta = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 3\}$ $\quad\delta = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 4\}$
$\sigma = \{X \leftarrow 1, Y \leftarrow 2, Z \leftarrow 5\}$ $\quad\tau = \{X \leftarrow 1, Y \leftarrow 5, Z \leftarrow 3\}$
One possible merging sequence is $(\theta \sqcup \delta) \sqcup \sigma$, that prevents further merging τ and yields the following set of multi-substitutions:
$\{\{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{3,4,5\}\},\ \{X \leftarrow \{1\}, Y \leftarrow \{5\}, Z \leftarrow \{3\}\}\}$
Another possibility is first merging $\theta \sqcup \tau$ and then $\delta \sqcup \sigma$, that cannot be further merged and hence yield:
$\{\{X \leftarrow \{1\}, Y \leftarrow \{2,5\}, Z \leftarrow \{3\}\},\ \{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{4,5\}\}\}$

The presented algorithm does not currently specify any particular principle according to which performing such a choice, but this issue is undoubtedly a very interesting one, and deserves a specific study (that would require a paper on its own) in order to understand if the quality of the result is actually affected by the ordering and, in such a case, if there are heuristics that can suggest in what order the multi-substitutions to be merged have to be taken in order to get an optimal result. Actually, many of such heuristics, often clashing with each other, can be developed according to the intended behavior of the system: for instance, some heuristics can be used to make the system faster in recognizing negative outcomes to the matching problem; others can be exploited to optimize the intermediate storage requirements when the matching exists. Preliminary tests on this issue revealed that sorting the constants in the multibinds, in addition to make easier the computation of their union and intersection, also generally leads to better compression with respect to the random case. Nevertheless, further insight could suggest better strategies that ensure the best compression (overall or at least on average).

Definition 6 (Intersection of Multi-substitutions). *The intersection of two multi-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 multi-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.*

Informally, the *intersection* $\sqcap$ of two multi-substitutions is a multi-substitution that inherits the multibinds concerning variables appearing in either of the starting multi-substitutions, and associates to each variable occurring in both the input multi-substitutions the intersection of the corresponding sets of constants (that is required not to be empty) in the input multi-substitutions.

Example 6. The intersection of two multi-substitutions
$\Sigma = \{X \to \{1,3,4\}, Z \to \{2,8,9\}\}$ and $\Theta = \{Y \to \{7\}, Z \to \{1,2,9\}\}$
is: $\Sigma \sqcap \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$.
The intersection of $\Sigma = \{X \to \{1,3,4\}, Z \to \{8,9\}\}$ and $\Theta = \{Y \to \{7\}, Z \to \{1,2\}\}$ is undefined.

The above $\sqcap$ operator is able to check if two multi-substitutions are compatible (i.e., if they share at least one of the substitutions they represent). Indeed, given two multi-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 multi-substitutions. Specifically, given two sets of multi-subistitutions $\mathcal{S}$ and $\mathcal{T}$, their intersection is defined as the set of multi-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 multi-substitution (and hence an intersection of multi-substitutions) is or is not defined, but cannot be empty, a set of multi-substitutions can be empty. Hence, an intersection of sets of multi-substitutions, in particular, can be empty (which happens when all of its composing intersections are undefined).

Proposition 1. *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* $\mathcal{S}_n \neq \emptyset$*.*

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 $(S_n \neq \emptyset)$
```

Proof.

(⇐) Let us prove (by induction on i) the thesis in the following form:
$\forall i \in \{1, \ldots, n\} : \mathcal{S}_i \neq \emptyset \Rightarrow \{l_1, \ldots, l_i\} \leq_\theta D$ (i.e., $\exists\theta$ s.t. $\{l_1, \ldots, l_i\}\theta \subseteq D$).
$[i = 1\ ]$ $\emptyset \neq \mathcal{S}_1 = \mathcal{T}_1 \Rightarrow \forall\Theta \in \mathcal{T}_1, \forall\theta \in split(\Theta) : \exists k \in D$ s.t. $l_1\theta = k \in D \Rightarrow \{l_1\}\theta = \{k\} \subseteq D \Rightarrow \{l_1\} \leq_\theta D$.
$[(i-1) \Rightarrow i\ ]$ $\mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i \neq \emptyset \Rightarrow^{def} \exists\Sigma \in \mathcal{S}_{i-1}, \Theta \in \mathcal{T}_i$ s.t. $\Sigma \sqcap \Theta$ defined $\Rightarrow \exists\sigma \in split(\Sigma), \theta \in split(\Theta) : \sigma, \theta$ compatible $\Rightarrow \{l_1, \ldots, l_{i-1}\}\sigma \subseteq D$ (by hypothesis) $\wedge\{l_i\}\theta \subseteq D$ (by definition of $\mathcal{T}_i$) $\Rightarrow \{l_1, \ldots, l_{i-1}\}\sigma \cup \{l_i\}\theta \subseteq D \Rightarrow \{l_1, \ldots, l_i\}\sigma\theta \subseteq D$.
This holds, in particular, for $i = n$, which yields the thesis.

(⇒) First of all, note that $\forall i = 1, \ldots, n : \mathcal{T}_i \neq \emptyset$. Indeed, (*ad absurdum*) $\exists\bar{i} \in \{1, \ldots, n\}$ s.t. $\mathcal{T}_{\bar{i}} = \emptyset \Rightarrow \exists l_{\bar{i}} \in C$ s.t. $merge(uni(C, l_{\bar{i}}, D)) = \emptyset \Rightarrow uni(C, l_{\bar{i}}, D) = \emptyset \Rightarrow \nexists\theta, \nexists k \in D$ s.t. $l_{\bar{i}}\theta = k \Rightarrow C \not\leq_\theta D$ (*Absurd!*).
Suppose (*ad absurdum*) that $\mathcal{S}_n = \emptyset$. Then $\exists\bar{i}$ s.t. $\forall i, j, 1 \leq i < \bar{i} \leq j \leq n : \mathcal{S}_i \neq \emptyset \wedge \mathcal{S}_j = \emptyset$. But then $\mathcal{S}_{\bar{i}} = \mathcal{S}_{\bar{i}-1} \sqcap \mathcal{T}_{\bar{i}} = \emptyset$, which implies that no substitution represented by $\mathcal{T}_{\bar{i}}$ is compatible with any substitution represented by $\mathcal{S}_{\bar{i}-1}$. Hence, while clause $\{l_1, \ldots, l_{\bar{i}-1}\}$ θ-subsumes D ($\forall\theta \in split(\Sigma)$, $\forall\Sigma \in \mathcal{S}_{\bar{i}-1}$), the clause obtained by adding to it literal $l_{\bar{i}}$ does not. Thus, C cannot θ-subsume D, which is an absurd since it happens by hypothesis.

This leads to the θ-subsumption procedure reported in Algorithm 2. It is worth explicitly noting that, if the number of substitutions by which clause C subsumes clause D grows exponentially, and these items are such that no merging can take place at all, it follows that exponential space will be required to keep them all. In this case there is no representation gain (remember that we are anyway forced to compute all solutions), but one could wonder how often this happens in real world (i.e., in non purposely designed cases). Moreover, as a side-effect, in this case the merge procedure will not enter the loop, thus saving considerable amounts of time.

Example 7. Let us trace the algorithm on $C = h : -p(X1, X2), r(X1, X2).$ and $D = h : -p(a, b), p(c, d), r(a, d)$. The two heads match by the empty substitution, thus S_0 is made up of just the empty substitution.
i = 1: $uni(C, p(X1, X2), D) = \{\{X1/a, X2/b\}, \{X1/c, X2/d\}\}$ that, represented as multi-substitutions, becomes: $\{\{X1 \rightarrow \{a\}, X2 \rightarrow \{b\}\}, \{X1 \rightarrow$

Table 1. Merge statistics on random problems

	Substitutions	Multi-Substitutions	Compression
Average	176,0149015	43,92929293	0,480653777
St-Dev	452,196913	95,11695761	0,254031445
Min	1	1	0,001600512
Max	3124	738	1

$\{c\}, X2 \rightarrow \{d\}\}\}$. Now, since the union between these two substitutions is not defined, the *while* loop is not entered and the *merge* procedure returns the same set unchanged: $merge(uni(C, p(X1, X2), D)) = \{\{X1 \rightarrow \{a\}, X2 \rightarrow \{b\}\}, \{X1 \rightarrow \{c\}, X2 \rightarrow \{d\}\}\}$. By intersecting such a set with S_0, we obtain $S_1 = \{\{X1 \rightarrow \{a\}, X2 \rightarrow \{b\}\}, \{X1 \rightarrow \{c\}, X2 \rightarrow \{d\}\}\}$.
i = 2: $uni(C, r(X1, X2), D) = \{\{X1/a, X2/d\}\}$ that, represented as multi-substitutions, becomes: $\{\{X1 \rightarrow \{a\}, X2 \rightarrow \{d\}\}\}$. It is just one multi-substitution, thus *merge* has no effect again: $merge(uni(C, r(X1, X2), D)) = \{\{X1 \rightarrow \{a\}$,
$X2 \rightarrow \{d\}\}\}$. By trying to intersect such a set with S_1, the algorithm fails because its intersections are both undefined, since $\{b\} \cap \{d\} = \emptyset$ for $X2$ in the former, and $\{a\} \cap \{c\} = \emptyset$ for $X1$ in the latter. Hence, C does not theta-subsumes D, since $S_2 = \emptyset$.

4 Experiments

Some preliminary artificial experiments were specifically designed to assess the compression power of multi-substitutions. First of all, 10000 instances of the following problem were generated and run: fixed at random a number of variables $n \in \{2, \ldots, 9\}$ and a number of constants $m \in \{2, \ldots, 5\}$, among all the possible m^n corresponding substitutions l were chosen at random and merged. Table 1 reports various statistics about the outcome: in particular, the last column indicates the compression factor, calculated as the number of multi-substitutions over the number of substitutions. It is possible to note that, on average, the multi-substitution structure was able to compress the 52% of the input substitutions, with a maximum of 99, 84%. As expected, there were cases in which no merging took place.

Two more experiments were designed, in order to better understand if (and how) the compression depends on the number of variables and constants. The reported results correspond to the average value reached by 33 repetitions of each generated problem. First we fixed the number of variables to 2, letting the number of constants m range from 2 to 10. For each problem $(2, m)$ and for each $i \in \{1, \ldots, m^2\}$, i substitutions were randomly generated and merged. Figure 1 reports the average number of multi-substitutions and the corresponding compression factor in these cases. Conversely, Figure 2 reports the average and the compression factor for a similar artificial problem in which the number of constants was set to 2, and the number of variables n was varied from 2 to 10.

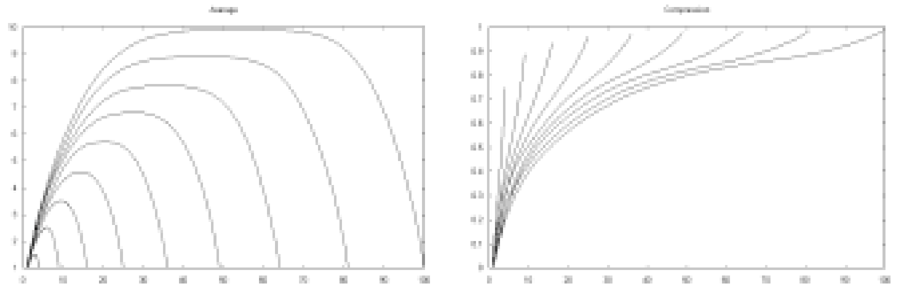

Fig. 1. Average number of multi-substitutions and compression factor for an artificial problem with 2 variables and $m \in \{2, \ldots, 10\}$ constants

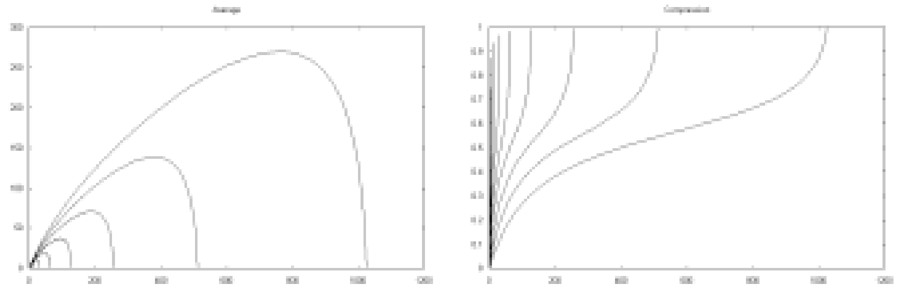

Fig. 2. Average number of multi-substitutions and compression factor for an artificial problem with 2 constants and $n \in \{2, \ldots, 10\}$ variables

The x-axis of the plots corresponds to the number of substitutions generated, the y-axis corresponds to the computed statistic. The curves in the two cases are similar, suggesting a stable (i.e., depending only on the percentage of the whole set of possible substitutions that is taken into account) compression behavior of the merge. Of course, growing parameters (represented by successive curves from left to right) lead to progressively higher values.

Then, a prototype of the proposed algorithm was implemented in Prolog, and integrated with INTHELEX (an incremental system for first-order logic learning from examples, refer to [3] for a more detailed description) in order to make it able to handle the problematic situation occurred in the document dataset. Note that, since INTHELEX relies on the Object Identity (OI) assumption [14], the Prolog implementation slightly modifies the algorithm in order to embed such a bias (the same effect is obtained in SLD-resolution by explicitly adding to each clause the inequality literals expressing OI). To check if any performance gain was obtained, INTHELEX was run on the document interpretation learning problem first exploiting the classical SLD-resolution procedure provided by Prolog (with proper inequality literals encoding the OI assumption), and then using the new algorithm (modified to embed OI). Note that our primary interest was

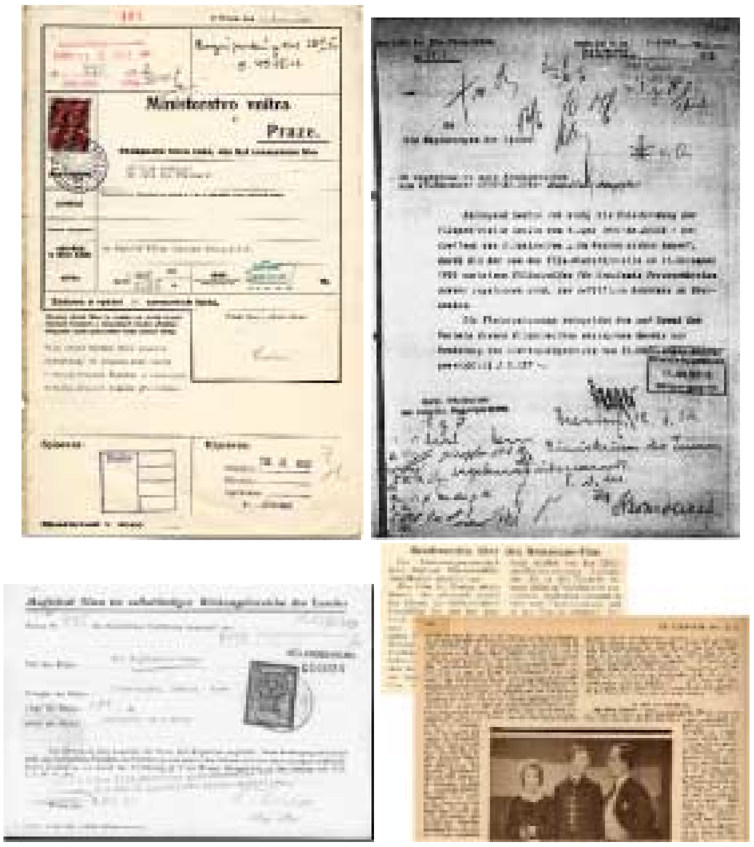
Ministerstvo vnitra
v Praze.

Fig. 3. Sample COLLATE documents: registration card from NFA (top-left), censorship decision from DIF (top-right), censorship card from FAA (bottom-left) and newspaper articles (bottom-right)

just obtaining acceptable and practically manageable runtimes for the learning task, and not checking the compression performance of the multi-substitution representation.

The original learning task concerned the EC project COLLATE (Collaboratory for Annotation, Indexing and Retrieval of Digitized Historical Archive Material), aimed at providing a support for archives, researchers and end-users working with digitized historic/cultural material. The chosen sample domain concerns a large corpus of multi-format documents concerning rare historic film censorship from the 20's and 30's, but includes also newspaper articles, photos, stills, posters and film fragments, provided by three major European film archives. Specifically, we considered 4 different kinds of documents: censorship decisions from Deutsches Filminstitut (*DIF*, in Frankfurt), censorship cards from Film Archive Austria (*FAA*, in Vienna), registration cards from Národni Filmový

Archiv (*NFA*, in Prague) and newspaper articles (from all archives), for each of which the aim was inducing rules for recognizing the role of significant layout components such as *film title*, *length*, *producer*, etc. Figure 3 reports some samples of this material. Specifically, for carrying out the comparison, we focused on one specific problem: learning the semantic label *object_title* for the document class *dif_censorship_decision*. Such a choice originates from the higher complexity of that label in that kind of document with respect to the others, due to its being placed in the middle of the page, thus having interrelations with more layout blocks than the other components.

The resulting experimental dataset, obtained from 36 documents, contained a total of 299 layout blocks, 36 of which were positive instances of the target concept, while all the others were considered as negative examples. The average length of the example descriptions ranges between 54 and 263 literals (215 on average), containing information on the size, content type, absolute and relative position of the blocks in a given document. The dataset was evaluated by means of a 10-fold cross validation. It should be noted that the learned theory must be checked for completeness and consistency for each new incoming example, and also on the whole set of processed examples each time a candidate refinement is computed, thus the matching procedure is very stressed during the learning and test tasks.

The number of variables in the clauses that make up the theory to be taken into account by the matching procedure, that is the critical factor for the exponential growth of the number of substitutions in the experiments, ranged between 50 (for some non-refined clauses) and 4 (for some very refined ones). The results of the comparison reveal a remarkable reduction of computational times when using the proposed algorithm[2]. Indeed, the average computational cost for learning in the 10 runs using SLD resolution (5297 sec) is much higher than using the new matching procedure (1942 sec), with a difference of 3355 sec on average. Going into more detail, 80% of the times are in favor of the matching procedure, in which cases the difference grows up to 4318 sec, while the average of times in favor of SLD resolution is limited to 225 sec only. It is noteworthy that in one case SLD resolution took 31572 sec to reach *one* solution, against 2951 sec of the matching procedure to return *all*. Another interesting remark is that the presented experiment was carried out by SLD resolution in reasonable, though very high, times; as already pointed out, we also faced cases (not taken into account here because we wanted to compute the average times) in which 20 days were still insufficient for it to get a solution.

The good results obtained under the OI assumption led us to try to make a comparison in the general case with other state-of-the-art systems. Since, as already pointed out, no algorithm is available in the literature to compute in one step the whole set of substitutions, the choice was between not making a comparison at all, or comparing the new algorithm to Django (the best-performing system in the literature so far for testing θ-subsumption, refer to [10] for a more

[2] Note that the difference is completely due to the matching algorithms, since the rest of the procedures in the learning system is the same.

detailed explanation). In the second case, it is clear that the challenge was not completely fair for our algorithm, since it *always* computes the whole set of solutions, whereas Django computes none (it just answers 'yes' or 'no'). Nevertheless, the second option was preferred, according to the principle that a comparison with a faster system could in any case provide useful information on the new algorithm performance, if its handicap is properly taken into account. The need for downward-compatibility in the system output forced to translate the new algorithm's results in the more generic answers of Django, and hence to interpret them just as 'yes' (independently of how many substitutions were computed, which is very unfair for our algorithm) or 'no' (if no subsuming substitution exists). Hence, in evaluating the experimental results, one should take into account such a difference, so that a slightly worse performance of the proposed algorithm with respect to Django should be considered an acceptable tradeoff for getting all the solutions whenever they are required by the experimental settings. Of course, the targets of the two algorithms are different, and it is clear that in case a binary answer is sufficient the latter should be used.

Specifically, a C language implementation of the new algorithm (this time in the general case, not restricted with the OI bias)[3] was made for carrying out the comparison on the same two tasks exploited for evaluating Django by its Authors. The former concerned *Phase Transition* [5], a particularly hard artificial problem purposely designed to study the complexity of matching First Order Logic formulas in a given universe in order to find their models, if any. Thus, this dataset alone should be sufficient to assess the algorithm performance. Nevertheless, it seemed interesting to evaluate it on real-world problems, whose complexity is expected not to reach the previous one. The *Mutagenesis* problem [16] is a good testbed to this purpose. No other experiment was run, because no other dataset is available in the literature having a complexity that allows to appreciate the two algorithms' power and performances, thus justifying their exploitation. All the experiments were run on a PC platform equipped with an Intel Celeron 1.3 GHz processor and running the Linux operating system. In both the two datasets referred to above, the general-case procedure (i.e., without the OI bias) will be exploited. The number of variables in clause C, denoted by n, is chosen according to the directions of the Authors that previously exploited them, for the sake of comparison; however, it is worth noting that the number of constants (which constitutes the base of the exponential formula) in the Phase Transition dataset is quite high (up to 50), which contributes to significantly raise the number of possible substitutions (50^{10} in some experiments).

In the Phase Transition setting, each clause ϕ is generated from n variables (in a set $\mathbf{X}$) and m binary predicates (in a set $\mathbf{P}$), by first constructing its *skeleton* $\varphi_s = \alpha_1(x_1, x_2) \wedge \ldots \wedge \alpha_{n-1}(x_{n-1}, x_n)$ (obtained by chaining the n variables through $(n-1)$ predicates), and then adding to φ_s the remaining $(m-n+1)$ predicates, whose arguments are randomly, uniformly, and without replacement selected from $\mathbf{X}$. Given Λ, a set of L constants, an example, against

[3] This implementation is publicly available on the Internet at the URL: `http://lacam.di.uniba.it:8000/systems/matching/`.

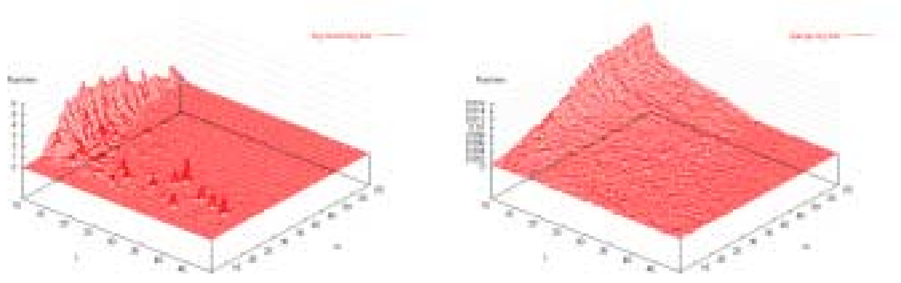

Fig. 4. Performance of the proposed algorithm (left-hand side) and of Django (right-hand side) on the Phase Transition problem (logarithm of times expressed in sec)

which checking the subsumption of the generated clause, is built using N literals for each predicate symbol in $\mathbf{P}$, whose arguments are selected uniformly and without replacement from the universe $\mathbf{U} = \Lambda \times \Lambda$. In such a setting, a *matching problem* is defined by a 4-tuple (n, m, L, N). Like in [10], n was set to 10, m ranges in $[10, 60]$ (actually, a wider range than in [10]) and L ranges in $[10, 50]$. To limit the total computational cost, N was set to 64 instead of 100: This does not affect the presence of the phase transition phenomenon, but just causes the number of possible substitutions to be less, and hence the height of the peaks in the plot to be lower. For each pair (m, L), 33 pairs $(\textit{hypothesis}, \textit{example})$ were constructed, and for each the computational cost was computed. Figure 4 reports the plots of the average θ-subsumption cost over all 33 trials for each single matching problem, measured as the logarithm of the seconds required by our algorithm and by Django on the phase transition dataset.

The two plots are similar in that both show their peaks in correspondence of low values of L and/or m, but the slope is smoother for Django than for the proposed algorithm, whose complexity peaks are more concentrated and abruptly rising. Of course, there is an orders-of-magnitude difference between the two performances (Django's highest peak is 0.037 sec, whereas our algorithm's top peak is 155.548 sec), but one has to take into account that the proposed algorithm also returns the whole set of substitutions by which θ-subsumption holds (if any, which means that a 'yes' outcome may in fact hide a huge computational effort when the solutions are very dense), and it almost always does this in reasonable time (only 5.93% of computations took more than 1 sec, and only 1.29% took more than 15 sec).

Such a consideration can be supported by an estimation of the expected number of solutions (i.e., substitutions) for each matching problem considered in the experiment.

According to the procedure for generating problem instances, the first literal in the clause can match any literal in the example, the following $(n-2)$ ones have one variable partially constrained by the previous ones, and the remaining

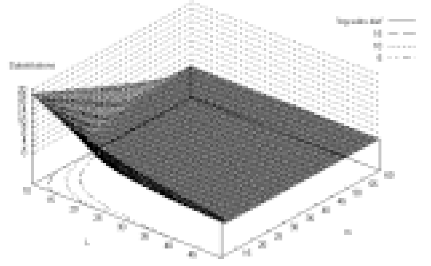

Fig. 5. Logarithm of the expected number of solutions

$(m - n + 1)$ are completely fixed, because they contain only variables already appeared in the first part of the formula. Then, the number of solutions for the skeleton φ_s is proportional to:

$$N \underbrace{\frac{N}{L} \cdots \frac{N}{L}}_{n-1} = \frac{N^{n-1}}{L^{n-2}} = S$$

The remaining literals decrease this number to:

$$S \underbrace{\frac{N}{L^2} \cdots \frac{N}{L^2}}_{m-n+1} = \frac{N^{n-1}}{L^{n-2}} \left(\frac{N}{L^2}\right)^{m-n+1} = \frac{N^m}{L^{2m-n}}.$$

Such a result, obtained in an informal way, agrees with the theoretical one (Expected number of solutions of a CSP) according to [15][4].

It is interesting to note that the shape of the plot of the logarithm of such a function, reported in Figure 5, resembles that of the proposed algorithm in Figure 4, in that both tend to have a peak for low values of L and m. This suggests, as expected, a proportionality between the computational times of the proposed algorithm and the number of substitutions to be computed. On the other hand, there is no such correspondence of this plot with Django performance. Even more, Django shows a tendency to increase computational times when keeping L low and progressively increasing m, just where our algorithm seems, on the contrary, to show a decrease (coherent with the number of expected solutions).

In the Mutagenesis dataset, artificial hypotheses were generated according to the procedure reported in [10]. 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 [7]. The cases in which $n > m + 1$ were not considered, since it is not

[4] Remember that the matching problem can be mapped onto a CSP.

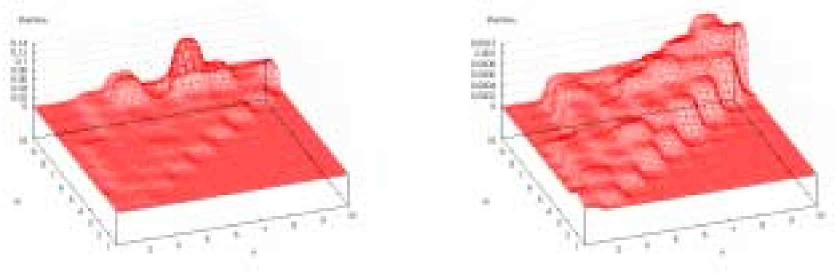

Fig. 6. Performance of the proposed algorithm (left-hand side) and of Django (right-hand side) on the Mutagenesis problem (sec)

Table 2. Mean time on the Mutagenesis problem for the three algorithms (sec)

SLD	Matching	Django
158,2358	0,01880281	0,00049569

possible to build a clause with m binary literals that contains more than $m+1$ variables and that fulfills the imposed linkedness constraint. 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-subsumption cost over all the times of the corresponding 10 hypotheses. Figure 6 reports the performance obtained by our algorithm and by Django on the θ-subsumption tests for the Mutagenesis dataset. Timings are measured in seconds.

Also on the second experiment, the shape of Django's performance plot is smoother, while that of the proposed algorithm shows sharper peaks in a generally flat landscape. Again, the proposed algorithm, after an initial increase, suggests a decrease in computational times for increasing values of n (when m is high). It is noticeable that Django shows an increasingly worse performance on the diagonal[5], while there is no such phenomenon in the left plot of Figure 6. In this case, however, there is no appreciable difference in computational times, since both systems stay far below the 1 sec threshold, which is a suggestion that real-world tasks normally do not present the difficult situations purposely created for the phase transition dataset.

Table 2 reports the mean time needed by the three considered algorithms to get the answer on the Mutagenesis Problem (backtracking was forced in SLD in order to obtain all the solutions). The Matching algorithm turned out to be

[5] Such a region corresponds to hypotheses with i literals and $i+1$ variables. Such hypotheses are particularly challenging for the θ-subsumption test since their literals form a chain of variables (because of linkedness).

$8415,5$ times more efficient than the SLD procedure (such a comparison makes no sense for Django because it just answers 'yes' or 'no'). To have an idea of the effort spent, the mean number of substitutions was $91,21$ (obviously, averaged only on positive tests, that are $8,95\%$ of all cases).

5 Conclusions and Future Work

This paper proposed a new algorithm for computing the whole set of solutions to the matching problem under θ-subsumption, whose efficiency derives from a proper representation of substitutions that allows to avoid backtracking (which may cause, in particular situations, unacceptable growth of computational times in classical matching mechanisms). Experimental results show that the new procedure significantly improves the performance of the learning system INTHELEX with respect to simple SLD resolution.

Also on two different datasets, a real-world one and an artificial one purposely designed to generate hard problems, it was able to carry out its task with high efficiency. Actually, it is not directly comparable to other state-of-the-art systems, since its characteristic of yielding all the possible substitution by which θ-subsumption holds has no competitors. Nevertheless, a comparison seemed useful to get an idea of the cost in time performance for getting such a plus. The good news is that, even on hard problems, and notwithstanding its harder computational effort, the new algorithm turned out to be in most cases comparable, and in any case at least acceptable, with respect to the best-performing system in the literature. A Prolog version of the algorithm is currently used in INTHELEX, a system for inductive learning from examples.

Future work will concern an analysis of the complexity of the presented algorithm, and the definition of heuristics that can further improve its efficiency (e.g., by guiding the choice of the best literal to take into account at any step in order to recognize as soon as possible the impossibility of finding a match).

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" (URL: `http://www.collate.de`). The authors would like to thank Michele Sebag and Jerome Maloberti for making available their system Django, and for the kind suggestions on its use and features. A grateful acknowledgement to the reviewers for their useful comments and suggestions on how to improve and make clearer ideas presented in this paper.

References

1. 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.
2. F. Esposito, N. Fanizzi, D. Malerba, and G. Semeraro. Downward refinement of hierarchical datalog theories. In M.I. Sessa and M. Alpuente Frasnedo, editors, *Proceedings of the Joint Conference on Declarative Programming - GULP-PRODE'95*, pages 148–159. Università degli Studi di Salerno, 1995.
3. 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.
4. M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.
5. A. Giordana, M. Botta, and L. Saitta. An experimental study of phase transitions in matching. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*, pages 1198–1203, S.F., July 31–August 6 1999. Morgan Kaufmann Publishers.
6. G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, 1985.
7. 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.
8. J.-U. Kietz and M. 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.
9. J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, New York, 2nd edition, 1987.
10. 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, pages 164–178. Springer, September 2001.
11. S.H. Muggleton and L. De Raedt. Inductive logic programming. *Journal of Logic Programming: Theory and Methods*, 19:629–679, 1994.
12. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer, 1997.
13. T. Scheffer, R. Herbrich, and F. 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.
14. 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.
15. Barbara M. Smith and Martin E. Dyer. Locating the phase transition in binary constraint satisfaction. *Artificial Intelligence*, 81(1–2):155–181, 1996.
16. 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.
17. P. Wadler. How to replace failure by a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1985.