A Complete Subsumption Algorithm

Stefano Ferilli, Nicola Di Mauro, Teresa M.A. Basile, and Floriana Esposito

Dipartimento di Informatica, Università di Bari
via E. Orabona, 4, 70125 Bari, Italia
{ferilli,nicodimauro,basile,esposito}@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 perform such a process with reduced time/space requirements (specifically when performing resolution). This paper proposes a new algorithm that is able to return the whole set of solutions to θ-subsumption problems by compactly representing substitutions. It could be exploited when techniques available in the literature are not suitable. Experimental results on its performance are encouraging.

1 Introduction

The classical Logic Programming [8] provability relation, logic implication, has been shown to be undecidable [12], which is too strict a bias to be accepted. Hence, a weaker but decidable generality relation, called θ-subsumption, is often used in practice. Given C and D clauses, C θ-subsumes D (often written $C \leq D$) iff there is a substitution θ such that $C\theta \subseteq D$. A substitution is a mapping from variables to terms, often denoted by $\theta = \{X_1 \rightarrow t_1, \ldots, X_n \rightarrow t_n\}$, whose application 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. Thus, since program execution corresponds to proving a theorem, efficiency of the generality relation used is a key issue that deserves great attention.

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 C θ-subsumes D. Note that D can always be considered ground (i.e., variable-free) without loss of generality. Indeed, in case it is not, each of its variables can be replaced by a new constant not appearing in C nor in D (*skolemization*), and it can be proven that C θ-subsumes D iff C θ-subsumes the skolemization of D. $|\cdot|$ will denote, as usual, the cardinality of a set (in particular, when applied to a clause, it will refer to the number of literals composing it). Since testing if C θ-subsumes D can be cast as a refutation of $\{C\} \cup \neg D$, a basic algorithm can be obtained in Prolog by skolemizing D, then asserting all the literals in the body of D an the clause C, and finally querying the head of D. The outcome is computed by Prolog through SLD resolution [10], which can be very inefficient under some conditions, as for C and D in the following example.

A. Cappelli and F. Turini (Eds.): AI*IA 2003, LNAI 2829, pp. 1–13, 2003.

Example 1. C = h(X_1) :- p(X_1,X_2),p(X_2,X_3),...,p(X_{n-1},X_n),q(X_n).
D = h(c_1) :- p(c_1,c_1), p(c_1,c_2), ..., p(c_1,c_n), p(c_2,c_1),
p(c_2,c_2), ..., p(c_2,c_n), ..., p(c_n,c_1), p(c_n,c_2), ..., p(c_n,c_n).

In the following, the next section presents past work in this field; Section 3 presents the new θ-subsumption algorithm, and Section 4 shows experimental results concerning its performance. Lastly, Section 5 concludes the paper.

2 Related Work

The great importance of finding efficient θ-subsumption algorithms is reflected by the amount of work carried out so far in this direction in the literature. In the following, we briefly recall some milestones in this research field.

Our brief survey starts from Gottlob and Leitsch [5]. After investigating two classical algorithms (by Chang & Lee [1] and Stillman [14]) in order to assess their worst-case time complexity, based on the number of performed literal unifications, they define a new backtracking algorithm that attacks the problem complexity through a *divide and conquer* strategy, by first partitioning the clause into independent subsets, and then applying resolution separately to each of them, additionally exploiting a heuristic that resolves each time the literal with the highest number of variables that occur also in other literals.

A more formal approach was then taken by Kietz and Lübbe in [7]. They start from the following definition:

Definition 1. *Let $C = C_0 \leftarrow C_{Body}$ and $D = D_0 \leftarrow D_{Body}$ be Horn clauses. C* deterministically *θ-subsumes D, written $C \vdash_{\theta DET} D$, by $\theta = \theta_0\theta_1 \ldots \theta_n$ iff $C_0\theta_0 = D_0$ and there exists an ordering $C'_{Body} = C_1, \ldots, C_n$ of C_{Body} such that for all i, $1 \leq i \leq n$, there exists exactly one θ_i such that $\{C_1, \ldots, C_i\}\theta_0\theta_1 \ldots \theta_i \subseteq D_{Body}$.*

Since in general $C \nvdash_{\theta DET} D$, in addition to identifying the subset of C that deterministically θ-subsumes D, C_{DET}, the algorithm can also return the rest of C, C_{NONDET}, to which other techniques can be applied according to the definition of *non-determinate locals*, corresponding to the independent parts of C_{NONDET} according to Gottlob and Leitsch. They can be identified in polynomial time, and handled separately by θ-subsumption algorithms.

The above ideas were extended by Scheffer, Herbrich and Wysotzki [11] by transposing the problem into a graph framework, in which additional techniques can be exploited. First, the authors extend the notion of 'determinism' in matching candidates by taking into account not just single literals, but also their 'context' (i.e., the literals to which they are connected via common variables). Indeed, by requiring that two literals have the same context in order to be matched, the number of literals in C that have a unique matching candidate in D potentially grows. Taking into account the context allows to test for subsumption in polynomial time a proper superset of the set of determinate clauses according to the definition by Kiets and Lübbe. The remaining (non-determinate) part of C is

then handled by mapping the subsumption problem onto a search for the maximum clique in a graph, for which known efficient algorithms can be exploited, properly tailored.

In sum, all the work described so far can be condensed in the following algorithm: First the 'extended' (according to the context definition) determinate part of C and D is matched; then the locals are identified, and each is attacked separately by means of the clique algorithm. Note that all the proposed 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 subsuming substitution found, even if many exist.

Finally, Maloberti and Sebag in [9] face the problem of θ-subsumption by mapping it onto a Constraint Satisfaction Problem (CSP). Different versions of a correct and complete θ-subsumption algorithm, named *Django*, were built, each implementing different (combinations of) CSP heuristics. Experiments are reported, proving a difference in performance of several orders of magnitude in favor of Django compared to the algorithms described above. 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 Approach

Previous successful results obtained on the efficiency improvement of the matching procedure under the Object Identity framework [3] led us to extend those ideas to the general case. The main idea to avoid backtracking and build in one step the whole set of subsumption solutions is to compress information on many substitutions by compactly representing them in a single structure. For this reason, some preliminary definitions are necessary.

3.1 Preliminaries

Let us start by recalling a useful definition from the literature.

Definition 2 (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 [2] $uni(C, l, D) = \{\mu \mid l \in C, l\mu \in D\}$

Now, it is possible to define the structure to compactly represent sets of substitutions.

Definition 3 (Multisubstitutions). *A* multibind *is denoted by $X \rightarrow T$, where X is a variable and $T \neq \emptyset$ is a set of constants. A* multisubstitution *is a set of multibinds $\Theta = \{X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\} \neq \emptyset$, where $\forall i \neq j : X_i \neq X_j$.*

In particular, a single substitution is represented by a multisubstitution in which each constants set is a singleton ($\forall i : \mid T_i \mid = 1$). In the following, multisubstitutions will be denoted by capital Greek letters, and normal substitutions by lower-case Greek letters.

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

Definition 4 (Split). *Given a multisubstitution* $\Theta = \{X_1 \to T_1, \ldots, X_n \to T_n\}$, split$(\Theta)$ *is the set of all substitutions represented by* Θ*:*
$split(\Theta) = \{\ \{X_1 \to c_{i_1}, \ldots, X_n \to c_{i_n}\}\ | \forall k = 1 \ldots n : c_{i_k} \in T_k \wedge i = 1 \ldots |T_k|\}$.

Example 3. Let us find the set of all substitutions represented by the multisubstitution $\Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$, $split(\Theta) = \{\{X \to 1, Y \to 7, Z \to 2\}, \{X \to 1, Y \to 7, Z \to 9\}, \{X \to 3, Y \to 7, Z \to 2\}, \{X \to 3, Y \to 7, Z \to 9\}, \{X \to 4, Y \to 7, Z \to 2\}, \{X \to 4, Y \to 7, Z \to 9\}\}$

Definition 5 (Union of Multisubstitutions). *The union of two multisubstitutions* $\Theta' = \{\overline{X} \to T', X_1 \to T_1, \ldots, X_n \to T_n\}$ *and* $\Theta'' = \{\overline{X} \to T'', X_1 \to T_1, \ldots, X_n \to T_n\}$ *is the multisubstitution*

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

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

Example 4. The union of two multisubstitutions $\Sigma = \{X \to \{1,3\}, Y \to \{7\}, Z \to \{2,9\}\}$ and $\Theta = \{X \to \{1,4\}, Y \to \{7\}, Z \to \{2,9\}\}$, is:

$$\Sigma \sqcup \Theta = \{X \to \{1,3,4\}, Y \to \{7\}, Z \to \{2,9\}\}$$

(the only different multibinds being those referring to variable X).

Definition 6 (Merge). *Given a set* $\mathcal{S}$ *of substitutions on the same variables,* merge$(\mathcal{S})$ *is the set of multisubstitutions obtained according to Algorithm 1.*

Example 5. 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 multisubstitution.

Definition 7 (Intersection of Multisubstitutions). *The intersection of two multisubstitutions* $\Sigma = \{X_1 \to S_1, \ldots, X_n \to S_n, Y_1 \to S_{n+1}, \ldots, Y_m \to S_{n+m}\}$ *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* $n, m, l \ge 0$ *and* $\forall j, k : Y_j \neq Z_k$*, is the multisubstitution 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.*

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

Require: $\mathcal{S}$: set of substitutions (each represented as a multisubstitution)
  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 6. The intersection of two multisubstitutions $\Sigma = \{X \to \{1,3,4\}, Z \to \{2,8,9\}\}$ 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.

Lemma 1. *The $\sqcap$ operator is monotonic in the set of variables. Specifically,*

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

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 multisubstitutions are compatible (i.e., if they share at least one of the substitutions they represent). Indeed, given two multisubstitutions 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 multisubstitutions. Specifically, given two sets of multisubstitutions $\mathcal{S}$ and $\mathcal{T}$, their intersection is defined as the set of multisubstitutions obtained as follows:

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

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

3.2 The Matching Algorithm

In the following, for the sake of readability, we use the expression $\theta \in \mathcal{T}$ to say that the substitution θ belongs to the split of some multisubstitution in the set of multisubstitutions $\mathcal{T}$.

Proposition 1. $\forall \theta : C\theta \subseteq D \Leftrightarrow \theta \in \mathcal{S}_n$.

Proof. 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 $\forall i = 2 \ldots n : \mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i$.
($\Leftarrow$) By induction on i: $\forall i \in \{1, \ldots, n\} : \mathcal{S}_i \neq \emptyset \Rightarrow \forall \theta \in \mathcal{S}_i : \{l_1, \ldots, l_i\}\theta \subseteq D$.
Base $\emptyset \neq \mathcal{S}_1 = \mathcal{T}_1 \Rightarrow \forall \theta \in \mathcal{T}_1 : \exists k \in D \ni' l_1\theta = k \in D \Rightarrow \{l_1\}\theta = \{k\} \subseteq D$.
Step $\mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i \neq \emptyset \Rightarrow$ (by definition of $\sqcap$) $\exists \Sigma \in \mathcal{S}_{i-1}, \Theta \in \mathcal{T}_i \ni' \Sigma \sqcap \Theta$

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 $(S_n \neq \emptyset)$

defined $\Rightarrow \forall\gamma \in \Sigma \sqcap \Theta : \gamma = \sigma\theta \nexists \sigma \in split(\Sigma), \theta \in split(\Theta) \wedge \sigma, \theta$ 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.
($\Rightarrow$) By induction on i: $\forall i \in \{1, \ldots, n\} : \{l_1, \ldots, l_i\}\theta \subseteq D \Rightarrow \theta \in \mathcal{S}_i$.
Base (*Ad absurdum*) $\nexists\theta|_{\{l_1\}} \Rightarrow \theta|_{\{l_1\}} \notin merge(uni(C, l_1, D)) \Rightarrow \theta|_{\{l_1\}} \notin uni(C, l_1, D) \Rightarrow \{l_1\}\theta|_{\{l_1\}} \notin D \Rightarrow \{l_1\}\theta \notin D$. But $\{l_1\}\theta \in D$ by hypothesis.
Step (*Ad absurdum*) $\theta|_{\{l_1,\ldots,i\}}(= \theta|_{\{l_1,\ldots,l_{i-1}\}}\theta|_{\{l_i\}}) \notin \mathcal{S}_i$. By construction, $\mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i$. By inductive hypothesis, $\theta|_{\{l_1,\ldots,l_{i-1}\}} \in \mathcal{S}_{i-1}$. Thus, $\theta|_{\{l_i\}} \notin \mathcal{T}_i \Rightarrow \{l_i\}\theta|_{\{l_i\}} \notin D \Rightarrow \{l_i\}\theta \notin D$. But, by hypothesis, $\{l_1, \ldots, l_i\}\theta \subseteq D \Rightarrow \{l_i\}\theta \in D$.

This leads to the θ-subsumption procedure reported in Algorithm 2. It should be noted that the set of multisubstitutions resulting from the merging phase could be not unique. In fact, it may depend on the order in which the two multisubstitutions to be merged are chosen at each step. 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 is outside the scope of this paper) in order to understand if the compression 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 multisubstitutions to be merged have to be taken in order to get an optimal result.

Example 7. 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 multisubstitutions:
$$\{\{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{3, 4, 5\}\}, \{X \leftarrow \{1\}, Y \leftarrow \{5\}, Z \leftarrow \{3\}\}\}$$
Another possibility is first merging $\theta \sqcup \tau$ and then $\delta \sqcup \sigma$, that cannot be further merged and hence yield:
$$\{\{X \leftarrow \{1\}, Y \leftarrow \{2, 5\}, Z \leftarrow \{3\}\}, \{X \leftarrow \{1\}, Y \leftarrow \{2\}, Z \leftarrow \{4, 5\}\}\}$$

3.3 Discussion

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. This aim inspired, for instance, the concepts of *determinate* (part of a) clause and of *k-locals*. However, after identifying such determinate and independent subparts of the given clauses, the only possible 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. Specifically, its search strategy consists in a kind of breadth-first in which the explored nodes of the search space are compressed; this means that, when no compression is possible for the substitutions of each literal, it becomes a normal breadth-first search (it would be interesting to discuss in what – non purposely designed – situations it happens). Hence, it is worth discussing the complexity of the different steps involved thereof. Because of the above considerations, in the following only linked clauses will be taken into account, so that neither determinate matching nor partitioning into k-locals apply.

Let p_i be the i-th distinct predicate in C, a_i its arity and m_i be the number of literals in D with predicate symbol p_i. Let l_j be the j-th literal in C. Call a the maximum arity of predicates in C (predicates with greater arity in D would not be considered for matching), and c the number of distinct constants in D. Each unifier of a given p_i with a literal on the same predicate symbol in D can be computed in a_i steps. There are m_i such unifiers to be computed (each represented as a multisubstitution), hence computing $uni(C, l, D)$ has complexity $a_i * m_i$ for any literal $l \in C$ built on predicate p_i. Note that the constants associated to each argument of p_i are the same for all literals in C built on it, hence such a computation can be made just once for each different predicate, and then tailored to each literal by just changing the variables in each multibind.

(Checking and) merging two multisubstitutions requires them to differ in at most one multibind (as soon as two different multibinds are found in the two multisubstitutions, the computation of their merging stops with failure). Hence, the complexity of merging two multisubstitutions is less than $a_i * 2m_i$, since there are at most a_i arguments to be checked, each made up of at most m_i constants (one for each compatible literal, in case they are all different)[1]. The multisubstitutions in the set $uni(C, l, D)$ can be merged by pairwise comparing (and, possibly, merging) any two of them, and further repeating this on the new sets stepwise obtained, until no merging is performed or all multisubstitutions have been merged

[1] Assuming that the constants in each multibind are sorted, checking the equality of two multibinds requires to scan each just once.

into one. At the k-th step ($0 \leq k \leq m_i - 1$), since at least one merging was performed at each step, the set will contain at most $m_i - k$ multisubstitutions, for a total of at most $\binom{m_i-k}{2}$ couples to be checked and (possibly) merged. Globally, we have a merge complexity equal to[2] $\sum_{k=1}^{m_i-1} \binom{m_i-k}{2} * a_i * 2m_i \sim O(a_i * m_i^4)$.

As to the intersection between two multisubstitutions, note that one of the two refers to a literal $l \in C$ built on a predicate p_i, and hence will be made up of a_i multibinds, each of at most m_i constants. In the (pessimistic) case that all of the variables in l are present in the other multisubstitution, the complexity of the intersection is therefore[3] $a_i * m_i * min(c, |D|)$.

When discussing the overall complexity of the whole procedure, it is necessary to take into account that a number of interrelations exist among the involved objects, such that a growth of one parameter often corresponds to the decrease of another. Thus, this is not a straightforward issue. Nevertheless, one intuitive worst case is when non merging can take place among substitutions for all literals[4] and each substitution of any literal is compatible with any substitution of all the others. In such a case, the number of intersections is $O(m^n)$ (supposing each literal in C has m matching substitutions in D), but it should be noted that in this case each intersection does not require any computation and is reduced to just an append operation. One intuitive best case is when all substitutions for each literal can be merged into one. In this case, the dominant complexity is that of merging, i.e. $O(n * a * m^4)$.

4 Experiments

The new algorithm has been implemented in C, and its performance in computing θ-subsumption between Horn clauses having the same predicate in their head was to be assessed. Actually, to the authors' knowledge, no algorithm is

[2] Actually, this is a pessimistic upper bound, that will not ever be reached. Indeed, it is straightforward to note that a number of simplifying interrelations (not taken into account here for simplicity) hold: e.g., the number of steps is *at most* m_i; the more the number of performed mergings, the less the number of possible steps, and the less the number of substitutions to be merged at each step; the more the number of steps, the less the number of merged multisubstitutions, and the less the number of constants in each of them; at each step, only the new merged multisubstitutions are to be taken into account for merging with the previous ones; and so on.

[3] Note that the other multisubstitution comes from past intersections of multisubsitutions referred to already processed literals, and hence each of its multibinds may contain at most a number of constants equal to the maximum number of literals in D that are compatible with a literal in C, i.e. $max_i(m_i) \leq |D|$, or to the maximum number of different constants, whichever is the less: $min(c, |D|)$.

[4] Remember that we suppose to deal only with linked clauses, otherwise the matching procedure can be applied separately to the single connected components and then the global substitution can be obtained by simply combining in all possible ways such partial solutions, that will be obviously compatible since they do not share any variable.

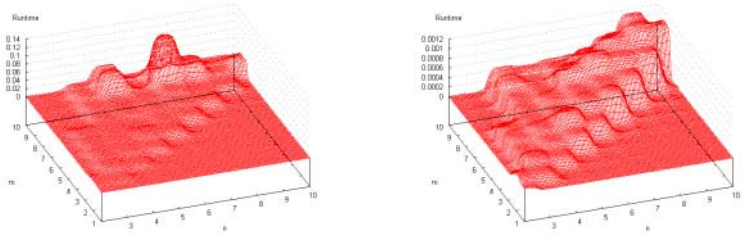

Fig. 1. Performance of the new algorithm and Django on Mutagenesis (sec)

available that computes the whole set of substitutions (except forcing all possible backtrackings in the previous ones, which could yield unacceptable runtimes), to which comparing the proposed one. Thus, the choice was between not making a comparison at all, or comparing the new algorithm to Django (the best-performing among those described in Section 2). 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 lower-level 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.

A first comparison was carried out on a task exploited for evaluating Django by its Authors: The *Mutagenesis* problem [13]. The experiment was run on a PC platform equipped with an Intel Celeron 1.3 GHz processor and running the Linux operating system. In the Mutagenesis dataset, artificial hypotheses were generated according to the procedure reported in [9]. 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 imposed linkedness constraint. Specifically, for each (m, n) pair $(1 \leq m \leq 10, 2 \leq n \leq 10)$, 10 artificial hypothe-

Table 1. Mean time on the Mutagenesis problem for the three algorithms (sec)

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

ses 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 1 reports the performance obtained by our algorithm and by Django (respectively) on the θ-subsumption tests for the Mutagenesis dataset. Timings are measured in seconds. The shape of Django's performance plot is smoother, while that of the proposed algorithm shows sharper peaks in a generally flat landscape. 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 plot on the left of Figure 1. However, there is no appreciable difference in computational times, since both systems stay far below the 1 sec threshold.

Table 1 reports the mean time on the Mutagenesis Problem for the three algorithms to get the answer (backtracking was forced in SLD in order to obtain all the solutions). It is possible to note that the Matching algorithm is $8415,5$ 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).

Another interesting task concernes *Phase Transition* [4], 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. A number of pairs clause-example were generated according to the guidelines reported in [4]. Like in [9], n was set to 10, m ranges in $[10, 60]$ (actually, a wider range than in [9]) 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. For each pair (m, L), 33 pairs $(hypothesis, example)$ were constructed, and the average θ-subsumption computational cost was computed as the seconds required by the two algorithms. Both show their peaks in correspondence of low values of L and/or m, but such peaks are more concentrated and abruptly rising in the new algorithm. Of course, there is an orders-of-magnitude difference between the two performances (Django's highest peak is 0.037 sec, whereas our

[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).

Table 2. Average θ-subsumption cost in the YES, NO an PT regions

		NO	Phase Transition	YES	NEG
Django	Mean	0,003907	0,00663	0,005189	0,003761
	St-Dev	0,004867	0,00756	0,004673	0,00455
Matching	Mean	0,1558803	3,5584	7,5501	0,1139
	St-Dev	0,75848	10,5046	20,954	0,5147
		39,8977	536,7119	1455,02023	30,2845

algorithm's top peak is 155.548 sec), but one has to take into account that the new algorithm also returns the whole set of substitutions (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 does this in reasonable time (only 5.93% of computations took more than 1 sec, and only 1.29% took more than 15 sec).

The mean θ-subsumption costs in various regions are summarized in Table 2. The region is assigned to a problem (m, L) according to the fraction f of clauses $\mathcal{C}$ subsuming examples Ex, over all pairs $(\mathcal{C}, Ex)$ generated for that problem. In particular, $f > 90\%$ denotes YES region, $10\% \leq f \leq 90\%$ denotes PT region, and $f < 10\%$ means NO region. While for Django the cost in PT is 1,7 times the cost in NO and 1,3 times the cost in YES, thus confirming the difficulty of that region, in the new algorithm the cost across the regions grows according to the number of substitutions, as expected. The last column reports the cost in a region (NEG) corresponding to the particular case $f = 0\%$ (i.e., there are no substitutions at all). The last row shows the gain of Django over the new algorithm. Again, as expected, the gain grows as the number of solution increases, because it stops immediately after getting an answer, whereas the new algorithm continues until all substitutions are found. The only region in which a comparison is feasible is NEG, where Django is 30 times better than Matching (this could be improved by introducing heuristics that can bias our algorithm towards recognizing a negative answer as soon as possible).

5 Conclusions and Future Work

This paper proposed a new algorithm for computing the whole set of solutions to θ-subsumption problems, 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 subsumption mechanisms). Experimental results suggest that it is 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 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., heuristics that may guide the choice of the best literal to choose at any step in order to recognize as soon as possible the impossibility of subsumption).

Acknowledgements

This work was partially funded by the EU project IST-1999-20882 COLLATE. The authors would like to thank Michele Sebag and Jerome Maloberti for making available the Django and for the suggestions on its use, and the anonymous reviewers for useful comments.

References

[1] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973. 2

[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

[3] S. Ferilli, N. Fanizzi, N. Di Mauro, and T. M. A. Basile. Efficient θ-subsumption under Object Identity. In *Atti del Workshop AI*IA su Apprendimento Automatico, Siena - Italy*, 2002. 3

[4] A. Giordana, M. Botta, and L. Saitta. An experimental study of phase transitions in matching. In Dean Thomas, editor, *Proceedings of IJCAI-99 (Vol2)*, pages 1198–1203, S. F., July 31–August 6 1999. Morgan Kaufmann Publishers. 10

[5] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, 1985. 2

[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. 9

[7] J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In W. Cohen and H. Hirsh, editors, *Proceedings of ICML-94*, pages 130–138, 1994. 2

[8] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, New York, 2 edition, 1987. 1

[9] J. Maloberti and M. Sebag. θ-subsumption in a constraint satisfaction perspective. In Céline Rouveirol and Michèle Sebag, editors, *Proceedings of ILP 2001*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 164–178. Springer, September 2001. 3, 9, 10

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

[11] T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient θ-subsumption based on graph algorithms. In Stephen Muggleton, editor, *Proceedings of ILP-96*, volume 1314 of *LNAI*, pages 212–228. Springer, August 26–28 1997. 2

[12] M. Schmidt-Schauss. Implication of clauses is undecidable. *Theoretical Computer Science*, 59:287–296, 1988. 1

[13] 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. 9

[14] R. B. Stillman. The concept of weak substitution in theorem-proving. *Journal of ACM*, 20(4):648–667, October 1973. 2