θ-Subsumption in a Constraint Satisfaction Perspective

Jérôme Maloberti[1] and Michèle Sebag[2,1]

(1) LRI, Bat 490, Université Paris-Sud, F-91405 Orsay
(2) LMS, Ecole Polytechnique, F-91128 Palaiseau
Jerome.Maloberti@lri.fr, Michele.Sebag@polytechnique.fr

Abstract. The covering test intensively used in Inductive Logic Programming, i.e. θ-subsumption, is formally equivalent to a Constraint Satisfaction problem (CSP). This paper presents a general reformulation of θ-subsumption into a binary CSP, and a new θ-subsumption algorithm, termed *Django*, which combines some main trend CSP heuristics and other heuristics specifically designed for θ-subsumption.
Django is evaluated after the CSP standards, shifting from a worst-case complexity perspective to a statistical framework, centered on the notion of *Phase Transition* (PT). In the PT region lie the hardest on average CSP instances; and this region has been shown of utmost relevance to ILP [4]. Experiments on artificial θ-subsumption problems designed to illustrate the phase transition phenomenon, show that *Django* is faster by several orders of magnitude than previous θ-subsumption algorithms, within and outside the PT region.

1 Introduction

Supervised learning intensively relies on the generality operator, or covering test, calculating whether a given hypothesis covers a given example. As the evaluation of a candidate hypothesis depends on its coverage, the covering test must imperatively be efficient.

The complexity of the covering test is one main concern facing Inductive Logic Programming (ILP) [10,12]. The covering test commonly used in ILP, i.e. θ-subsumption [13], is exponentially complex in the size of the candidate hypothesis. How to manage this complexity has motivated numerous studies on learning biases, restricting the size and/or the number of hypotheses explored through syntactic or search biases [11]. In parallel, new algorithms for achieving efficient θ-subsumption [8,18] and ILP learners based on a correct approximation of θ-subsumption [19], have been proposed.

In this paper, is presented a new correct and complete θ-subsumption algorithm termed *Django*, based on a Constraint Satisfaction Problem (CSP) approach.

Although it is long known that θ-subsumption is equivalent to a Constraint Satisfaction problem (CSP), ILP problems have only recently been put in a

CSP perspective [3,4]. The focus is thereby shifted from a worst-case complexity analysis, to a statistical approach [7].

The covering test complexity is handled as a random variable, measuring the computational cost of θ-subsumption for some order parameters (e.g. the number of variables & predicates in the hypothesis, the number of literals & constants in the example). Surprisingly, the computational cost is almost zero for most problems, referred to as trivial. For instance, assuming that all predicates in the hypotheses also appear in the examples, a short hypothesis will cover almost surely all examples; inversely, a long hypothesis will almost surely cover no example at all. In both cases, the θ-subsumption cost remains low as the θ-subsumption problem corresponds to an under or over-constrained satisfaction problem. But in a narrow region, termed *phase transition* (PT), where the probability for a hypothesis to cover an example is close to 50%, the covering test reaches its maximum complexity on average [3].

The PT phenomenon is of utmost importance for ILP, for two reasons. First, there is ample evidence of phase transition in artificial problems statistically modeled from ILP real-world problems [3]. Second, intensive experimentations on artificial problems have shown that this region behaves as an attractor on existing ILP learners[1] [4].

This paper is concerned with designing a θ-subsumption algorithm with good average performances on the most relevant and critical instances of θ-subsumption problems, i.e. lying within the PT. To this aim, is first presented a general transformation of a θ-subsumption problem referred to as *primal problem*, into another constraint satisfaction problem, termed *dual problem*. Along the transformation, each literal (involved in the hypothesis, primal CSP) becomes a constrained variable in the dual CSP; conversely, a variable in the primal CSP derives a set of dual constraints; furthermore, specific constraints encoding the θ-subsumption structure are automatically generated. On the dual CSP is applied a combination of well-known CSP algorithms, forming the *Django* system. The approach is validated on artificial θ-subsumption problems designed after [4] to sample the Phase Transition region. Intensive experiments show that *Django* improves by several orders of magnitude on average on all problems within and outside the PT in the considered range, compared to previous θ-subsumption algorithms [8,18].

The paper is organized as follows. Next section briefly introduces θ-subsumption and reviews existing θ-subsumption algorithms [8,18]. Section 3 presents the Constraint Satisfaction framework and the main heuristics used to solve CSPs. Section 4 describes the transformation of a θ-subsumption problem into a dual binary CSP, and presents the combinations of CSP heuristics involved in the *Django* system. Experimental setting and results are reported and discussed in section 5, and the paper ends with some perspectives for further research.

[1] E.g. for almost all target concepts, FOIL [15] selects its final hypotheses in the PT region.

2 θ-Subsumption, Definition and Algorithms

Let hypothesis $\mathcal{C}$ denote a conjunction of literals with no function symbols, and $arg(\mathcal{C})$ denote the set of variables in $\mathcal{C}$. Example Ex likewise is a conjunction of literals with no function symbols, $arg(Ex)$ being the set of variables and constants in Ex.

By definition, $\mathcal{C}$ θ-subsumes Ex according to θ, iff θ is a mapping from $arg(\mathcal{C})$ onto $arg(Ex)$, mapping variables in $\mathcal{C}$ onto variables and constants in Ex such that $\mathcal{C}\theta$ be included in Ex. Instead of mapping variables in $\mathcal{C}$ onto variables and constants in Ex, it is often more computationally efficient to map literals in $\mathcal{C}$ onto those literals in Ex built on the same predicate symbol. Through a literal mapping, each variable in $\mathcal{C}$ is associated to a set of variables and constants in Ex; the literal mapping is termed *consistent* if it maps each variable in $\mathcal{C}$ onto a single variable or constant in Ex.

The main stream algorithm for θ-subsumption is based on Prolog SLD resolution [16]. It performs a depth first exploration of literal mappings (associating to the first literal in $\mathcal{C}$ the first literal built on the same symbol in Ex, and so on), and it backtracks if an inconsistency occurs (e.g. one variable in $\mathcal{C}$ is associated to two constants in Ex). Literals in $\mathcal{C}$ and E are explored in their order of apparition, which has a significant impact on the SLD efficiency, as known by all Prolog programmers.

A first improvement has been proposed by [8], based on the notion of determinate matching. It consists of reordering the literals in $\mathcal{C} = p_1..p_K$ in such a way that, if possible, there is a single candidate literal $p' = \theta(p_i)$ in Ex for p_i in $\mathcal{C}$, which is consistent with the previous assignments (such that $\{p_1/\theta(p_1), ..p_i/\theta(p_i)\}$ is consistent). After all determinate literals in $\mathcal{C}$ have been mapped onto literals in Ex, and if necessary, the search resumes using SLD resolution.

The scope of determinate matching is extended by [18], using a *graph context* to prune the candidate literals. To each literal p in $\mathcal{C}$ (resp. in Ex) is associated its neighborhood; the 1-neighbors of p are all literals sharing at least one variable (resp. one variable or one constant) with p; the i-th neighbors are recursively constructed, as 1-neighbors of $i-1$-neighbors. It is shown, that, unless all predicate symbols occurring in p i-th neighbors also occur among p' i-neighbors, p cannot be mapped onto p', and the latter can be removed from the candidate literals for p. [18] further define a *substitution graph*, connecting two pairs of literals (p, p') and (q, q') iff mapping $(p/p', q/q')$ is consistent. The SLD search is replaced by a maximal clique search in the substitution graph. The worst-case complexity remains exponential, but the advantage is to perform the consistency check only once.

Another heuristics used by [8] proceeds by decomposing the substitution graph into mutually independent components (*k-locality*). Such a decomposition significantly reduces the complexity of the problem..

3 Constraint Satisfaction Problem

This section briefly introduces CSPs together with the main stream heuristics; the reader is referred to [21] for a comprehensive presentation.

A CSP involves i) a set of variables $X_1, .., X_n$, with $dom(X_i)$ being the value domain for X_i, and ii) a set of constraints, specifying the simultaneously admissible values of the variables. A constraint can conveniently be thought of as a predicate $r(X_{i_1}, ..X_{i_M})$, while the admissible values are described as a set of literals $r(a_{1,i_1}, ..a_{1,i_M}), \ldots, r(a_{l,i_1}, ..a_{l,i_M})$, $a_{j,i_k} \in dom(X_{i_k})$. The constraint scope, noted $arg(r)$ is the set of variables $X_{i_1}, ..X_{i_k}$. The constraint domain, noted $dom(r)$, is the set of literals built on[2] r.

A CSP solution assigns to each variable X_i a value a_i in $dom(X_i)$ such that all constraints are satisfied; it can be viewed as a mapping $\theta = \{X_i/a_i\}$ such that for each constraint $r(X_{i_1}, ..X_{i_k})$, $r\theta = r(a_{i_1}, ..a_{i_k})$ belongs to $dom(r)$. In other words, the CSP defined by constraints $r_1, ..r_K$ is satisfiable iff $r_1, ..r_K$ θ-subsumes the conjunction $dom(r_1), ..dom(r_K)$. Likewise, the CSP complexity is exponential in the number n of variables, and linear in the number m of constraints: if $|a|$ is the number of possible values for a variable, the complexity is $\mathcal{O}(|a|^n \times m)$. (A first way of decreasing the complexity is by decomposing the CSP into fewly related subproblems – hierarchizing the set of variables [6] or the set of constraints [2] – in the same spirit as k-locality [8]).

Two CSPs are *equivalent* iff they are defined on same variables and admit same solutions. As any CSP can be embedded into a binary CSP, i.e. with binary constraints only, most CSP algorithms only consider binary and unary constraints. Further, with no loss of generality, one assumes that there exists at most one constraint on each variable pair.

CSP algorithms are made up of two kinds of heuristics. Reduction heuristics are meant to transform a CSP into an equivalent CSP of lesser complexity, through reducing the variable domains. Search heuristics are concerned with the search strategy.

3.1 Reduction

Reduction proceeds by pruning the candidate values for each variable X. Value a in $dom(X)$ is locally *consistent* if, for all variables Y such that there exists a constraint $r(X, Y)$, there exists some candidate value b in $dom(Y)$ such that $r(a, b)$ holds (belongs to $dom(r)$). Clearly, if a is not locally consistent, X cannot be mapped onto a, which can thus soundly be removed from $dom(X)$.

Local consistency is extended as follows; a is k-consistent with X if for each set of constraints $r_1(X, Y_1), r_2(Y_1, Y_2), ..r_k(Y_{k-1}, Y_k)$, there exists a $k-1$-tuple $(b_1, .., b_k)$ such that $r_1(a, b_1), r_2(b_1, b_2), ..r_k(b_{k-1}, b_k)$, holds.

[2] In all generality, the constraint domain can be infinite (e.g. a numerical constraint on real-valued variables). Only finite domains are considered in the rest of the paper.

A CSP is k-consistent iff each value domain $dom(X_i)$ includes k-consistent values only – and is not empty[3]. Checking k-consistency is exponentially complex with respect to k; therefore, only 2-consistency, or *arc consistency* is used in practice. The best complexity of reduction algorithms is $\mathcal{O}(m|a|^2)$, with m being the number of constraints and $|a|$ the value domain size.

3.2 Search

CSP algorithms incrementally construct a solution $\{X_i/a_i\}$ through a depth first exploration of the substitution tree; a node corresponds to a variable X_i, to which is assigned some candidate value a_i. On each assignation, consistency is checked; on failure, another candidate value for the current node is considered; if no other value is available, the search backtracks.

Several approaches have been proposed in order to improve: (i) the backtracking procedure (*look-back* heuristics); (ii) the choice of the next variable and candidate value to consider (*look-ahead* heuristics).

Look-back heuristics aim at preventing the repeated exploration of a same substitution subtree on backtracking (*thrashing*). For instance, Conflict Based Jumping (CBJ) [14] registers all variable conflicts occurred during the exploration, which allows for backtracking directly to the appropriate tree level. On the other hand, it may happen that the overhead due to maintaining the conflict registers offsets the look-back advantages for some particular CSP instances.

Look-ahead heuristics aim at minimizing the number of assignments considered. The best known look-ahead heuristics is constraint propagation; in each step, the candidate values which are inconsistent with the current assignment, are pruned. This way, inconsistencies are detected earlier and less nodes are visited; in counterpart, the assignment operation becomes more expensive as it involves the constraint propagation step.

Forward checking (FC) employs a limited propagation, only pruning the candidate values for the next variable (partial arc-consistency). Maintaining arc consistency (MAC) checks the arc-consistency on each variable assignment. Again, the overhead due to constraint propagation might offset its advantages on medium-size weakly constrained CSPs. Currently, the most generally efficient algorithms combine FC and CBJ.

In addition, the variable order can be optimized, either statically (once for all), or dynamically (the yet unassigned variables are reordered on each assignment). Dynamic variable ordering is generally more efficient than static variable ordering. One criterion for reordering the variables is based on the First Fail Principle [1], preferring the variable with the smallest domain. This way, failures will occur sooner rather than later.

Last, the candidate values can be ordered too; the value with less conflicts with the other variable domains is commonly preferred.

[3] The use of graph contexts to prune the candidate literals [18] can be viewed as a k-consistency check (more on this in section 4.2).

4 CSP Heuristics for θ-Subsumption

This section formalizes θ-subsumption as a binary CSP problem, and presents a new combination of CSP heuristics for θ-subsumption, *Django*.

4.1 Representation

It has been shown (section 2) that a CSP problem is equivalent to a θ-subsumption problem. However, θ-subsumption generally considers n-ary predicates. An ad hoc representation is thus necessary to enable the use of standard CS heuristics.

$\mathcal{C}$: $tc(X_0), p(X_0, X_1), p(X_1, X_2), p(X_2, X_3), q(X_0, X_2, X_3)$
Ex : $tc(a_0), p(a_0, a_1), p(a_1, a_2), p(a_2, a_3), p(a_3, a_4), p(a_0, a_3), q(a_0, a_2, a_3), q(a_0, a_1, a_3)$

We choose to consider the dual constraint satisfaction problem defined as follows. Each dual variable $Y_{p.i}$ corresponds to a literal in $\mathcal{C}$, namely the i-th literal built on predicate symbol p (subscript $.i$ will be omitted for readability when there is a single literal built on the predicate symbol); its domain $dom(Y_{p.i})$ is the set of all literals in Ex built on the same predicate symbol p, e.g. $dom(Y_{p.1}) = \{p(a_0, a_1), p(a_1, a_2), p(a_2, a_3), p(a_3, a_4), p(a_0, a_3)\}$.
A dual constraint $r(Y_{p.i}, Y_{q.j})$ is set on a variable pair $(Y_{p.i}, Y_{q.j})$ iff the corresponding literals in $\mathcal{C}$ share a (primal) variable; for instance, as $tc(X_0)$ and $p(X_0, X_1)$ share variable X_0, there is a dual constraint linking Y_{tc} and $Y_{p.1}$. This constraint specifies that, for each literal p' in $dom(Y_{p.i})$, there must be a literal q' in $dom(Y_{q.j})$ such that the literal mapping $\{p.i/p', q.j/q'\}$ is consistent with respect to θ-subsumption. In our toy example, dual constraint $r(Y_{tc}, Y_{p.1})$ is only satisfied for the dual value pair $(tc(a_0), p(a_0, a_1))$.
The difference between such dual constraints and the substitution graph in [18] is that the substitution graph specifies whether a given literal assignment $\{p/p'\}$ is consistent with another one $\{q/q'\}$. In contrast, dual constraints require that, for each pair of literals p, q in $\mathcal{C}$ sharing one variable, there exists a pair p', q' of literals in Ex, such that $\{p/p', q/q'\}$ is consistent.

The dual CSP is enriched by associating to each dual variable (literal in $\mathcal{C}$) and candidate value (literal in Ex) a *signature*, encoding the literal links (shared variables) with all others literals. For instance, the signature associated to $p(X_0, X_1)$ states that the first variable appear in a literal built on symbol tc, position 1, and a literal built on p, position 1; and the second variable appear in a literal built on symbol p, positions 1 and 2. Signatures allow one to prune the candidate literals through arc-consistency, in a similar way to graph contexts [18]; the signature of the literal in $\mathcal{C}$ must be included in the signature of the candidate literal in Ex. The difference is that signatures are deliberately limited to depth 1 (only 1-neighborhoods are considered), which allows for an optimized implementation.

Last, the case of literals sharing several variables is considered; signatures associated to pairs of such literals, termed 2-signatures, are designed and used to prune candidate literals too.

4.2 Resolution

As mentioned earlier, there is no such thing as a universally efficient CSP heuristics; it is thus desirable to evaluate carefully how relevant a given CSP heuristics is wrt θ-subsumption problems. Several combinations of heuristics have been experimented in *Django* (summarized in Table 1).

The baseline version **V1** combines arc-consistency checking [9] and forward checking (the propagation of the current assignment is restricted to the next variable domain).

Table 1. *Django*, Versions **V1** to **V8**.

	Base line
V1	Arc consistency + simple Forward Checking (propagation of the current assignment wrt the next variable)
	Dynamic Variable Ordering
V2	**V1** + DVO based on minimal domain (random choice in case of tie)
V3	**V1** + DVO based on maximal connectivity (random choice in case of tie)
V4	**V1** + DVO based on min. domain + max. connectivity (minimal domain, then maximal connectivity)
V5	**V1** + DVO based on max. connectivity + min. domain (maximal connectivity then minimal domain)
	Forward Checking
V6	**V4** + improved Forward Checking propagation of forced assignments (singleton candidate value)
	Arc Consistency
V7	**V6** + AC based on signatures
V8	**V7** + AC based on signatures and 2-signatures

We first investigate the influence of variable ordering on the search efficiency. Versions **V2** to **V5** implement several dynamic variable orderings, all based on the First Fail Principle. In **V2**, variables with minimal domain are ranked first[4]. In **V3**, variables subject to a maximal number of dual constraints are ranked first (prefer the literals in $\mathcal{C}$ which are most connected to others literals). Both criteria are combined in versions **V4** and **V5**, with different priorities.

Secondly, we investigate the influence of Forward Checking. In Version **V6**, besides the 1-step propagation of the current assignment, forced assignments (singleton candidate value for any variable) are propagated.

Last, we investigate the influence of arc consistency, using signatures and 2-signatures. Version **V7** differs from Version **V6** as it considers the literal signatures; version **V8** considers both signatures and 2-signatures.

[4] Note that the determinate matching heuristics in the primal θ-subsumption problem [8] corresponds to a particular case of the minimal domain heuristics with regard to the dual CSP.

5 Experimental Validation

5.1 Experimental Setting

As mentioned earlier, CSP algorithms are mainly tested in the PT region, which concentrates the hardest on average problems.

Following [3], artificial data were constructed to examine the algorithm behavior within and outside the PT region. Artificial θ-subsumption problems (pairs (hypothesis $\mathcal{C}$, example Ex)), are constructed from four order parameters: the number n of variables in $\mathcal{C}$; the number m of literals in $\mathcal{C}$, all built on distinct predicate symbols; the number N of literals built on each predicate symbol in Ex; the number L of constants in Ex.

In order to keep the total computational cost beyond reasonable limits, n is set to 10, N is set to 100, m varies in $[10, 50]$ and L varies in $[10, 50]$. For each pair (m, L), 1,000 pairs (hypothesis $\mathcal{C}$, example Ex) are constructed with random uniform distribution [3], and $cost(m, L)$ is reported as the average θ-subsumption cost over all 1,000 trials, measured in seconds (*Django* is implemented in C^{++} and runs on a PC Pentium2). All *Django* versions are experimented and compared with three θ-subsumption reference algorithms, respectively SLD Prolog, determinate matching [8] and graph contexts [18]; in the latter cases, we used the algorithm implementation kindly given by T. Scheffer.

5.2 Results and Discussion

As might have been expected, Prolog SLD does not keep up when hypothesis $\mathcal{C}$ involves more than a few literals, and it had to be stopped for $m > 5$ (being reminded that example Ex involves $100 \times m$ literals). Determinate matching [8] does significantly better than SLD for small size hypotheses; however, it runs out of resources for $m > 10$; in retrospect, this heuristics is poorly suited to the random structure of the examples.

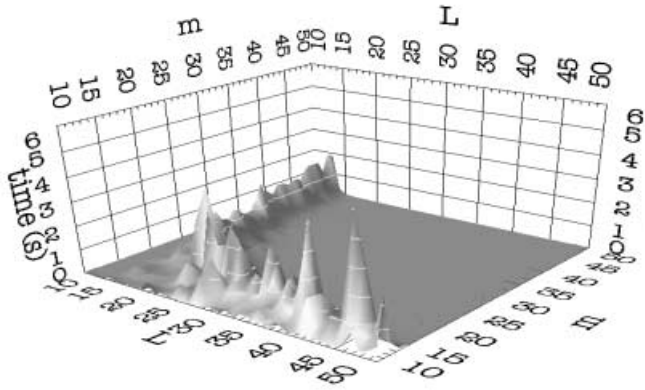

Fig. 1. θ-subsumption cost(m, L) for *Django*.**V1**, averaged on 1,000 pairs $(\mathcal{C}, Ex)$ with m : nb predicates in $\mathcal{C}$ in[10,50], L nb constants in Ex in[10,50]

Graph contexts also turned out to be hardly applicable, mostly for efficiency reasons (see below); finally only the maximal clique search (MCS) [18] was experimented in the same range as *Django*.

The behavior of each algorithm is conveniently pictured as the surface $cost(m, L)$. Fig. 1 displays the cost landscape obtained for the baseline version of *Django*.

In a CPS perspective [7], three complexity regions are distinguished. The PT region appears as a mountain chain of hyperbolic shape in the (m, L) plane; it concentrates the hardest on average θ-subsumption problems.
The *YES* region, besides the PT (for low values of m or L), contains trivial (hypothesis,example) pairs, where the hypothesis almost surely subsumes the example; in this region typically lie overly general, complete and incorrect hypotheses wrt the dataset.
The *NO* region, beyond the PT, contains trivial (hypothesis,example) pairs, where the hypothesis almost never subsumes the example; in this region lie the hypotheses covering no training examples, which are thus found to be correct.

The cost landscape obtained for MCS [18] is depicted in Fig. 2 (higher costs by factor 6 compared to Fig. 1). Interestingly, the phase transition region is larger than for *Django*.**V1**. Note that the complexity is not negligible in the *NO* region. This suggests that MCS does not early detect the inconsistencies, achieving unnecessary exploration of the substitution graph in the *NO* region.

On the other hand, MCS [18] first step concerns the construction of the whole substitution graph, which is exploited in the second step along a maximal clique search. This first step is computationally heavy; it represents a significant amount of the total cost, unless the problem size is large. For large-size prob-

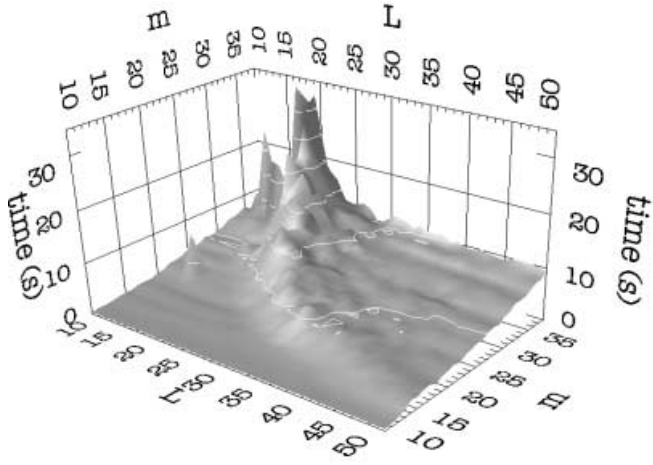

Fig. 2. θ-subsumption cost(m, L) for MCS, averaged on 1,000 pairs $(\mathcal{C}, Ex)$ m : nb predicates in $\mathcal{C}$ in[10,38], L nb constants in Ex in[10,50] (scale factor $\times$ 6 compared to Fig. 1)

lems, the graph construction effort is negligible compared to the maximal clique search, and worthwhile as it significantly speeds up the maximal clique search. Unfortunately, the memory resources needed to store the substitution graph for large problems, are hardly tractable; no 2-graph contexts could be used with MCS for $m > 10$.

In contrast, *Django* interleaves the search and the constraint propagation; this way, the construction of the whole graph is avoided whenever a solution might be found along the search.

The θ-subsumption costs (with multiplicative factor 100) are summarized in Table 2, averaged over all three regions.

Table 2. Average θ-subsumption cost ($\times 100$) in the *YES*, *NO* and PT regions.

	YES region		Phase Transition		NO region	
	cost	$\pm$	cost	$\pm$	cost	$\pm$
MCS	344.69	589.93	841.53	1325.83	800.74	908.22
Django.**V1**	20.25	32.80	116.83	142.01	6.67	17.25
V2	4.18	5.97	4.99	6.92	2.19	2.71
V3	4.80	6.70	8.79	11.84	2.44	3.33
V4	4.25	6.09	4.56	6.45	2.22	2.75
V5	4.51	6.44	6.70	9.77	2.33	3.09
V6	4.24	5.86	4.79	6.32	1.98	2.45
V7	2.20	3.15	3.53	4.55	1.58	1.93
V8	2.48	3.78	3.15	4.40	0.95	1.41

Cost (m, L) is counted in the *YES*, PT or *NO* region, depending on the fraction f of clauses $\mathcal{C}$ subsuming examples Ex, over all pairs $(\mathcal{C}, Ex)$ generated to estimate $cost(m, L)$ (with *YES* region $=_d$ $[f > 90\,\%]$; PT region $=_d$ $[f \in [10\,\%,\ 90\,\%]]$; *NO* region $=_d$ $[f < 10\,\%]$).

Some care must be exercised when interpreting the results, due to the high variability of the measures; this variability was hardly reduced by increasing the number of experiments for a given pair (m, L) from 100 to 1,000 trials.

This variability is explained as only "simple" Forward Checking and Arc-Consistency heuristics were considered. Though these heuristics are very efficient on average, they do not manage well with "pathological" cases, which considerably increases the average resolution cost. Further experiments will consider the use of Maintained Arc Consistency heuristics, and see whether the gain achieved on hard θ-subsumption problems (as MAC is optimal with regard to worst-case complexity) compensates for the loss on easier problem instances.

The phase transition phenomenon is most marked for *Django*.**V1** (Fig. 1), the cost in the PT region being 10 times the cost in the *YES* region and 20 times the cost in the *NO* region. Note that the performance gain of *Django*.**V1** over MCS is not uniform; the gain factor is about 30 in the *YES* region, 7 in the PT, and 200 in the *NO* region.

The addition of dynamic variable ordering heuristics visibly improves the performances, especially in the PT region, smoothening the complexity peak. Other heuristics, especially signature-based heuristics, also seem to contribute to the global efficiency[5]. The best gain factor compared to MCS is about 100 in the *YES* region, 200 in the PT region, and 700 in the *NO region* (Fig. 3).

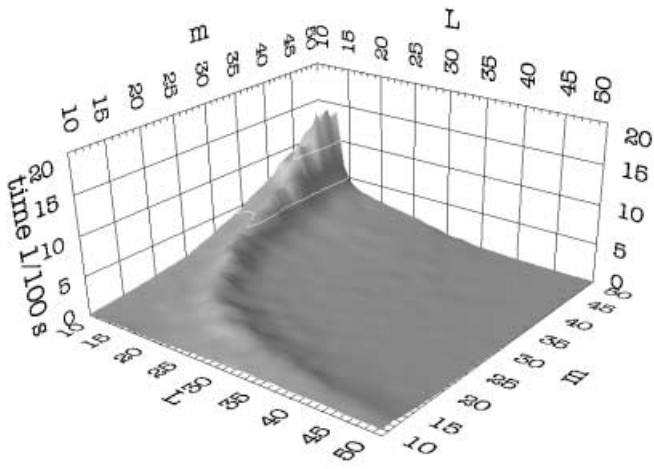

Fig. 3. θ-subsumption cost(m, L) for *Django*.**V8**, averaged on 1,000 pairs $(\mathcal{C}, Ex)$ (scale factor $\div$ 25 compared to Fig. 1)

Similar gain factors have been obtained for experiments (not shown for space limitations) on the N-queen problem, for $N = 10..30$.

Last, artificial problems derived from the real-world Mutagenesis problem [20] have been used to compare *Django* and MCS. Each hypothesis $\mathcal{C}$ considered involves m literals and n variables, where m and n respectively range in 1..10 and 2..10; $\mathcal{C}$ is tested against all 229 examples in the training set. For a given m and n, $\mathcal{C}$ is randomly generated from m *bond* literals $bond(X_i, X_j)$, where X_i and X_j are each selected among n variables in such a way that $X_i \neq X_j$ and $\mathcal{C}$ is connected.

Results obtained with *Django* show the presence of phase transition when the number of literals and variables in the hypothesis are around 4 and 5 respectively, though this change in the covering probability is not coupled with a complexity peak. The worst effective complexity is observed for hypotheses with n literals and $n + 1$ variables (chains of atoms).

MCS obtains good results on the "artificial mutagenesis" θ-subsumption problems. Since a single predicate symbol is actually considered, the substi-

[5] This contrasts with the inefficiency observed for graph contexts [18], though formally equivalent to signatures. However, this seems to be mostly due to implementation matters: for the sake of generality, 1-neighborhoods are implemented as lists of lists, whereas signatures are coded as boolean vectors.

tution graph is not relevant, and the maximal clique search efficiently solves the search. On this problem, *Django* outperforms MCS by a gain factor between 50 and 700.

5.3 Scope and Relevance of the Experimental Study

Artificial problems considered in the paper differ from real-world θ-subsumption problems encountered in ILP in three respects.

In the general case, a major issue is to decompose the problem at hand into fewly or not related subproblems [6,2] (e.g. decomposing the hypothesis into k-local components [5,8]), as successful decomposition entails exponential savings in the resolution cost.
In this study, artificial problems are designed in such a way that they are *not* decomposable into two or more disjoint CSPs [3]. The θ-subsumption average cost reported for a given m-literal clause and L-constant example thus corresponds to a pessimistic (non-decomposable case) estimate.

A second issue regards the uniform distribution of the θ-subsumption problems considered. Each predicate symbol occurs once in the hypothesis, and $N = 100$ times in the example.
In real-world problems, some predicate symbols occur more frequently than others in the examples. Wrt the dual CSP, this means that constrained variables have domains with diverse sizes. Such a diversity makes CSP heuristics, e.g. constraint propagation or dynamic variable ordering, more effective. In this respect, considering predicate symbols with same number of literals built on them leads to a pessimistic estimate of the average θ-subsumption cost.

The third issue concerns the arity of the predicate symbols, which is restricted to 2 in our artificial setting. With respect to worst-case complexity, the predicate arity does not affect the dual CSP; the dual CSP size is $\mathcal{O}(N^m)$, the size of dual domains exponentiated by the number of dual constraints, which does not depend on the arity. But the predicate arity dictates the number of dual constraints. Assume that a (primal) variable in hypothesis $\mathcal{C}$ occurs in o distinct literals in $\mathcal{C}$; this is accounted for in the dual CSP by $\frac{o \times (o-1)}{2}$ dual constraints[6].

Assuming that all predicates are k-ary, their $m \times k$ arguments are selected among n primal variables. Assuming this selection is uniform (which is not as one has to ensure the clause connectivity), each variable intervenes on average in $\frac{m \times k}{n}$ literals in $\mathcal{C}$, and thus the total number of dual constraints is $\mathcal{O}\left(\frac{m^2 \times k^2}{n}\right)$. According to this preliminary analysis, increasing the predicate arity by a factor $\sqrt{t}$ can be likened to decreasing the number of variables by factor t. Experimentally, decreasing the number of variables causes the phase transition to move toward shorter hypotheses everything else being equal (left region in Fig. 1), with exponential decrease of the complexity peak [3].

[6] Note that if a primal variable occurs twice in a single literal p in $\mathcal{C}$, this amounts to a dual unary constraint on the dual constrained variable Y_p, directly accounted for by reducing the associated dual domain.

In summary, the artificial θ-subsumption problems considered were meant to study the worst *average* case with respect to decomposability and distributional diversity.

6 Conclusion and Perspectives

This paper presents a new θ-subsumption algorithm, *Django*, operating on a constraint satisfaction-like representation of θ-subsumption. *Django* combines well-known CS heuristics (arc consistency, forward checking and dynamic variable ordering) with θ-subsumption-specific heuristics (signatures). Intensive experimental validation on artificial worst average instances show that *Django* outperforms previous θ-subsumption algoritms [8,18] by several orders of magnitude in computational cost.

This computational gain might be a good news as ILP systems routinely perform thousands of subsumption tests.

Even more interesting is the fact that the θ-subsumption complexity gives indications regarding the current situation of the ILP search, as located in the *YES*, PT or *NO* regions after the CSP framework [7].

This might open several perspectives to ILP.
On one hand, the relevance of the *YES*, *NO* and PT regions might be questioned in regard to real-world examples, whose distribution model could be arbitrarily different from the uniform model used in the artificial problems. How to characterize and exploit a generative model in order to refine and simplify the representation of an ILP problem, is investigated in the field of reformulation and abstraction (see [17] among others).

On the other hand, it appears reasonable that, unless the target concept belongs to the *YES* region, relevant hypotheses lie in the PT region. This is due to the fact that most ILP learners prefer most general hypotheses provided that they are sufficiently correct (Occam's Razor); therefore, no learner will engage in the *NO* region. In this perspective, new refinement operators directly searching the PT region would be most appreciated.

Further research is first concerned with improving *Django*, checking whether other CS heuristics such as path-consistency are appropriate to θ-subsumption. In the same spirit, the CSP translation proposed for θ-subsumption will be extended to θ-reduction. The idea is that matching a clause with itself might give some information about the redundant literals.

Another perspective is to use *Django* to compare alternative representations for an ILP problem, and select the representation with minimal θ-subsumption cost for randomly generated hypotheses.
Last, an interesting question is whether and how the partial results of *Django* (values or variable links leading to most failures) can be used to navigate in the PT region, by repairing a clause into a clause with same complexity.

Acknowledgments

We gratefully acknowledge Lorenza Saitta, Attilio Giordana and Marco Botta, for many lively discussions about Phase transitions and its consequences on ILP. Thanks also to Tobias Scheffer, who kindly gave us his implementation of graph contexts and determinate matching.

References

1. C. Bessière and J.-C. R/'egin. Mac and combined heuristics: Two reasons to forsake FC (and CBJ ?) on hard problems. In *2nd Int. Conf. on Principles and Practice of Constraint Programming*, pages 61–75, 1996.
2. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
3. A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 2:217–251, 2000.
4. A. Giordana, L. Saitta, M. Sebag, and M. Botta. Analyzing relational learning in the phase transition framework. In P. Langley, editor, *17th Int. Conf. on Machine Learning*, pages 311–318. Morgan Kaufmann, 2000.
5. G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, 1985.
6. M. Gyssens, P. G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.
7. T. Hogg, B. Huberman, and C. Williams. Phase transitions and the search problem. *Artificial Intelligence*, 81:1–15, 1996.
8. J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In W. Cohen and H. Hirsh, editors, *11th Int. Conf. on Machine Leaning*. Morgan Kaufmann, 1994.
9. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
10. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.
11. C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. de Raedt, editor, *Advances in ILP*, pages 82–103. IOS Press, 1996.
12. S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer Verlag, 1997.
13. G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163, 1970.
14. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
15. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
16. J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
17. L. Saitta and J.-D. Zucker. Semantic abstraction for concept representation and learning. In AAAI, editor, *Symposium on Abstraction, Reformulation and Approximation (SARA98)*, 1998.

18. T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient θ-subsumption based on graph algorithms. In S. Muggleton, editor, *Proceedings Int. Workshop on Inductive Logic Programming*. Springer-Verlag, 1997.
19. M. Sebag and C. Rouveirol. Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning*, 38:41–62, 2000.
20. A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg, and R.D. King. Theories for mutagenicity: a study in first order and feature-based induction. *Artificial Intelligence*, 85:277–299, 1996.
21. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.