Fast Theta-Subsumption with Constraint Satisfaction Algorithms

Jérôme Maloberti (jerome.maloberti@lri.fr) and Michèle Sebag (michele.sebag@lri.fr)
Laboratoire de Recherche en Informatique, CNRS UMR 8623
Bât 490, Université Paris-Sud, F-91405 Orsay

Abstract. Relational learning and Inductive Logic Programming (ILP) commonly use as covering test the θ-subsumption test defined by Plotkin. Based on a reformulation of θ-subsumption as a binary constraint satisfaction problem, this paper describes a novel θ-subsumption algorithm named *Django*[1], which combines well-known CSP procedures and θ-subsumption-specific data structures. *Django* is validated using the stochastic complexity framework developed in CSPs, and imported in ILP by Giordana et Saitta. Principled and extensive experiments within this framework show that *Django* improves on earlier θ-subsumption algorithms by several orders of magnitude, and that different procedures are better at different regions of the stochastic complexity landscape. These experiments allow for building a control layer over *Django*, termed Meta-*Django*, which determines the best procedures to use depending on the order parameters of the θ-subsumption problem instance. The performance gains and good scalability of *Django* and Meta-*Django* are finally demonstrated on a real-world ILP task (emulating the search for frequent clauses in the mutagenesis domain) though the smaller size of the problems results in smaller gain factors (ranging from 2.5 to 30).

1. Introduction

Supervised learning relies heavily on the generality relation, stating whether a hypothesis covers a (training) example. As the assessment of a hypothesis depends on its coverage, the implementation of the generality relation, or covering test, must be efficient.

The complexity of the covering test is among the greatest challenges facing supervised learning from relational examples, or *Inductive Logic Programming* (ILP) (MDR94; NCdW97). ILP systems commonly use as generality relation, a decidable restriction of logical implication named θ-subsumption (Plo70), with exponential worst-case complexity in the size of the candidate hypothesis. This drawback has motivated the use of learning biases, more precisely syntactic biases and/or search biases favoring short hypotheses (NRA$^+$96; RB93; Qui90; Mug95; BR98; Sri01). In parallel, efficient implementations of θ-subsumption have been proposed (KL94; SHW97); and ILP systems based on an approximation of θ-subsumption have been developed, see for instance (SR00; KLF01).

[1] The *Django* system is available at URL http://www.lri.fr/~malobert/.

This paper presents a new correct and complete algorithm implementing θ-subsumption, termed *Django*. The originality of the approach is that it formalizes θ-subsumption as a Constraint Satisfaction problem (Tsa93), and thoroughly exploits efficient CSP algorithms in order to resolve θ-subsumption. *Django* is evaluated along an original framework developed by A. Giordana and L. Saitta (GS00), which transposes in the context of ILP the principled framework developed in the Constraint Satisfaction literature since the early 90's (CKT91; HHE96).
In this framework, the focus is shifted from a worst-case complexity analysis to a statistical complexity analysis. This new complexity paradigm is based on the definition of order parameters. Typically, the order parameters associated to θ-subsumption are the number of variables and predicates in the hypothesis, the number of literals and constants in the example. Computational complexity is thus viewed as a random variable, ranging over the empirical complexity of all problem instances with same order parameters.

The θ-subsumption landscape associated to these order parameters has been investigated experimentally and intensively (GS00). In accordance with earlier findings in CSPs (HHE96), three complexity regions appear. In a wide region, corresponding to under-constrained, satisfiable CS problems (the hypothesis is "short", or general, relatively to the example) θ-subsumption almost always succeeds. In another wide region, corresponding to over-constrained, unsatisfiable CS problems (the hypothesis is "long", or specific, relatively to the example), θ-subsumption almost always fails. These two wide regions are respectively referred to as YES and NO regions; they are separated by a narrow region, termed phase transition (PT), where the probability for θ-subsumption to succeed abruptly drops from almost 1 to almost 0.

The main goal of the paper is to provide a θ-subsumption algorithm with good performance in the PT region, without sacrificing the performance in the YES and NO regions. This goal is motivated by two remarks. Firstly, after the CSP literature, the PT region concentrates the hardest problem instances on average, i.e. the most challenging problems; therefore the scalability of a novel θ-subsumption algorithm must be investigated in this particular region. Secondly, experiments on artificial relational learning problems suggest that this PT region is particularly relevant to ILP (BGSS03), for it includes most of the sought hypotheses. Indeed, relevant hypotheses are unlikely to belong to the YES or NO regions since hypotheses in these regions tend to cover all or none of the examples.

Experimental results show that the *Django* algorithm improves on previous θ-subsumption algorithms (KL94; SHW97) by one or several orders of magnitude in all three regions of the θ-subsumption landscape. These results are confirmed by experimentations in the real-world mutagenesis domain (KSS95).

Lastly, as could have been expected, there is nothing like a universally efficient CSP procedure: some procedures are more efficient than others in particular regions of the θ-subsumption landscape. The respective efficiency of the procedures is demonstrated by their *competence map*, summarizing principled experimentations inspired by (GS00). Based on these competence maps, a control layer termed Meta-*Django* is defined; it achieves the built-in selection of the best *Django* procedures depending on the problem instance at hand (Section 7). This way, Meta-*Django* benefits from the most efficient procedures implemented in *Django*, and does not suffer from their computational overhead in the regions where these procedures are not adequate. Meta-*Django* is validated on the well-known ILP benchmark, the mutagenesis domain (KSS95).

The paper is organized as follows. The next section briefly introduces θ-subsumption and reviews existing θ-subsumption algorithms (KL94; SHW97). Section 3 presents the Constraint Satisfaction framework and the main algorithms used to solve CSPs. Section 4 presents the *Django* algorithm, first described in (MS01). *Django* first transforms the θ-subsumption problem referred to as *primal problem*, into a binary CSP referred to as *dual problem*; the dual problem is then handled using diverse procedures developed for binary CSPs and θ-subsumption-specific data structures. The experimental setting inspired by (BGSS03), is described and discussed in Section 5, together with the experimental goals. Empirical results are reported in Section 6 and discussed with respect to the state of the art. Section 7 is devoted to Meta-*Django*. Some related works are discussed in Section 8 and the paper ends with some perspectives for further research.

2. Theta-subsumption, definition and algorithms

After defining θ-subsumption, this section reviews the main algorithms proposed in the ILP literature for optimizing the θ-subsumption test.

2.1. Definition and Standard subsumption test

In the rest of the paper, $\mathcal{C}$ and Ex denote Horn clauses in Datalog language (including no function symbols other than constants).

DEFINITION 1. Theta-subsumption
Clause $\mathcal{C}$ θ-subsumes Ex iff there exists a substitution $\theta = \{X_i/V_{i_j}\}$,

where X_i ranges over the variables in $\mathcal{C}$ and V_{i_j} ranges over the variables and constants in Ex, such that $\mathcal{C}\theta \subseteq Ex$.

Example 1.

$\mathcal{C}$ $tc(X_0) : -atm(X_0, X_1), atm(X_0, X_2), atm(X_0, X_3), bond(X_1, X_2), bond(X_1, X_3)$

Ex $tc(m) : -atm(m, m_1), atm(m, m_2), bond(m_1, m_2)$

$\mathcal{C}$ θ-subsumes Ex according to substitution $\theta = \{X_0/m,\ X_1/m_1,\ X_2/m_2,\ X_3/m_2\}$.

The natural search space for θ-subsumption therefore is the set of substitutions, mapping the set of variables in $\mathcal{C}$ noted $arg(\mathcal{C})$, onto the set of variables and constants in Ex, noted $arg(Ex)$.

A more computationally efficient approach is to consider the set of mappings from the set of literals in $\mathcal{C}$ onto the set of literals in Ex (built on the same predicate symbol), referred to as literal mappings. In the above toy example, each one of the three literals $atm(X_0, X_1)$, $atm(X_0,X_2)$, $atm(X_0, X_3)$ is mapped onto either $atm(m, m_1)$ or $atm(m, m_2)$, and both literals $bond(X_1, X_2)$ and $bond(X_1, X_3)$ are mapped onto $bond(m_1,m_2)$.

Through a literal mapping, each variable in $\mathcal{C}$ is associated to a set of variables and constants in Ex. The literal mapping is said to be consistent if it associates each variable in $\mathcal{C}$ to a single variable or constant in Ex.

DEFINITION 2. Consistency of a literal mapping
A mapping $m = \{p_i/p'_i\}$ from the literals in $\mathcal{C}$ onto the literals in Ex is said to be consistent *if there exists a substitution θ such that, for each literal p_i in $\mathcal{C}$, $p'_i = p_i\theta$.*

The standard algorithm for θ-subsumption is based on Prolog SLD resolution (Rob65). Within SLD resolution, all mappings from the literals in $\mathcal{C}$ onto the literals in Ex built on the same predicate symbol are constructed and checked for consistency. The order of construction depends on the order of the literals in $\mathcal{C}$ and Ex; as known by all Prolog programmers, this order has a significant impact on the SLD efficiency.

2.2. Determinate Matching

Kietz and Lubbe have proposed a first improvement over SLD resolution, known as determinate matching (KL94). In determinate matching, the literals in $\mathcal{C} = p_1..p_K$ are reordered in such a way that, if possible, for each p_i in $\mathcal{C}$ there exists a single candidate literal p'_i in Ex which is consistent with the previous assignments, i.e. such that $\{p_1/p'_1, ..p_i/p'_i\}$

is consistent; p_i is then termed determinate literal. After all determinate literals in $\mathcal{C}$ have been mapped onto literals in Ex, and if necessary, the search resumes using SLD resolution.

Clause $\mathcal{C}$ is said to be determinate *with respect to* Ex if it only contains determinate literals wrt Ex.

Example 2.

$$\begin{array}{ll} \mathcal{C}: & tc(X_0) \leftarrow p(X_0, X_1), p(X_1, X_2), p(X_2, X_3) \\ Ex: & tc(a_0) \leftarrow p(a_0, a_1), p(a_1, a_2), p(a_2, a_3) \end{array}$$

It is clear that $\mathcal{C}$ θ-subsumes Ex according to substitution $\theta = \{X_0/a_0, X_1/a_1, X_2/a_2, X_3/a_3\}$: determinate matching succeeds for clauses $\mathcal{C}$ and Ex.

The determinacy property is desirable as it guarantees the polynomial complexity of the θ-subsumption test. The limitation is that this property depends on the example considered.

Example 3.

$$\begin{array}{ll} \mathcal{C}: & tc(X_0) \leftarrow p(X_0, X_1), p(X_1, X_2), p(X_2, X_3) \\ Ex: & tc(a_0) \leftarrow p(a_0, a_1), p(a_1, a_2), p(a_2, a_3), p(a_0, a_4) \end{array}$$

Adding literal $p(a_0, a_4)$ to the body of Ex causes the determinacy test to fail as $p(X_0, X_1)$ now admits $p(a_0, a_1)$ and $p(a_0, a_4)$ as candidate literals.

2.3. Graph Context

In order to circumvent this problem, Scheffer et al. (SHW97) propose a method similar to the *vertex invariants* (RC77) used in graph isomorphism to prune candidate literals and thus, extend the scope of determinate matching. The idea is the following: when graph isomorphisms are required to preserve some attributes for the vertices or nodes in the graph, such as the degree and label of a node, or the labels of the adjacent vertices, these attributes can be used to prune and partition the candidate set into equivalence classes.

(SHW97) build two graphs respectively associated to $\mathcal{C}$ and Ex, where the vertices correspond to literals, and the edges connect all pairs of literals that share a variable in $\mathcal{C}$ (resp. a variable or a constant in Ex).

To each literal p in $\mathcal{C}$ (resp. in Ex) is associated its neighborhood or graph context, recursively defined as follows. The 1-neighbors of p

are all literals in $\mathcal{C}$ (resp. in Ex) adjacent to p. The i-th neighbors are recursively constructed, as 1-neighbors of $(i-1)$-neighbors.

One easily shows that a necessary condition for a consistent literal mapping m is that it maps every literal p in $\mathcal{C}$ onto literal p' in Ex, such that all predicate symbols occurring in p' i-neighbors also are i-neighbors of p.

Example 4. Let $\mathcal{C}$ and Ex be defined as in Example 3, and let us consider their associated *graph context of depth 1*, noted 1-GC.
The 1-GC of $\mathcal{C}$ gives for each literal and each variable X in this literal, the set of pairs (predicate p, argument k) such that X also appears in a literal built on predicate p as k-th argument:

$$\begin{aligned} tc(X_0):&\ \{\langle 1/p.1\rangle\} \\ p(X_0,X_1):&\ \{\langle 1/tc.1\rangle, \langle 2/p.1\rangle\} \\ p(X_1,X_2):&\ \{\langle 1/p.2\rangle, \langle 2/p.1\rangle\} \\ p(X_2,X_3):&\ \{\langle 1/p.2\rangle\} \end{aligned}$$

The 1-GC of literal $tc(X_0)$, i.e. $1/p.1$, indicates that literal $tc(X_0)$ is linked by its first variable to the first variable of a literal built on predicate symbol p.
Likewise, the 1-GC of Ex (Example 3) is given as:

$$\begin{aligned} tc(a_0):&\ \{\langle 1/p.1\rangle\} \\ p(a_0,a_1):&\ \{\langle 1/tc.1\rangle, \langle 1/p.1\rangle, \langle 2/p.1\rangle\} \\ p(a_0,a_4):&\ \{\langle 1/tc.1\rangle, \langle 1/p.1\rangle\} \\ p(a_1,a_2):&\ \{\langle 1/p.2\rangle, \langle 2/p.1\rangle\} \\ p(a_2,a_3):&\ \{\langle 1/p.2\rangle\} \end{aligned}$$

This example shows that the use of graph contexts extends the scope of determinate matching. Literal $p(X_0,X_1)$ now admits a single candidate, $p(a_0,a_1)$; candidate $p(a_0,a_4)$ is pruned as its 1-GC does not contain that of $p(X_0,X_1)$.

As graph contexts help to reduce the set of candidate literals, their use extends the set of θ-subsumption problems that can be resolved in polynomial time. Nevertheless, the determinacy of the θ-subsumption test still depends on the clause Ex at hand.

Example 5.

$$\begin{aligned} \mathcal{C}:&\quad tc(X_0) \leftarrow p(X_0,X_1), p(X_1,X_2), p(X_2,X_3) \\ Ex:&\ \ tc(a_0) \leftarrow p(a_0,a_1), p(a_0,a_2), p(a_1,a_2), p(a_2,a_3) \end{aligned}$$

In this example, literal $p(a_0,a_2)$ cannot be pruned on the basis of its 1-GC. Graph contexts of depth 2 would be required to ensure determinate matching.

In many cases, the matching test can be made deterministic by developing the graph context up to some depth d; but the construction of the graph context is exponential in d.

2.4. Other approaches

Scheffer et al. further define a *substitution graph*-based θ-subsumption test (SHW97). The nodes of the substitution graph are all pairs of literals (p/p') where p and p' respectively are literals in $\mathcal{C}$ and Ex built on the same predicate symbol. Two nodes (p/p') and (q/q') are linked by an edge iff mapping $(p/p', q/q')$ is consistent (SHW97).

In (SHW97), the SLD search is replaced by a maximal clique search in the substitution graph. The worst-case complexity remains exponential, but the advantage is that the consistency check is performed only once.

Another approach presented by Kietz and Lubbe proceeds by decomposing the substitution graph into mutually independent components (*k-locality*) (KL94). Whenever such a decomposition applies, it significantly reduces the complexity of the θ-subsumption test.

Indeed, all above heuristics can be viewed as particular cases of Constraint Satisfaction algorithms. Since we aim at a principled exploitation of CSP algorithms for θ-subsumption, the next section is devoted to a brief presentation of CS.

3. Constraint Satisfaction Problem

This section introduces Constraint Satisfaction problems (CSPs) together with the main resolution algorithms; the reader is referred to (Tsa93) for a comprehensive presentation.

3.1. Overview of CSPs

DEFINITION 3. Constraint Satisfaction Problem *A CSP involves:*

- *A set of n variables $\mathcal{X} = \{X_1, \ldots, X_n\}$; to each variable X_i is associated its variable domain $dom(X_i)$, including all possible values for X_i ;*

- *A set of m constraints, specifying the simultaneously admissible values of the variables.*

Each constraint r is defined on some variables in $\mathcal{X}$:

- *it can be thought of as a k-ary predicate $r(X_{i_1}, ..X_{i_k})$;*

– *the constraint scope noted* $arg(r)$ *is the set of variables* $\{X_{i_1},..X_{i_k}\}$*;*

– *the set of admissible values for the variables in the constraint scope is termed the constraint relation, noted* $rel(r)$*. Constraint relation* $rel(r)$ *can be thought of as a set of* N *literals* $\{r(a_{1,i_1},..a_{1,i_k}),\ldots, r(a_{N,i_1},..a_{N,i_k})\}$*, with* a_{l,i_j} *a value in* $dom(X_{i_j})$ *for* $l = 1..N$ *and* $j = 1..k$*.*

Although a constraint relation can be infinite in the general case (e.g. a numerical constraint on real-valued variables), only finite constraint relations are considered in the rest of the paper.

A CSP is *satisfiable* iff it admits a solution, assigning to each variable X_i a value a_i in $dom(X_i)$ such that all constraints are satisfied. Such a solution 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 $rel(r)$. *In other words, the CSP defined by constraints* $r_1,..r_m$ *is satisfiable iff conjunction* $r_1,..,r_m$ θ*-subsumes the conjunction of all literals in* $rel(r_1),\ldots,rel(r_m)$*.*

Example 6. Let us consider the CSP defined as:
- $\mathcal{X} = \{X_0, X_1, X_2, X_3\}$, where each variable has domain $\{m, m_1, m_2\}$;
- six constraints are defined as follows: tc is defined on X_0, with $rel(tc) = \{m\}$; atm_1 (respectively atm_2 and atm_3) is defined on (X_0, X_1) (resp. (X_0, X_2) and (X_0, X_3)), with domain $\{\langle m, m_1\rangle, \langle m, m_2\rangle\}$; $bond_1$ and $bond_2$ are respectively defined on (X_1, X_2) and (X_1, X_3) with domain $\{\langle m_1, m_2\rangle\}$.

The above CSP is equivalent to the θ-subsumption test in Example 1, and its solution is given by $\theta = \{X_0/m,\ X_1/m_1,\ X_2/m_2,\ X_3/m_2\}$.

Two CSPs are *equivalent* iff they are defined on the same variables and admit the same solutions. As any CSP can be embedded into a binary CSP, i.e. involving only binary constraints, most CS algorithms consider binary CSPs. In the following, we assume with no loss of generality that at most one constraint is set on any variable pair.

Complexity. The CSP complexity is exponential in the number n of variables, and linear in the number m of constraints. More precisely, assuming that all variables have the same domain size L, the CSP resolution complexity is $\mathcal{O}(L^n \times m)$. A first way of decreasing the complexity thus is by decomposing the CSP into independent or scarcely related subproblems – hierarchizing the set of variables (GJC94; GLS00) or the set of constraints (DP89) – in the same spirit as k-locality (KL94).

CS algorithms basically combine two kinds of procedures: reduction procedures and search procedures. Reduction procedures reduce the variable domains, thereby transforming a CSP into an equivalent one of lower complexity. Search procedures are concerned with the search strategy, and how to best explore the search space.

3.2. Reduction Procedures

The simplest reduction algorithm, termed *node consistency*, is concerned with unary constraints. Let $r(X)$ be a unary constraint defined on variable X, and let $rel(r)$ denote its set of admissible values. With no loss of generality, the domain of X can be restricted to those values appearing in $rel(r)$. Conversely, r is trivially satisfied if $dom(X)$ is included in $rel(r)$.

Node consistency is extended based on the *2-consistency value* definition.

DEFINITION 4. 2-Consistency of a value
Let X be a constrained variable and let a *denote a value in $dom(X)$. Value* a *is* 2-consistent *(or consistent if there is no ambiguity from the context) if, for every variable Y such that there exists a constraint $r(X, Y)$, there exists some value* b *in $dom(Y)$ such that r(a,b) belongs to $rel(r)$;* b *is termed* support *of* a *wrt r.*

Example 7.

$$\begin{array}{lll} p(X,Y) & rel(p) & = & \{\langle a,b\rangle,\ \langle c,d\rangle\} \\ q(X,Z) & rel(q) & = & \{\langle a,e\rangle,\ \langle f,g\rangle\} \end{array}$$

Value a is consistent for X (supported by b for Y and e for Z); value c is not consistent.

Clearly, if a has no support value with respect to some constraint r involving X, then X cannot be mapped onto a. Therefore, all non 2-consistent values can soundly be removed from $dom(X)$.

The above pruning is achieved by *arc consistency*; this algorithm considers each value in the domain of every variable, and removes all values that are not 2-consistent.

Node and arc consistency can be generalized using k-consistency:

DEFINITION 5. k-Consistency of value tuples
A partial assignment θ assigning a value to a subset $\mathcal{X}_\theta$ of the constrained variables $\mathcal{X}$, is consistent if it violates no constraint, i.e. it satisfies all constraints defined on (a subset of) $\mathcal{X}_\theta$.
A CSP is k-consistent iff for any consistent assignment θ over $k-1$

variables, for any variable X not in $\mathcal{X}_\theta$ there exists a value a_X such that $\theta' = \theta \bigcup \{X/a_X\}$ is consistent.
A CSP is strongly k-consistent if it is j-consistent for all j less than k.

If a consistent partial solution over some $k-1$ variables of a CSP cannot be extended to another variable, then this partial solution can soundly be removed (with all its specializations) from the assignment search space. This pruning algorithm is termed k-consistency.

Particular cases of k-consistency are given below:

1-consistency	=	node consistency
2-consistency	=	arc consistency
3-consistency	=	path consistency

For complexity and implementation reasons, arc consistency is generally preferred to k-consistency with $k > 2$. On one hand, checking k-consistency is exponentially complex in k. On the other hand, k-consistency aims at pruning $(k-1)$-tuples of values, which requires specific data structures for $k > 2$; managing these structures might partially or totally offset the computational gains for some problem instances.

Complexity. The best known worst-case complexity of arc consistency algorithms is $\mathcal{O}(mL^2)$, with m being the number of constraints and L the variable domain size.

Note that in the particular case of tree-structured CSPs, arc consistency is a necessary *and sufficient* condition for the CSP to be satisfiable. More generally, a CSP with an acyclic hyper graph can be solved with polynomial complexity (DP89; GL85).

3.3. Search Procedures

CSP algorithms construct an assignment solution $\{X_i/a_i\}$ through a depth first exploration of the assignment space, referred to as substitution tree. Each node corresponds to a variable X_i; the edge attached to this node is the candidate value a_i tentatively assigned to X_i. On each assignment, 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* algorithms); (ii) the choice of the next variable and candidate value to consider (*look-ahead* algorithms).

Look-back algorithms aim at preventing the repeated exploration of a same substitution subtree on backtracking (*thrashing*). For instance, Conflict directed BackJumping (CBJ) (Pro93) 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 algorithms aim at minimizing the number of assignments considered. The best known look-ahead procedure is constraint propagation, pruning all candidate values which are inconsistent with the current assignment, also known as arc-consistency. This way, inconsistencies are detected earlier and less nodes are visited; on the other hand, the assignment operation is computationally heavier as it involves the constraint propagation check.

Forward checking (FC) employs a limited propagation, only checking the variables connected to the assigned variable (partial arc-consistency). Maintaining Arc Consistency (MAC) extends FC by checking arc-consistency on adjacent variables if a value has been removed from their domain. Again, the overhead due to constraint propagation might offset its advantages on medium-size weakly constrained CSPs. Currently, the CSP resolution algorithms considered as most efficient combine FC and CBJ procedures.

The variable order can be optimized too. The ordering is achieved 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. Variable ordering is most often based on the First Fail Principle (BR96), favoring the variable with the smallest domain. This way, failures will occur sooner rather than later. Last, the candidate values can also be ordered; the value with less conflicts (greatest support) with the other variable domains is commonly preferred.

4. CS Resolution for Theta-subsumption

In this section, we first discuss how to best transform a θ-subsumption problem into a binary CSP, in order to delegate the θ-subsumption resolution to efficient CSP procedures. Based on this transformation, the *Django* algorithm is presented; it combines standard CSP procedures with θ-subsumption specific data structures and procedures.

4.1. Standard Binary Representation

The equivalence of θ-subsumption and CS problems follows from their definitions in Sections 2 and 3. However, θ-subsumption generally involves n-ary predicates, while most CSP procedures consider binary constraints. A specific change of representation is thus required to allow θ-subsumption to exploit standard CS algorithms.

The usual way of binarizing an n-ary CSP replaces each n-ary constraint $r(X_{i_1}, \ldots, X_{i_n})$ using an additional variable Y_r and n additional binary constraints $r_1, \ldots, r_n$ linking $X_{i_1}, \ldots, X_{i_n}$ to Y_r. Constraint $r_j(X_{i_j}, Y_r)$ expresses that X_{i_j} is the j-th argument of the initial constraint r.

Let us illustrate the binarization procedure on an example.

Example 8.

$$\begin{array}{ll} \mathcal{C}: & tc(X_0) \leftarrow p(X_0, X_1), q(X_0, X_2, X_3) \\ Ex: & tc(a_0) \leftarrow p(a_0, a_1), p(a_1, a_2), q(a_0, a_2, a_3), q(a_0, a_1, a_3) \end{array}$$

The θ-subsumption problem is first rewritten into a CSP problem:

- The set of variables $\mathcal{X}$ is $\{X_0, X_1, X_2, X_3\}$,
- The domain variables are defined and reduced using arc consistency (Section 3.2). The domain of X_0 is reduced to a_0 by consistency with the unary constraint tc.
  $dom(X_1) = dom(X_2) = dom(X_3) = \{a_0, a_1, a_2, a_3\}$;
- Two constraints r_1 and r_2 respectively account for literals $p(X_0, X_1)$ and $q(X_0, X_2, X_3)$:

  $r_1(X_0, X_1)$ is associated to $rel(r_1) = \{\langle a_0, a1\rangle, \langle a_1, a_2\rangle\}$
  $r_2(X_0, X_2, X_3)$ is associated to $rel(r_2) = \{\langle a_0, a_2, a_3\rangle, \langle a_0, a_1, a_3\rangle\}$.

This CSP is represented as a hyper graph (Figure 1).

In order to binarize the above CSP, an additional variable is associated to each literal in $\mathcal{C}$. (Though literal $p(X_0, X_1)$ is binary, we still introduce variable Y_p for the sake of pedagogy). The resulting binary CSP (Figure 2) is given as:

- Two additional variables Y_p and Y_q are considered besides variables X_0 to X_3,
- The variable domains associated to Y_p and Y_q are respectively defined as $dom(Y_p) = \{p(a_0, a_1), p(a_1, a_2)\}$, and
  $dom(Y_q) = \{q(a_0, a_2, a_3), q(a_0, a_1, a_3)\}$,

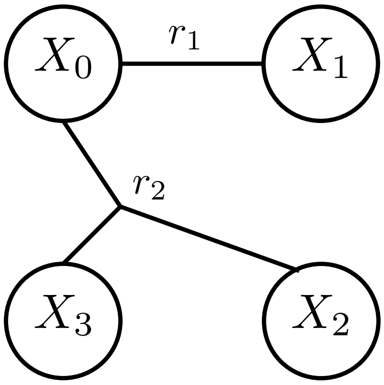

Figure 1. Hyper graph representation of Example 8.

- The set of constraints now includes 5 binary constraints:

  $r_1(X_0, Y_p)$ is associated to $rel(r_1) = \{\langle a_0, p(a_0, a_1)\rangle, \langle a_1, p(a_1, a_2)\rangle\}$;
  $r_2(X_1, Y_p)$ is associated to $rel(r_2) = \{\langle a_1, p(a_0, a_1)\rangle, \langle a_2, p(a_1, a_2)\rangle\}$;
  $r_3(X_0, Y_q)$ is associated to $rel(r_3) = \{\langle a_0, q(a_0, a_2, a_3)\rangle, \langle a_0, q(a_0, a_1, a_3)\rangle\}$;
  $r_4(X_2, Y_q)$ is associated to $rel(r_4) = \{\langle a_2, q(a_0, a_2, a_3)\rangle, \langle a_1, q(a_0, a_1, a_3)\rangle\}$;
  $r_5(X_3, Y_q)$ is associated to $rel(r_5) = \{\langle a_3, q(a_0, a_2, a_3)\rangle, \langle a_3, q(a_0, a_1, a_3)\rangle\}$;

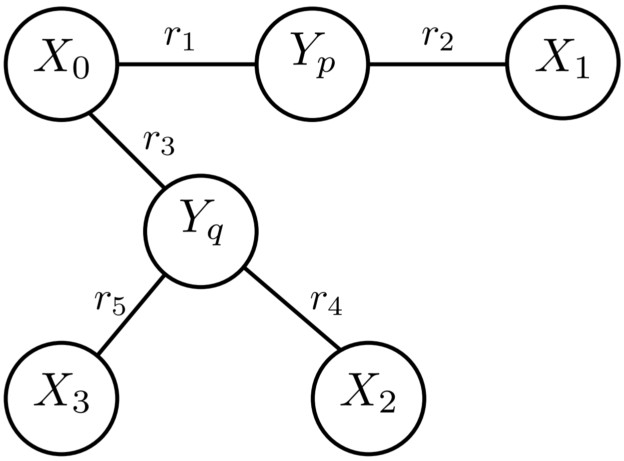

Figure 2. Binary graph representation of Example 8.

This standard binarization creates two types of constrained variables: the actual constrained variables (X_0 to X_3 in the above example) and the additional variables standing for the literals in $\mathcal{C}$ (Y_p and Y_q in the above example). As the θ-subsumption algorithm must handle both types of variables in different ways, standard binarization is not compatible with a straightforward use of CSP algorithms.

4.2. Dual Constrained Representation

For this reason, we devised another transformation of a θ-subsumption problem into a binary CSP, making a clear distinction between literals and variables in $\mathcal{C}$.

Concretely, each θ-subsumption problem $(\mathcal{C}, Ex)$ is transformed into a binary CSP termed *dual CSP*, where:
i) Each literal p in $\mathcal{C}$ gives rise to a constrained variable Y_p termed *dual variable*, as opposed to the variables in $\mathcal{C}$ referred to as *primal variables*.
ii) The domain of a dual variable Y_p is the set of all literals in Ex built on predicate symbol p.
iii) The constraints in the dual CSP, termed *dual constraints*, implements the consistency requirement (Definition 1): literals in $\mathcal{C}$ must be mapped onto literals in Ex in such a way that it induces the mapping of (primal) variables in $\mathcal{C}$ onto a single constant or variable in Ex. Formally,

DEFINITION 6. Dual representation
The dual representation of a θ-subsumption problem $(\mathcal{C}, Ex)$ is defined as follows.

- *To each literal in $\mathcal{C}$ is associated a dual constrained variable. Specifically, to the i-th literal built on predicate symbol p in $\mathcal{C}$ noted $p_{.i}$ is associated the dual constrained variable $Y_{p.i}$. Subscript i will be omitted for readability when there is a single literal built on the predicate symbol p.*

- *The domain of a dual constrained variable $Y_{p.i}$ involves all literals in Ex built on the predicate symbol p. All dual variables related to the same predicate symbol p thus have same domain.*

- *For each pair of dual constrained variables $(Y_{p.i}, Y_{q.j})$ such that literals $p_{.i}$ and $q_{.j}$ in $\mathcal{C}$ share at least one variable in $\mathcal{C}$, a binary dual constraint $r(Y_{p.i}, Y_{q.j})$ is defined. This dual constraint expresses the fact that the assignment of variables $Y_{p.i}$ and $Y_{q.j}$ must be consistent in the sense of Definition 1. (The particular case where a primal variable occurs more than once in a literal $p_{.i}$, is also handled through a binary dual constraint $r(Y_{p.i}, Y_{p.i})$).*

Every dual constraint $r(Y_{p.i}, Y_{q.j})$, similar to the substitution graph in (SHW97), is satisfied by all pairs (p', q') of literals in Ex, such that mapping $\{p_{.i}/p', q_{.j}/q'\}$ is consistent according to Definition 1. In other words, any admissible assignment of (dual) variables $Y_{p.i}$ and $Y_{q.j}$ corresponds to a consistent assignment of the primal variables in $\mathcal{C}$.

Example 9. The θ-subsumption problem presented in Example 8 is now binarized as follows:

- The set of variables is $\mathcal{X} = \{Y_{tc}, Y_p, Y_q\}$,
- The variable domains are: $dom(Y_{tc}) = \{tc(a_0)\}$, $dom(Y_p) = \{p(a_0, a_1), p(a_1, a_2)\}$, $dom(Y_q) = \{q(a_0, a_2, a_3), q(a_0, a_1, a_3)\}$;
- The set of constraints is now:

$r_1(Y_{tc}, Y_p)$ is associated to $rel(r_1) = \{\langle tc(a_0), p(a_0, a_1)\rangle\}$;
$r_2(Y_{tc}, Y_q)$ is associated to $rel(r_2) = \{\langle tc(a_0), q(a_0, a_2, a_3)\rangle, \langle tc(a_0), q(a_0, a_1, a_3)\rangle\}$;
$r_3(Y_p, Y_q)$ is associated to $rel(r_3) = \{\langle p(a_0, a_1), q(a_0, a_2, a_3)\rangle, \langle p(a_0, a_1), q(a_0, a_1, a_3)\rangle\}$;

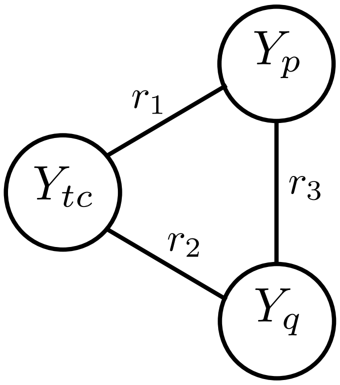

Figure 3. Dual representation of Example 8.

4.3. Discussion

As mentioned above, dual constraints encode the same information as substitution graphs (Section 2.4, (SHW97)). In both cases, the point is to check whether a consistent substitution can be derived from partial literal mappings.

A first difference is that dual constraints are more parsimonious, in the sense that they only consider literals p and q in $\mathcal{C}$ that do share variables. In opposition, substitution graphs systematically consider all pairs of mappings $\{p/p'\}$ and $\{q/q'\}$, where p and q (respectively p' and q') belong to $\mathcal{C}$ (resp Ex).

A second difference is that substitution graphs materially store the list of pairs $(\{p/p'\}, \{q/q'\})$ such that assignment $\{p/p', q/q'\}$ is consistent. Although this list can be pruned by using graph contexts, it remains that the construction and exploitation of graph contexts is computationally expensive in some regions of the θ-subsumption landscape (we shall return to this point in Section 6.3). In opposition, we do not store the domain of relation $r(Y_p, Y_q)$; rather, this constraint is implemented as a function, checking for each pair p' and q' respectively in the domain of Y_p and Y_q whether assignment $\{p/p', q/q'\}$ is consistent.

Dual constraints are exploited through arc consistency, i.e. with complexity $\mathcal{O}(mL^2)$, where m is the number of constraints and L the variable domain size. Arc consistency enables the pruning of the dual variables domains, thereby restricting the search space of consistent mappings.

To further restrict the search space, constraint propagation is activated when a dual variable domain is limited to a single value (literal in Ex). Note that this is always the case for the head of $\mathcal{C}$, which is thus directly handled and propagated over all determinate literals (e.g. dual variable Y_p can only be assigned to $p(a_0, a_1)$ in Example 9, due to dual constraint $r_1(Y_{tc}, Y_p)$ and $dom(Y_{tc}) = \{tc(a_0)\}$).

Proposition.
Let AC-Pol *denote the set of θ-subsumption problems which can be resolved with polynomial complexity using the above binarization combined with arc consistency, and let* GC-Pol *similarly denote the set of θ-subsumption problems which can be resolved with polynomial complexity using graph contexts (SHW97).*

Then, GC-Pol *is a subset of* AC-Pol.

Proof.
By induction, we show that i-GC test is equivalent to arc consistency with $(i-1)$ steps of propagation of the eliminated values.
The 1-GC test of a literal p in $\mathcal{C}$ wrt a literal p' in Ex, built on the same predicate symbol as p, checks whether p' shares the same links as p (variables in $\mathcal{C}$, variables or constants in Ex) with the other literals. Therefore, the 1-GC check succeeds if the arc-consistency check of p' in the value domain of p succeeds.
However, if the arc-consistency test fails, literal p' is removed from the domain of p *— and its elimination is propagated on the adjacent literals.* Since a GC of depth 2 is recursively defined as 1-GC on the neighbors of p and p', p' will be removed from the domain if a value that supports p' is not itself supported and thus, 2-GC test is equivalent to an arc-

consistency check with a propagation limited to one step.
In the general case, a GC of depth i is thus equivalent to arc consistency with $(i-1)$ steps of propagation of the eliminated values.
This shows that *GC-Pol* is a subset of *AC-Pol*.
The strict inclusion is shown as follows. Let $(\mathcal{C}, Ex)$ be a *GC-Pol* problem instance, and consider the addition of a new literal to Ex. In the GC-based approach, this literal can destroy the θ-subsumption determinacy, entailing an exponential complexity of the GC-based resolution of θ-subsumption.
In the dual binarization approach, this literal simply intervenes as an additional value in the (dual) constrained variable domain; this does not modify the complexity class of the problem, as the resolution complexity is quadratic in the variable domain size. □

This shows that the presented approach extends over (SHW97) the set of problems which can be polynomially addressed.

4.4. Signature

An additional data structure is defined to further restrict the search space.

To each literal in $\mathcal{C}$ and Ex is associated a *signature*, similar to the 1-depth graph context (Section 2.3, (SHW97)). The signature encodes the links of the literal with other literals through sharing (primal) variables in $\mathcal{C}$ or constants in Ex. It is clear that a necessary condition for assigning a dual variable Y_p (literal in $\mathcal{C}$) to a dual constant p' (literal in Ex) is that the signature of Y_p be included into that of p'. Signatures thus enable the pruning of variable domain by arc consistency.

For instance in Example 4, the signature associated to $p(X_0, X_1)$ states that the first variable appears in a literal built on symbol tc, position 1, and a literal built on p, position 1; and the second variable appears in a literal built on symbol p, positions 1 and 2.

Contrasting with the generality of graph contexts (SHW97), signatures are restricted to 1-depth graph contexts, which allows for an optimized implementation.

Naturally, signatures suffer from the same limitations as 1-GC:

Example 10.

$$\begin{array}{rl} \mathcal{C}: & p(X,Y), q(Y,X), \\ Ex: & p(x_0,y_0), p(x_1,y_1), q(y_0,x_1), q(y_1,x_0) \end{array}$$

Though all literals in Ex have a signature including that of literals in $\mathcal{C}$, the θ-subsumption test fails.

Additional signatures, termed 2-signatures, are thus defined. The 2-signature associated to a literal p describes the literals sharing at least *two* variables with p; more precisely, it encodes the relative position of all variables in each literal.

For instance, the 2-signature associated to literal $p(X, Y)$ in the above example is $\{\langle q(1.2, 2.1)\rangle\}$; it expresses the fact that $p(X, Y)$ shares two variables with a literal built on symbol q, such that the first variable in $p(X, Y)$ is the second in the literal built on q, and the second variable in $p(X, Y)$ appears as first variable in this same literal. Literals with several occurrences of a primal variable are also handled through signatures. For instance, the signature associated to the literal $p(X, X)$ is $\{\langle 1/p.2\rangle, \langle 2/p.1\rangle\}$.

Like signatures, 2-signatures are used to prune the θ-subsumption search space: a necessary condition for a literal p in $\mathcal{C}$ to be mapped onto a literal p' in Ex, is that the 2-signature associated to p be included in that of p'.

5. Experimental goal and setting

This section describes the experimental setting chosen for the validation of the proposed θ-subsumption approach. This setting is inspired from the now standard setting for validating constraint satisfaction algorithms, which has been imported to the ILP framework by Giordana and Saitta (GS00).

The experimental goals are then discussed.

5.1. CSP validation paradigm

As mentioned in the introduction, the worst- and average-case complexity analysis of constraint satisfaction problems has been extended using a stochastic modeling.

Stochastic complexity modeling relies on the definition of order parameters on CSP problems (Hog96). Two main order parameters are considered: constraint density and constraint tightness. For the sake of clarity, only binary constraints will be considered in the rest of the section.

The constraint density p_1 is defined as the fraction of variable pairs which are related by a constraint; letting n be the number of variables and m the number of constraints,

$$p_1 = \frac{m}{\frac{n(n-1)}{2}} = \frac{2m}{n(n-1)}$$

The constraint tightness p_2 is the average fraction of value pairs ruled out by a constraint; letting L be the size of the variable domain, and N the size of the relation associated to a constraint,

$$p_2 = 1 - \frac{N}{L^2}$$

Stochastic computational complexity is viewed as a random variable depending on the constraint density and constraint tightness parameters. For a given density $p_1 = v_1$ and tightness $p_2 = v_2$, this random variable is realized as the computational effort required to solve any CSP instance with density v_1 and tightness v_2.

Experimentally, a sample of CSP instances associated to each value pair (v_1, v_2) is drawn; every CS algorithm is assessed from its average computational effort over the sample.

Stochastic analysis offers a more realistic perspective on CS complexity than worst- and average-case analysis. It accounts for the fact that, although the worst-case complexity is exponential ($m \times L^n$, where m is the number of constraints, n the number of variables and L the size of the domain variable), still the empirical cost of resolution is moderate for a vast majority of CSP instances (CKT91).

More precisely, the CS complexity landscape defined by the order parameters p_1 and p_2 involves three regions:

- The so-called YES region, including under-constrained problems, is associated to a negligible stochastic complexity: a problem instance admits many solutions on average, and finding one of them is easy.

- The so-called NO region, including over-constrained problems, also is associated to a negligible stochastic complexity: a problem instance admits no solution on average, and showing that there is no solution is easy.

- The YES and NO regions are separated by a narrow region termed the *Phase Transition* region (PT). The probability of satisfiability abruptly drops from 1 (in the YES region) to 0 (in the NO region), and the stochastic complexity reaches a peak in the PT region; most hard-to-solve CS instances lie in this region. The height of the peak depends on the algorithm (see Section 6), while it is conjectured that the location of the peak depends on the distribution used to sample the problem instances.

It must be emphasized that stochastic complexity analysis only delivers general trends. Indeed, there exist satisfiable problems in the NO region and unsatisfiable problems in the YES region; and some problems in the PT can be solved with moderate computational effort, while some problems in the YES or NO regions are hard to solve (GBCK00).

However, this stochastic modeling offers a principled framework to evaluate the performance and regions of competence of a CS algorithm. Further, this framework allows for a visual analysis of the algorithm competence, through a *competence map*, characterizing the average algorithmic performance with respect to the order parameters (see Section 5.3).

Actually, these competence maps can be used to decide which algorithm is best in a particular region. This facility is highly desirable as it is now widely recognized that no CS algorithm dominates over the others in all regions.

5.2. A stochastic complexity model for θ-subsumption

Stochastic modeling of CS problems has been ported to θ-subsumption problems and Inductive Logic Programming by Giordana et al. (GS00), as follows.

A θ-subsumption problem instance is given as a pair (clause $\mathcal{C}$, example Ex) defined as Horn clauses (after the learning by entailment ILP setting (Rae97)). With no loss of generality, only the bodies of $\mathcal{C}$ and Ex are considered thereafter (as noted earlier on, the head of $\mathcal{C}$ can only be mapped onto the head of Ex).
Three assumptions are made to facilitate the definition of order parameters; their implications are discussed further in Section 5.4:

- All literals in $\mathcal{C}$ are built on distinct binary predicate symbols and clause $\mathcal{C}$ is connected. The latter requirement prevents the θ-subsumption problem from being decomposable into simpler problems; its effective complexity is therefore representative of the complexity attached to a given constraint density and tightness.

- Example Ex is a conjunction of ground literals. Each literal in Ex is built on a predicate symbol occurring in $\mathcal{C}$ (other literals are irrelevant to the θ-subsumption problem). The number of literals in Ex per predicate symbol is constant.

- All arguments of all predicates have same value domain.

After these assumptions, four order parameters for θ-subsumption problems $(\mathcal{C}, Ex)$ are defined:

n The number of variables in $\mathcal{C}$;

m The number of literals in $\mathcal{C}$ (= the number of distinct predicate symbols);

N The number of literals built on each predicate symbol in Ex;

L The number of constants in Ex, i.e. the size of each variable domain.

Within the stochastic framework, the algorithmic complexity of θ-subsumption $(\mathcal{C}, Ex)$ is a random variable depending on the sizes of $\mathcal{C}$ (the number m of predicates and the number n of variables) and Ex (the number L of constants and the number N of literals *per* predicate symbol; in total Ex includes $N \times m$ literals).

5.3. The competence map format

The competence map describes the performance of the algorithm in a given region of interest, measured after a sampling mechanism.

The region of interest specifies the ranges for the order parameters n, N, m and L. To keep the total computational cost within reasonable limits, the number n of variables in $\mathcal{C}$ is set to 10 and the number N of literals per predicate symbol is set to 100 (complementary experiments have been done with $n = 12, 15$ and $N = 120, 150$ to confirm the general trends reported in Section 6).

The number m of literals in $\mathcal{C}$ varies in $[10, 50]$ and the number L of constants in example Ex varies in $[10, 50]$. The region of interest thus is defined as $n = 10, N = 100, m \in [10, 50]$ and $L \in [10, 50]$.

This region corresponds to larger θ-subsumption problems than encountered in typical ILP benchmark problems. For instance in the mutagenesis problem (KSS95), the clauses found involve four or five chemical atoms ($n = 4$), with a few literals ($m < 10$) besides the attributes of the molecule (e.g. determinate predicates such as logp); the examples involve two predicate symbols (*atm* and *bond*), $L < 40$ chemical atoms, and $N < 80$ literals per predicate.

However, from a multi-relational learning perspective (DL01), this region corresponds to middle-sized problems ($L \leq 50$ identifiers per example, $m \leq 50$ tables with $N = 100$ rows *per* table, and a target clause involving $n = 10$ independent objects).

The competence map describes the **computational cost** of the algorithm under study. The regions in the competence map refer to the (algorithm-independent) probability of θ-subsumption success, measured from the percentage of instances such that $\mathcal{C}$ θ-subsumes Ex (Figure 4; see also (GS00)). The YES region of under-constrained CS problems (left part of the θ-subsumption landscape) corresponds to "short" clauses $\mathcal{C}$ respectively to "long" examples Ex; accordingly, $\mathcal{C}$

almost surely subsumes Ex. The NO region of over-constrained problems (right part of Figure 4) corresponds to "long" clauses respectively to "short" examples, where $\mathcal{C}$ almost never subsumes Ex. In the PT region, the probability for $\mathcal{C}$ to subsume Ex abruptly drops from 1 to 0.

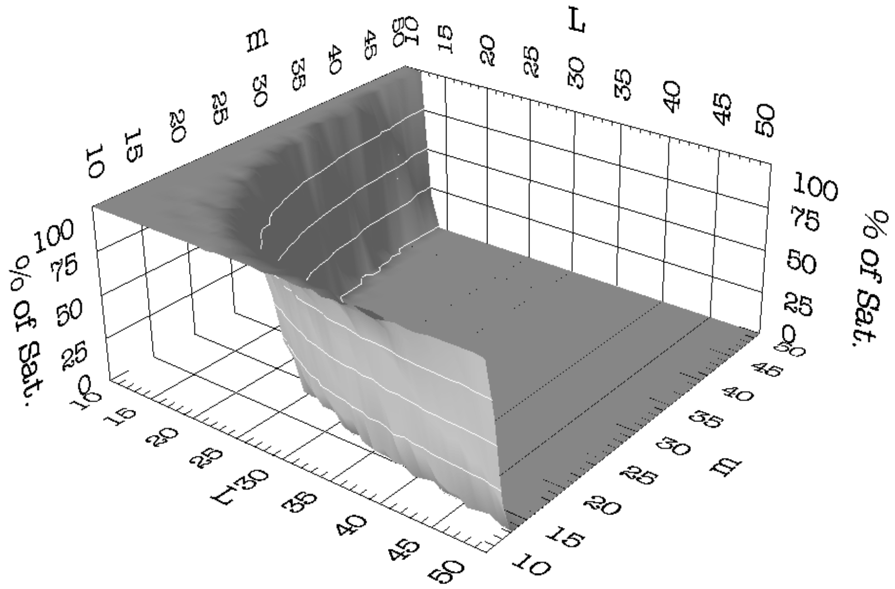

Figure 4. Percentage of θ-subsumption tests satisfied on 100 pairs $(\mathcal{C}, Ex)$, where $\mathcal{C}$ is uniformly generated with $n = 10$ variables and m literals (m in [10,50]), and Ex is uniformly generated with $N = 100$ literals built on each one of the m predicate symbols in $\mathcal{C}$, and L constants (L in [10,50]).

In the rest of the paper, the competence map associated to an algorithm describes the average computational effort for solving the θ-subsumption test.

The competence map last depends on **the sampling mechanism**, used to select problem instances $(\mathcal{C}, Ex)$ conditioned by the number of variables n and the number of literals m in $\mathcal{C}$, and the number of literals $N \times m$ and the number of constants L in Ex. Following (GS00), we use a uniform sampling mechanism.

The $2 \times m$ arguments in $\mathcal{C}$ (since all m predicates are binary) are partitioned into $n = 10$ subsets (the variables in $\mathcal{C}$). Further, one ensures that $\mathcal{C}$ is connected (see Section 5.4).

The $N = 100$ literals $p_i(a_j, a_k)$ built on each predicate symbol p_i in Ex are constructed by uniformly selecting without replacement the pairs (a_i, a_j), where i, j are in $[1, L]$.

A sample of 100 problems instances $(\mathcal{C}, Ex)$ is uniformly drawn for each value of $(n = 10, N = 100, m \in [10, 50], L \in [10, 50])$.

This relatively small sample size is justified as no heavy tailed distribution was observed for θ-subsumption cost in the parameter range considered, although such distributions were reported for large size CSPs (GBCK00).

5.4. Discussion

Before presenting the experimental results in Section 6, we discuss the scope and limitations of our experimental setting. The reliability of the competence map depends on the representativity of the artificial problems generated as above. These artificial problems might differ from θ-subsumption problems encountered in ILP benchmarks in several respects.

A first issue is the non-decomposability of clause $\mathcal{C}$. In the general CS case, the decomposition of the problem at hand into (scarcely related) subproblems is a major practical issue (GJC94; DP89); similarly, the decomposition of $\mathcal{C}$ into k-local components is a key issue in relational logic and ILP (GL85; KL94). Such a decomposition is desirable as it entails exponential savings in the resolution cost.

However, on one hand decomposability is not often encountered in ILP problems. (SHW97) shows empirically that determinate matching was rarely operational on the mesh problem (DM91); (CSC$^+$04) shows that the level of non-determinism is high, with a branching factor *circa* 30 in the mutagenesis (KSS95) and carcinogenesis (Sri97) domains.

On the other hand, evaluating a θ-subsumption algorithm on decomposable clauses introduces an optimistic bias, as the cost of simple problems is wrongly reported for complex ones (GS00). Such an optimistic bias reduces the reliability of the competence map, which is not desirable.

A second issue concerns the uniform distribution of the θ-subsumption problems considered. Each predicate symbol occurs exactly once in the clause $\mathcal{C}$, and $N = 100$ times in example Ex. Moreover, each predicate argument has same domain $\{a_1, \ldots, a_L\}$.
In real-world examples, some predicate symbols occur more frequently than others; predicate arguments are typed, and their domains have diverse sizes. The use of types enables significant savings as it restricts the θ-subsumption search space (Llo03); furthermore, the diversity of domain sizes results in a better efficiency of CSP heuristics, e.g. con-

straint propagation or dynamic variable ordering. In this respect, the fact that Ex involves N literals for every predicate symbol, all instantiated from pairs of L constants, leads to a pessimistic estimate of the average θ-subsumption cost. On the other hand, this restriction permits a simple and direct interpretation of the competence map.

The third issue concerns the arity of the predicate symbols, as the presented setting is limited to binary predicate symbols. However, the predicate arity does not intervene directly on computational complexity (note that the worst-case complexity L^n only depends on the variable domain size, exponentiated by the number of variables). More precisely, the key complexity factor depends on how many arguments in $\mathcal{C}$ ($m \times k$, if k denotes the predicate arity) are grouped into how many subsets (the number n of variables). The θ-subsumption equivalent for CS density is the probability for two literals in $\mathcal{C}$ to share a variable (meaning that they cannot be independently mapped onto literals in Ex).

This probability is $1 - \frac{(k\ n-k)}{(k\ n)}$ (two literals are independent iff the k arguments in the second literal, are mapped onto k among the $n-k$ variables not involved in the first literal), of order $\mathcal{O}(1-(1-\frac{k}{n})^k)$. According to this analysis, increasing the predicate arity k by factor t can be likened to increasing the total number of variables n by approximately t^2. The competence map drawn for binary predicates can thus be used to estimate the θ-subsumption algorithm behavior for non-binary predicates, by moving toward higher values of n.

Experimentally, increasing the number of variables n causes the phase transition to move toward longer hypotheses everything else being equal (toward the region on the right in Figure 5), with exponential increase of the complexity peak (GS00).

We restrict therefore the experimental setting to binary predicates, to keep the total computational cost within reasonable limits.

In conclusion, the assumptions made in Section 5.2 are meant to construct a reliable θ-subsumption competence map, though it might be pessimistic for ILP problems with distributional diversity.

5.5. Experimental Goals

As mentioned already, there is no universally efficient CSP algorithm. One primary goal of the paper thus is to determine which CSP algorithms and procedures are most efficient wrt θ-subsumption problems in general. A secondary goal is to characterize where these procedures are particularly efficient.

To this aim, we studied 10 variants of the *Django* algorithm, involving various combinations of CSP procedures and summarized in Table 1.

For each of these variants, a competence map is constructed as detailed in Sect. 5.3.

Table I. *Django*, Variants **V1** to **V10**.

	Baseline
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)
V6	**V1** + DVO based on min. domain ÷ max. connectivity (random choice in case of tie)
	Forward Checking
V7	**V6** + improved Forward Checking propagation of forced assignments (singleton candidate value)
	Arc Consistency
V8	**V7** + AC based on signatures
V9	**V7** + AC based on signatures and 2-signatures
	Maintaining Arc Consistency
V10	**V6** + Full Arc Consistency on assignments + AC based on signatures and 2-signatures

Table 1. *Django*, Versions **V1** to **V10**.

The baseline algorithm *Django*.**V1** combines arc-consistency checking (Mac77) and forward checking (the propagation of the current assignment is restricted to the next variable domain).

The influence of variable ordering is investigated through *Django*.**V2** to *Django*.**V6**. These variants involve distinct heuristics, all based on the First Fail Principle. In *Django*.**V2**, variables with minimal domain are ranked first[1]. In *Django*.**V3**, variables subject to a maximal number of dual constraints are ranked first (favoring the literals in $\mathcal{C}$ which are most connected to others literals). Both criteria are combined in *Django*.**V4** and *Django*.**V5**, with different priorities.

In *Django*.**V6**, the order criterion is heuristically defined as the domain size, divided by the number of neighbors in the graph; variables are sorted by increasing value of this criterion.

The influence of Forward Checking is investigated in *Django*.**V7**; in addition to the 1-step propagation of the current assignment, forced assignments (singleton candidate value for any variable) are propagated.

The influence of arc consistency is investigated using signatures and 2-signatures. *Django*.**V8** differs from *Django*.**V7** as it considers the literal signatures; *Django*.**V9** considers both signatures and 2-signatures. Signatures and 2-signatures allow for pruning the domains of the literals in a preprocessing step, and redundant tests are avoided during the constraint propagation.

Finally, *Django*.**V10** implements a Maintaining Arc Consistency (MAC) procedure, where each assignment is fully propagated in each resolution step, using both signatures and 2-signatures.

6. Experimental validation

The CSP-inspired approach of θ-subsumption implemented within the *Django* system is compared to several reference algorithms. These algorithms are introduced and the format of results is presented. The experimental results are then discussed and some interpretations are proposed.

6.1. Reference Algorithms

Django (written in C^{++}) is compared to three reference algorithms (written in C): SLD Prolog, determinate matching (KL94) and graph contexts (SHW97). We used the reference algorithm implementations kindly given by T. Scheffer. Run times are measured on a PC-Pentium II, 300 MHz.

As might have been expected, Prolog SLD does not keep up when $\mathcal{C}$ involves more than a few literals, and it had to be stopped for $m > 5$ (note that example Ex involves $100 \times m$ literals).

[1] Note that the determinate matching heuristic in the primal θ-subsumption problem (KL94) corresponds to a particular case of the minimal domain heuristic applied on the dual CSP.

Determinate matching (KL94) does significantly better than SLD for small size hypotheses; however, it runs out of resources for $m > 10$. Indeed, determinate matching is inappropriate to the uniform structure of the examples.

Graph contexts (SHW97) also turned out to be not applicable, for efficiency reasons (discussed further in Section 6.3).

Finally only the maximal clique search (MCS) (SHW97) could be used in the same problem range as *Django*.

6.2. Presentation of the results

The behavior of each algorithm is graphically displayed through a competence map (Section 5.3), reporting for each $(n = 10, N = 100, m \in [10, 50], L \in [10, 50])$ the computational cost averaged over 100 pairs $(\mathcal{C}, Ex)$ uniformly drawn, where $\mathcal{C}$ involves $n = 10$ variables and m literals, and Ex involves $N = 100$ literals for each of the m predicate symbols, and L constants.

Figures 5 and 6 respectively display the competence maps associated to *Django*.**V1** and MCS.

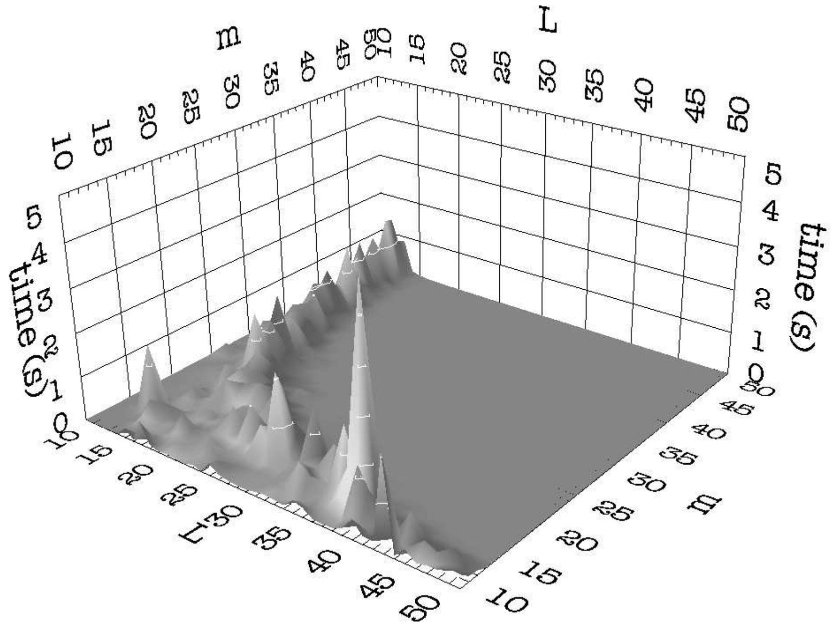

Figure 5. Competence map of *Django*.**V1**, reporting the average θ-subsumption computation cost measured as in Figure 4.

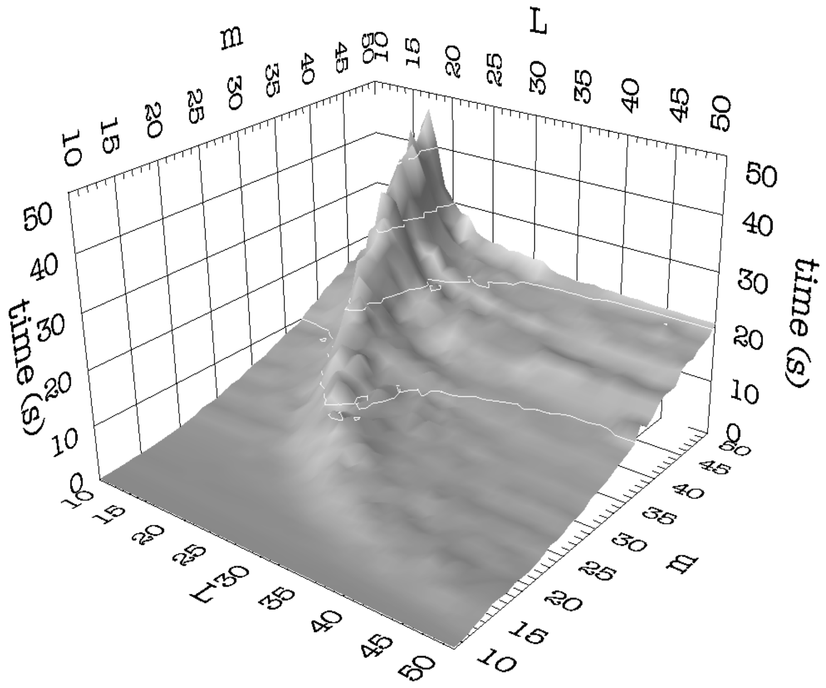

Figure 6. θ-subsumption cost(m, L) for MCS, averaged on 100 pairs $(\mathcal{C}, Ex)$, with scaling factor $\times 10$ wrt *Django*.**V1** (Figure 5).

With respect to the θ-subsumption phase transition (Figure 4), both competence maps show similar trends in the YES, NO and PT regions. The PT region coincides with the region of high complexity peaks, like a mountain chain of hyperbolic shape in plane (m, L). The peak values increase with L as expected from the worst complexity analysis.

The NO region, on the left of the PT (high values of m and/or L), is a low complexity region with moderate variance.

The YES region, on the right of the PT, is a moderate complexity region, with high variance.

For the sake of precise comparisons, the cost of each algorithm is also aggregated in the YES, NO and PT regions. Table II gives the average cost over all points (m, L) such that the θ-subsumption probability of success is respectively greater than 90% (YES region), less than 10% (NO region), and in $[10\%, 90\%]$ (PT region); the standard deviation is graphically displayed in Figures 9 and 10.

Table II. *Django*: Average θ-subsumption cost in milliseconds (Pentium II) in the *YES*, *NO* and PT regions, Artificial setting.

	YES region	Phase Transition	NO region
MCS	4582.4	10509.5	12680.9
Django.**V1**	158.3	764.1	38.1
V2	43.6	51.3	20.5
V3	50.5	83.3	22.3
V4	43.2	50.1	20.4
V5	46.8	68.1	21.3
V6	43.0	48.7	20.4
V7	40.9	45.0	19.0
V8	24.3	33.7	16.8
V9	26.4	30.9	10.3
V10	29.1	28.9	10.3

6.3. Comparison with MCS

Django.**V1** significantly outperforms MCS; the gain factor is about 30 in the *YES* region, 15 in the PT, and 330 in the *NO* region (a scaling factor of 10 is used in Figure 6 compared to Figure 5).

The phase transition phenomenon is most marked for *Django*.**V1** (Figure 5), the cost in the PT region being 5 times the cost in the *YES* region and 20 times the cost in the *NO* region. In comparison, the cost of MCS in the PT region is only twice the cost in the YES region, and the cost reaches its maximum in the NO region, close to the PT. This suggests that MCS does not detect early the inconsistencies during the search, and unnecessarily explores large portions of the search tree.

The lesser efficiency of MCS is blamed in part on the construction of the substitution graph, which must be completed before the maximal clique search takes place. This construction is computationally heavy for small sized problem instances (in $\mathcal{O}(m^2N^2)$). For large sized problem instances, the construction cost is negligible compared with the (exponential) maximal clique search and well worth the effort as the substitution graph significantly speeds up the maximal clique search. Unfortunately, the storage of the substitution graph then becomes unaffordable.

Also for complexity reasons, 2-graphs contexts could not be used for $m > 10$ (as k-graphs are constructed and checked with exponential complexity in k).

In contrast, *Django*.**V1** interleaves the search and the constraint propagation, avoiding the construction of the whole graph whenever a solution is found along the search.

The computational cost of MCS also shows a high variance compared to *Django*.**V1**. Practically, a time limit of one minute was set on every θ-subsumption test to ensure the feasibility of the experiments with MCS; in preliminary runs, the resolution of some problems instances required more than 30 minutes on Pentium II. This variance is also blamed on the lack of adequate pruning mechanism within MCS, ensuring an early detection of inconsistencies.

6.4. Comparison of CSP procedures

The impact of the diverse CSP procedures in *Django* is presented in Table II. The first set of procedures is concerned with dynamic variable ordering (DVO, *Django*.**V2** to *Django*.**V6**). *Django*.**V2** significantly improves on *Django*.**V1**, with a dynamic variable ordering based on minimal domain, which can be viewed as a non-deterministic generalization of determinate matching (KL94). Sorting the variables (literals in $\mathcal{C}$) based on their connectivity (*Django*.**V3** and *Django*.**V5**) is less efficient, though it also improves on *Django*.**V1**. The best sorting criterion is that of *Django*.**V6**, where literals are sorted by increasing value of their domain size divided by their number of 1-neighbors (literals sharing at least one variable).

Again, the efficiency of DVO depends on the region considered; the gain factor is less than 2 in the NO region, 3 in the YES region, and circa 15 in the PT region. The major effect of DVO is to smoothen the complexity peak in the PT region.

The addition of Forward Checking with propagation of singleton domains (*Django*.**V7**) very slightly improves on *Django*.**V6**, with a gain factor circa 5% in all three regions.

The effects of arc consistency (*Django*.**V8** uses signatures in a pre-processing step, and *Django*.**V9** likewise uses signatures and 2-signatures) are significantly beneficial compared to *Django*.**V7**. Note that *Django*.**V8** is more efficient in the YES region than *Django*.**V9** (with a gain factor 40% on *Django*.**V7**), and *Django*.**V9** is more efficient in the PT and NO regions (with a gain factor respectively 30% and 45% over *Django*.**V7**). This is interpreted as the construction and exploitation of 2-signatures is not worth the effort for small sized problem instances.

The efficiency of signature-based heuristics surprisingly contrasts with the relative inefficiency of graph contexts (SHW97). Indeed, graph contexts and signatures alike proceed by reducing the variable domains. The difference is that all consistency checks are computed and stored in the substitution graph; in contrast, in *Django* signatures are only meant to speed up consistency checks, which are performed as needed along constraint propagation; note that a domain not reduced through signatures does not undergo arc consistency. Furthermore, it appears that graph contexts suffer from a too general and thereby inefficient implementation, devised for handling k-GC though $k \leq 2$ in practice.

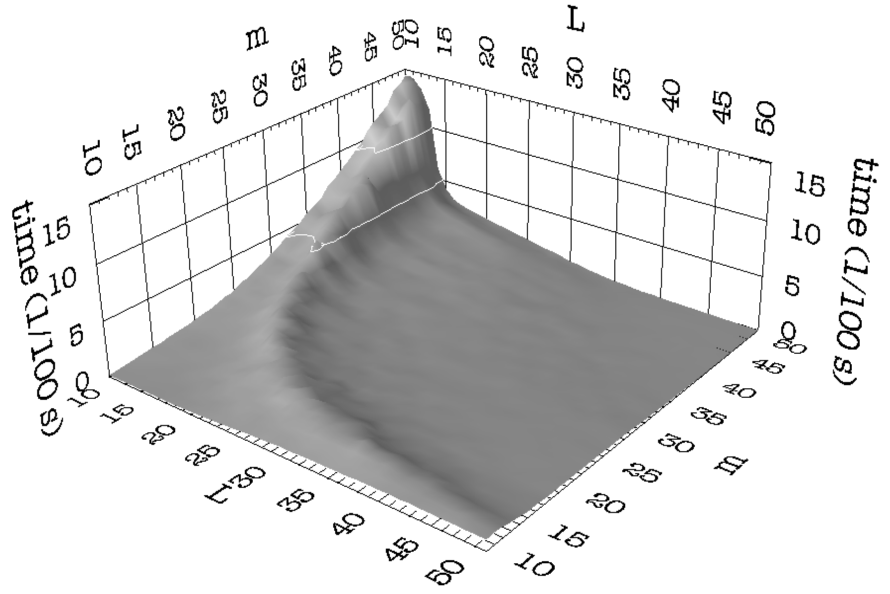

Figure 7. θ-subsumption cost(m, L) for *Django*.**V9**, averaged on 100 pairs $(\mathcal{C}, Ex)$. A factor $\div$ 25 is observed compared to *Django*.**V1** (Figure 5).

Last, a Maintaining Arc Consistency (MAC) procedure is integrated in *Django*.**V10** (Figure 8). This heuristic is mostly beneficial in the PT region: the additional constraint checks remove inconsistencies that would otherwise lead to backtracks (gain factor of 7% over *Django*.**V9**).

Globally, the extra cost of MAC is not worth the effort in the YES region, since few backtracks occur in this region anyway (loss factor of 10% over *Django*.**V9**). Further, MAC makes no difference in the NO region compared to *Django*.**V9**. This is explained as MAC is only called

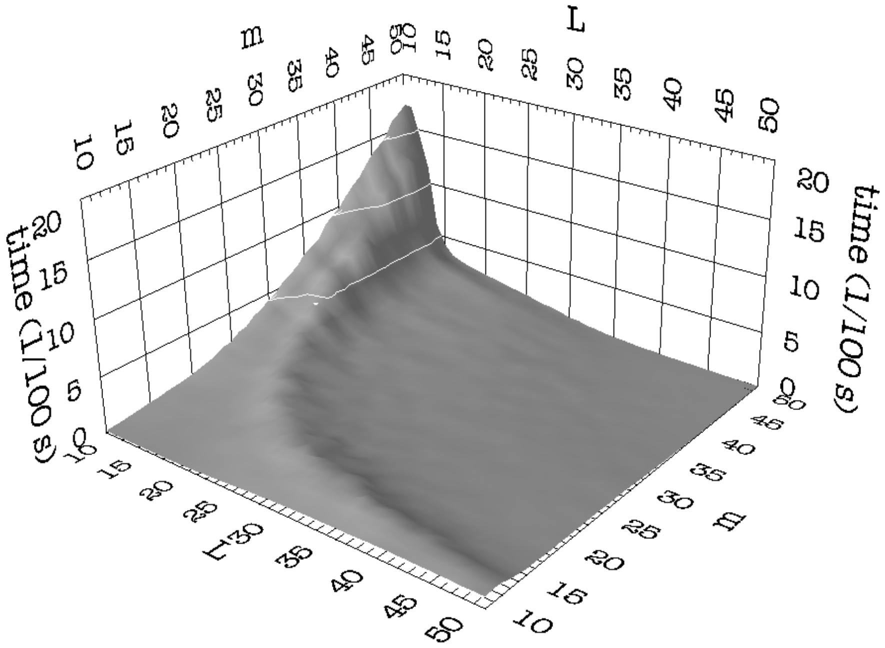

Figure 8. θ-subsumption cost(m, L) for *Django*.**V10**, averaged on 100 pairs $(\mathcal{C}, Ex)$. A factor $\div$ 25 is observed compared to *Django*.**V1** (Figure 5).

after arc consistency and forward checking, while arc consistency fail most of the times in the NO region.

Still, MAC brings a significant improvement in the PT region, increasing with the instance size. This improvement is explained as MAC avoids the dramatic extra cost due to the hardest problems handled by *Django*.**V9**. This way, the use of MAC smooths both the computational cost, and its variance, in the PT region. Figures 9 and 10 respectively show the standard deviation of the computational θ-subsumption cost of *Django*.**V9** and *Django*.**V10**.

Finally, even though MAC only slightly improves on Forward Checking in terms of computational gain, *Django*.**V10** appears much more stable than *Django*.**V9** as it drastically decreases the cost of the hardest problems. From an operational perspective, this would recommend the use of *Django*.**V10** as more appropriate to real-time or manufacturing applications than *Django*.**V9**.

The *Django* system at its best (*Django*.**V9** and *Django*.**V10**), thus improves on MCS with a gain factor about 150 in the *YES* region, 360 in the PT region, and 1200 in the *NO* region (Figure 7 and 8) on the

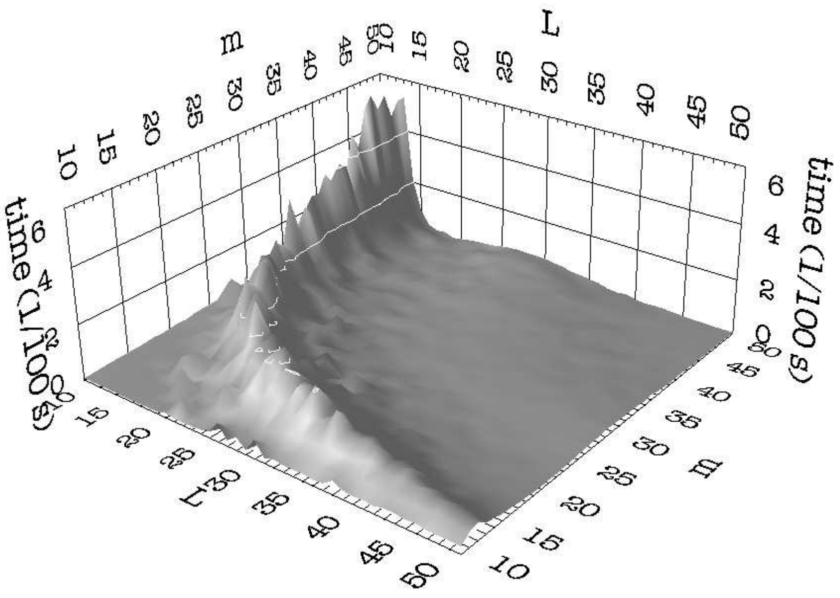

Figure 9. Standard deviation for *Django*.**V9**, (average results reported in Figure 7).

artificial problems. Real-world problems will be considered in the next section.

7. Meta-*Django*

This section is concerned with the choice of the best algorithm in *Django*. The goal is defined and discussed with respect to the Meta-learning goal (PBGC00; BK01). Then the Meta-*Django* algorithm is presented, which exploits the competence maps built in the previous section. Experimental validation considers artificial problems and the real-world mutagenesis problem (KSS95).

7.1. Meta-learning

The experimental results presented in Section 6 confirm that none of *Django*.**V9** and *Django*.**V10** dominates the other. This raises the question of how to choose the best algorithm depending on the problem instance.

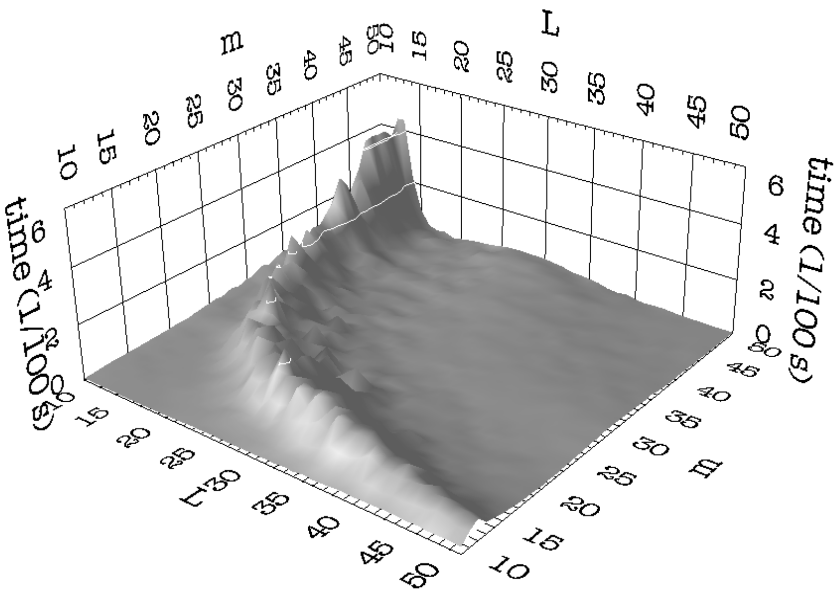

Figure 10. Standard deviation for *Django*.**V10**, (average results reported in Figure 8).

This general question, recognized a key question in Machine Learning since the 90's, was formulated as a Meta-Learning problem in (BGH94; PBGC00; BK01): learn which algorithm is most suited to the problem instance at hand. Training examples for the meta-learning task are made of a pair (problem, algorithm), labeled with the success of the algorithm for the problem. Depending on whether the label is numerical (predictive accuracy, rank of the algorithm) or boolean (the algorithm reaches the best predictive accuracy), the meta-learning task is one of regression or classification. Another meta-learning setting considers examples made of a triplet (problem, algorithm1, algorithm2), labeled as *true* when algorithm1 dominates algorithm2 on this problem (FP01).

The very elegant approach of meta-learning unfortunately depends on the quantity and quality of available training data, like all learning applications. A first difficulty regards the representation of the examples (problem, algorithm). Algorithms are represented by their name (categorical attribute), together with their parameters. Problems are described along many attributes, either static (size, number of missing values,..) or dynamic ("landmarking" the problem instance with the predictive accuracy of a k nearest neighbor, naive Bayes,..) (PBGC00).

A second difficulty is the choice of the problem instances. In (BK01), the problem instances were constructed through principled perturbations of problems in the Irvine repository (BKM98), (e.g. increasing the rate of missing values, incorporating irrelevant attributes,..).

However, the choice of the perturbations is a strong bias on Meta-learning; for instance, the importance of the "missing value rate" attribute in the meta-rules directly reflects its use in the perturbations (Kal02).

Returning to the selection of the best θ-subsumption algorithm, the stochastic complexity approach taken in this paper can be viewed as an alternative to meta-learning. The representation of the problem instances, set to the order parameters, is firstly defined; based on these order parameters, principled experiments yield the competence map of each algorithm; the competence maps can be used as look-up tables. These look-up tables encapsulate the same knowledge as sought for by meta-learning: they predict the behavior of the algorithms conditioned by the general characteristics of the problem instance. This prediction supports the built-in selection of the best algorithm, achieved in Meta-*Django*.

According to experimental evidence, Meta-*Django* must select *Django*.**V9** (Forward checking) in the YES region and *Django*.**V10** (Maintaining Arc Consistency, MAC) in the NO and PT region. (From an implementation perspective, these procedures involve the same data structure; the selection is thus integrated seamlessly to the acquisition of the θ-subsumption problem instance).

Meta-*Django* thus only requires to determine whether the current problem instance lies in the YES region.

7.2. Localizing the problem instance

To this aim, we borrow to the CSP literature the κ parameter defined by (GMPW96):

$$\kappa = \frac{-\sum_{r \in \mathcal{R}} log_2(1-p_2)}{log_2(|dom(\mathcal{X})|)}$$

where $\mathcal{R}$ is the set of constraints, p_2 is the constraint tightness (defined in Section 3 as the number of assignments ruled out by a constraint r), and $|dom(\mathcal{X})|$ the total size of the variable domains (L^n since all n variables have same domain of size L). After (GMPW96), problems in the PT region are associated with κ values between 0.75 and 1; the

YES region is associated with $\kappa < .75$ and the NO region is associated with $\kappa > 1$.

The κ parameter is first attuned to θ-subsumption problems. Term $(1 - p_2)$ corresponds to the probability of an assignment to be admissible wrt a given constraint. With respect to θ-subsumption (before binarization), each constraint corresponds to some literal p in $\mathcal{C}$; the number of admissible assignments is thus upper-bounded by the number of literals in Ex built on the same predicate symbol noted $|rel(p)|$; in the meanwhile, the number of possible assignments is estimated as the total number of constants or variables in Ex, exponentiated by the arity of the current predicate symbol ($|arg(Ex)|^{arity(p)}$).

Finally:

$$\kappa = \frac{-\sum_{p \in \mathcal{C}} log_2(|rel(p)|) - arity(p) \times log_2(|arg(Ex)|)}{log_2(|dom(\mathcal{X})|)}$$

$$= \frac{-\sum_{p \in \mathcal{C}} log_2(|rel(p)|) - arity(p) \times log_2(|arg(Ex)|)}{n log_2(|arg(Ex)|)}$$

$$= \frac{m \times (2 log_2 L - log_2 N)}{n log_2 L}$$

where n denotes the number of variables in $\mathcal{C}$.

There is a good empirical agreement between parameter κ and the actual location of the θ-subsumption problem instances. Practically, Meta-*Django* uses the FC procedure (Version **V9**) for problems such that $\kappa < .7$ (Yes region) and the MAC procedure (Version **V10**) is used in all other cases.

7.3. Experimentation on artificial problems

Using the same experimental setting as in Section 5.3, by construction Meta-*Django* reaches the best cost among *Django***V9** and *Django.***V10** (Table III), and the competence map of Meta-*Django* combines the best of respectively *Django.***V9** (in the YES region) and *Django.***V10** (in the NO and PT regions).

In the YES region, *Django.***V9** (FC) is selected; Meta-*Django* thus avoids checking and reducing the variable domains (as in MAC), which is in any case useless in the YES region[2]. However, the FC procedure is less efficient than MAC on exceptionally hard problem instances; this is demonstrated as the standard deviation of Meta-*Django* computational cost in the YES region is worse than for *Django.***V10**.

[2] Incidentally, our last experiments show that removing the arc consistency step is beneficial in the YES region.

Table III. Meta-*Django*: Average θ-subsumption cost in milliseconds (Pentium II) in the *YES*, *NO* and PT regions defined as in Table II, Artificial setting.

	YES region	Phase Transition	NO region
V9	26.4	30.9	10.3
V10	29.1	28.9	10.3
Meta-*Django*	26.4	28.9	10.3

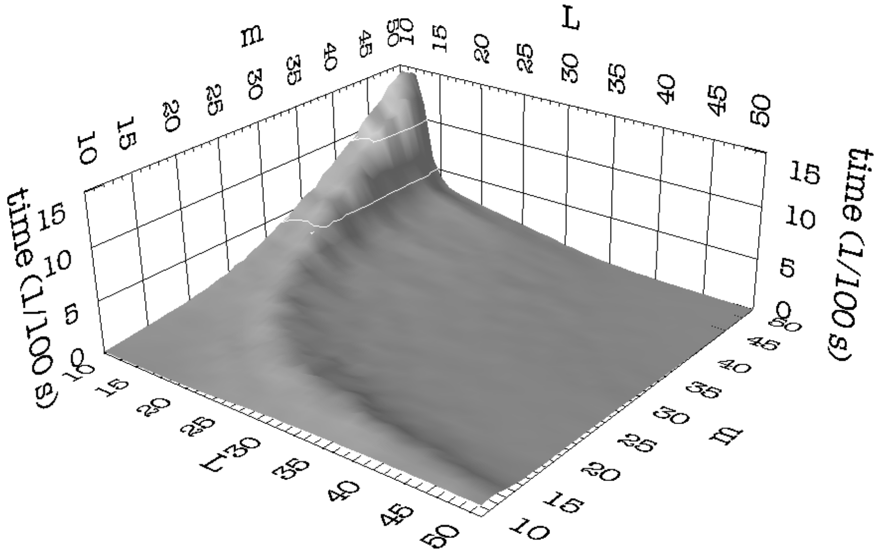

Figure 11. Meta-*Django*: θ-subsumption cost(m, L) averaged on 100 pairs $(\mathcal{C}, Ex)$. A factor $\div$ 25 is observed compared to *Django*.**V1** (Figure 5).

Django.**V10** (MAC) is selected in the PT region, with an additional propagation of assignments compared to FC. This propagation results in significant reductions of the variable domains, preventing further expensive backtracks.

Django.**V10** is selected in the NO region for the sake of simplicity, as there is no actual difference in this region between MAC and FC. Actually, the resolution is mostly achieved during the preliminary arc consistency step, which is common to both FC and MAC procedures.

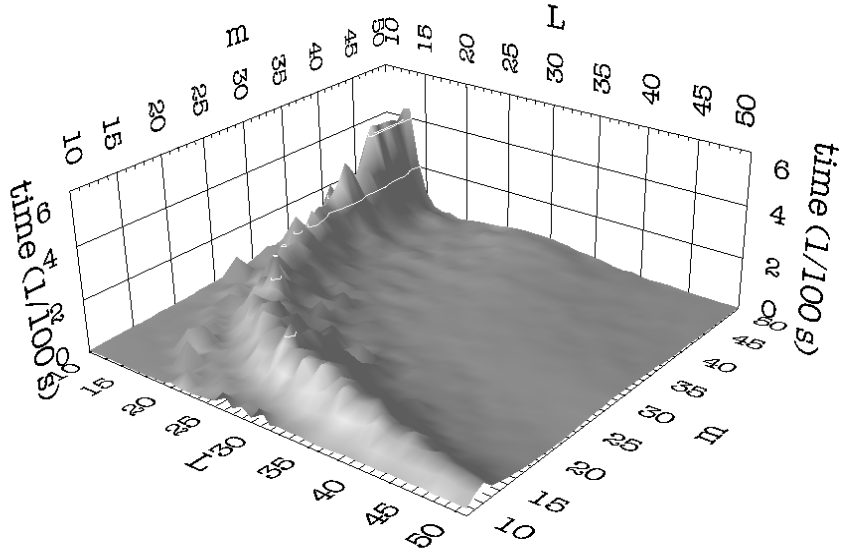

Figure 12. Meta-*Django*: Standard deviation on cost(m, L)

7.4. Experimentation on the mutagenesis domain

In order to validate *Django* and Meta-*Django* on a real-world like domain, they were integrated into an ILP learner inspired by Warmr (DR97) and presented in (MS03), with the goal of finding the clauses with support $\geq 10\%$ in the mutagenesis domain (KSS95). However the focus is here on comparing the θ-subsumption algorithms, rather than on the frequent clauses.

For the sake of computational feasibility, the following restrictions were made:

- The learner proceeds in a level-wise fashion (AS94). At each level, only 10,000 clauses are retained out of the solution clauses. These retained clauses are specialized by addition of another literal.
- Clauses are limited to 10 literals. They are connected and they only involve variables (though *Django* and Meta-*Django* support constants); further, all variables in a literal are distinct.

Furthermore, the constants corresponding to the molecule identifier in the examples are omitted. In the learning from entailment setting (Rae97) considered in this paper, the body of the example is made of all literals relevant to the description of this example; therefore there is no need for linking these literals explicitly through the example identifier.

Each candidate clause is tested against all examples in the dataset; the CPU time is measured on a Pentium IV 3GHz. The θ-subsumption cost reported in Table IV gives the average cost of checking one candidate clause against all examples; the θ-subsumption tests made to prune the logically equivalent solution clauses, or the candidate clauses that cover an infrequent clause, are not considered.

The graph context algorithm (SHW97) could not be used besides the third level and its results are omitted.

Experimental evidence in Table IV shows some differences compared to the artificial setting (Table II).

Table IV. MCS, *Django* and Meta-*Django*: Average θ-subsumption cost in milliseconds (Pentium IV) in the *YES*, *NO* and PT regions defined as in Table II, Mutagenesis domain.

	YES region	Phase Transition	NO region
MCS	532.4	644.4	389.5
Django.**V1**	35.1	370.3	408.O
V2	33.5	246.9	59.7
V3	34.3	413.5	61.6
V4	33.6	239.6	45.3
V5	34.1	351.1	57.5
V6	33.6	237.9	45.2
V7	35.7	303.9	53.3
V8	17.5	274.3	8.2
V9	18.3	269.7	8.0
V10	21.7	264.8	7.0
Meta-*Django*	21.6	263.6	6.9

MCS is still outperformed by all *Django* variants (except for *Django*.**V1** in the *NO* region). However the gain factor is less than for the artificial problems. This is explained by the smaller size of the θ-subsumption problems, reducing the cost of the substitution graphs and making MCS more efficient. However, MCS still does not detect inconsistencies as early as possible, and the best gain factor still occurs in the *NO* region.

Dynamic variable ordering based on minimal domains (*Django*.**V2**, *Django*.**V4** and *Django*.**V6**) likewise improves the results compared to *Django*.**V1**, and the gain factor is less than for the artificial problems. In fact, DVO appears to be beneficial mostly in the *NO* region.

Surprisingly, the forced propagation of singleton domains (*Django*.**V7**) appears to be at best useless and most often harmful in all regions. This might be explained as the connectivity of the literals is more regular in the mutagenesis domain than for artificial problems; the overhead of singleton propagation is not compensated for by the exploitation of highly constrained variables.

Quite the contrary, the use of signatures (*Django*.**V8** and *Django*.**V9**) appears to be very efficient, with a gain factor of 2 and 6 respectively in the *YES* and *NO* regions. As could have been expected, 2-signatures (*Django*.**V9**) does not improve significantly the performances, as literals share at most one variable in general (being reminded that the molecule identifier was omitted). But as the type of both *bond* and *atm* is coded by an integer, some literals happen to share two variables. Of course, in a real-world application the example description can be inspected to determine *a priori* whether the use of 2-signatures is relevant or should better be discarded.

Last, Maintaining Arc Consistency (*Django*.**V10**) slightly improves over *Django*.**V9** in the *PT* and *NO* regions, and does no good in the *YES* region. On these problems, the computational overhead of full arc consistency penalizes *Django*.**V10** compared to *Django*.**V9**. As noted earlier on, the extra-cost of MAC is worth the effort on large-sized problems only. Complementary experiments show that MAC catches up and significantly outperforms FC for clauses with 12 literals or more. Regarding Meta-*Django*, since the algorithm selection is built in from the competence maps in Section 6, there is no wonder that it selects *Django*.**V10** in the *NO* and *PT* regions, though *Django*.**V8** was the best one in the *NO* region on the mutagenesis domain. What is more surprising, Meta-*Django* also tends to select *Django*.**V10** in the *YES* region. This is interpreted as the κ factor is not sufficient to localize the PT when dealing with a non-uniform example distribution. Other factors, such as the clause size, should also be considered.

As an alternative, Meta-*Django* can also be directly updated and refined from Table IV: clearly, *Django*.**V8** improves on *Django*.**V10** in the *YES* region for this problem.

Nevertheless, in its actual version Meta-*Django* improves on MCS by a gain factor ranging from 2.5 (in the *PT* region) to 24 (in the *YES* region) and 55 (in the *NO* region) on this real-world problem.

7.5. Conclusions regarding Meta-Django

Complementary experiments show that the comparative performance of *Django*.**V8** and *Django*.**V10** is sensitive to the size of the cache memory (128 Kb and 512 Kb). It appears that a small cache size hinders MAC (*Django*.**V10**) as frequent memory access might not be compensated for by effective pruning. However, for "sufficiently" large domain sizes *Django*.**V10** catches up and ultimately dominates *Django*.**V8**.

In summary, experiments on artificial problems and on the mutagenesis domain show three main trends. First of all, Dynamic Variable Ordering with minimal domain (*Django*.**V2**, *Django*.**V4** and *Django*.**V6**) significantly improves on *Django*.**V1**, particularly in the PT and *NO* region. Second, the use of signatures (*Django*.**V8**) also improves the performance though to a lesser extent, particularly in the *YES* region. Last, the use of MAC (*Django*.**V10**) is only beneficial for problems with large domain sizes.

8. Related works

The θ-subsumption test was recently reconsidered in the ILP literature along several lines.

A novel generality relation, termed *subsumption under object identity* (OI) was defined by Esposito et al. (EFFS00). OI requires substitution θ to be injective, i.e. to map distinct variables onto distinct constants. OI is mostly based on semantic motivations (the roles of variables in a hypothesis should be distinct), though the additional requirement on substitutions also restricts the search space and thereby increases the efficiency of θ-subsumption; note that *Django* would achieve OI θ-subsumption by adding *diff constraints* on all pairs of variables. Along the same CSP lines, a novel θ-subsumption algorithm has been proposed to find all OI-based successful substitutions in (MBF$^+$03).

Another approach was proposed by Blockeel et al., to limit the computational cost of hypothesis assessment (BDD$^+$00). This approach proceeds by sharing the data structures and cost of θ-subsumption among several similar candidate clauses. Some modifications are needed (and planned for further research) in order to handle query packs in *Django*. Typically, the arc consistency and constraint propagation algorithms must effect a limited propagation of the domain reductions; the check of signatures and 2-signatures must handle in a different way the variables shared with the parent of the clauses.

Also, Costa et al. propose the combined use of several transformations to speed up θ-subsumption, based on additional information on the variable types (CSC^+04). This additional information allows for decomposing the clause (*cut-transformation*) into independent parts up to the instantiation of "grounded variables". The *cut-transformation* is extended by the *once-transformation*, recursively decomposing the independent parts in independent subparts that share only variables instantiated in a previous step. In parallel, the *smartcall-transformation* prunes independent subsets of the clause for which a solution was found. However, it is noted that *once-transformation* is limited to acyclic clauses, where the instantiations of variables can be hierarchically ordered.

9. Conclusion and perspectives

The thesis advocated in this paper is that many findings and algorithms in the CSP literature can be adapted and exploited for theoretical and algorithmic advances in Inductive Logic Programming, as first advocated in (GS00).

The main result of the paper is a novel θ-subsumption system first presented in (MS01), *Django*, which improves on earlier algorithms (KL94; SHW97) by one or several orders of magnitude. Such a computational gain is good news as ILP systems routinely perform thousands of subsumption tests. This result is obtained by combining a specific re-description of θ-subsumption problems into CSPs and data structures (signatures), and well-known CSP algorithms (dynamic variable ordering, forward checking and arc consistency).

Principled experiments are exploited to build a control layer on the top of *Django*, termed Meta-*Django*, which achieves the selection of the algorithm most suited to the problem at hand.

Further work is concerned with extending *Django*, using more sophisticated CSP algorithms (e.g. replacing the AC3 arc consistency with worst-case cubic complexity, by an AC6, AC7 or AC8, with worst-case quadratic complexity; adding a Conflict Based Jumping into the backtracking mechanism), or adapting other CSP algorithms (e.g. path consistency (Aug00)). The extensions will be documented with their competence map and the Meta-*Django* algorithm will be extended accordingly.

Along the same lines, the use of Forward Checking on the primal and dual formulations of a θ-subsumption problems will be investigated in more depth; as shown by (Che00), the FC efficiency might vary by an exponential factor between both formulations, depending on the CS problem at hand.

This work also opens several perspectives of research in ILP.

The CSP approach can be used to speed up other ILP operators than θ-subsumption, such as for instance θ-reduction (Hir03) or propositionalization (AR00). θ-reduction is used as post processing for the least general generalization (lgg) operator defined by Plotkin (Plo70); it reduces the lgg output into a minimal unique (up to variable renaming) clause (NCdW97). θ-reduction algorithms currently proceed by removing one literal at a time and checking whether the initial clause still subsumes the resulting one. By considering the CSP problem defined as the θ-subsumption of the lgg by itself, one might exploit the indeterminacies of the θ-subsumption solutions to detect groups of equivalent literals. Also, the decomposition of the lgg can be exploited to restrict the search space.

A second perspective concerns the choice of a representation for a particular domain application, and the assessment of hypothesis and example languages. As known by all ILPers, the predicate structure (arity, links) strongly influences learning performances. Better predicates can be introduced dynamically through refinement operators, as for instance in (LF03; PCW02). But the adequacy of a hypothesis language can also be experimentally assessed through systematic θ-subsumption tests, prior to learning. By testing random clauses against the training examples, the phase transition region can be localized. This region likely includes the target clauses: the target clauses hardly belong to the YES and NO regions as they separate the training examples. Therefore the localization of the PT region indicates the complexity of the learning task, i.e. the number of variables and literals in the target clauses within the hypothesis language. This complexity estimate, which also reflects the characteristics of the training examples, can be used to compare several hypothesis languages.

Acknowledgments

We gratefully acknowledge Lorenza Saitta, Attilio Giordana and Marco Botta, for many discussions about Phase transitions and its consequences on ILP. We also thank Tobias Scheffer, who kindly gave us his implementation of graph contexts and determinate matching.
Let Sébastien Augier and Fabien Torre be particularly thanked for their support and their many critics during the elaboration of *Django*.
We last thank the anonymous referees for their remarks and suggestions, that significantly helped us to improve the paper.

References

E. Alphonse and C. Rouveirol. Lazy propositionalisation for relational learning. In Horn W., editor, *Proceedings of the 14^{th} European Conference on Artificial Intelligence, ECAI'00*, pages 256–260. IOS Press, 2000.

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20^{th} Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

S. Augier. *Apprentissage Supervisé relationnel par algorithmes d'évolution*. PhD thesis, Université de Paris 11 – Orsay, 2000.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In *Proceedings of the 10^{th} International Conference on Inductive Logic Programming*, LNAI 1866, pages 60–77. Springer-Verlag, Berlin, 2000.

P. Brazdil, J. Gama, and B. Henery. Characterizing the applicability of classification algorithms using meta-level learning. In *Proceedings of the 7^{th} European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 83–102. Springer, 1994.

M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

H. Bensusan and A. Kalousis. Estimating the predictive accuracy of a classifier. In P. Flach and L. De Raedt, editors, *Proceedings of the 12^{th} European Conference on Machine Learning*, pages 25–36. Springer Verlag, 2001.

C. Blake, E. Keogh, and C.J. Merz. *UCI Repository of machine learning databases*. http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

C. Bessière and J.-C. Régin. Mac and combined heuristics: Two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of the 2^{nd} Int. Conf. on Principles and Practice of Constraint Programming*, pages 61–75, 1996.

H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

X. Chen. *A theoretical comparison of selected CSP solving and modeling techniques*. PhD thesis, University of Alberta, Canada, 2000.

P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of Int. Joint Conf. on Artificial Intelligence, IJCAI'91*, pages 331–337, 1991.

V. Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2004.

S. Dzeroski and N. Lavrac, editors. *Relational Data Mining*. Springer Verlag, 2001.

D. Dolsak and S. Muggleton. The application of ILP to finite element mesh design. In S. Muggleton, editor, *Proceedings of the first International Workshop on Inductive Logic Programming*, pages 453–472, 1991.

R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In *Proceedings of the 7^{th} International Workshop on Inductive Logic Programming*, LNCS 1297, pages 125–132. Springer-Verlag, Berlin, 1997.

F. Esposito, N. Fanizzi, S. Ferilli, and G. Semeraro. Ideal theory refinement under object identity. In P. Langley, editor, *Proceedings of the 17^{th} International Conference on Machine Learning*, pages 263–270. Morgan Kaufmann, 2000.

J. Fürnkranz and J. Petrak. An evaluation of landmarking variants. Technical report, Austrian Research Institute for Artificial Intelligence, 2001.

C. P. Gomes, B.Selman, N. Crato, and H. A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

M. Gyssens, P. G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.

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

G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *AAAI/IAAI, Vol. 1*, pages 246–252, 1996.

A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 2:217–251, 2000.

T. Hogg, B.A. Huberman, and C.P. Williams (Eds). *Artificial Intelligence: Special Issue on Frontiers in Problem Solving: Phase Transitions and Complexity*, volume 81(1-2). Elsevier, 1996.

K. Hirata. On condensation of a clause. In T. Horvath and A. Yamamoto, editors, *Proceedings of the 15^{th} Int. Conf. on Inductive Logic Programming*, LNAI, pages 164–179. Springer Verlag, 2003.

T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81:127–154, 1996.

A. Kalousis. *Algorithm Selection via Meta-Learning*. PhD thesis, Université de Genève, Centre Universitaire d'Informatique, 2002.

J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In W. Cohen and H. Hirsh, editors, *Proceedings of the 11^{th} International Conference on Machine Learning*, pages 130–137. Morgan Kaufmann, July 1994.

S. Kramer, N. Lavrac, and P. Flach. Propositionalization approaches to relational data mining. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 262–291. Springer Verlag, 2001.

R.D. King, A. Srinivasan, and M.J.E. Sternberg. Relating chemical activity to structure: an examination of ILP successes. *New Gen. Comput.*, 13, 1995.

N. Lachiche and P. A. Flach. A true first-order bayesian classifier. In S. Matwin, editor, *Proceedings of the 14^{th} Int. Conf. on Inductive Logic Programming*, LNAI, pages 133–148. Springer, 2003.

J. Lloyd. *Logic for Learning : Learning Comprehensive Theories from Structured Data*. Springer Verlag, 2003.

A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

N. Di Mauro, T. M.A. Basile, S. Ferilli, F. Esposito, and N. Fanizzi. An exhaustive matching procedure for the improvement of learning efficiency. In T. Horvath and A. Yamamoto, editors, *Proceedings of the 15^{th} Int. Conf. on Inductive Logic Programming*, LNAI, pages 112–129. Springer Verlag, 2003.

S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

J. Maloberti and M. Sebag. Theta-subsumption in a constraint satisfaction perspective. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 13^{th} Inductive Logic Programming*, LNAI 2157, pages 164–178. Springer Verlag, 2001.

J. Maloberti and E. Suzuki. Improving efficiency of frequent query discovery by eliminating non-relevant candidates. In *Proceedings of the 6th International Conference on Discovery Science (DS 2003)*, LNAI 2843, pages 219–231, Berlin, Heidelberg, New York, 2003. Springer-Verlag.

S. Muggleton. Inverse entailment and PROGOL. *New Gen. Comput.*, 13:245–286, 1995.

S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming.* Springer Verlag, 1997.

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.

B. Pfahringer, H. Bensusan, and C. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proceedings of the 17th International Conference on Machine Learning*, pages 743–750. Morgan Kaufmann, 2000.

L. Peña-Castillo and S. Wrobel. Macro-operators in multirelational learning: a search-space reduction technique. In H. Mannila et al., editor, *Proceedings of the 13th European Conference on Machine Learning*, LNCS 2430, pages 357–368. Springer Verlag, Berlin, 2002.

G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95(1):187–201, 1997.

L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of Int. Joint Conf. on Artificial Intelligence, IJCAI'93*, pages 1058–1063. Morgan Kaufmann, 1993.

R. C. Read and D. G. Corneil. The graph isomorph disease. *Journal of Graph Theory*, 1:339–363, 1977.

J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

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*, pages 212–228. Springer-Verlag, 1997.

M. Sebag and C. Rouveirol. Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning*, 38:41–62, 2000.

A. Srinivasan. The predictive toxicology evaluation challenge. In *Proceedings of Int. Joint Conf. on Artificial Intelligence, IJCAI'97*, pages 4–8. Morgan Kaufmann, 1997.

A. Srinivasan. The Aleph manual, 2001. http://web.comlab.ox.ac.uk/oucl/research/-areas/machlearn/Aleph/.

Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.