Constraints (2013) 18:144–165
DOI 10.1007/s10601-013-9141-7

Formulating the template ILP consistency problem as a constraint satisfaction problem

Roman Barták · Radomír Černoch · Ondřej Kuželka · Filip Železný

Published online: 12 February 2013

Abstract Inductive Logic Programming (ILP) deals with the problem of finding a hypothesis covering positive examples and excluding negative examples, where both hypotheses and examples are expressed in first-order logic. In this paper we employ constraint satisfaction techniques to model and solve a problem known as template ILP consistency, which assumes that the structure of a hypothesis is known and the task is to find unification of the contained variables. In particular, we present a constraint model with index variables accompanied by a Boolean model to strengthen inference and hence improve efficiency. The efficiency of models is demonstrated experimentally.

Keywords Constraint modeling · Inductive logic programming · Meta-reasoning

1 Introduction

Inductive logic programming (ILP) is a subfield of machine learning which uses first-order logic as a uniform representation for examples, background knowledge and

R. Barták (✉)
Faculty of Mathematics and Physics, Charles University in Prague, Praha 1, Czech Republic
e-mail: bartak@ktiml.mff.cuni.cz

R. Černoch · O. Kuželka · F. Železný
Faculty of Electrical Engineering, Czech Technical University in Prague,
Praha 1, Czech Republic

R. Černoch
e-mail: cernorad@fel.cvut.cz

O. Kuželka
e-mail: kuzelon2@fel.cvut.cz

F. Železný
e-mail: zelezny@fel.cvut.cz

hypotheses [19]. ILP provides a powerful framework for relational data mining [11]. To analyze the complexity of ILP, the formalization of core ILP tasks was proposed by the seminal paper by Gottlob et al. [15] still serving as a reference framework in newer studies [1]. Gottlob et al. define two basic ILP problems: the *bounded consistency problem* and the *template consistency problem*. In both, it is assumed that examples are clauses and the goal is to find a *consistent* hypothesis H, that is a clause entailing all *positive examples* and no *negative example*. Entailment is checked using θ-subsumption [20] which is a decidable restriction of logical entailment.

In the bounded consistency formulation, the number of literals in H is polynomially bounded by the number of examples. In the template consistency formulation, adopted by this paper, it is instead required that $H = T\vartheta$ for some substitution ϑ, where T is a given clause called a *template*. The template defines atoms in the hypothesis, but usually it does not specify the connection between the atoms (via shared variables). Hence the substitution ϑ is in fact unification of variables in the template.

Gottlob et al. show that both problems are equivalent in terms of computational complexity. In both cases, the complexity arises from two sources: (1) "the subsumption test for checking whether a clause subsumes an example" and (2) "the choice of the positions of variables in the atoms (of the clause)". Informally, (2) corresponds to the task of *searching* the space of admissible clauses, and (1) corresponds to *evaluating* an explored clause.

Previously, Maloberti and Sebag [18] addressed the complexity source (1) through constraint satisfaction techniques. In particular, they proposed a θ-subsumption algorithm called *Django* that is based on reformulation of θ-subsumption as a binary constraint satisfaction problem. Thanks to powerful CSP inference techniques, Django brought dramatic speed-up for θ-subsumption and consequently for the entire ILP system. This result clearly motivates to explore the application of constraint satisfaction techniques also for the complexity source (2). We pursue this goal here.

We frame our approach in the template consistency problem. Thus, given learning examples and a template T, the purpose of our CSP model is to find a substitution ϑ making $T\vartheta$ consistent with the examples. As another contribution of this work, we show how this model is connected with a modified Django model and how the search for substitution ϑ is realized.

Since our approach translates ILP tasks into CSP tasks, which is a solving approach different from classical methods in ILP, we justify the validity of our approach experimentally by direct comparison of runtimes and solution quality. Firstly, we perform an evaluation of our proposed CSP models. Then, the model obtaining the best results is compared with two standard ILP systems and it is shown to provide a significant speed-up.

The paper is organized as follows. Section 2 provides necessary background on constraint satisfaction and inductive logic programming including the formal definition of the template consistency problem. In Section 3 we describe how to abstract from the template consistency problem into a set of interconnected constraint satisfaction problems defining the subsumption problems for given examples. We also present there the constraint model for subsumption checking based on the Django algorithm. In Section 4 we present the search algorithm finding a set of unifications defining the substitution ϑ that transforms a given template to a consistent hypothesis. Section 5 introduces two constraint models describing possible unifications. These models help the search algorithm from Section 4 to restrict the set

of unifications that the search algorithm needs to explore. The proposed constraint models are later in Section 6 experimentally evaluated and compared to other ILP systems. Section 7 concludes the paper.

2 Background information

2.1 Constraint satisfaction techniques

A *constraint satisfaction problem* (CSP) is a triple (X, D, C), where X is a finite set of decision variables, for each $x \in X$, $D_x \in D$ is a finite set of possible values for the variable x (the domain), and C is a finite set of constraints [10]. A constraint c is a pair $(scope(c), rel(c))$ such that $scope(c) \subseteq X$ is a set of constrained variables and $rel(c)$ determines the value tuples satisfying the constraint: $rel(c) \subseteq \prod_{x \in scope(c)} D_x$. A constraint thus restricts the possible combinations of values to be assigned to the variables in its scope. Let us assume variables A and B, both with domains {1,2,3}. Then $scope(A < B) = \{A, B\}$ and $rel(A < B) = \{(1, 2), (1, 3), (2, 3)\}$. A *solution* to a CSP is a complete instantiation of variables such that the values are taken from respective domains and all constraints are satisfied. A constraint c is satisfied for a given assignment I of variables, if I restricted to $scope(c)$ is in $rel(c)$. For example, $I = \{A = 1, B = 3, C = 3\}$ satisfies the constraint $A < B$. If a CSP has a solution then we call this CSP *satisfiable*.

In this paper we will use two well known constraints element and lex. In the constraint element$(X, List, Y)$, X and Y are variables and $List$ is a list of variables (so element is a $(k + 2)$-ary constraint, where k is the length of $List$). The semantics of element$(X, List, Y)$ is as follows: Y equals to the X-th element of $List$, $Y = List_X$. For example the constraint element$(2, [3, 4, 5], 4)$ is satisfied, while element$(3, [3, 4, 5], 4)$ is not satisfied. The constraint element is frequently used to define n-ary constraints specified in extension, but as we shall show later, we will use it in an innovative way to define uniqueness of unifications (to break so called value symmetries). The constraint lex(L), where L is a list of variable tuples, ensures that the variable tuples in list L are lexicographically ordered. In this paper, we assume that the variable tuples are strictly ordered. For example lex$([(X_1, X_2), (X_3, X_4), (X_5, X_6)])$ can be seen as a compact way of writing $(X_1, X_2) < (X_3, X_4) < (X_5, X_6)$. Hence, lex$([(1, 2), (1, 3), (2, 1)])$ is satisfied, while lex$([(1, 2), (2, 3), (2, 1)])$ is not satisfied because $\neg\ (2, 3) < (2, 1)$. The constraint lex is frequently used to break symmetries in the problem as we shall show later.

Constraint satisfaction techniques are frequently based on the combination of inference techniques (called consistency techniques) and search. The consistency techniques remove inconsistencies from the problem specification (for example, the values violating any constraint) and hence they prune the search space. Arc consistency plays a major role among the general consistency techniques thanks to its good ratio between the strength (how many inconsistencies are removed) and efficiency (how long it takes and how much memory is necessary).

We say that a constraint is *arc consistent* if for any value in the domain of any variable in its scope there exist values in the domains of the remaining variables in the constraints scope such that the value tuple satisfies the constraint. To make the constraint consistent, it is enough to remove values violating the above condition.

This is done by a filtering algorithm attached to each constraint, for example, the filtering algorithm behind the lex constraint is described in [8].

Arc consistency is a local inference technique meaning that in general it does not remove all values that do not belong to the solution. For example, if we assume the constraints $X = Y$ and $X \neq Y$ and the domains for X and Y are $\{1, 2\}$ then both constraints are arc consistent, but the problem has no solution. Therefore a search algorithm is necessary to instantiate the variables. The search algorithm is usually integrated with the inference procedure in the following way: after each search decision (posting a particular constraint such as $X = Y$ or $X \neq Y$ that splits the search space into disjoint subareas, a so called semantic branching [14]) the problem is made arc consistent. If the problem is still consistent, the inference procedure might remove some inconsistencies (for example, inconsistent values of variables) and hence prune the remaining search space. If the problem is found inconsistent then the search algorithm explores the alternative branch(es) or backtracks (if no alternative branch remains).

There are two critical steps when applying the constraint satisfaction techniques to combinatorial problems. The first step is formulating the problem as a CSP—this is called *constraint modeling*. The constraint model defines the search space via the variables and their domains and it also determines how many inconsistencies are removed by the used inference technique such as arc consistency. The second step is defining the *search strategy*. The search strategy determines how the search space is being split to disjoint subparts and in which order these subparts are explored. Naturally, the search strategy influences how fast we reach the solution. In this paper we focus on constraint models for ILP, but we also show how the models are integrated with a search strategy and how some deficiencies of arc consistency (see the above example) can be removed by adding a special inference technique.

2.2 Template consistency problem

As we already mentioned, inductive logic programming deals with the problem of finding a hypothesis that separates positive and negative examples—the hypothesis entails all positive examples and does not entail any negative example. Hypotheses and examples are represented in first order logic as clauses. For simplicity of notation, we will assume clauses to be expressed as sets of literals and, without loss of generality, we will only work with positive literals, that is, non-negated atoms. The negative literals can be encoded as special positive literals (for example *not p* becomes a new atom neg_p).

All terms in learning examples (hypotheses, respectively) will be constants (variables) written in lower (upper) cases. For instance, $E^+ = \{\texttt{arc}(a, b), \texttt{arc}(b, c), \texttt{arc}(c, a)\}$ is a positive example and $H = \{\texttt{arc}(X, Y), \texttt{arc}(Y, Z), \texttt{arc}(Z, X)\}$ is a hypothesis.

As usual in ILP, we use θ-subsumption [20] to approximate the entailment relation between clauses. Hypothesis H subsumes example E, if there exists a substitution θ of variables such that $H\theta \subseteq E$. In the above example, substitution $\theta = \{X/a, Y/b, Z/c\}$ implies that H subsumes E^+. Because the hypothesis contains only variables and examples contain only constants in the atoms, the subsuming

substitution is basically an assignment of values to the variables in the hypothesis. The requirement that a negative example E^- is not subsumed by hypothesis H means that there must not exist any substitution θ such that $H\theta \subseteq E^-$.

The hypothesis is frequently obtained from a *template*, which is again a set of positive literals where all terms are variables. Briefly speaking, the template describes which atoms will appear in the hypothesis and the hypothesis then specifies particular relations between the atoms (expressed as shared variables). For generality, we assume that all variables in the template T are mutually different, that is each variable occurs exactly once in T, as in $T = \{\texttt{arc}(X_1, X_2), \texttt{arc}(X_3, X_4), \texttt{arc}(X_5, X_6)\}$. Note, that this is the most general template containing given atoms as any other possible template with the same atoms can be obtained by unification of certain variables.

To solve the *template consistency problem* given a template T, we look for a substitution ϑ making the hypothesis $H = T\vartheta$ consistent with the learning examples. Since all terms in H are supposed to be variables, the task lies in determining which subsets of variables in T should be unified. The hypothesis H from the last example may be obtained from the above template T by applying unifications $X_2 = X_3$, $X_4 = X_5$ and $X_6 = X_1$ (and then suitably renaming the variables). Clearly, if a hypothesis H obtained from the template T by unifying some variables subsumes a positive example, then T also subsumes that example. The reason for introducing unifications is thus to prevent H from subsuming the negative examples.

In summary, the problem we are addressing in this paper can be formulated as follows. Given a template, a set of positive examples, and a set of negative examples, find unification of variables in the template that will make it subsume all positive examples and no negative example. In the above examples we used atoms describing arcs in graphs. We can see the descriptions of particular graphs as the learning examples. The positive examples represent the graphs containing some common but hidden substructure that is not present in the graphs from the negative examples. The above example template T specifies that this hidden substructure consists of three arcs, but it does not say what the relations between the arcs are. Finally, the hypothesis H obtained from template T then describes how these arcs share the nodes (unification of variables). So the substructure that we looked for is represented by the consistent hypothesis, which in our example is a loop of three arcs.

While in this paper we are not elaborating the question of how to obtain a template, we obviously do need a template generator for designing an ILP algorithm based on our proposed constraint models. We briefly discuss some options now. Given the example sets, one can always produce a template that will certainly subsume a consistent hypothesis if one exists. As follows from [16], such a template would be exponentially large (in the number of positive examples) in the worst case where the examples represent graph-cycles of mutually co-prime lengths. More efficient methods are based for example on the bottom clause concept [19]. We however want our method to be optimal in that it will find the smallest consistent hypothesis if one exists. Therefore we choose an iterative approach where we solve the consistency problem repeatedly with gradually growing templates. Let $\pi_1/\alpha_1, \ldots \pi_n/\alpha_n$ be the symbols and arities of all predicates occurring in the positive example set. We start with an empty template and $i = 1$. Then, in each iteration, we add $\pi_i(v_1, \ldots v_{\alpha_i})$ ($v_1, \ldots v_{\alpha_i}$ are fresh variables) into the template and increment i (modulo n) so we loop over all possible predicates. In specific domains, larger steps

can be made in extending the templates without losing the optimality guarantee. This will be exemplified in the description of experiments in Section 6. There also exist stochastic approaches to template generation that are much faster but do not guarantee optimality [9].

3 Problem abstraction

3.1 An abstract solution approach

The template consistency problem is the problem to find out whether it is possible to unify certain variables in a given template in such a way that the obtained hypothesis (after unifying the variables) subsumes each positive example and does not subsume any negative example. We already mentioned that the subsumption check can be seen as the problem of finding values for the variables in the hypothesis. Obviously, different assignments may be necessary for different examples and hence the subsumption check must be done separately for each example. Let n be the number of variables in the template. Then for each example E_i we can formulate a constraint satisfaction problem C_i defined over n variables $\{X_{i,1}, X_{i,2}, ..., X_{i,n}\}$ such that each problem C_i encodes the problem whether the hypothesis with variables $\{X_{i,1}, X_{i,2}, ..., X_{i,n}\}$ subsumes the example E_i. The constraint satisfaction problem C_i can be derived from the example E_i as described in [18]. We shall show the particular constraint model for this problem in the next sub-section.

We can assume that the set of problems C_i is divided into two disjoint sets *Pos* and *Neg*. This clearly corresponds to the positive and the negative examples in ILP. Now, the template consistency problem is equivalent to that of finding a set *Uni* of pairs (k, l), where $k, l \in \{1, 2, ..., n\}$, such that the following two conditions hold:

$$\forall C_i \in Pos \quad C_i \cup \{X_{i,k} = X_{i,l} \mid (k,l) \in Uni\} \text{ has a solution,}$$
$$\forall C_j \in Neg \quad C_j \cup \{X_{j,k} = X_{j,l} \mid (k,l) \in Uni\} \text{ has no solution.}$$

A solution is an instantiation of variables $\{X_{i,1}, X_{i,2}, ..., X_{i,n}\}$ satisfying all the specified constraints in C_i. In other words, we have a set of constraint satisfaction problems that "share" some equality constraints defined by the set *Uni*.

The above task corresponds exactly to the template consistency problem. We have a template T with n different variables and the set *Uni* describes which variables in the template should be unified to obtain a consistent hypothesis. This is because the constraint satisfaction problem $C_i \cup \{X_{i,k} = X_{i,l} \mid (k,l) \in Uni\}$ defines the subsumption problem for example E_i, where the added equality constraints $X_{i,k} = X_{i,l}$ describe how the hypothesis is obtained from the template.

In summary, to solve the template consistency problem we need to formulate the constraint satisfaction problems encoding the subsumption check—let us call them the *subsumption models*. Similarly, it would be useful to have a constraint model describing how the variables are unified—let us call it a *unification model*. The reason why we are proposing the unification model instead of simply posting an equality constraint each time two variables should be unified in the template is that we will exploit explicit information about which variables are unified in further reasoning.

We shall now describe the subsumption model which is derived from the model in [18]. Then we will present the major contribution of this paper which are two unification models and their possible improvements.

3.2 Subsumption model

Maloberti and Sebag [18] showed how to formulate the subsumption problem as a binary constraint satisfaction problem that can be solved using ad-hoc implementation of forward checking combined with depth-first search. Based on their model, we propose a more traditional constraint model with k-ary constraints that can be solved by any constraint solver with support for tabular constraints. As a main advantage, our approach is more general and hence easy to integrate with other constraints (as we will do in this work). We did not do a direct experimental comparison of our model with the original Django algorithm, but it is known that for hard combinatorial problems the look-ahead techniques (maintaining arc consistency during search as used in our approach) are more efficient than forward checking [21].

Let us recall that a hypothesis H subsumes an example E if there exists a substitution θ of variables such that $H\theta \subseteq E$. The atoms in the hypothesis H can be seen as constraints where the example E defines the compatible tuples of values for these constraints. Let π be a predicate symbol of arity α appearing in some example E. We will call all atoms of this predicate π-atoms. Then we define an α-ary *subsumption constraint* c_π, where $rel(c_\pi)$ is a set of α-tuples such that each tuple is defined by the arguments of one π-atom from the example. For example, for $E = \{\mathtt{arc}(a, b), \mathtt{arc}(b, c), \mathtt{arc}(c, a)\}$ we obtain $rel(c_{\mathtt{arc}}) = \{(a, b), (b, c), (c, a)\}$. Such a constraint is introduced for each π-atom from the hypothesis and this constraint forces the π-atom from the hypothesis to match some π-atom from the example.

Now, for the hypothesis H and a given example E we formulate the θ-subsumption problem as a constraint satisfaction problem in the following way. We take the variables from the hypothesis and define their domains as a set of all constants in the example E. For each atom $\pi(Y_1, Y_2, ..., Y_\alpha)$ in the hypothesis we define a constraint c_π over the variables in the atom—$scope(c_\pi) = \{Y_1, Y_2, ..., Y_\alpha\}$. For example, for hypothesis $H = \{\mathtt{arc}(X, Y), \mathtt{arc}(Y, Z)\}$ and for the above example E we formulate the θ-subsumption problem as a constraint satisfaction problem with variables $\{X, Y, Z\}$, all with domain $\{a, b, c\}$, and constraints $\{c_{\mathtt{arc}}(X, Y), c_{\mathtt{arc}}(Y, Z)\}$. Clearly, a solution to such a CSP, that is an instantiation of the variables satisfying the constraints, defines the substitution θ of variables such that $H\theta \subseteq E$. Hence, the hypothesis subsumes the example if and only if the corresponding CSP has a solution. Note that each example requires a separate set of variables because the subsuming substitutions (instantiations of the variables) may be different between the examples. This is exactly what we did when formulating the problems C_i over the variables $\{X_{i,1}, X_{i,2}, ..., X_{i,n}\}$ in the abstract model in Section 3.1.

4 Solution approach

In the previous section we fully abstracted from the template consistency problem so that we can focus on the formal task specified there, which is finding the set *Uni* satisfying the specified properties. Notice that the set *Uni* adds constraints

to problems C_i so the main reason for including a pair (k, l) in *Uni* is to break satisfaction of some "negative problem" $C_i \in Neg$ (we do not need these equality constraints to make the "positive problems" satisfiable).

It would be possible to model the problem as a *Quantified CSP* (QCSP) [6] supporting the description of a non-existence of a solution for the negative problems. The idea is to use existential quantifiers to express that there is a solution for each positive problem and to use universal quantifiers (or the negation of the existential quantifiers) to express that every assignment of variables violates at least some constraint in the negative problems (or alternatively, there is no solution for the negative problems). We rather decided to mimic the solving ideas behind a QCSP (that is exploring all assignments to model universal quantifiers) in an ad-hoc search procedure implemented on top of existing constraint solvers. The major advantage is that there are many existing classical CSP solvers while only a few experimental QCSP solvers. We can also exploit the powerful inference procedures (arc consistency and global constraints) behind the CSP solvers. In particular, we explore the solutions of the negative problems and break these solutions by adding equality constraints for the variables assigned to different values in the solution. These equality constraints define the unification of variables in the template. The following pseudo-code describes the idea formally.

$Uni \leftarrow \{\}$
for $C_i \in Pos$ do *post_constraint*(C_i); end
for $C_j \in Neg$ do
  for $Sol \in solution(C_j \cup \{X_{j,k} = X_{j,l} \mid (k, l) \in Uni\})$ do
    branch by selecting (k,l) s.t. $X_{j,k} \neq X_{j,l}$ in *Sol*
      $Uni \leftarrow Uni \cup \{(k, l)\}$
      for $C_i \in Pos$ do *post_constraint*($X_{i,k} = X_{i,l}$) end
    end
  end
end
for $C_i \in Pos$ do *instantiate*($X_{i,1}$, $X_{i,2}$, ..., $X_{i,n}$) end

The solver works as follows. First, we post the constraints from the subsumption models of all positive examples. These subsumption constraints must be satisfied independently on the set *Uni*; the set *Uni* only brings additional equality constraints to each subsumption model. Posting these subsumption constraints allows us to exploit the inference techniques behind these constraints (maintaining arc consistency). This helps the search algorithm to detect earlier if any such subsumption constraint is violated and hence any problem from *Pos* cannot be satisfied after adding some equality constraints.

Then we incrementally build the set *Uni* by taking the negative problems one by one and trying to find the solutions for them while satisfying the equality constraints defined by the current set *Uni* (initially empty). If the negative problem has no solution then the algorithm continues as the template remains consistent (the obtained hypothesis does not subsume that negative example). Otherwise, each solution *Sol* of the negative problem $C_j \in Neg$, that is the instantiation of variables $\{X_{j,1}, X_{j,2}, ..., X_{j,n}\}$, needs to be "broken". This is done by taking a pair of variables, say $X_{j,k}$, $X_{j,l}$, that are instantiated to different values in *Sol* and adding an equality

constraint between these variables. This equality constraint is encoded as a pair (k, l) added to *Uni* so that we ensure that *Sol* is no more a solution to the problem $C_j \cup \{X_{j,p} = X_{j,q} | (p, q) \in Uni\}$.

Note that there might be more such pairs (k, l) for a given solution *Sol* so we introduce a choice point here and when a failure is detected later we backtrack to the closest choice point and select another pair to be added to *Uni*. Assuming the example from the previous section $E = \{\texttt{arc}(a, b), \texttt{arc}(b, c), \texttt{arc}(c, a)\}$ with hypothesis $H = \{\texttt{arc}(U, V), \texttt{arc}(X, Y)\}$, the solution could be $\{U/a, V/b, X/a, Y/b)\}$ and we can break this solution by adding one of the constraints $U = V$, $U = Y$, $V = X$, or $X = Y$.

Notice that as soon as we add the pair (k, l) to *Uni* we post the corresponding equality constraints to all positive problems so the underlying inference techniques can find out if there is any inconsistency. These constraints are removed if the pair (k, l) is removed from *Uni* upon backtracking.

After we process all negative problems, we ensure that no negative problem has a solution. However, as the inference techniques are usually incomplete, we still need to validate that all positive problems have a solution by instantiating their variables. If this is not the case, we backtrack to the closest choice point and select some alternative unification there. For solving both positive and negative problems, we use a standard constraint satisfaction approach that instantiates the variables using min-dom heuristic (the variables with the smallest domains are instantiated first) and maintains arc consistency during search. The values for variables are tried in the order of appearance in the problem.

5 Constraint models for unification

The above-described solving approach uses a base constraint model (the constraints from the "positive" problems) that is extended by the equality (unification) constraints derived from the solutions of "negative" problems. In each decision (choice) point we solve another CSP (a negative problem) whose solution defines the possible options (branches) for the search algorithm. Notice that if we refuse the unification of some variables X_k and X_l at some step then it may still happen that the solver tries this unification again and again to break other solutions of the same problem or of other negative problems. This approach is known as *syntactic branching* and it follows the following principle. Assume that we need to resolve two disjunctive constraints, say $p \vee q$ and $p \vee r$ in this order (in our task each disjunction defines possible unifications breaking a certain solution). We first branch on adding p to the model with the alternative branch of adding q to the model. Assume that the first branch with p failed and we are now exploring the branch with q. When resolving the second disjunctive constraint, we are again trying to add p though we already proved that this is not possible. This is clearly undesirable behavior as it decreases the time efficiency of search by exploring branches that are known to fail. Giunchiglia and Sebastiani [14] proposed an alternative way called *semantic branching* that is based on the following idea. When the branch with p fails, we add $\neg p$ together with q in the alternative branch. The inference algorithm (unit propagation in this case, which

is similar to arc consistency) can then deduce that to resolve $p \vee r$ we must add r to the model and the branch with p is not tried to resolve this disjunction.

Note that the idea of semantic branching cannot be directly applied to prevent the above inefficiency of our search procedure—if X_k and X_l cannot be unified then posting the negated constraints in the form $X_{i,k} \neq X_{i,l}$ among the corresponding variables in all positive problems cannot be done because the variables $X_{i,k}$ and $X_{i,l}$ can be instantiated to identical value in some positive problems C_i but not in all. The reason is that the negation of the conjunction of equality constraints ($\neg(X_1 = X_2 \wedge Y_1 = Y_2)$) is a disjunction of inequality constraints ($X_1 \neq X_2 \vee Y_1 \neq Y_2$). General disjunctive constraints do not propagate well unless a dedicated filtering algorithm is proposed for a particular type of disjunction (see for example the filtering algorithms for the constraints modeling unary resources [2]). We suggest to use meta-reasoning on unifications that is based on keeping the information about which variables were unified and which variables were decided not to be unified in the hypothesis.

5.1 Index model

Our first model is based on the observation that if a set of variables is unified then we can take the variable with the smallest index to represent this set and all other variables in the set are mapped to this variable. For example, unification $X_2 = X_3 = X_4$ can be represented by a mapping $X_3 \rightarrow X_2$ and $X_4 \rightarrow X_2$. The proposed constraint model uses index variable I_i for the i-th variable in the template to describe the mapping. The domain of I_i is $\{1, ..., i\}$. Each time a pair (k, l) is added to *Uni*, the constraint $I_k = I_l$ is posted to describe that these two variables are unified with (mapped to) the same variable. If it is decided that this pair is not unified (the alternative branch in the choice point), constraint $I_k \neq I_l$ is posted. Hence, the semantic branching can be applied to the index variables. Note that this is correct as constraint $I_k \neq I_l$ only says that these variables should not be unified in the future but it does not force the corresponding variables in the positive problems to be different.

To ensure that each variable is mapped to the first variable in the set of unified variables we use a constraint $\forall i = 1, ..., n$ element$(I_i, [I_1, .., I_n], I_i)$, where n is the total count of variables. Recall that element$(I_i, [I_1, .., I_n], I_i)$ means that at the position I_i in the list $[I_1, .., I_n]$ there must be value I_i. For example, [1, 1, 2] is not a valid list of indexes (it represents $X_1 = X_2$ and $X_2 = X_3$) as it violates the constraint element(2, [1, 1, 2], 2). In other words we map variable X_3 to variable X_2 which maps further to X_1. This is allowed by the domains of variables I_i, but as we showed it violates the element constraint. Note also that [3, 3, 3] is not a valid list too, though it satisfies the element constraints. The reason is that value 3 is not in domains of I_1 and I_2. The correct representation of this unification should be [1, 1, 1], where both variables X_3 and X_2 are mapped to X_1. The element constraints thus ensure that each set of unifications is represented by exactly one list of indexes—each variable maps to the first variable in the set of unified variables and this first variable is unique in each set of unified variables. We call this set of constraints an *index model*. Note that the index model is a form of *value symmetry*, where bijection on values preserves solutions. For example, assume that X_2 is mapped to X_1 ($I_1 = I_2 = 1$). Then mapping X_3 to X_1 gives a solution identical to mapping X_3 to X_2 and hence values 1 and 2 are interchangeable for the variable I_3. The element constraint forbids value 2 for I_3 in

this case and hence breaks this symmetry. Notice also that due to the domains of variables I_i, we cannot map a variable to a variable with a larger index (X_2 to X_3), because the maximal value for I_i is i. Hence both the specification of domains and the constraints element are important in the model.

Channeling constraints When we have the model describing which variables should be unified, we can connect this model to the positive problems rather than posting the special equality constraints $X_{i,k} = X_{i,l}$ each time a decision about unification of variables is done as in Section 4. The advantage of this direct connection is that if inference in some positive example deduces that two variables are different then this information is propagated back to the index model. This connection is realized via so called *channeling constraints*. In our model, this is implemented again using the element constraints connecting the index model with each positive problem C_i as follows: $\forall j = 1, ..., n$ element$(I_j, [X_{i,1}, .., X_{i,n}], X_{i,j})$. Recall that I_j is the index of a variable to which variable X_j maps, that is variables X_j and X_{I_j} are unified. Hence, for the problem C_i the element constraints are identical to constraints $X_{i,j} = X_{i,I_j}$. These channeling constraints allow propagation of information from the positive problems to the index model and vice versa.

Reducing symmetries The index model can also exploit the structure of the template. Typically, the template represents a set of atoms, for example $\{\texttt{arc}(X_1, X_2), \texttt{arc}(X_3, X_4), \texttt{arc}(X_5, X_6)\}$. However, the template itself is modeled as a list (variables are ordered) so swapping atoms generates a different unification set. Moreover, unifying some variables may collapse the template to a smaller hypothesis. In our example, $X_1 = X_3$ and $X_2 = X_4$ collapses the first two atoms to one. This is not a desirable behavior in situations where we know that there is no smaller consistent hypothesis. This is for example the case when we have already explored all possible hypotheses provided by smaller templates (as in our approach, see Section 2.2). To remove this deficiency we suggest the following constraint. We collect the tuples of index variables belonging to variables in atoms with the same name, in our example, we obtain pairs $(I_1, I_2), (I_3, I_4), (I_5, I_6)$. For each atom and a corresponding list of variable tuples L, we post a constraint lex(L) forcing the variable tuples in list L to be lexicographically ordered [8]. In our example, it means $(I_1, I_2) < (I_3, I_4) < (I_5, I_6)$, which ensures that no atom arc is unified with another atom arc in the template (and hence no atom will disappear from T). This constraint can also be seen as a form of symmetry breaking. The example in Fig. 1 demonstrates the complete index model (without the channeling constraints).

We are aware of other permutation symmetries that appear in the model but are not forbidden by the *lex* constraint: $\{\texttt{arc}(X_1, X_2), \texttt{arc}(X_2, X_4), \texttt{arc}(X_4, X_6)\}$ is identical (after renaming the variables) to $\{\texttt{arc}(X_1, X_2), \texttt{arc}(X_3, X_4), \texttt{arc}(X_4, X_1)\}$, but both hypotheses are allowed in our constraint model. These symmetries appear because we model the set of atoms as a list. Consequently, one hypothesis corresponds to several instantiations of the index variables and hence, when we will be exploring possible candidates for the hypothesis by instantiating the index variables (deciding the unifications), we may obtain the same hypothesis several times. This is not a desirable behavior, but it seems that it cannot be completely avoided in polynomial time as identifying identical hypotheses (after renaming the

Template:

$$\texttt{arc}(X_1, X_2), \texttt{arc}(X_3, X_4), \texttt{arc}(X_5, X_6), \texttt{red}(X_7), \texttt{red}(X_8), \texttt{red}(X_9), \texttt{green}(X_{10})$$

Unification model with index variables:

variables	domains	constraints
$I_1, ..., I_{10}$	$\forall i = 1, \ldots, 10$	$\forall i = 1, \ldots, 10$
	$D_i = \{1, ..., i\}$	$\mathsf{element}(I_i, [I_1, .., I_{10}], I_i)$
		$\mathsf{lex}([(I_1, I_2), (I_3, I_4), (I_5, I_6)])$
		$\mathsf{lex}([(I_7), (I_8), (I_9)])$

Fig. 1 An example of index model for a given template

variables) inherently includes the graph isomorphism problem which is one of the NP problems neither known to be soluble in polynomial time nor NP-complete [13].

5.2 Boolean model

One of the problems of the index model is that it cannot easily detect conflicting constraints $I_i \neq I_j$ and $I_i = I_j$ unless the domains are singleton. This deficiency of local inference, namely arc consistency, can be resolved by applying global reasoning.

Assume that we have n variables in the template. We propose to use a matrix U of size $n \times n$ to describe which variables are unified (1) and where the unification is forbidden (0). Initially, the matrix consists of unbounded variables $U_{i,j}$ such that $U_{i,j}$ is identical to (unified with) $U_{j,i}$ (the matrix is symmetric, since the unification is an equivalence relation) and $U_{i,i} = 1$ (bounded). When a pair (i, j) is added to *Uni*, we simply unify rows i and j of the matrix U, that is we post equality constraints $\forall k\ U_{i,k} = U_{j,k}$ (and also $U_{k,i} = U_{k,j}$). In particular we obtain $U_{i,j} = U_{j,i} = 1$. This unification operation implicitly keeps the transitive closure of the equality constraints between the variables (if $U_{i,l} = 1$ for some l then after the unification $U_{j,l} = 1$ also holds and vice versa). When a pair (i, j) is decided not to be unified then we set $U_{i,j} = 0$. If this is not possible because of $U_{i,j} = 1$ then we can immediately deduce a failure (we have a conflict between requiring X_i and X_j to be unified and not to be unified at the same time). Similarly, if sometime later we deduce that $X_i = X_j$ either directly via posting this constraint or indirectly via the transitive closure of equality (unification) constraints then we can also deduce a failure. The Boolean matrix for template $\{\texttt{arc}(X_1, X_2), \texttt{arc}(X_3, X_4), \texttt{arc}(X_5, X_6)\}$, where we already decided that $X_1 \neq X_2$, $X_3 \neq X_4$, $X_5 \neq X_6$ (see the next section on hints), will look like this:

	X_1	X_2	X_3	X_4	X_5	X_6
X_1	1	0	A	B	C	D
X_2	0	1	E	F	G	H
X_3	A	E	1	0	I	J
X_4	B	F	0	1	K	L
X_5	C	G	I	K	1	0
X_6	D	H	J	L	0	1

Now, if we decide during search that $X_2 = X_3$ then we obtain the following table by unifying the second and third rows (and columns):

	X_1	X_2	X_3	X_4	X_5	X_6
X_1	1	0	0	B	C	D
X_2	0	1	1	0	G	H
X_3	0	1	1	0	G	H
X_4	B	0	0	1	K	L
X_5	C	G	G	K	1	0
X_6	D	H	H	L	0	1

Notice that we can immediately deduce that for example $X_2 \neq X_4$ so if we ever try unification $X_2 = X_4$ it will immediately fail. Or if we decide that $X_2 \neq X_5$ ($G = 0$) then we immediately get also $X_3 \neq X_5$.

The global reasoning behind the Boolean model can be implemented as a special global constraint but the above operations over the Boolean matrix do the same job. Note also that the Boolean model can be used independently of the index model or both models can be combined to exploit their complementary strengths (each model represents some form of local inference). If the Boolean model is used independently then the connection to positive problems is realized easily by adding the variables from each positive problem as a separate column to the Boolean matrix. Then unifying the rows of the matrix automatically unifies the corresponding variables in the positive problems.

It may seem that the Boolean model infers more information than the index model, but it is not clear how to implement the symmetry breaking from the index model there. Hence both models have their value (see the section on experiments).

Hints

We already explored some structural information in the symmetry breaking constraints of the index model, but we can do more. Assume that some constraint in some positive problem C_i forbids a pair of variables $X_{i,k}$ and $X_{i,l}$ to be instantiated to the same value, for example the constraint allows only the pairs $\{(a, b), (b, c), (c, a)\}$. If we post constraint $X_{i,k} = X_{i,l}$ then arc consistency does not deduce a failure immediately (domains of both variables are $\{a, b, c\}$) because the constraints are treated independently and the failure is only detected when we attempt to instantiate the variables or prune their domains. Nevertheless, it is possible to identify such pairs $(X_{i,k}, X_{i,l})$ of non-unifiable variables in positive problems in advance and to express this information in the Boolean model by setting $U_{k,l} = 0$.

We call it a *hint* and in our experiment we deduce hints only from individual constraints. More precisely, $U_{k,l} = 0$ if variables $X_{i,k}$ and $X_{i,l}$ appear in the same constraint in some positive problem C_i and there is no tuple satisfying that constraint such that $X_{i,k} = X_{i,l}$. In terms of ILP, from the positive example $E = \{\texttt{arc}(a, b), \texttt{arc}(b, c), \texttt{arc}(c, a)\}$ we can deduce that $X \neq Y$ for any pair of variables appearing in any atom $\texttt{arc}(X, Y)$ of the template. Note finally that such hints can be found fully automatically just by exploring the value tuples in each constraint.

6 Experimental results

Searching for unifications in the template is a novel approach to obtain hypothesis in ILP. As we are not aware about another published approach for solving the template consistency problem, we focused on the experiments comparing the efficiency of the proposed models. In particular, the experimental results demonstrate the contribution of individual models/constraints to overall efficiency.

Afterwards the performance of the fully enhanced system is compared to the popular ILP system *Aleph* [22] that we use as a gold standard for comparison. We also did a comparison with another ILP system called *nFOIL* [17]. To get a domain-independent insight into the performance of the systems, we are using example sets sampled from various graph-generative models (Barabási–Réka model [3] and Erdős–Rényi model [12]) and predicate-generative models (Botta model [7]).

6.1 Internal validation

The constraint models were implemented in SICStus Prolog 4.0.8 and the experiments run on 2.53 GHz Core 2 Duo processor with 4 GB RAM under Windows 7 Professional system.

We considered the problems of identifying common structures in randomly generated structured graphs. In particular, we generated random structured graphs with 20 nodes where the new nodes are attached to the graph with 3 arcs according to the Barabási–Réka model [3]. Each dataset consists of ten positive and ten negative examples representing the graphs and we generated the problems in such a way that the positive examples share a substructure of a given size that was not present in the negative examples. This structure consists of 5 nodes represented by 10 atoms constructed from a base set of predicate symbols $\{\mathtt{a}/2, \mathtt{b}/1, \mathtt{c}/1\}$ (binary atoms describe arcs and unary atoms describe "colors" of nodes). Hence, we had the consistent hypothesis in our hand. We removed the unification of variables from the hypothesis to obtain a template that is used in the experiments (the template has 15 unique variables). Consequently, the template contained the right atoms and the task was to re-discover the unification of variables to obtain a consistent hypothesis.

Table 1 shows the results, in particular, the runtimes in milliseconds for generated datasets. The first two columns show the results for the Index model and the Boolean

Table 1 Comparison of runtimes (milliseconds) for identifying the common structures in graphs (Barabási–Réka)

Index	Boolean	Combined	Decoupled		
			Full	No SB	No hints
32	202	47	**31**	250	32
250	7956	343	250	10312	**249**
54351	85192	12495	**6365**	108780	54163
49421	14679	52541	**6552**	18362	49858
>600000	531670	132288	**38579**	>600000	>600000
>600000	**38735**	347804	108717	50171	>600000
>600000	**65302**	>600000	378005	84817	>600000
>600000	>600000	>600000	**158308**	>600000	>600000

The bold numbers indicate the best runtime

model from Section 5. The Boolean model scales better because it can infer more information about possible and forbidden unifications. We also put the models together, meaning that we used the subsumption models connected to the Index model and to the Boolean model. This is called a "Combined model" and the Index model and the Boolean model communicate indirectly via the subsumption model there. Combining models is a form of adding *redundant constraints*. Each model describes the problem completely so the constraints from the other model are not necessary to define the solution (such constraints are called redundant). However, redundant constraints may increase the inference power and prune the search space more. As the results show, the combination of models indeed helped in some problems while it led to worse runtimes in other problems. We realized that the overhead is mainly due to the channeling constraints connecting the Index model with the subsumption models as these constraints have very limited influence on inference. Hence we removed these channeling constraints so the Index model was actually decoupled from the other models and it was used mainly to detect impossible unifications (recall that during search we basically explore unifications of variables). If we decide to unify some variables then we post an equality constraint between the corresponding index variables and we unify the corresponding rows in the Boolean matrix (and also the variables in the positive examples). If we decide that some variables are not unified, then we post an inequality constraint between the corresponding index variables and we place value zero to a corresponding cell in the Boolean matrix. In both cases, either model can deduce that a particular equality or inequality cannot hold and hence we must backtrack. We call the model "Decoupled full" in Table 1. The experimental results confirm that this decoupling significantly reduced the overhead and led to the best runtimes (we will comment later on the examples where the Boolean model was better). It may seem that in the case of decoupling the Index model is useless and only adds overhead for propagating its constraints. Hence, we removed the symmetry breaking lex constraints from this model, this is called "Decoupled no SB". The experimental results clearly demonstrate that these constraints are important and because they are expressed for the Index model only, this also shows that the Index model is useful. Notice that if we remove the symmetry breaking constraints we obtain the runtimes that were about 20–30 % larger than the runtimes from the Boolean model. This is the overhead for propagating the constraints in the Index model, which, on the other side, does not contribute to stronger pruning of the search space. Hence, if there are no symmetry breaking constraints in the Index model then this model is fully redundant without any benefit (the Index model describes the possible unifications that are also described by the Boolean model). Finally, we evaluated the role of hints that were deduced from the positive examples and were reflected in the Boolean matrix by setting values of some cells and by posting equality or inequality constraints in the Index model before starting the search procedure. Again, the results clearly demonstrate that the hints are very important for runtime efficiency.

There are two problems where the Boolean model beats all other models. The reason is that the random graphs in these problems contain a common substructure that is a subset of the template. When generating the problems, we know that there is a common substructure shared by the positive examples and non present in the negative examples, but there may be a smaller substructure with the same properties and hence a smaller hypothesis may exist. Recall that the lex constraints forbid collapsing atoms in the template and hence the Index model does not allow finding hypothesis

with a smaller number of atoms than specified in the template. On the other side, the Boolean model does not have such restriction and it allows a smaller hypothesis (easier to find) than the other models. This behavior can also be observed at the results for the model without symmetry breaking constraints. The runtimes for these two problems are also very good there because the missing symmetry breaking constraints allow us to find a smaller hypothesis even when the Index model is present.

We also did experiments with other data sets namely we generated random structured graphs using the Erdős–Rényi model [12]. Each dataset again consisted of ten positive and ten negative examples (graphs). The random graphs contain 20 nodes with the density of arcs 0.2 and the template contains 5 nodes. Again, the base set of predicate symbols is {a/2, b/1, c/1}. Table 2 shows the results, in particular, it describes the size of the template (the number of atoms and variables) and the runtimes in milliseconds. The results confirm that the conclusions from the previous paragraph about the contribution of symmetry breaking and hints to overall efficiency and about the importance of decoupling the models are valid. Again, the Boolean model is better for one problem due to the same reason as before.

6.2 Comparative Evaluation

We further compared our approach (the "Decoupled full" model) to the standard ILP system Aleph [22] and the state-of-the-art system nFOIL [17]. Apart from the Erdős–Rényi and Barabási–Réka data-generating models, we also used the data generator from [7] which has become an ILP benchmark representing hard ILP learning tasks. We refer to this latter model as the *Botta model*.

Parameters for each data-generating model are stated in Table 3. Given the large performance gap between Aleph and nFOIL, two variants were prepared (*easy* for comparing the CSP solver against Aleph and *hard* for comparing against nFOIL). The datasets were obtained in the following way: For each data-generating model and each combination of parameters in Table 3 we randomly constructed a target hypothesis. Given the target hypothesis we randomly generated 15 positive and 15 negative examples (50 and 50 in the *hard* version), which serve as the input to both systems. The objective is to discover a hypothesis, which can discriminate all the positive examples from all the negative ones. We measured the time taken by all the approaches to achieve this goal. Processes running longer than 30 min were

Table 2 Comparison of runtimes (milliseconds) for identifying the common structures in graphs (Erdős–Rényi)

#atoms	#vars	Index	Boolean	Combined	Decoupled		
					Full	No SB	No hints
7	9	0	15	16	**0**	16	**0**
8	11	0	16	15	**0**	15	**0**
8	11	31	281	31	**21**	343	**32**
8	11	94	343	109	**78**	421	**78**
9	13	94	1046	125	**93**	1373	94
9	13	328	1810	436	**312**	2309	**312**
9	13	1232	9626	1606	**1170**	12324	1185
10	15	>600000	**86425**	>600000	236514	110981	>600000

The bold numbers indicate the best runtime

Table 3 Parameters of graph models used for comparative evaluation

Easy	Hard
Erdős–Rényi model:	
$n \in \{2, 4, 6, 8\}$	$n \in \{4, 5, 6, 7\}$
$p \in \{0.1, 0.3, 0.6\}$	$p \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$
$N \in \{\lfloor n \cdot p \cdot x \rfloor \mid x \in \{4, 7, 10\}\}$	$N \in \{6, 10, 14\}\}$
$P \in \{0.1, 0.3, 0.6\}$	$P \in \{0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$
Barabási–Réka model:	
$n \in \{2, 4, 6, 8\}$	$n \in \{2, 3, 4, 5, 6\}$
$k \in \{\lfloor x + n/5 \rfloor \mid x \in \{1, 3, 5\}\}$	$k \in \{2, 4, 6, 8, 10, 12, 14, 16, 18\}$
$N \in \{\lfloor n \cdot x \rfloor \mid x \in \{1.8, 2.5, 3.6\}\}$	$N \in \{6, 7, 8, 9, 10\}$
$K \in \{\lfloor k \cdot x \rfloor \mid x \in \{2.5, 3.6, 5.1\}\}$	$K \in \{2, 4, 6, 8, 10, 12, 14, 16, 18\}\}$
Botta model:	
$m \in \{2, 4, 6, 8\}$	$m \in \{6, 10, 14, 18\}$
$n \in \{\lfloor (m-1) \cdot x \rfloor \mid x \in \{2, 2.5, 3\}\}$	$n \in \{6, 10, 14, 18\}$
$N \in \{5, 10, 20\}$	$N \in \{6, 10, 14, 18\}$
$L \in \{\lfloor m \cdot x \rfloor \mid x \in \{1.7, 3.4, 5.1\}\}$	$L \in \{2, 3, 4, 5, 6, 7, 8\}$

Symbol $\lfloor \cdot \rfloor$ represents the *floor* function. **Erdős–Rényi model:** n is the number of nodes in the target concept, p the density of arcs in the target concept, N the number of nodes in the (pos. or neg.) example and P the density of arcs in the example. **Barabási–Réka model:** n is the number of nodes in the target concept, k the number of arcs for a new node in the target concept, N the number of edges in the (pos. or neg.) example and K the number of arcs for a new node in the example. **Botta model:** m is the number of predicates in the target concept, n the number of variables in the target concept, N the number of literals per predicate symbol in each example and L the number of constants in each example

terminated and we counted the number of successfully solved examples. Notice that the template is not given as an input (see further for the description of the method used to generate templates for the CSP solver).

System parameters As it is well known by ILP researchers, Aleph's performance is very sensitive to the setting of its user-adjustable parameters. By preliminary validation, we attempted to find Aleph's optimal settings by trying about 800 combinations of parameter setting, using 6 examples randomly drawn from the Botta model. The values that turned out to be best performing are shown in Table 4. These settings were maintained during the whole testing process.

The nFOIL system has only one tuneable parameter relevant to our settings, which is the *beam width* of the internal beam search algorithm. Figure 5 shows the effect on nFOIL's performance. Apparently the default greedy search (*beam width* = 1) cannot solve a single instance (the program was interrupted after 30 min) and

Table 4 Parameter settings for Aleph

`search`	`nodes`	`clauselength`	`depth`	`noise`
`df`	10^{10} (i.e. no limit)	20	100	0

The meaning of the settings is as follows: depth-first strategy is used for clause search, the number of clauses explored during the search is not limited, the maximum size of a considered clause is 20 literals, the maximum depth of the resolution tree in clause evaluation is 100 and learning examples are not noisy, meaning that clauses are not allowed to entail a negative example

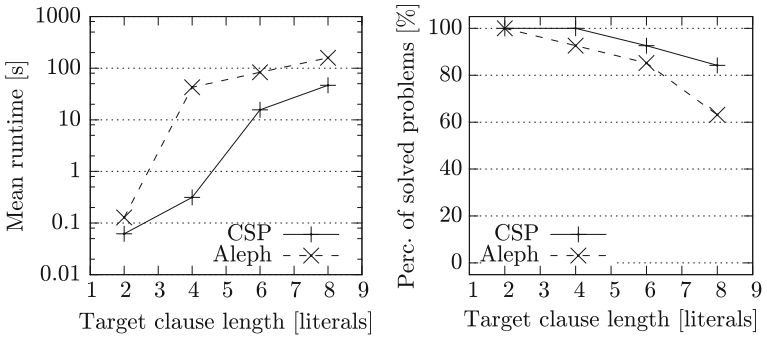

Fig. 2 Results for the Erdős–Rényi model in the *easy* variant

also increasing the parameter beyond the value 5 does not improve the performance significantly. Finally, all experiments were executed with *beam width* = 9.

Template generation As the proposed method solves a template-consistency problem and the evaluation is formulated as the hypothesis discovery problem, the template must be formulated externally. For this purpose we have used an incrementally growing template: In the Erdős–Rényi and Barabási–Réka domain, where the hypothesis are constructed from a base set of predicate symbols {a/2, b/1, c/1}, the template in the n-th iteration consists of the base set repeated n times. However in cases where the target clause does not contain an equal number of occurrences of each predicate symbol, the template will contain redundant literals. Nevertheless they do not limit the solubility of the problem, because the θ subsumption allows two literals to be mapped one onto another by unifying all their variables. In the Botta domain, all literals forming a hypothesis have different predicate symbol (always of arity 2) and therefore the template in the n-th iteration contains the first n literals from an infinite list [lit1/2, lit2/2, lit3/2, ...]. In this case no redundancy can occur.

Testing platform The experiments were performed on a 3 GHz Intel Core i3 with 8 GB RAM and Debian 6. We used SICStus 4.1.1 for running the CSP code and YAP 6.0.3 for running Aleph 5. Unfortunately it is not possible to run both systems

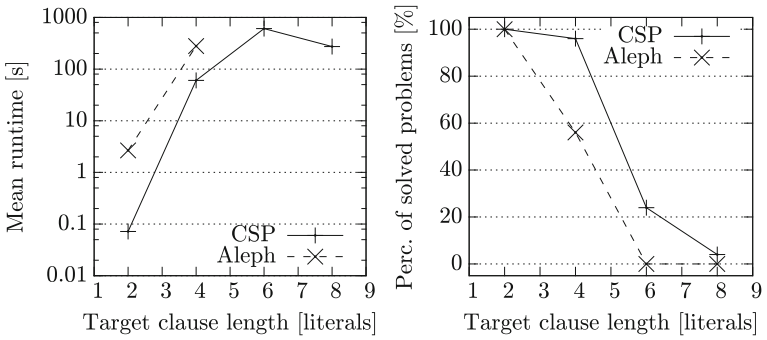

Fig. 3 Results for the Barabási–Réka model int the *easy* variant

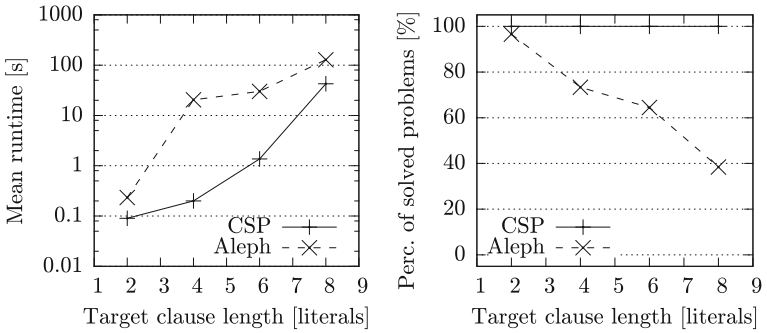

Fig. 4 Results for the Botta model in the *easy* variant

using the same Prolog, because Aleph is not compatible with SICStus and the presented system needs the constraint satisfaction libraries present in SICStus.

Reproducibility of experiments The code necessary for reproducing the experiments is available at http://ida.felk.cvut.cz/ilp2csp/test.tar.bz2. The commercial SICStus Prolog compiler is needed to run the experiments.

Results Figures 2, 3 and 4 clearly show that our constraint-based approach equipped with the iterative template generation finds the target clause significantly faster than Aleph does. The largest speedup is apparent in clause lengths smaller than 6 literals, where it exceeds orders of magnitude.

To interpret the drops in mean runtimes observed for the longest target clauses, note that here a large percentage of runs fail to find a solution (right-hand side diagrams) and these runs do not contribute to the average runtime.

The diagrams showing the percentage of solved problems also bring an interesting comparison. Firstly in all domains the CSP algorithm was able to solve problems of higher complexity than Aleph. Moreover in the Botta domain, the curve for the CSP algorithm remains constant meanwhile in other domains the success-rate drops

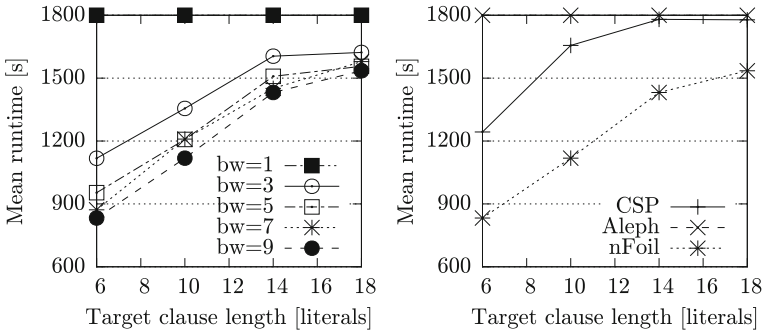

Fig. 5 *Left* effect of the *beam-width* parameter on the performance of the nFOIL algorithm (measured on the *hard* variant). *Right* Results for the Botta model in the *hard* variant

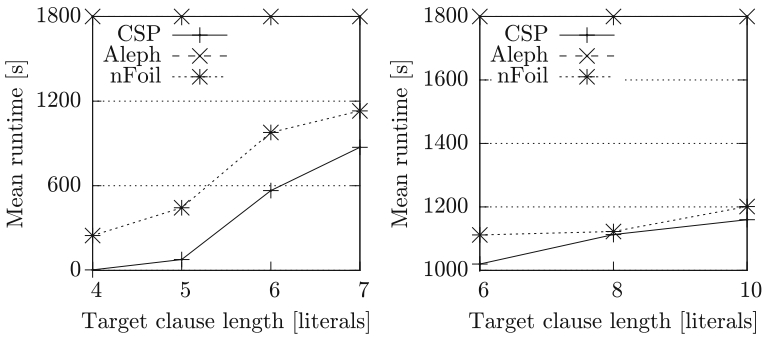

Fig. 6 Results for the Erdős–Rényi and Barabási–Réka models in the *hard* variant

rapidly with the growing complexity of the problem. We can ascribe it to the fact that the first consistent template in the Botta domain are already optimal.

Figures 5 and 6 compare the system with nFOIL on the *hard* variant of the models.[1] Results of both systems in the Barabási–Réka model are comparable, nFOIL being slightly slower. In the Erdős–Rényi model, the constraint-based approach clearly outperforms nFOIL by tens of percents. The Botta domain the result is opposite, where the constraint-based approach loses by a margin of approximately 5 min regardless of the task's complexity. We can only speculate about the true reason for this result. It is likely that the nFOIL's inbuilt heuristics can select relevant literals quickly, meanwhile our system iterates over all templates of insufficient length.

7 Conclusions

We proposed a constraint satisfaction approach for solving the template ILP consistency problem formulated in [15]. Whereas constraint satisfaction techniques have previously been used in ILP to enhance subsumption checking, as far as we know, our work is the first attempt to use them to tackle the problem of searching a consistent clause, despite the central role of this latter problem in ILP. We proposed two core constraint models to describe unification of variables in the template and we connect these models with a constraint model for subsumption checks motivated by the Django algorithm. The idea of the index model was sketched in [5] and further investigated and complemented by the Boolean model in [4]. In addition to using some classical modeling techniques, such as symmetry breaking, we proposed several enhancements specific to the template consistency problem. The efficiency of our approach is demonstrated experimentally with training data sampled from three different generative models. In particular, we compared our approach to the existing ILP systems Aleph and nFOIL. This paper thus contributed to the general efforts of making ILP systems fast. Some improvements focusing on more efficient

[1] Aleph was unable to solve a single instance, which is shown by a straight line at 1,800 s.

generation of templates are already ongoing [9] and promise further improvement of time efficiency.

While most of previous research was concerned with 'noisy' scenarios where a *sufficiently good* clause is searched, we provided a fast method for the alternative 'crisp' ILP scenario where one seeks the smallest possible clause *perfectly separating* positive examples from negative examples.

Acknowledgements Roman Barták is supported by the Czech Science Foundation through project GAP202/12/G061 "Centrum excelence—Institut teoretické informatiky." Radomír Černoch, Ondřej Kuželka, are Filip Železný are supported by the Czech Science Foundation through project 103/10/1875 "Learning from Theories."

References

1. Alphonse, E., & Osmani, A. (2009). Empirical study of relational learning algorithms in the phase transition framework. In *Machine learning and knowledge discovery in databases* (pp. 51–66).
2. Baptiste, P., Le Pape, C., Nuijten, W. (2001). *Constraint-based scheduling: Applying constraint programming to scheduling problems*. Kluwer Academic Publishers.
3. Barabási, A.-L., & Réka, A. (1999). Emergence of scaling in random networks. *Science, 286*(5439), 509–512.
4. Barták, R. (2010). Constraint models for reasoning on unification in inductive logic programming. In *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA 2010)* (pp. 101–110). Springer Verlag.
5. Barták, R., Kuželka,O., Železný, F. (2010). Using constraint satisfaction for learning hypotheses in inductive logic programming. In *Proceedings of the 23rd international Florida AI Research Society conference (FLAIRS 2010)* (pp. 440–441). AAAI Press.
6. Bordeaux, L., & Monfroy, E. (2002). Beyond NP: Arc-Consistency for quantified constraints. In *Principles and practice of Constraint Programming—CP 2002* (pp. 17–32). Springer Verlag.
7. Botta, M. Challenging relational learning—dipartimento di informatica—università di torino. http://www.di.unito.it/~mluser/challenge/index.html. Accessed 6 February 2013.
8. Carlsson, M., & Beldiceanu, N. (2002). Arc-Consistency for a chain of lexicographic ordering constraints. http://soda.swedish-ict.se/2267. Accessed 6 February 2013.
9. Chovanec, A., & Barták, R. (2011). On generating templates for hypothesis in inductive logic programming. In *Advances in artificial intelligence (proceedings of 10th Mexican International Conference on Artificial Intelligence (MICAI 2011), Part 1* (pp. 162–173). Springer Verlag
10. Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann Publishers Inc.
11. Džeroski, S., & Lavrač, N. (2001). *Relational data mining*. Springer Verlag.
12. Erdős, P., & Rényi, A. (1959). On the evolution of random graphs. *Publicationes Mathematicae, 6*, 290—297.
13. Garey, M.R., & Johnson, D.S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness*. W. H. Freeman & Co.
14. Giunchiglia, F., & Sebastiani, R. (1996). Building decision procedures for modal logics from propositional decision procedures—the case study of modal K(m). In *CADE13: Proceedings of 13th international conference on automated deduction* (pp. 583–597). Springer Verlag.
15. Gottlob, G., Leone, N., Scarcello, F. (1999). On the complexity of some inductive logic programming problems. *New Generation Computing, 17*(1), 53–75.
16. Horváth, T., Sloan, R.H., Turán, G. (1997). Learning logic programs by using the product homomorphism method. In *COLT '97: Proceedings of the 10th annual conference on computational learning theory* (pp. 10–20). New York, NY: ACM.
17. Landwehr, N., Kersting, K., De Raedt, L. (2005). nFOIL: Integrating naive bayes and FOIL. In *Proceedings of the 20th national conference on Artificial intelligence—Volume 2* (pp. 795–800). AAAI Press.
18. Maloberti, J., & Sebag, M. (2004). Fast Theta-Subsumption with constraint satisfaction algorithms. *Machine Learning, 55*(2), 137–174.

19. Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: theory and methods. *Journal of Logic Programming, 19*(20), 629–679.
20. Plotkin, G., Meltzer, B., Michie, D. (1970). A note on inductive generalization. *Machine Intelligence, 5*, 153–163.
21. Sabin, D., & Freuder, E.C. (1994). Contradicting conventional wisdom in constraint satisfaction. *Principles and practice of constraint programming* (pp. 162–173). Springer Verlag.
22. Srinivasan, A. Aleph manual. http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph. Accessed 6 February 2013.