Formulating Template Consistency in Inductive Logic Programming as a Constraint Satisfaction Problem

Roman Barták

Charles University, Faculty of Mathematics and Physic
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
bartak@ktiml.mff.cuni.cz

Ondřej Kuželka, Filip Železný

Czech Technical University, Faculty of Electrical Engineering
Karlovo nám. 13, 121 35 Praha 2, Czech Republic
{zelezny, kuzelon2}@fel.cvut.cz

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 a unification of the contained variables such that no negative example is subsumed by the hypothesis and all positive examples are subsumed.

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 hypotheses (Muggleton and De Raedt 1994). ILP provides a powerful framework for relational data mining (Džeroski and Lavrač 2001). For the sake of complexity analysis, a formalization of core ILP tasks was proposed by the seminal paper (Gottlob et al 1999) still serving as a reference framework in newer studies (e.g. Alphonse and Osmani 2009). Gottlob defines 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* (Plotkin 1970) 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 here, it is instead required that $H = T\theta$ for some substitution θ, where T is a given clause called a *template*. Gottlob shows that both problems are equivalent in terms of computational complexity and belong among Σ_2^P complete problems. 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 (2004) addressed the above 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 heuristics, Django brought dramatic speed-up for θ-subsumption and consequently for the entire ILP system. This result clearly motivates the exploration of using constraint satisfaction techniques also for the above complexity source (2). We pursue this goal here. We frame our approach in the template consistency problem. Thus, given learning examples and template T, the general purpose of our CSP model is to find a substitution θ making Tθ 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.

The paper is organized as follows. We will first formally introduce the problem addressed in this paper and give some background on constraint satisfaction. Then, we will describe the constraint models used to solve the problem and show how the constraint models are integrated within a search procedure. After that, we will propose some extensions to improve efficiency. Finally, we will present experimental results demonstrating the efficiency of the proposed models.

Problem Formulation

For simplicity of notation and as usually done in ILP, 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. All terms in learning examples (hypotheses, respectively) are constants (variables) written in lower (upper) cases. For instance, E = {arc(a,b), arc(b,c), arc(c,a)} is an example and H = {arc(X,Y), arc(Y,Z)} is a hypothesis.

As usual in ILP, we use *θ-subsumption* (Plotkin 1970) 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 θ = {X/a, Y/b, Z/c} implies that H subsumes E (there are more such substitutions, for example {X/b, Y/c, Z/a}). The requirement that a negative example E^- is not subsumed by hypothesis H means that there does not exist any substitution θ such that $H\theta \subseteq E^-$. For example, hypothesis G = {arc(X,Y), arc(Y,X)} does not subsume the above example E.

To solve the template consistency problem given a template T, we look for a substitution σ making hypothesis $H = T\sigma$ 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. For generality, we assume that all variables in T are mutually different, that is, each variable occurs exactly once in T, as in $T = \{arc(X_1,X_2), arc(X_3,X_4)\}$. An exemplary hypothesis H may be obtained from this T by applying unification $X_2 = X_3$ (and then suitably renaming the variables). Clearly, if no unification is applied and the template consists only of the predicates in the example (arc/2 in our case) then the hypothesis subsumes that example. The reason for introducing unifications is thus to prevent H from subsuming negative examples. In our case, hypothesis obtained by applying unifications $X_2 = X_3$ and $X_1 = X_4$ to T does not subsume the above example E.

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 a unification of variables in the template that will make it subsume all positive examples and no negative example.

Constraint Satisfaction in a Nutshell

Constraint satisfaction is a technology for solving combinatorial (optimization) problems. A *constraint satisfaction problem* (CSP) is a triple (X, D, C), where X is a finite set of decision variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints (Dechter 2003). A constraint is a relation over a subset of variables (its *scope*) that restricts possible combinations of values to be assigned to the variables. Constraints can be expressed in extension using a set of compatible value tuples. Such constraints are also called *tabular constraints* and we will use them in the model for *θ-subsumption*. There also exist combinatorial constraints, where the semantics of the constraint defines the compatible value tuples. In this paper we will use two well known constraints *element* and *lex*. In constraint *element*(X,List,Y) X and Y are variables and L is list of variables (so it is a (*k*+2)-ary constraint, where *k* is the length of L). The semantics of *element*(X,List,Y) is as follows: Y equals to the X-th element of List. Constraint *lex*(M) is defined over a matrix M of variables (described as a list of rows, where row is a list of variables) and it ensures that rows of the matrix are lexicographically ordered. For example *lex*([[X_1,X_2], [X_3,X_4], [X_5,X_6]]) can be seen as abbreviation for $[X_1,X_2] < [X_3,X_4] < [X_5,X_6]$.

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. CSPs are usually solved by combination of inference techniques and search. The major inference technique is *(generalized) arc consistency* which ensures that each constraint is arc consistent. We say that a constraint is *arc consistent* if for any value in the domain of any variable in the scope of the constraint there exist values in the domains of the remaining variables in the constraint's 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 (Carlsson and Beldiceanu 2002).

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 constraints X = Y and $X \neq Y$ and 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. Typically, search interleaves with inference in the sense that after each search decision (posting a particular constraint such as X = 1 or $X \neq 1$) the problem is made arc consistent.

The critical part when applying constraint satisfaction techniques is formulating the problem as a CSP – so called *constraint modeling*. 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.

Base Constraint Model

Let us first describe the whole approach for finding the unifications in the template. We propose a constraint model describing how the variables are unified; let us call it a *unification model*. The reason, why we are proposing a unification model instead of simply posting an equality constraint each time two variables should be unified in the template is that we need to keep explicit information about which variables are unified for further reasoning.

To conduct a subsumption check, we use a constraint model motivated by Django, let us call it a *subsumption model*. The subsumption models for positive examples are connected to the unification model via so called channeling constraints. These constraints make a channel between the models as they ensure that any decision about unification

of variables is immediately propagated to all subsumption models (and vice versa) and can be discarded in case any subsumption model detects a failure (a given positive example is not subsumed).

The situation with negative examples is more complicated because negative examples cannot be subsumed which implies that there is no solution for the subsumption model for the negative example. This can be modeled by using a *Quantified CSP* (Bordeaux and Monfroy 2002), where nonexistence of valid instantiation of variables can be modeled. However, a Quantified CSP is not yet a part of mainstream constraint solvers so we decided for a different approach. Every solution of the subsumption model for the negative example is translated to unification of some variables in the template which breaks the solution (the unification of variables is in conflict with the subsumption substitution). We will explain later how this is realized.

The goal of unification model is to keep and propagate information about unification of variables in the template. Recall that each variable appears exactly once in the template so we can order the variables. Indices in the following example show this ordering $T = \{arc(X_1,X_2), arc(X_3,X_4), arc(X_5,X_6)\}$. Our model is based on the observation that if a set of variables is unified then we can select 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 $X_2 = X_3$ can be represented by mapping X_3 to X_2. The proposed constraint model uses index variable I_i for each variable X_i in the template to describe the mapping. The domain of I_i is $\{1,\ldots,i\}$ (variable X_i can only be mapped to itself or to some preceding variable). To express that variables X_i and X_j are unified we simply post a constraint $I_i = I_j$ (both variables are mapped to an identical variable). To ensure that each variable is mapped to the first variable in the set of unified variables we use a constraint $\forall i=1,\ldots,n$ *element*$(I_i, [I_1,.., I_n], I_i)$, where n is the total number of variables. In other words, if variable X_i is mapped to X_j ($I_i = j$) then X_j is not mapped to any preceding variable ($I_j = j$, i.e., $I_{I_i} = I_i$). For example, [1,1,2] is not a valid list of indices (it represents $X_1 = X_2$ and $X_2 = X_3$), the correct representation of this unification should be [1,1,1] ($X_1 = X_2$ and $X_1 = X_3$). The *element* constraints thus ensure that each set of unifications is represented by a single list of indices.

Notice that the same predicate symbol may appear several times in the template. For example, *arc* appears three times in above template T. We may assume that the structure and the size of the hypothesis should be preserved after unifying some variables (otherwise a shorter template was generated). In particular, we should ensure that no atom will disappear after unifying the variables. For example, index list [1,2,1,2,5,6] satisfies the element constraints and represents the following hypothesis $\{arc(X_1,X_2), arc(X_1,X_2), arc(X_5,X_6)\}$. However, it is actually hypothesis $\{arc(X_1,X_2), arc(X_5,X_6)\}$ because the first two atoms are identical. To remove this ambiguity we suggest using constraint *lex*(M) for each predicate symbol where matrix M consists of indices of variables in atoms of given predicate (one row per atom), in our example $M = [[I_1,I_2], [I_3,I_4], [I_5,I_6]]$. It means $[I_1,I_2] < [I_3,I_4] < [I_5,I_6]$ which forbids identical atoms in the hypothesis and also introduces ordering between the atoms that prevents some symmetrical solutions. For example $\{arc(X_1,X_2), arc(X_1,X_4), arc(X_5,X_6)\}$ is identical (after renaming the variables) to $\{arc(X_1,X_2), arc(X_3,X_4), arc(X_1,X_6)\}$, but the second hypothesis is not assumed thanks to *lex* constraint. We are aware about other permutation symmetries that appear in the model but are not forbidden by the *lex* constraint: $\{arc(X_1,X_2), arc(X_2,X_4), arc(X_4,X_6)\}$ is identical (after renaming the variables) to $\{arc(X_1,X_2), arc(X_3,X_4), arc(X_4,X_1)\}$, but both hypotheses are allowed in our constraint model. These symmetries appear because we model the set of atoms as a list of atoms. 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 variables) inherently includes the graph isomorphism problem which is one of problems belonging to NP neither known to be solvable in polynomial time nor NP-complete (Garey and Johnson 1979). The following example demonstrates the complete unification model:

template:	
$arc(X_1,X_2), arc(X_3,X_4), arc(X_5,X_6),$ $red(X_7), red(X_8), red(X_9),$ $green(X_{10})$	
unification model:	
variables	$I_1, \ldots, I_{10}$
domains	$D_i = \{1,\ldots,i\}\ \forall i=1,\ldots,10$
constraints	*element*$(I_i, [I_1,.., I_{10}], I_i)\ \forall i=1,\ldots,10$ *lex*$([[I_1,I_2], [I_3,I_4], [I_5,I_6]])$ *lex*$([[I_7], [I_8], [I_9]])$

Constraint Model for Subsumption Checks

In this section we describe how to model the subsumption check when the unification of variables is defined by the list of index variables $[I_1,.., I_n]$. To find out if hypothesis H subsumes example E, one needs to find substitution θ such that $H\theta \subseteq E$. As examples contain constants only, the substitution can be seen as instantiation of variables in the hypothesis. Taking in account that different examples may require different substitution (instantiation of variables) we should standardize apart the hypothesis before applying the substitution. In particular, for each example E_j, we plug a set $X_{j,1},\ldots, X_{j,n}$ of fresh variables into H, where n is the number of variables in the template. Then we unify these variables according to the index list $I_1,\ldots, I$, which can be done by constraints *element*$(I_i, [X_{j,1},.., X_{j,n}], X_{j,i})$ for each $i=1,\ldots,n$. These are exactly the channeling constraints between the unification and subsumption models.

The rest of subsumption check is realized using the idea proposed in (Maloberti and Sebag 2004). The main difference of our approach is in using k-ary constraints and standard constraint satisfaction techniques instead of binary constraints and ad-hoc implementation. The constraint model for each example looks as follows. First, for each predicate symbol *p* with arity *k* we collect all *k*-tuples of values from atoms of this predicate in the example. These value tuples define in extension a k-ary constraint c_p. Now, for each atom of predicate *p* with variables $\{Y_1,\ldots, Y_k\}$ in the hypothesis we post constraint c_p over these variables. Clearly, based on instantiation of variables $\{Y_1,\ldots, Y_k\}$ we can find an atom in the example to which a given atom from the hypothesis is mapped to. Let {arc(a,b), arc(b,c), arc(c,a)} be all atoms of predicate arc/2 in the example. Then binary constraint c_{arc} is defined in extension by a set of value pairs {(a,b), (b,c), (c,a)}. Atom arc(X,Y) from the hypothesis is represented by constraint $c_{arc}(X,Y)$ and instantiation X = a, Y = b means that that this atom is mapped to arc(a,b) in the example. In summary, any solution to a CSP defined by constraints c_p describes a substitution θ such that $H\theta \subseteq E$. The following example demonstrates the subsumption model (without channeling constraints, the variables are already unified).

example:	
arc(a,b), arc(b,c), arc(c,a), red(a), red(c)	
hypothesis:	
arc(Y_1,Y_2), arc(Y_2,Y_3), red(Y_2)	
subsumption model:	
variables	Y_1, Y_2, Y_3
domains	{a,b,c}
constraints	$c_{arc}(Y_1,Y_2)$, $c_{arc}(Y_2,Y_3)$, $c_{red}(Y_2)$
solutions	$\{Y_1 = c, Y_2 = a, Y_3 = b\}$ $\{Y_1 = b, Y_2 = c, Y_3 = a\}$

Integration and Search Strategy

Before going into extensions of the base constraint model let us describe briefly, how the unified variables are being found. First, we generate the unification model with the index variables. Then for each positive example, we generate a subsumption model with fresh set of variables and connect these variables to the index variables (see the previous section). It remains to include the negative examples. It is possible to use a naïve generate and test approach, where the subsumption of negative examples is checked after the unifications are decided. We applied a more advanced concept where the negative examples directly participate in the decisions about variable unifications as follows.

After the unification model and subsumption models for all positive examples are generated we take the first not-yet explored negative example (we simply take the examples in the order specified in the input data). For this example, we generate the subsumption model and connect it to the index variables exactly like we did it for positive examples. Then we instantiate the variables in this subsumption model for the negative example only. If no solution is found then the negative example cannot be subsumed so we remove the constraints and variables for this example and move to the next negative example. Assume now that we found an instantiation of variables satisfying all the constraints, that is, the negative example E^- is subsumed by the hypothesis H. It means that for hypothesis H containing variables $Y_1,\ldots, Y_n$, $H\theta \subseteq E^-$ turns out to hold. As the negative example is required not to be subsumed by H, we need to break substitution θ. This can be done by selecting a pair of variables Y_i and Y_j such that $Y_i\theta \neq Y_j\theta$ and forcing unification of these variables by adding a constraint $I_i = I_j$ and unifying corresponding variables in all subsumption models. Frequently, there are several such pairs so we introduce a choice point here. If the selected pair of variables for unification is found wrong later (that is, unifying the pair makes it impossible to subsume some positive example), we try another pair (we take the pairs of variables in the lexicographical order of their indices). To prevent trying the same pair of variables repeatedly for different substitutions subsuming negative examples, we post a constraint $I_i \neq I_j$ in the alternative search branch before we select another pair for unification. The above process is repeated until we break all possible substitutions for all negative examples (for each substitution such that $H\theta \subseteq E^-$ we need to find unification that breaks it).

In summary, the search procedure consists of three levels. First, we select the unification constraints by using the negative examples, then we instantiate the index variables which fixes the unifications, and finally we check subsumption of positive examples by instantiating variables in the corresponding subsumption models. Note also, that the underlying constraint solver maintains consistency of all posted constraints which helps in detecting some failures earlier in the search tree (it is called a look-ahead approach). The following pseudo-code summarizes this *base solving approach*.

1) Generate a unification model with index variables I
2) **For each** positive example *p* **do**
   a. Generate a subsumption model with fresh hypothesis variables X_p
   b. Connect hypothesis variables X_p to index variables
3) **For each** negative example *e* **do**
   a. Generate a subsumption model with fresh hypothesis variables Y_e
   b. Connect hypothesis variables Y_e to index variables
   c. **While** exists instantiation θ of hypothesis variables Y_e **do**
      i. Select variables $Y_{e,i}$ and $Y_{e,j}$ such that $Y_{e,i}\theta \neq Y_{e,j}\theta$
      ii. Introduce *choice point* $I_i = I_j$ or $I_i \neq I_j$
   d. Remove the variables Y_e with corresponding constraints
4) Instantiate index variables I
5) **For each** positive example *p* **do**
   a. Instantiate hypothesis variables X_p

Enhancements

The base solving approach guarantees finding a valid unification of variables, if it exists (otherwise it fails which is a proof that no such unification exists). Nevertheless, when experimenting, we identified some deficiencies that decrease overall performance. In particular, there were two types of deficiency – one in the search procedure and one in the constraint model. We will now describe how we resolved these deficiencies.

When instantiating the index variables (step 4) we may introduce other unifications than those necessary to break subsumption of negative examples! Assume that for each j ($j < i$) no pair of variables Y_j and Y_i was explored for negative examples (step 3ci). Then the search procedure can still instantiate index variable I_i to index $j < i$, which corresponds to unifying X_j and X_i in the hypothesis. Note that we are looking for a single solution so it is useless to explore unifications that are not forced by the negative examples. If the hypothesis with these additional unifications subsumes the positive examples then also the hypothesis without them subsumes the examples. Hence, we should avoid exploring these additional unifications during search in step (4). This can be done by substituting the search procedure in step (4) by the following procedure. We explore the set of index variables and for each index variable I_i such that i is in the current domain of I_i (note that domains were pruned by maintaining consistency) we set $I_i = i$. This approach completely avoids search in step (4) because all index variables will be instantiated either via the constraint $I_i = i$ or via propagation of unification constraints $I_i = I_j$. This is easy to prove by induction. I_1 is always set to 1 (the only value in its initial domain). Assuming the variables $I_1,.., I_{i-1}$ are instantiated then either there is no (even implied) constraint $I_j = I_i$ ($\forall j < i$), hence $i \in D_i$ and constraint $I_i = i$ is posted in step (4) or there is a constraint $I_j = I_i$ for some $j < i$ and the value of I_i is set to I_j (via propagation) which is already instantiated. Note that if the above instantiation violates some constraint $I_i \neq I_j$ posted in step (3cii) then there is no other instantiation satisfying all the constraints. If neither I_i nor I_j was set by the new constraint in step (4) then both I_i and I_j unify with some I_k ($k < i$, $k < j$) and there is no consistent instantiation of variables I_i, I_j, I_k. The other option is that I_i is set to i by the constraint $I_i = i$ posted in step (4) while I_j unifies with some I_k where $k < j$ (and vice versa). Let us take the smallest such k. If $k \neq i$ then $I_i \neq I_j$ holds ($I_j = k$). If $k = i$ then there is no consistent instantiation of variables I_i, I_j, I_k. Clearly, if I_i is set to i and I_j is set to j then $I_i \neq I_j$ holds.

The second group of deficiencies is hidden in constraint propagation that does not detect some trivial infeasibilities. We already discussed the conflict X = Y and X ≠ Y that cannot be discovered by arc consistency if the domains are not singleton. Because our constraint model for unification is based mainly on these two types of constraints, this deficiency may have significant impact on efficiency as the conflict is discovered late (typically in step (4)). Hence we propose a global inference procedure that can detect such conflicts immediately. Assume that we have n index variables. We propose using Boolean 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) $U_{j,i}$ and $U_{i,i} = 1$ (bounded). When constraint $I_i = I_j$ is posted in step (3cii) we simply unify rows i and j of the matrix U, that is, $\forall k\ U_{i,k} = U_{j,k}$ (and also $U_{k,i} = U_{k,j}$). In particular we obtain $U_{i,j} = U_{j,i} = 1$. Moreover, the matrix keeps a transitive closure of equality constraints between the index variables (if $U_{i,l} = 1$ for some l then after the unification $U_{j,l} = 1$ also holds and vice versa). When constraint $I_i \neq I_j$ is posted in step (3cii) 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 constraints $I_i = I_j$ and $I_i \neq I_j$). Similarly, if sometime later we deduce that $I_i = I_j$ either directly via posting this constraint in step (3cii) or indirectly via transitive closure of equality constraints then we can also deduce a failure. The Boolean matrix for variables $\{X_1, X_2, X_3, X_4, X_5, X_6\}$, where we decided that $X_1 \neq X_2$, $X_3 \neq X_4$, $X_5 \neq X_6$, $X_2 = X_3$ will look like this:

1	**0**	**0**	A	B	C
0	**1**	**1**	**0**	D	E
0	**1**	**1**	**0**	D	E
A	**0**	**0**	**1**	F	G
B	D	D	F	**1**	**0**
C	E	E	G	**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 $X_2 \neq X_5$ (D = 0) then we immediately get also $X_3 \neq X_5$.

It may seem that the above model completely overrides the unification model (if we also include the variables from the subsumption models in these unifications which we actually did to strengthen propagation of channeling constraints), but notice that the symmetry breaking *lex* constraints are not covered by matrix U and there is no propagation from the subsumption models to matrix U and hence co-existence of the unification model and matrix U might still be appropriate (it seems easier to express the symmetry breaking constraints using the index variables).

We can further exploit the unification matrix in the following way. When building the tabular constraints for the subsumption model, we may deduce that certain variables cannot be unified because if they are unified then the positive example is not subsumed. Assume the positive example {arc(a,b), arc(b,c), arc(c,a)} which defines the constraint with compatible pairs {(a,b), (b,c), (c,a)}. If this binary constraint between variables X and Y is made arc consistent then we obtain domains {a,b,c} for both variables. Hence if constraint X = Y is posted later then no conflict is deduced by making the problem arc consistent. To avoid such behavior, we can initially set $U_{i,j} = 0$ when some variables X_i and X_j are found different in any positive example (during pre-processing).

Experimental Results

To validate the proposed approach, we implemented the model in SICStus Prolog 4.0.8 and did some preliminary experiments with problems of identifying common structures in randomly generated structured graphs used as benchmarks in ILP. The experiments run on 2.53 GHz Core 2 Duo processor with 4 GB RAM under Windows 7. As we are not aware about another direct approach for solving the template ILP consistency problem, we present the comparison of various models that we proposed.

In the first experiment, we generated random structured graphs using the Erdős-Rényi model (1959). Each dataset consists of ten positive and ten negative examples (graphs). The graphs contain 20 nodes with density of arcs 0.2 and the template describes a sub-graph with 5 nodes. Table 1 shows the results, in particular, it describes the size of template (number of atoms and variables) and runtimes in milliseconds for base model, base model with improved search strategy, base model with improved constraint reasoning, and base model with both enhancements. Clearly, the enhancement of the constraint model contributed most to better efficiency, though the improved search strategy alone also brought a similar gain here.

#atoms	#vars	base	search	constr	search+constr
6	7	0	0	0	0
7	9	16	16	0	0
7	9	15	16	16	15
8	11	47	46	47	46
8	11	125	16	16	15
8	11	281	124	109	109
9	13	2044	1997	1684	1700
9	13	483	484	468	468
9	13	2075	141	140	140

Table 1. Comparison of runtimes (milliseconds) for identifying common structures in graphs (Erdős-Rényi).

To support further the observation that the proposed model enhancements contribute significantly to efficiency of the system, we did another experiment with random structured graphs based on Barabási-Réka model (1999). Again, each dataset consists of ten positive and ten negative examples representing the graphs. The graphs contain 20 nodes, where new nodes are attached to graph with 3 arcs. The template consists of 10 atoms and 15 variables for each dataset (a subgraph with 5 nodes). Table 2 shows the runtimes for various models. The contribution of enhanced constraint model is undoubted. The speedup seems to be between one and two orders of magnitude

base	search	constr	search+constr
69498	390	375	374
202786	107718	14087	13834
>600000	>600000	216063	213330
>600000	>600000	510777	491277

Table 2. Comparison of runtimes (milliseconds) for identifying common structures in graphs (Barabási-Réka).

Conclusions

The paper suggests using constraint satisfaction techniques for solving a template ILP consistency problem formulated in (Gottlob, Leone, and Scarcello 1999). 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 crucial importance of this latter problem in ILP. We propose a constraint model to describe unification of variables in the template and we connect this model with a constraint model for subsumption checks motivated by Django system. In additional to using some classical modeling techniques, such as symmetry breaking, we propose several enhancements motivated by the nature of the problem. The efficiency of models is demonstrated experimentally using problems of identifying common structures in randomly generated structured graphs.

Acknowledgements: The research is supported by the Czech Science Foundation under the project 201/08/0509.

References

Alphonse E. and Osmani A. 2009. Empirical Study of Relational Learning Algorithms in the Phase Transition Framework. *European Conference on Machine Learning – ECML,* Springer.

Barabási, A-L. and Réka, A, 1999. Emergence of scaling in random networks. *Science*, 286:509-512.

Bordeaux, L. and Monfroy, E. 2002. Beyond NP: Arc-consistency for quantified constraints. *Principles and Practice of Constraint Programming - CP 2002.* LNCS 2470, pp. 371–386, Springer Verlag.

Carlsson, M. and Beldiceanu, N. 2002. Arc-Consistency for a Chain of Lexicographic Ordering Constraints. *SICS Technical Report* T2002-18.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Džeroski, S.; Lavrač N. (Eds.) 2001. *Relational Data Mining*. Springer Verlag.

Erdős, P.; Rényi, A. 1959. On Random Graphs I. *Publicationes Mathematicae* 6: 290–297.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco.

Gottlob, G., Leone, N., and Scarcello, F. 1999. On the complexity of some inductive logic programming problems. *New Generation Computing*, 17, 53-75, Omsha.

Maloberti, J. and Sebag, M. 2004. Fast Theta-Subsumption with Constraint Satisfaction Algorithms. *Machine Learning*, 55, 137–174. Kluwer Academic Publishers.

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

Plotkin, G., 1970. A note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine Intelligence*, 5, 153–163. Edinburgh University Press.