On Learning Constraint Problems

Arnaud Lallouet, Matthieu Lopez, Lionel Martin, Christel Vrain

On Learning Constraint Problems

Arnaud Lallouet
GREYC, University of Caen

Matthieu Lopez, Lionel Martin, Christel Vrain
LIFO, University of Orléans

***Abstract*— It is well known that modeling with constraints networks require a fair expertise. Thus tools able to automatically generate such networks have gained a major interest. The major contribution of this paper is to set a new framework based on Inductive Logic Programming able to build a constraint model from solutions and non-solutions of related problems. The model is expressed in a middle-level modeling language. On this particular relational learning problem, traditional top-down search methods fall into blind search and bottom-up search methods produce too expensive coverage tests. Recent works in Inductive Logic Programming about phase transition and crossing plateau shows that no general solution can face all these difficulties. In this context, we have designed an algorithm combining the major qualities of these two types of search techniques. We present experimental results on some benchmarks ranging from puzzles to scheduling problems.**

I. Introduction

Constraint Programming (CP) is a very successful formalism to model and solve a wide range of decision problems, from arithmetic puzzles to timetabling and industrial scheduling problems. However, it has been recognized by the community [1] that modeling in CP requires expert knowledge to be achieved successfully. A major problem encountered by novice users is that they have a very limited knowledge on how to choose the variables, how to find the constraints and how to improve their model in order to make it efficient. In this process, the activity of finding the constraints to be stated is a crucial part and a lot of work has been spent on the understanding [2] and automation [3] of modeling tasks.

The problem we address in this paper is the automatic acquisition of a constraint model from examples and counter-examples of related problems. To our best knowledge, learning a constraint network has only been addressed by the system CONACQ in [4] and subsequent papers [3], [5] with a version-space algorithm. However, while efficient, a major limitation of this approach is that the user has to provide the exact set of variables as well as solutions and non-solutions for her very problem. It is questionable that the user still wants to build a model after having found some of its solutions. In contrast, and from a cognitive point of view, it is more likely that the user wants to model a new problem, having on the shelf a set of examples and counter-examples only for some related problems. To illustrate them, we can consider that the user wants to model school timetabling problem, only having solutions and non-solutions generated by hand to some historical instances from the past few years. These problems are almost the same but the number of teachers, groups, rooms may be different. Despite the generalization to an active learning framework [5], it is still required with CONACQ to provide judgements on potential solutions of the actual problem.

Some modeling languages like OPL [6], Essence' [7] or MiniZinc [8] provide an actual framework for modeling constraint problems at middle-level of abstraction. The user provides rules and parameters which are combined in a rewriting process to generate a Constraint Satisfaction Problem (CSP) adapted to the very problem to solve. Learning such a specification from already solved problems (historical data) would provide a model that could be reused in a new context with different parameters. For example, after the generation of the school timetabling problem, the model can be fed with current parameter data like the number of classes, the number of teachers, new available classrooms, etc.

In this paper, we present a way of acquiring such a constraint specification using Inductive Logic Programming (ILP). The examples and counter-examples for the concept to be learned are defined as interpretations in a logic language we call the *description language* and the output CSP is expressed by constraints in a *constraint language*. The specification is expressed by first-order rules which associate to a set of predicates in the description language (body of a rule) a set of constraints of the constraint language (head of a rule). We do not use directly a modeling language like Essence or Zinc to stay closer to a rule system but the rules we learn are at the *same* level of abstraction as the ones of intermediate modeling languages like Essence', Minizinc or OPL. In particular, they allow the use of arithmetics and parameters and they can be rewritten to generate constraint problems of different size. It happens that finding such rules is a genuine challenge for ILP techniques since the discovery process falls almost every time into ILP pathological cases of blind plateau search at phase transition [9].

The contributions of this paper are first setting the framework of learning CSP specifications, then the choice of the rule language and its rewriting into a CSP and the learning algorithm which allows to guide search when traditional methods fail. The learning algorithm is a slightly improved version of the one presented in [19].

The paper is organized as follows. We first give an informal overview of the framework (section II), introduce the rule language (section III), then we present the learning framework and give keys to understand and to implement our algorithm and describe how rules can be rewritten in a CSP. We present the results of different experimentations on classical constraint problems like timetabling, job-shop scheduling and the clas-

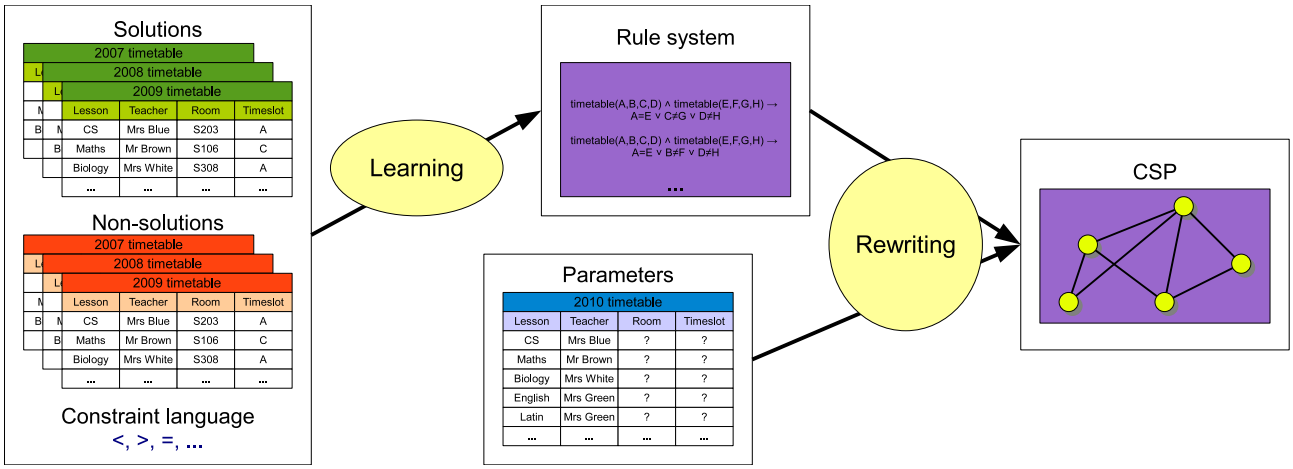

Fig. 1. Constraint Problem Learning workflow

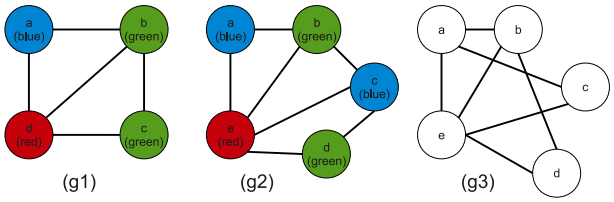

Fig. 2. (g1) wrong coloration, (g2) good coloration, (g3) to be colored

sical n-queens.

II. General presentation of the framework

In order to bridge the gap between constraint programming modeling language and ILP, we use a rather complex framework. In this section, we provide a quick overview of the framework, as well as some justifications of our choices as depicted in the workflow of Figure 1. To illustrate the concepts, we use the very simple example of graph coloring.

First, examples and counter-examples are each defined by a logical interpretation, which can be seen as a set of ground literals. For the graph coloring example, two graphs are depicted in Figure 2 one, called $g1$, with a wrong coloration and one, called $g2$, with a good one. For the first graph *g1*, its logical description is given by the interpretation { *nwc(g1), n(g1,a), n(g1,b), n(g1,c), n(g1,d), adj(g1,a,b), adj(g1,a,d), adj(g1,b,c), adj(g1,b,d), adj(g1,c,d), col(g1,a,blue), col(g1,b,green), col(g1,c,green), col(g1,d,red), $a \neq b$, ..., red $\neq$ blue, ...* } where the predicate *nwc* stand for non-well colored, *n* for node, *adj* for adjacent and *col* for color. We assume that all the pointwise difference constraints between constants ($a \neq b, \ldots$) are present in the description. We can give a similar description of the well-colored graph $g2$ using a predicate *wc*. Note that the examples and counter-examples may range on different sets of variables. This gives our framework an increased flexibility over the previous state-of-the-art system CONACQ [4]. To be able to infer a CSP from a problem, all variables and constants have types. The rule we aim to learn for graph coloring is:

$$col(G, X, A) \wedge col(G, Y, B) \wedge adj(G, X, Y) \rightarrow A \neq B$$

This rule describe *colorability* of a graph independently of the actual graph to be colored. We assume that each variable is universally quantified. We can further use this rule to build a CSP for coloring a specific graph, like the graph $g3$ of Figure 1. We assume that the user provides a description of the graph she wants a coloration of, by giving a similar but incomplete description of the graph: { *wc(g3), n(g3,a), n(g3,b), n(g3,c), n(g3,d), n(g3,e), col(g3,a,ColA), col(g3,b,ColB), col(g3,c,ColC), col(g3,d,ColD), col(g3,e,ColE), adj(a,b), adj(a,c), adj(a,e), adj(b,d), adj(b,e), adj(c,e), $a \neq b$, ..., red $\neq$ blue, ...* }. Then the left-hand side of the rule is matched against the specification and all possible substitutions are computed. For example, the substitution { *G/g3, X/ColA, Y/ColB* } allows to set the constraint *ColA* $\neq$ *ColB* of the CSP by applying the substitution to the right-hand side of the rule. By considering all possible substitutions allowed by the variable types, we obtain the CSP defining the colorability of $g3$.

Expressed as a clause, the above rule gives:

$$\neg col(G, X, A) \vee \neg col(G, Y, B) \vee \neg adj(G, X, Y) \vee A \neq B$$

A specification is composed of a conjunction of clauses like this one. It happens that most ILP systems rather learn DNF instead of CNF. Then we simply consider the negation of the rules to be learned, and exchange examples and counter-examples. The conjunction to be learned is as follows and describes a wrong coloration:

$$col(G, X, A) \wedge col(G, Y, B) \wedge adj(G, X, Y) \wedge A = B$$

In theory, we could simply give this problem at this step to a relational learning tool [13], [14], [20], [15], [16], [17] but in practice, all of them fail because of a too large or unstructured search space: either a top-down search falls into random plateau search or a bottom-up search faces too expensive coverage tests. To stay informal in this section, we can say that

our learning algorithm is based on a specific rule construction in which the search space is top-down recursively divided into zones. Each zone is then explored bottom up until a rule is found. Then, following a separate-and-conquer technique, the examples covered by the rules are discarded and the search for a new rule is launched until all examples are covered.

III. Modeling language

Let V be a set of variables and $D = (D_X)_{X \in V}$ their domains. A *constraint* is a relation c on a subset of the variables. We denote by $var(c)$ the variables on which c is defined and by $sol(c) \subseteq D^{var(c)}$ the set of tuples defining c. A CSP is a triple (V, D, C) in which V and D are defined as above and C is a set of constraints. A modeling language provides a way to specify a CSP in an abstract manner. A lot of existing modeling languages allow the use of high level variables like sets, functions, arithmetic operators, universal and existential quantifiers or arguments to parameterize their models [7], [10]. In practice, the parameters are usually provided in a separate file and mixed with the model to produce a CSP. As described previously, we aim at learning such an abstract model of the target problem. However, even if we would like to learn a specification in such languages, we think that dealing directly with them is too hard due to their excess of expressivity. In this paper, we propose to focus on a simpler language but still allowing to express a large number of problems. We choose a first-order logic language retaining the notion of parameter. We have discarded features like functions commonly found in human-targeted modeling languages but kept the crucial ability to be able to generate CSP for a set of instances of the problem.

A side contribution of this paper is to propose a rule-based intermediate modeling language suited for Machine Learning purpose. A Constraint Problem Specification (or CPS) in this language consists in a set of rules describing when a constraint should be posted in the CSP instance. Let T be a set of types. Let $V = (V_t)_{t \in T}$ and $(Const_t)_{t \in T}$ be respectively a set of typed variables and constants. A term is either a variable or a constant. Predicates also have types and are divided into two disjoint sets P_D and P_C corresponding respectively to the body and the head of a rule. Body predicates form the *description language*. They are used to express examples and counter-examples and to introduce the variables of the rules. They also have mode declaration: each argument has a mode describing if the argument is used as input or output. Input mode is denoted by $+$ while output mode is denoted by $-$. For example, the predicate *sum(X,Y,Z)* with semantics $X + Y = Z$ and mode *sum(+,+,-)* defines the last argument to be the computation of the sum of the two first ones. Head predicates form the *constraint language* and are used to define the constraints which hold when body predicates are true. These predicates are precursors of constraints and will be turned into constraints in the rewriting phase (see Figure 1 and section V). An *atom* is an expression $P(t_1, \ldots, t_k)$, where P is a k-ary predicate and $t_1, \ldots, t_k$ are terms.

The syntax of our rules is:

$$
\begin{array}{rcl}
\textit{rule} & ::= & \forall\ \textit{variables} :\ \textit{body}\ \rightarrow \textit{head} \\
\textit{variables} & ::= & \textit{vs} \in \texttt{TYPE} \mid \textit{variables}, \textit{variables} \\
\textit{vs} & ::= & \texttt{VARIABLE} \mid \textit{vs}, \textit{vs} \\
\textit{body} & ::= & \texttt{BODY\_ATOM} \mid \textit{body}\ \wedge\ \textit{body} \\
\textit{head} & ::= & \texttt{HEAD\_ATOM} \mid \neg\texttt{HEAD\_ATOM} \\
 & & \mid \textit{head}\ \vee\ \textit{head}
\end{array}
$$

Figure 3 presents some examples of problems specified in our language. Due to the lack of space, the universal quantification of the variables is omitted.

The first example corresponds to the well-known graph coloring problem where two neighbors must have different colors. The second one is a simplified school timetabling problem where $timetable(L, T, R, S)$ means that a lesson L is taught by a teacher T in the room R at time slot S. The first rule imposes two lessons not to be during the same time slot if they are in the same room. The second rule ensures that a teacher does not give two lessons during the same time slot. Finally, the last example, used in experiments, is a simplified job-shop problem where $schedule(J, T, B, E, M)$ means that a job J of type T is processed by machine M between the time points B and E. The first rule specifies that the beginning of a job must be before its end; the second one that two jobs cannot be performed at the same time by the same machine; the last one describes that some jobs must be done before others according to their types ($prev$ depicts the order on job types).

In contrast with classical intermediate modeling languages, the presence of disjunctions makes it more difficult to be understood by a human user. However, a model in this language is supposed to be automatically compiled into a CSP and in the simplified expressions, most disjunctions usually disappear.

IV. Learning process

The first step of our framework consists in learning a CPS describing the target problem. In this section, we start by presenting the Inductive Logic Programming framework and its application to our learning problem. Then, we focus our presentation on the main part: learning a rule. We explain the different strategies we have considered and show results obtained by the experiments.

A. Learning a CPS as an ILP problem

First, we present what the learning process needs as inputs. If we refer to Figure 1, the learning phase needs a set of solutions, for instance examples of correct timetable, a set of non-solutions, like examples of incorrect timetables, and background knowledge. This one contains a definition of each constraint predicate given either in extension or by Horn clauses. In addition, mode declarations are put into the background knowledge.

The goal is to learn in our rule language a definition of a CPS, denoted by CS, that correctly discriminates positive (solutions) from negative examples (non-solutions). The discriminative power of a rule is computed w.r.t. a covering

Graph coloring problem

$$n(X) \wedge n(Y) \wedge col(X, A) \wedge col(Y, B) \rightarrow A \neq B \ \vee \ \neg adj(X, Y)$$

Simplified school timetable

$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow L_1 = L_2 \ \vee \ R_1 \neq R_2 \ \vee \ S_1 \neq S_2$$
$$\wedge$$
$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow T_1 \neq T_2 \ \vee \ L_1 = L_2 \ \vee \ S_1 \neq S_2$$

Simplified jobshop

$$schedule(J, T, B, E, M) \rightarrow B < E$$
$$\wedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ M_1 \neq M_2 \ \vee \ B_1 > E_2 \ \vee \ E_1 < B_2$$
$$\wedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ E_1 < B_2 \ \vee \ prev(T_1, T_2)$$

N-queens problem

$$position(Q_1, L_1, C_1) \wedge position(Q_2, L_2, C_2) \rightarrow Q_1 = Q_2 \vee L_1 \neq L_2$$
$$position(Q_1, L_1, C_1) \wedge position(Q_2, L_2, C_2) \rightarrow Q_1 = Q_2 \vee C_1 \neq C_2$$
$$position(Q_1, L_1, C_1) \wedge position(Q_2, L_2, C_2) \wedge gap(L_1, L_2, I_1) \wedge gap(C_1, C_2, I_2) \rightarrow Q_1 = Q_2 \vee I_1 \neq I_2$$

Fig. 3. Some CPS examples: all variables are universally quantified

relation: informally, a definition C covers an example e with respect to a background knowledge $\mathcal{B}$ if e can be deduced from $\mathcal{B}$ and C. On the other hand, an example is rejected when it is not covered. The definition C is said to be *complete* if it covers all positive examples and *consistent* if it rejects all negative ones. To sum up, we can formalize our learning problem as follow: given two sets of examples E^+, E^-, and a background knowledge $\mathcal{B}$, find a definition CS such that:

- $\forall e^+ \in E^+ : e^+$ is covered by CS
- $\forall e^- \in E^- : e^-$ is rejected by CS

To illustrate the learning problem, we use the example of school timetabling problem. Let us notice that on example in the Figure 1, is usually composed of one or more tables (e.g. for graph coloring, there are two tables : one for color and another for adjacent). For timetabling, we only need one table giving the features of each lesson. This table can be viewed as the 4-ary relation $timetable$, the arguments of which are the lesson name, the teacher, the room and the time slot. A positive example of a simple timetable could be:

$e^+ : \{timetable(CS, MrsBlue, s203, A),$
$timetable(Maths, MrsBrown, s106, C)$
$timetable(Biology, MrsWhite, s308, A),$
$timetable(Maths, MrsBrown, s106, B)\}$

whereas a negative example could be:

$e^- : \{timetable(CS, MrsBlue, s203, A),$
$timetable(Maths, MrsBrown, s106, C)$
$timetable(Biology, MrsWhite, s203, A),$
$timetable(Maths, MrsBrown, s105, C)\}$

Note that the counter-example e^- contains two errors. Background knowledge contains useful predicates like equalities to compare lessons, teachers, rooms and time slots.

A set of rules defining a CPS is equivalent to a Conjunctive Normal Form (CNF). However, the majority of existing ILP systems handle Disjunctive Normal Form (DNF) or equivalent. Passing from CNF to DNF is a simple operation (see [11], section 3.4.3) consisting in searching a definition for the negation of the target concept.

Considering the timetable example, the CNF corresponding to the CPS from fig. 3 is:

$$\forall \ L_1, L_2 \in Lessons, T_1, T_2 \in Teachers, \ldots : \neg timetable(L_1, T_1, R_1, S_1) \vee \neg timetable(L_2, T_2, R_2, S_2) \vee L_1 = L_2 \ \vee \ R_1 \neq R_2 \ \vee \ S_1 \neq S_2$$
$$\bigwedge$$
$$\forall \ L_1, L_2, \ldots : \neg timetable(L_1, T_1, R_1, S_1) \vee \neg timetable(L_2, T_2, R_2, S_2) \vee T_1 \neq T_2 \ \vee \ L_1 = L_2 \ \vee \ S_1 \neq S_2$$

The negation produces the DNF:

$$\exists \ L_1, L_2 \in Lessons, T_1, T_2 \in Teachers, \ldots : timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \wedge L_1 \neq L_2 \ \wedge \ R_1 = R_2 \ \wedge \ S_1 = S_2$$
$$\bigvee$$
$$\exists \ L_1, L_2, \ldots : timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \wedge T_1 = T_2 \ \wedge \ L_1 \neq L_2 \ \wedge \ S_1 = S_2$$

To learn this concept, the sets of positive and negative examples must be inverted. Positive examples become negative and vice versa. In the sequel of this section, we only focus on learning DNF.

B. Background in ILP

The state-of-the-art framework of ILP consists in learning a definition composed of rules. Separate-and-conquer is a heavily used family of algorithms in ILP (the interested reader can refer to [12] for a complete state-of-the-art). The algorithm iterates a single rule learning algorithm until positive examples are correctly discriminated from all negative ones. Since we use separate-and-conquer, this section only focuses on the single rule learning step.

First, we introduce the search space of ILP. A *search state* is represented by a conjunction of literals representing a rule with respect to background knowledge. To evaluate the interest of a state, we can consider different criteria such as the length of the rule or its coverage score over positive and negative

examples. An example e is *covered* by a rule r if there exists a substitution σ such that $\sigma(r) \subseteq e$.

The search space can be organized as a lattice. In this paper, we consider the following search lattice bounded by two rules denoted $\top$ (top) and $\bot$ (bottom) and ordered by the inclusion relation. A clause c_1 is included in an other one c_2 if all the literals of c_1 appear in c_2, up to variable renaming. In this case, we say that c_1 is *more general* than c_2 and, similarly, that c_2 is *more specific* than c_1.

The $\top$ clause is the most general clause of the lattice. It covers the maximum number of (positive and negative) examples. The empty clause which covers all the examples and counter-examples can be considered as a $\top$ clause. But in order to reduce the search space, many algorithms choose as $\top$ clause a more specialized one.

The bottom clause is the most specific one and should reject most examples. In certain algorithms like FOIL [13], this bound is theoretical since $\bot$ is an infinite clause (and therefore we have an infinite search space) because new variables can be introduced at each step. This is why a bottom limit is chosen by saturation of an example called a *seed* (see [14] for complete explanation). Given a positive seed example s, its saturation $sat(s)$ consists in s augmented of all entailed background knowledge literals. To obtain this limit, modes are used and new variables corresponding to output modes may be introduced. This set can be parameterized by a certain limit to obtain a finite set and so a finite search space. Typically, we can limit the number of new variables introduced under a limit k.

To explore the search space, there exists different strategies like *top-down* (See Figure 4), which starts from the hypothesis $\top$ and progressively specializes the hypothesis by adding new literals, or *bottom-up*, which starts from $\bot$ and generalizes the hypothesis by removing some literals. These two strategies consist in a sequence of operations, called *refinements*, the goal of which is to specialize or generalize the current hypothesis. Each refinement produces a new hypothesis. For a hypothesis, there exists generally several possible refinements due to the lattice structure. To choose the best one, algorithms use a *heuristic function* based on a coverage score and/or the length of the rule.

Let us illustrate this with an example. A very simple refinement operator, in a top-down search, could be an operation selecting a literal in the $\bot$ clause and adding it to the hypothesis. Then, the number of possible refinements would be equal to the length of $\bot$. To select a literal, we can choose one with the best *purity* heuristic value defined by $\frac{p}{p+n}$ where p and n correspond to the coverage score over positive and negative examples.

We have presented the rule learning process with a hill-climbing strategy but many versions exist with, for instance, beam search or A^* methods. *Top-down* are illustrated by algorithms like FOIL [13], Progol [14], ICL [15], Beth [16] and Propal [17]. But these algorithms fail on our CSP benchmarks(see section IV-D). Recent works on phase transition problems in ILP [18] and blind search or crossing "plateau" [9] indicate that searching a solution may fall into a very difficult zone and CPS learning clearly belongs to this kind of problems. *Bottom-up* approaches are not adapted because of the expensive cost of coverage test at the beginning of the search. To take advantages of the structure of CPS, we have developed a new algorithm [19] based on a bidirectional search. We describe it in the next section.

C. Bidirectional search

We have designed a new algorithm taking advantages of top-down and bottom-up approaches. Our algorithm is a bidirectional search where each refinement step is characterized by a couple of hypothesis $(H^i_\top, H^i_\bot)$ In this couple, ($H^i_\top$ and $H^i_\bot$) are two rules where $H^i_\top$ is a specialization of $\top$ and $H^i_\bot$ a generalization of $\bot$ (see fig. 4).

The search starts with $H^0_\top$ reduced to an empty rule, and $H^0_\bot$ obtained by saturation of a positive uncovered example. The main characteristics of the approach are:

- $H^{i+1}_\top$ is obtained from $H^i_\top$ by adding a set of literals. The candidate sets are selected from $H^i_\bot$ w.r.t. the analysis of layers in the initial saturation $H^0_\bot$. We start by searching singletons and the size of candidate sets is increased until a satisfying candidate is found. Each candidate set S is associated to a candidate clause obtained by adding S to the body of $H^i_\top$.
- Candidate clauses are not evaluated w.r.t. their discriminating power but w.r.t. the discriminating power of their saturation. This choice reduces the phenomenon of blind search.
- We restrict the search to candidate sets so that if $H^0_\bot$ contains a discriminating clause, then this clause can be found by our bidirectional search.
- Once a candidate clause $H^{i+1}_\top$ is selected, $H^{i+1}_\bot$ is obtained by a saturation of $H^{i+1}_\top$ where only literals from $H^0_\bot$ are added.

This process is repeated until $H^i_\top = H^i_\bot$. Each step produces a refinement of the previous top clause, but coverage tests are less expensive than in usual bottom-up search. At the same time, it produces a generalization of the bottom clause which both allows to reduce the search space and is used for evaluating and choosing among refinements (to be more accurate than top-down searches). We have obtained with this technique very interesting and encouraging results compared to other tested approaches. The detail of the algorithm can be found in [19].

D. Experiments and discussion

To evaluate different existing strategies, we have generated several benchmarks for the examples specified in Figure 3. In doing so, we have generated a set of solutions and non-solutions. To produce solutions, we have chosen a random size for each problem (*e.g.* for the graph coloring problem, the number of vertices and colors), and then solved the CSP. For negative ones, we have proceeded in a similar way but constraints have been relaxed and we have checked that there is at least one unsatisfied constraint.

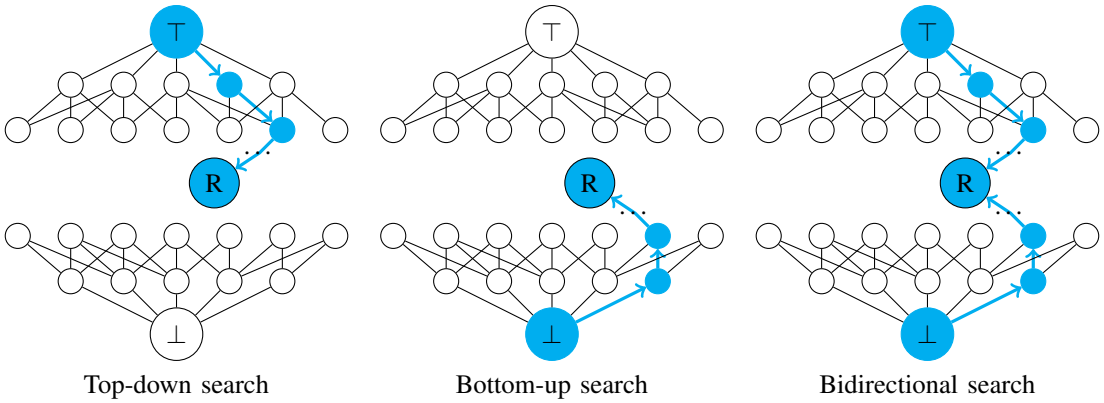

Fig. 4. Search strategies

We have tested most usual ILP algorithms. Results of experiments are detailed in Figure 5. To compute the accuracy of a learned CPS (third column), we have generated new examples and computed the ratio of examples correctly discriminated by the learned rules (covered for solutions and rejected for non-solutions). We have concentrated our effort on recognized top-down approaches. Bottom-up approaches have been totally ineffective due to the large size of $\bot$ rules and the expensive coverage test. For top-down searches, results are more interesting but not enough to be acceptable. Only three top-down approaches have given interesting results: Propal and two configurations of Aleph [20][1]. Propal is based on the data driven strategy (a way to reject certain refinements). The first configuration for Aleph, called Aleph1, is a breadth first search with a maximum of 200000 visited search states and an infinite open list. The second, named Aleph2, only differs from the first one by the search strategy in which the breath-first search has been replaced by a heuristic search. The more complex the target concept is, the more Propal and Aleph2 find an incorrect theory. For the n-queens problem, Propal has been stopped after ten hours. These results show the main limitation of *top-down* approaches when they face to plateau phenomena [9]. Aleph1 corresponds to a quasi complete search which may succeeds but with an important computation time (depending on the problems).

In contrast, our bidirectional method has succeeded with all benchmarks: it finds accurate definition in a short amount of time. But even if the meaning is identical, the learned rules are not always the expected ones. For example, for n-queens, the learned constraint for column is (*gap* is a predicate expressing the difference of two integers):

$$position(Q1, X1, Y1) \wedge\ position(Q2, X2, Y2)$$
$$\wedge\ gap(Y1, Y2, V1) \wedge\ gap(Y2, Y2, V2)$$
$$\rightarrow\ Q1 = Q2 \vee\ V2 \neq V1$$

Our prototype, as well as the benchmarks, can be obtained by sending a mail to the authors. This method, built as shown in [19], works very efficiently with CSP benchmark. It relies on a stratified representation of the examples, which unfortunately is not satisfied on all kind of learning problems.

[1]Aleph is a general system allowing the emulation of several other ILP systems

V. Translation of CPS to CSP

Let us consider as example the school timetabling problem. We assume the user has obtained a CPS during the learning phase. To produce an actual CSP, she needs to provide a partially completed table representing the predicate $timetable$ (fig. 1). In general, there may be several tables, called *partial extension*. They set the parameters of the CPS. The objective, for the user, is to obtain a CSP which is able, once solved, to complete partial extensions. In Figure 1, the problem is to determine rooms and time slots where teachers could do their lessons. The user must give domains corresponding to rooms and time slots in order to obtain the CSP model corresponding to her problem. In particular, there may be new teachers, a different number of groups. These data are very natural to provide given an actual problem to solve.

A *partial extension* of a predicate p is a pair (p, E), where E is a set of tuples $\langle x_1, x_2, \ldots, x_k \rangle$ defining this predicate, and x_i is either a constant or ?, the latter meaning that the user does not know the value of the attribute. We denote by ext the set of partial extensions given by the user. The translation from CPS to CSP is composed of two steps. First, partial extensions are completed with CSP variables which are set to the corresponding domains. Second, for each rule of the CPS, all possible substitutions of the body are produced and the corresponding constraints (the head of the rule) are posted. Algorithm V computes these two steps.

The first step (line 3) consists in completing ext, replacing all ? by CSP variables with the right domain (given by the user). The second step consists in producing constraints from CPS rules with respect to the user instance. In doing so, we generate all possible substitutions of the body G allowing to satisfy thee body. Given a substitution σ, the body is satisfied if each atom $p(t_1, \ldots, t_k)$ of G is satisfied with σ. An atom $p(t_1, \ldots, t_k)$ is satisfied if $\sigma(p(t_1, \ldots, t_k))$ has a support in ext or, in the case where p is intensionally defined, if $\sigma(p(t_1, \ldots, t_k))$ is valid with respect to the definition. A problem exists when p is intensional. When there are CSP

benchmark	Propal			algorithm from [19]		
	# learned rules	time (s)	acc.	# learned rules	time (s)	acc.
Graph coloring	1	0	100%	1	0.17	100%
School timetable	3	11	98,33%	2	0.69	100%
Job-shop	6	103	87,78%	5	7.37	100%
N-queens	-	-	-	3	29.11	100%
	Aleph1			Aleph2		
Graph coloring	1	0.24	100%	1	0.14	100%
School timetable	1	1.24	100%	1	0.31	100%
Job-shop	3	1051.03	100%	6	1130.88	96%
N-queens	3	489.49	100%	3	4583.84	61.67%

Fig. 5. Experiments with different learning strategies

variables among positions with input mode of p, we generate auxiliary CSP variables for the output. For instance, considering the following atom $sum(X, Y, Z)$ and the substitution $\{X/2, Y/v_1, Z/?\}$ where the domain of v_1 is $[2..6]$. The substitution is completed with Z/v_2 and the domain of v_2 is $[4..8]$. When all the substitutions are computed, the algorithm substitutes the head of the rule to produce the constraints (line 21). These constraints are disjunctions of constraints. However, if we consider the example of school timetabling problem, many variables of the constraints have already a value allowing to simplify the constraint. For instance, consider the substitution $\{L_1/Latin,\ T_1/Mrs\ Green,\ R_1/v_1,\ S_1/v_2,\ L_2/English,\ T_2/Mrs\ Green,\ R_2/v_3,\ S_2/v_4\}$ of the second rule. The computed constraint will be $Mr\ Green \neq Mrs\ Green \vee Latin = English \vee v_2 \neq v_4$. It can be simplified in $v_2 \neq v_4$. Constraint cannot always be simplified; for instance, if the substitution is applied to the first rule, the disjunction remains.

Algorithm : TRANSLATE$(CS, ext, domains)$

```
1.  //Complete the partial extension
2.  // with CSP variables with domains
3.  ext ← COMPLETE(ext, domains)
4.  //Initialize the variables set
5.  //with these in ext
6.  vars ← GETVAR(ext)
7.  constraints ← ∅
8.  for each G → C ∈ CS
9.      // Generate all substitution of the
body
10.     subst ← GENERATEALLSUBST(G, ext)
11.     for each σ ∈ subst
12. //If there are atoms with no matching
13. // in ext,it adds aux. variables
14. // and the constraint
15.       for each atoms p(t1, ..., tk) ∈ G
16.       such that (p, _) ∉ ext
17.         vars.add(GETVAR(σ(p(t1, ..., tk))))
18.         constraints.add(σ(p(t1, ..., tk)))
19.  //It adds the constraint
20.  //corresponding to the head
21.       constraint.add(σ(C))
22.  //Finally, it returns the CSP
23.  return CSP(vars, domains, constraints)
```

Fig. 6. Translation of a CPS in CSP

VI. CONCLUSION

The motivation of our work is to avoid some of the limitations encountered with systems like CONACQ[3]. To our best knowledge, CONACQ and our system are the only works concerning the acquisition of CSP. In [21], Bessière et al. propose a method to automatically generate viewpoints from examples. However, no method is given to generate constraints on these variables. More generally, several works exist about reformulation of models, including the discovery of implied or redundant constraints [22], [23]. In this case, the learning task consists in learning only from positive examples since the discrimination of solutions and non-solutions are already made by the simple model. This is complementary to our approach since our learned models would gain to be reformulated into more efficient ones.

In this paper, we have presented a framework to obtain automatically an abstract model of a CSP. Our approach, based on Inductive Logic Programming, received examples of what the user considers as solutions and non-solutions of *related* problems. Then, she gets a specification (a CPS) which can be further translated into a CSP by adding data from her very problem. Even with a specification expressed in first order logic, and which can potentially take advantage of a lot of systems designed to learn such definitions, this learning problem has proved to be very difficult. It falls into well known limitations of ILP: blind search with plateau phenomena for top-down searches and too expensive tests for bottom-up search. In order to succeed in our learning task, we have designed a new algorithm taking advantages of these two strategies. This algorithm present a very good behavior on CSP benchmarks. The final step of our framework consists in a translation of the CPS and the data of the user's very problem into a CSP. This work constitutes an encouraging step towards the automatic acquisition of a CSP. Obviously, the CPS language has to be improved to handle larger CSP problem classes. An easy way would be to add, similarly to the "for all" block presented in this paper, an existential block. The method should be the same. But an important improvement would be the addition of aggregates (e.g. the sum of a list of variables). However, each improvement has a cost which needs to be taken in account to preserve the system efficiency.

REFERENCES

[1] J.-F. Puget, "Constraint programming next challenge : Simplicity of use," in *International Conference on Constraint Programming*, ser. LNCS, M. Wallace, Ed., vol. 3258. Toronto, CA: Springer, 2004, pp. 5–8, invited paper.

[2] B. M. Smith, "Modelling," in *Handbook of Constraint Programming*, T. W. F. Rossi, P. van Beek, Ed. Elsevier, 2006, ch. 11, pp. 377–406.

[3] C. Bessière, R. Coletta, F. Koriche, and B. O'Sullivan, "Acquiring constraint networks using a sat-based version space algorithm," in *AAAI*. AAAI Press, 2006.

[4] R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton, "Semi-automatic modeling by constraint acquisition," in *CP*, ser. Lecture Notes in Computer Science, F. Rossi, Ed., vol. 2833. Springer, 2003, pp. 812–816.

[5] C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin, "Query-driven constraint acquisition," in *IJCAI*, M. M. Veloso, Ed., 2007, pp. 50–55.

[6] P. V. Hentenryck, *The OPL optimization programming language*. Cambridge, MA, USA: MIT Press, 1999.

[7] A. M. Frisch, M. Grum, C. Jefferson, B. M. Hernández, and I. Miguel, "The design of essence: A constraint language for specifying combinatorial problems," in *IJCAI*, M. M. Veloso, Ed., 2007, pp. 80–87.

[8] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *CP*, 2007, pp. 529–543.

[9] É. Alphonse and A. Osmani, "On the connection between the phase transition of the covering test and the learning success rate in ilp," *Machine Learning*, vol. 70, no. 2-3, pp. 135–150, 2008.

[10] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace, "The design of the zinc modelling language," *Constraints*, vol. 13, no. 3, pp. 229–267, 2008.

[11] W. V. Laer, "From propositional to first order logic in machine learning and data mining," Ph.D. dissertation, Katholieke Universiteit Leuven, June 2002. [Online]. Available: http://www.cs.kuleuven.ac.be/ wimv/PhD/

[12] J. Furnkranz, "Separate-and-conquer rule learning," *Artificial Intelligence Review*, vol. 13, pp. 3–54, 1999.

[13] J. R. Quinlan and R. M. Cameron-Jones, "Foil: A midterm report." in *ECML*, ser. Lecture Notes in Computer Science, P. Brazdil, Ed., vol. 667. Springer, 1993, pp. 3–20.

[14] S. Muggleton, "Inverse entailment and progol," *New Generation Computing, Special issue on Inductive Logic Programming*, vol. 13, no. 3-4, pp. 245–286, 1995. [Online]. Available: http://citeseer.ist.psu.edu/muggleton95inverse.html

[15] L. D. Raedt and W. V. Laer, "Inductive constraint logic," in *ALT*, 1995, pp. 80–94.

[16] L. P. R. Tang, "Integrating top-down and bottom-up approaches in inductive logic programming: applications in natural language processing and relational data mining," Ph.D. dissertation, Department of Computer Sciences, University of Texas, 2003, supervisor-Mooney, Raymond J.

[17] É. Alphonse and C. Rouveirol, "Extension of the top-down data-driven strategy to ilp," in *ILP*, ser. Lecture Notes in Computer Science, S. Muggleton, R. P. Otero, and A. Tamaddoni-Nezhad, Eds., vol. 4455. Springer, 2006, pp. 49–63.

[18] A. Serra, A. Giordana, and L. Saitta, "Learning on the phase transition edge," in *IJCAI*, 2001, pp. 921–926.

[19] M. Lopez, L. Martin, and C. Vrain, "Learning discriminant rules as a minimal saturation search," in *ILP*, 2010.

[20] A. Srinivasan, *A learning engine for proposing hypotheses (Aleph)*.

[21] C. Bessiere, J. Quinqueton, and G. Raymond, "Mining historical data to build constraint viewpoints," in *Proceedings CP'06 Workshop on Modelling and Reformulation*, 2006, pp. 1–16.

[22] J. Charnley, S. Colton, and I. Miguel, "Automatic generation of implied constraints," in *ECAI*, 2006, pp. 73–77.

[23] C. Bessière, R. Coletta, and T. Petit, "Learning implied global constraints," in *IJCAI*, 2007, pp. 44–49.

[24] M. M. Veloso, Ed., *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.