Acquiring Constraint Networks using a SAT-based Version Space Algorithm

Christian Bessière, Frédéric Koriche, Remi Coletta, Barry O'Sullivan

Acquiring Constraint Networks using a SAT-based Version Space Algorithm

Christian Bessiere, Remi Coletta, Frédéric Koriche
LIRMM (CNRS/University of Montpellier)
161 rue Ada, 34392 Montpellier, France
{bessiere|coletta|koriche}@lirmm.fr

Barry O'Sullivan
Cork Constraint Computation Centre
University College Cork, Ireland
b.osullivan@4c.ucc.ie

Abstract

Constraint programming is a commonly used technology for solving complex combinatorial problems. However, users of this technology need significant expertise in order to model their problems appropriately. We propose a basis for addressing this problem: a new SAT-based version space algorithm for acquiring constraint networks from examples of solutions and non-solutions of a target problem. An important advantage of the algorithm is the ease with which domain-specific knowledge can be exploited.

Introduction

Constraint programming provides a powerful paradigm for solving combinatorial problems. However, modelling a combinatorial problem in the constraints formalism requires significant expertise in constraint programming. One of the reasons for this is that, for any problem at hand, different models of this problem are possible, and two distinct constraint networks for it can critically differ on performance. An expert in constraint programming typically knows how to decompose the problem into a set of constraints for which very efficient propagation algorithms have been developed. Such a level of background knowledge precludes novices from being able to use constraint networks on complex problems without the help of an expert. Consequently, this has a negative effect on the uptake of constraint technology in the real world by non-experts.

To overcome this problem we envision the possibility of *acquiring* a constraint network from a set of examples and a library of constraints. The constraint acquisition process is regarded as an interaction between a user and a learner. The user has a combinatorial problem in mind, but does not know how this problem can be modelled as an efficient constraint network. Yet, the user has at her disposal a set of solutions (positive examples) and non-solutions (negative examples) for this problem. For its part, the learner has at its disposal a library of constraints for which efficient propagation algorithms are known. The goal for the learner is to induce a constraint network that uses combinations of constraints defined from the library and that is consistent with the solutions and non-solutions provided by the user.

In this paper we summarise our contributions in this area (Bessiere *et al.* 2004; 2005). Our main contribution is a SAT-based algorithm, named CONACQ (for CONstraint ACQuisition), that is capable of learning a constraint network from a set of examples and a library of constraints. The algorithm is based on the paradigm of version space learning (Mitchell 1982; Hirsh, Mishra, & Pitt 2004). In the context of constraint acquisition, a version space can be regarded as the set of all constraint networks defined from a given library that are consistent with the received examples. The key idea underlying the CONACQ algorithm is to consider version space learning as a satisfiability problem. Namely, any example is encoded as a set of clauses using as atoms the constraint vocabulary defined from the library, and any model of the resulting satisfiability problem captures a constraint network consistent with the corresponding acquisition problem.

This approach has a number of distinct advantages. Firstly and most importantly, the formulation is generic, so we can use any SAT solver as a basis for version space learning. Secondly, we can exploit powerful SAT concepts such as unit propagation and backbone detection (Monasson *et al.* 1999) to improve learning rate. Finally, we can easily incorporate domain-specific knowledge in constraint programming to improve the quality of the acquired network. Specifically, we describe two generic techniques for handling redundant constraints in constraint acquisition. The first is based on the notion of *redundancy rules*, which can deal with some, but not all, forms of redundancy. The second technique, based on *backbone detection*, is more powerful.

The Constraint Acquisition Problem

A constraint network consists of a set of variables X, a set of domain values D, and a set of constraints. We assume that the set of variables and the set of domain values are finite, pre-fixed and known to the learner. This vocabulary is, thus, part of the common knowledge shared between the learner and the user. We implicitly assume that every variable in X uses the same set D of domain values, but this condition can be relaxed in a straightforward way.

The learner has at its disposal a constraint library from which it can build and compose constraints. The problem is to find an appropriate combination of constraints that is consistent with the examples provided by the user. For sake of

clarity, we shall assume that every constraint defined from the library is binary. However, we claim that the results presented here can be easily extended to constraints of higher arity. A *binary constraint* is a tuple $\mathsf{c} = (var(\mathsf{c}), rel(\mathsf{c}))$ where $var(\mathsf{c})$ is a pair of variables in X, known as the *scope* of the constraint, and $rel(\mathsf{c})$ is a binary *relation* defined on D. With a slight abuse of notation, we shall often use c_{ij} to refer to the constraint with relation $rel(\mathsf{c})$ defined on the scope (x_i, x_j). For example, $\leq_{12}$ denotes the constraint specified on (x_1, x_2) with relation "less than or equal to".

A *constraint library* is a collection B of binary constraints. From a constraint programming point of view, any library B is a set of constraints for which (efficient) propagation algorithms are known. A constraint network is *admissible* for some library if each constraint in the network is defined as the intersection of a set of constraints from the library.

An *example* is a map e that assigns to each variable x in X a domain value $e(x)$ in D. An example e *satisfies* a binary constraint c_{ij} if the pair $(e(x_i), e(x_j))$ is an element of c_{ij}. If e satisfies every constraint in C then e is called a *solution* of C; otherwise, e is called a *non-solution* of C. In the following, $sol(\mathsf{C})$ denotes the set of solutions of C.

Finally, a *training set* consists of a pair (E^+, E^-) of sets of examples. Elements of E^+ are called *positive* examples and elements of E^- are called *negative* examples. A constraint network C is said to be *consistent* with a training set (E^+, E^-) if every example in E^+ is a solution of C and every example in E^- is a non-solution of C.

Definition 1 (Constraint Acquisition Problem) *Given a constraint library* B *and a training set* (E^+, E^-)*, the* Constraint Acquisition Problem *is to find a constraint network* C *admissible for the library* B *and consistent with the training set* (E^+, E^-).

Example 1 (Fundamentals) *Consider the vocabulary defined by the set* $X = \{x_1, x_2, x_3\}$ *and the set* $D = \{1, 2, 3, 4, 5\}$. *In the following, the symbols* $\top$ *and* $\bot$ *refer to the total relation and the empty relation over* D*, respectively. Let* B *be the constraint library defined as follows:* $\mathsf{B} = \{\top_{12}, \leq_{12}, \neq_{12}, \geq_{12}, \top_{23}, \leq_{23}, \neq_{23}, \geq_{23}\}$. *Note that the constraints* $=_{12}$*,* $<_{12}$*,* $>_{12}$*,* $\bot_{12}$ *and* $=_{23}$*,* $<_{23}$*,* $>_{23}$*,* $\bot_{23}$ *can be derived from the intersection closure of* B. *Consider the two following networks* $\mathsf{C}_1 = \{\leq_{12} \cap \geq_{12}, \leq_{23} \cap \neq_{23}\}$ *and* $\mathsf{C}_2 = \{\leq_{12} \cap \geq_{12}, \leq_{23} \cap \geq_{23}\}$. *Each network is admissible for* B. *Finally, consider the training set* $E = (\{e_1^+\}, \{e_2^-, e_3^-\})$ *with* $e_1^+ = ((x_1, 2), (x_2, 2), (x_3, 5))$, $e_2^- = ((x_1, 1), (x_2, 3), (x_3, 3))$, *and* $e_3^-((x_1, 1), (x_2, 1), (x_3, 1))$. e_1^+ *is positive and the others are negative. We can easily observe that* C_1 *is consistent with* E*, while* C_2 *is inconsistent with* E. ▲

The Core Result

The CONACQ Algorithm

We have proposed a SAT-based algorithm for acquiring constraint networks based on version spaces. Informally, the version space of a constraint acquisition problem is the set of all constraint networks that are admissible for the given library and that are consistent with the given training set. In the SAT-based framework this version space is encoded in a clausal theory K, and each model of the theory represents a candidate constraint network. Given a constraint b_{ij} in B, the literal b_{ij} in K stands for the presence of the constraint in the acquired network. Notice that $\neg\mathsf{b}_{ij}$ is *not* a constraint: it merely captures the absence of b_{ij} from the acquired network. A clause is a disjunction of literals, and the clausal theory K is a conjunction of clauses.

The CONACQ algorithm provides the theory K, which is an implicit representation of the version space $V_{\mathsf{B}}(E^+, E^-)$ for the constraint acquisition problem on library B and training set (E^+, E^-). This representation allows the learner to perform several useful operations in polynomial time.

$V_{\mathsf{B}}(E^+, E^-)$ has *collapsed* if it is empty: there is no constraint network C admissible for B such that C is consistent with the training set (E^+, E^-). The *membership* test involves checking whether or not a constraint network belongs to $V_{\mathsf{B}}(E^+, E^-)$. The *update* operation involves computing a new version space once a new example e has been added to the training set. Consider a pair of training sets (E_1^+, E_1^-) and (E_2^+, E_2^-). The *intersection* operation requires computing the version space $V_{\mathsf{B}}(E_1^+, E_1^-) \cap V_{\mathsf{B}}(E_2^+, E_2^-)$. In the following, we assume that (E_1^+, E_1^-) and (E_2^+, E_2^-) contain m_1 and m_2 examples, respectively. Finally, given a pair of training sets (E_1^+, E_1^-) and (E_2^+, E_2^-), we may wish to determine whether $V_{\mathsf{B}}(E_1^+, E_1^-)$ is a *subset* of (resp. *equal* to) $V_{\mathsf{B}}(E_2^+, E_2^-)$.

Query/Operation	Time Complexity
Collapse	$\mathcal{O}(bm)$
Membership	$\mathcal{O}(bm)$
Update	$\mathcal{O}(b)$
Intersection	$\mathcal{O}(b(m_1 + m_2))$
Subset	$\mathcal{O}(b^2 m_1 m_2)$
Equality	$\mathcal{O}(b^2 m_1 m_2)$

Table 1: Time complexities of standard version space operations on the SAT representation used by CONACQ.

In Table 1 we present a summary of the complexities of these operations (see (Bessiere *et al.* 2005) for details). In each case we consider a library B containing b constraints and a training set (E^+, E^-) containing m examples.

Exploiting Domain-specific Knowledge

In constraint programming, constraints can be interdependent. For example, two constraints such as $\geq_{12}$ and $\geq_{23}$ impose a restriction on the relation of any constraint defined on the scope (x_1, x_3). This is a crucial difference with propositional logic where atomic variables are pairwise independent. As a consequence of such interdependency, some constraints in a network can be *redundant*. For example, the constraint $\geq_{13}$ is redundant given $\geq_{12}$ and $\geq_{23}$. An important difficulty for the learner is its ability to "detect" redundant constraints. The notion of redundancy is formalised as follows. Let C be a constraint network and c_{ij} a constraint in C. We say that c_{ij} is *redundant* in C if $sol(\mathsf{C} \setminus \{\mathsf{c}_{ij}\}) = sol(\mathsf{C})$. In other words, c_{ij} is redundant

if the constraint network obtained by deleting c_{ij} from C is equivalent to C.

Redundancy is a crucial notion that must be carefully handled if we need to allow version space convergence, or at least if we want to have a more accurate view of which parts of the target network are not precisely learned. This problem is detailed in the following example.

Example 2 (Redundancy) *Consider a vocabulary formed by a set of variables $\{x_1, x_2, x_3\}$ and a set of domain values $D = \{1, 2, 3, 4\}$. The learner has at its disposal the constraint library $\mathsf{B} = \{\top_{12}, \leq_{12}, \neq_{12}, \geq_{12}, \top_{23}, \leq_{23}, \neq_{23}, \geq_{23}, \top_{13}, \leq_{13}, \neq_{13}, \geq_{13}\}$. We suppose that the target network is given by $\{\geq_{12}, \geq_{13}, \geq_{23}\}$. The training set is given in Table 2. In the third column of the table, we present the growing clausal theory K, maintained by* CONACQ*, after processing each example and performing unit propagation.*

	x_1	x_2	x_3	K
e_1^+	4	3	1	$(\neg \leq_{12}) \wedge (\neg \leq_{13}) \wedge (\neg \leq_{23})$
e_2^-	2	3	1	$(\neg \leq_{12}) \wedge (\neg \leq_{13}) \wedge (\neg \leq_{23}) \wedge (\geq_{12})$
e_3^-	3	1	2	$(\neg \leq_{12}) \wedge (\neg \leq_{13}) \wedge (\neg \leq_{23}) \wedge (\geq_{12}) \wedge (\geq_{23})$

Table 2: A set of examples and the corresponding set of clauses K (unit propagated), showing the effect of redundancy.

After processing each example in the training set, the constraints $\geq_{12}$ and $\geq_{23}$ have been found. Yet, the redundant constraint $\geq_{13}$ has not. For the scope (x_1, x_3) the version space contains the four possible constraints $>_{13}, \geq_{13}, \neq_{13}$ or $\top_{13}$. In fact, in order to converge further we need a negative example e where $e(x_1) < e(x_3)$, $e(x_1) \geq e(x_2)$ and $e(x_2) \geq e(x_3)$. Due to the semantics of inequality constraints, no such example exists. Consequently, the inability of the learner to detect redundancy can hinder the convergence process and, hence, overestimate the number of networks in the version space. ▲

In (Bessiere *et al.* 2004) we proposed two approaches to dealing with redundancy. These will be explained briefly here. In the following section each approach will be demonstrated on some simple problems.

Redundancy Rules. A *redundancy rule* is a Horn clause:

$$\forall x_1, x_2, x_3, \mathsf{b}(x_1, x_2) \wedge \mathsf{b}'(x_2, x_3) \rightarrow \mathsf{b}''(x_1, x_3)$$

such that for any network C for which a substitution θ maps $\mathsf{b}(x_1, x_2)$, $\mathsf{b}'(x_2, x_3)$ and $\mathsf{b}''(x_1, x_3)$ into variables in C, the constraint $\mathsf{b}''_{\theta(x_1)\theta(x_3)}$ is redundant in C. As a form of background knowledge, the learner can use redundancy rules in its acquisition process. Given a library of constraints B and a set R of redundancy rules, the learner can start building each possible substitution on R. Namely, for each rule $\mathsf{b}(x_1, x_2) \wedge \mathsf{b}'(x_2, x_3) \rightarrow \mathsf{b}''(x_1, x_3)$ and each substitution θ that maps $\mathsf{b}(x_1, x_2)$, $\mathsf{b}'(x_2, x_3)$, and $\mathsf{b}''(x_1, x_3)$ to constraints b_{ij}, b'_{jk} and b''_{ik} in the library, a clause $\neg\mathsf{b}_{ij} \vee \neg\mathsf{b}'_{jk} \vee \mathsf{b}''_{ik}$ can be added to the clausal theory K. As argued in (Bessiere *et al.* 2005), even with redundancy rules, CONACQ remains polynomial.

Backbone Detection. While redundancy rules can handle a particular type of redundancy, there are cases where they are not sufficient to find all redundancies. The reason for this behaviour is that the redundancy rules are in the form of Horn clauses that are applied only when *all* literals in the left-hand side are true (i.e., unit propagation is performed on these clauses). However, the powerful concept of the *backbone* of a propositional formula can be used here. Informally, a literal belongs to the backbone of a formula if it belongs to all models of the formula (Monasson *et al.* 1999). Once the literals in the backbone are detected, we use them to update the theory representing the version space.

If an atom b_{ij} appears positively in all models of $\mathsf{K} \wedge \mathsf{R}$, then it belongs to its backbone and we can deduce that $\mathsf{c}_{ij} \subseteq \mathsf{b}_{ij}$. By construction of $\mathsf{K} \wedge \mathsf{R}$, the constraint c_{ij} cannot reject all negative examples in E^- and, at the same time, be more general than b_{ij}. Thus, given a new negative example e in E^-, we simply need to build the corresponding clause κ_e, add it to K, and test if the addition of κ_e causes some literal to enter the backbone of $\mathsf{K} \wedge \mathsf{R}$. This process can guarantee that all the possible redundancies will be detected.

Examples

We consider the constraint library used in Example 1. This constraint library can be used to represent networks of simple temporal constraints. The Horn clause $\forall x, y, z, (x \geq y) \wedge (y \geq z) \rightarrow (x \geq z)$ is a redundancy rule since any constraint network in which we have two constraints '$\geq$' such that the second argument of the first constraint is equal to the first argument of the second constraint implies the '$\geq$' constraint between the first argument of the first constraint and the second argument of the second constraint.

Example 3 (Redundancy Rules) *We can apply the redundancy rule technique to Example 2. After performing unit propagation on the clausal theory K obtained after processing the examples $\{e_1^+, e_2^-, e_3^-\}$, we know that $\geq_{12}$ and $\geq_{23}$ have to be set to 1. When instantiated on this constraint network, the redundancy rule above becomes $\geq_{12} \wedge \geq_{23} \rightarrow \geq_{13}$. Since all literals of the left part of the rule are forced by K to be true, we can fix literal $\geq_{13}$ to 1.* ▲

	x_1	x_2	x_3	K
e_1^+	2	2	2	$(\neg \neq_{12}) \wedge (\neg \neq_{13}) \wedge (\neg \neq_{23})$
e_2^-	3	3	4	$(\neg \neq_{12}) \wedge (\neg \neq_{13}) \wedge (\neg \neq_{23}) \wedge (\geq_{13} \vee \geq_{23})$
e_3^-	1	3	3	$(\neg \neq_{12}) \wedge (\neg \neq_{13}) \wedge (\neg \neq_{23}) \wedge (\geq_{13} \vee \geq_{23})$ $\wedge (\geq_{12} \vee \geq_{13})$

Table 3: A set of examples and the corresponding set of clauses K (unit propagated), showing the effect of higher-order redundancy.

Consider the example in Table 3 where the target network comprises the set of constraints $\{=_{12}, =_{13}, =_{23}\}$ and all negative examples differ from the single positive example by *at least* two constraints. The version space in this example contains 4 possible constraints for each scope, due to the disjunction of possible reasons that would classify the negative examples correctly. Without any further information, particularly negative examples that differ from the positive example by one constraint, redundancy rules cannot restrict the version space any further. However, there is a constraint

Redundant Pattern		CONACQ		CONACQ +rules		CONACQ +rules +backbone		
Length	$\{constraints\}$	$U(V_\mathsf{B})$	time (s)	$U(V_\mathsf{B})$	time (s)	$U(V_\mathsf{B})$	time (s)	$\#Exs$
none		32	0.11	26	0.32	24	2.67	1000
n/3	$\{\leq, \geq\}$	12	0.11	6	0.31	0	2.61	360
n/2	$\{\leq, \geq\}$	34	0.11	12	0.32	0	2.57	190
n	$\{\leq, \geq\}$	57	0.11	18	0.32	0	2.54	90
n/3	$\{<, =, >\}$	28	0.11	10	0.32	0	2.60	280
n/2	$\{<, =, >\}$	66	0.11	25	0.32	0	2.58	170
n	$\{<, =, >\}$	113	0.11	36	0.32	0	2.54	70
n	$\{<, =, >\}$	53	0.11	11	0.32	0	0.24	1000

Table 4: Comparison of the CONACQ variants (problems have 12 variables, 12 values, 18 constraints). $U(V_\mathsf{B})$ is the average (over 100 experiments) of the number of constraints not yet decided in V_B (i.e., the associated literal is not fixed in our version space representation K). $2^{U(V_\mathsf{B})}$ is an upper bound to the size of V_B. We also report the average time to process an example on a Pentium IV 1.8 GHz running Linux. $\#Exs$ is the number of examples used in each experiment. In all but the first and last lines, $\#Exs$ is the number of examples needed to obtain convergence of at least one of the algorithms. The training set contains 10% of positive examples and 90% of negative examples.

that is implied by the set of negative examples but redundancy rules are not able to detect it.

Example 4 (Backbone Detection) *We now apply this method to the example in Table 3. To test if the literal $\geq_{13}$ belongs to the backbone, we solve $\mathsf{R} \cup \mathsf{K} \cup \{\neg \geq_{13}\}$. If the redundancy rule $\geq_{12} \wedge \geq_{23} \rightarrow \geq_{13}$ belongs to R, we detect inconsistency. Therefore, $\geq_{13}$ belongs to the backbone. Our representation of the version space can now be refined, by setting the literal $\geq_{13}$ to 1, removing from the version space the constraint networks containing $\leq_{13}$ or $\top_{13}$.* ▲

Experiments

We summarise, in Table 4, an empirical evaluation reported in (Bessiere *et al.* 2005). We implemented CONACQ using SAT4J.[1] The target constraint networks were sets of binary constraints defined from the set of relations $\{\leq, \neq, \geq\}$. We did not assume to know the scopes of the constraints in the target problem, so the available library involved all 66 possibilities (12 variables). The level of redundancy was controlled by introducing constraint "patterns" of various lengths and type. Patterns were paths of the same constraint selected either from the set $\{\leq, \geq\}$ (looser constraints) or $\{<, =, >\}$ (tighter constraints). For example, a pattern of length k based on $\{<, =, >\}$ could be $x_1 > x_2 > \ldots > x_k$. The remaining constraints in the problem were selected randomly. Negative examples were *partial* non-solutions to the problem involving a subset (2-5 in size) of variables.

We report the number of non-fixed literals in K, that is the number of constraints not yet decided in the version space. We note that the rate of convergence improves if we exploit domain-specific knowledge. In particular, CONACQ using redundancy rules and backbone detection can eliminate all redundant networks in all experiments with redundant patterns. In contrast, the performance of the first two algorithms degrades as the length of the redundant patterns increases. Secondly, we observe that for patterns involving tighter constraints ($<$, $=$, or $>$), better improvements are obtained as we employ increasingly powerful techniques for exploiting redundancy. Thirdly, the learning time progressively increases with the sophistication of the method used. The basic CONACQ algorithm is about 3 times faster than CONACQ+ $rules$ and 25 times faster than CONACQ+$rules$ + $backbone$. However, the running times are still satisfactory for an interactive application.

[1] Available from `http://www.sat4j.org`.

Conclusions

Users of constraint programming technology need significant expertise in order to model their problems appropriately. We have proposed a SAT-based version space algorithm that learns constraint networks. This approach has a number of distinct advantages. Firstly, the formulation is generic, so we can use any SAT solver as a basis for version space learning. Secondly, we can exploit powerful SAT techniques such as unit propagation and backbone detection to improve learning rate. Finally, we can easily incorporate domain-specific knowledge into constraint acquisition to improve the quality of the acquired network.

Acknowledgments. This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

References

Bessiere, C.; Coletta, R.; Freuder, E.; and O'Sullivan, B. 2004. Leveraging the learning power of examples in automated constraint acquisition. In *CP*, 123–137.

Bessiere, C.; Coletta, R.; Koriche, F.; and O'Sullivan, B. 2005. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *ECML*, 23–34.

Hirsh, H.; Mishra, N.; and Pitt, L. 2004. Version spaces and the consistency problem. *AI Journal* 156(2):115–138.

Mitchell, T. 1982. Generalization as search. *AI Journal* 18(2):203–226.

Monasson, R.; Zecchina, R.; Kirkpatrick, S.; Selman, B.; and Ttroyansky, L. 1999. Determining computational complexity from characteristic 'phase transition'. *Nature* 400:133–137.