Query-driven Constraint Acquisition

Christian Bessiere
LIRMM-CNRS
U. Montpellier, France
bessiere@lirmm.fr

Remi Coletta
LRD
Montpellier, France
coletta@l-rd.fr

Barry O'Sullivan
4C, Computer Science Dept.
UCC, Ireland
b.osullivan@4c.ucc.ie

Mathias Paulin
LIRMM-CNRS
U. Montpellier, France
paulin@lirmm.fr

Abstract

The modelling and reformulation of constraint networks are recognised as important problems. The task of automatically acquiring a constraint network formulation of a problem from a subset of its solutions and non-solutions has been presented in the literature. However, the choice of such a subset was assumed to be made independently of the acquisition process. We present an approach in which an interactive acquisition system actively selects a good set of examples. We show that the number of examples required to acquire a constraint network is significantly reduced using our approach.

1 Introduction

Constraint Programming (CP) provides a powerful paradigm for solving combinatorial problems. However, the specification of constraint networks still remains limited to specialists in the field. An approach to automatically acquiring constraint networks from examples of their solutions and non-solutions has been proposed by [Bessiere *et al.*, 2005]. Constraint acquisition was formulated as a concept learning task. The classical version space learning paradigm [Mitchell, 1982] was extended so that constraint networks could be learned efficiently. Constraint networks are much more complex to acquire than simple conjunctive concepts represented in propositional logic. While in conjunctive concepts the atomic variables are pairwise independent, in constraint satisfaction there are dependencies amongst them.

In [Bessiere *et al.*, 2005] the choice of the subset of solutions and non-solutions to use for learning was assumed to be made before and independently of the acquisition process. In this paper we present an approach in which the acquisition system actively assists in the selection of the set of examples used to acquire the constraint network through the use of learner-generated queries. A query is essentially a complete instantiation of values to the variables in the constraint network that the user must classify as either a solution or non-solution of her 'target' network. We show that the number of examples required to acquire a constraint network is significantly reduced if queries are selected carefully.

When acquiring constraint networks computing good queries is a hard problem. The classic query generation strategy is one in which, regardless of the classification of the query, the size of the version space is reduced by half. Therefore, convergence of the version space can be achieved using a logarithmic number of queries. Furthermore, in the classic setting, a query can be generated in time polynomial in the size of the version space. When acquiring constraint networks, query generation becomes NP-hard. This is further aggravated by the fact that in constraint acquisition, while the ordering over the hypothesis space is most naturally defined in terms of the solution space of constraint networks, we usually learn at the constraint level, i.e. a compact representation of the set of solutions of a hypothesis. Our main contribution is a number of algorithms for identifying good queries for acquiring constraint networks. Our empirical studies show that using our techniques the number of examples required to acquire a constraint network is significantly reduced. This work is relevant to interactive scenarios where users are actively involved in the acquisition process.

2 Constraint Acquisition using CONACQ

A constraint network is defined on a (finite) set of variables X and a (finite) set of domain values D. This common knowledge shared between the learner and the user is called the vocabulary. Furthermore, 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 the sake of notation, we shall assume that every constraint defined from the library is binary. However, the results presented here can be easily extended to constraints of higher arity, and this is demonstrated in our experiments.

A *binary constraint* c_{ij} is a binary relation defined on D that specifies which pairs of values are allowed for variables x_i, x_j. The pair of variables (x_i, x_j) is called the *scope* of c_{ij}. For instance, $\leq_{12}$ denotes the constraint specified on (x_1, x_2) with relation "less than or equal to". A *binary constraint network* is a set C of binary constraints. A *constraint bias* is a collection B of binary constraints built from the constraint library on the given vocabulary. A constraint network C is said to be *admissible* for a bias B if for each constraint c_{ij} in C there exists a set of constraints $\{\mathsf{b}^1_{ij}, \cdots, \mathsf{b}^k_{ij}\}$ in B such that $\mathsf{c}_{ij} = \mathsf{b}^1_{ij} \cap \cdots \cap \mathsf{b}^k_{ij}$.

An *instance* e is a map that assigns to each variable x_i in

X a domain value $e(x_i)$ in D. Equivalently, an instance e can be regarded as a tuple in D^n. An instance e *satisfies* a binary constraint c_{ij} if the pair $(e(x_i), e(x_j))$ is an element of c_{ij}; otherwise we say that c_{ij} *rejects* e. If an instance e *satisfies* every constraint in C, then e is called a *solution* of C; otherwise, e is called a *non-solution* of C.

Finally, a *training set* E^f consists of a set E of instances and a classification function $f : E \rightarrow \{0, 1\}$. An element e in E such that $f(e) = 1$ is called *positive example* (often denoted by e^+) and an element e such that $f(e) = 0$ is called *negative* example (often denoted by e^-). A constraint network C is said to be *consistent* with a training set E^f if every positive example e^+ in E^f is a solution of C and every negative example e^- in E^f is a non-solution of C. We also say that C *correctly classifies* E^f. Given a constraint bias B and a training set E^f, the *Constraint Acquisition Problem* is to find a constraint network C admissible for the bias B and consistent with the training set E^f.

A SAT-based algorithm, called CONACQ, was presented in [Bessiere *et al.*, 2005] 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 vocabulary and bias, and that are consistent with the given training set. We denote as $V_{\mathsf{B}}(E^f)$ the version space corresponding to the bias B and the training set E^f. In the SAT-based framework this version space is encoded in a clausal theory K. Each model of the theory K is a constraint network of $V_{\mathsf{B}}(E^f)$.

More formally, if B is the constraint bias, a literal is either an atom b_{ij} in B, or its negation $\neg\mathsf{b}_{ij}$. Notice that $\neg\mathsf{b}_{ij}$ is *not* a constraint: it merely captures the absence of b_{ij} in the acquired network. A clause is a disjunction of literals (also represented as a set of literals), and the clausal theory K is a conjunction of clauses (also represented as a set of clauses). An *interpretation* over B is a map I that assigns to each constraint atom b_{ij} in B a value $I(\mathsf{b}_{ij})$ in $\{0, 1\}$. A *transformation* is a map ϕ that assigns to each interpretation I over B the corresponding constraint network $\phi(I)$ defined according to the following condition: $\mathsf{c}_{ij} \in \phi(I)$ iff $\mathsf{c}_{ij} = \bigcap\{\mathsf{b}^p_{ij} \in \mathsf{B} : I(\mathsf{b}^p_{ij}) = 1\}$. An interpretation I is a *model* of K if K is true in I according to the standard propositional semantics. The set of all models of K is denoted $Models(\mathsf{K})$. For each instance e, $\kappa(e)$ denotes the set of all constraints b_{ij} in B rejecting e. For each example e in the training set E^f, the CONACQ algorithm iteratively adds to K a set of clauses so that for any $I \in Models(\mathsf{K})$, the network $\phi(I)$ correctly classifies all already processed examples plus e. When an example e is positive, unit clauses $\{\neg\mathsf{b}_{ij}\}$ are added to K for all $\mathsf{b}_{ij} \in \kappa(e)$. When an example e is negative, the clause $\{\bigvee_{\mathsf{b}_{ij} \in \kappa(e)} \mathsf{b}_{ij}\}$ is added to K. The resulting theory K encodes all candidate networks for the constraint acquisition problem. That is, $V_{\mathsf{B}}(E^f) = \{\phi(m) \mid m \in Models(\mathsf{K})\}$.

Example 1 (CONACQ's Clausal Representation) *We wish to acquire a constraint network involving 4 variables, $x_1, \ldots, x_4$, with domains $D(x_1) = \ldots = D(x_4) = \{1, 2, 3, 4\}$. We use a complete and uniform bias, with $L = \{\leq, \neq, \geq\}$ as a library. That is, for all $1 \leq i < j \leq 4$, B contains $\leq_{ij}, \neq_{ij}$ and $\geq_{ij}$. Assume that the network we wish to acquire contains only one constraint, namely $x_1 \neq x_4$; there is no constraint between any other pair of variables. For each example e (first column), Table 1 shows the clausal encoding constructed by* CONACQ *after e is processed, using the set $\kappa(e)$ of constraints in the bias B that can reject e.* ▲

Table 1: An example of the clausal representation built by CONACQ, where each example $e^?_i = (x_1, x_2, x_3, x_4)$.

E^f	example	clauses added to K
$\{e^+_1\}$	(1,2,3,4)	$\neg \geq_{12} \wedge \neg \geq_{13} \wedge \neg \geq_{14} \wedge \neg \geq_{23} \wedge \neg \geq_{24} \wedge \neg \geq_{34}$
$\{e^+_2\}$	(4,3,2,1)	$\neg \leq_{12} \wedge \neg \leq_{13} \wedge \neg \leq_{14} \wedge \neg \leq_{23} \wedge \neg \leq_{24} \wedge \neg \leq_{34}$
$\{e^-_3\}$	(1,1,1,1)	$(\neq_{12} \vee \neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$

The learning capability of CONACQ can be improved by exploiting domain-specific knowledge [Bessiere *et al.*, 2005]. In constraint programming, constraints are often interdependent, e.g. 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 conjunctive concepts where atomic variables are pairwise independent. Because of such interdependency, some constraints in a network can be *redundant*. c_{ij} is redundant in a network C if the constraint network obtained by deleting c_{ij} from C has the same solutions as C. The constraint $\geq_{13}$ is redundant each time $\geq_{12}$ and $\geq_{23}$ are present.

Redundancy must be carefully handled if we want to have a more accurate idea of which parts of the target network are not precisely learned. One of the methods to handle redundancy proposed in [Bessiere *et al.*, 2005], was to add *redundancy rules* to K based on the library of constraints used to build the bias B. For instance, if the library contains the constraint type $\leq$, for which we know that $\forall x, y, z, (x \leq y) \wedge (y \leq z) \rightarrow (x \leq z)$, then for any pair of constraints $\leq_{ij}, \leq_{jk}$ in B, we add the Horn clause $\leq_{ij} \wedge \leq_{jk} \rightarrow \leq_{ik}$ in K. This form of background knowledge can help the learner in the acquisition process.

3 The Interactive Acquisition Problem

In reality, there is a cost associated with classifying instances to form a training set (usually because it requires an answer from a human user) and, therefore, we should seek to minimise the size of training set required to acquire our target constraint network. The *target network* is the constraint network C_T expressing the problem the user has in mind. That is, given a vocabulary X, D, C_T is the constraint network such that an instance on X is a positive example if and only if it is a solution of C_T.

During the learning process the acquisition system has knowledge that can help characterise what next training example would be ideal from the acquisition system's point of view. Thus, the acquisition system can carefully select 'good' training examples (which we will discuss in Section 4 in more depth), that is, instances which, depending on how the user classifies them, can help reduce the expected size of the version space as much as possible. We define a query and the classification assigned to it by the user as follows.

Definition 1 (Queries and Query Classification) *A query q is an instance on X that is built by the learner. The user*

classifies a query q using a function f such that $f(q) = 1$ if q is a solution of C_T and $f(q) = 0$ otherwise.

Angluin [Angluin, 2004] defines several classes of queries, among which the *membership query* is exactly the kind used here. The user is presented with an unlabelled instance, and is asked to classify it. We can now formally define the interactive constraint acquisition problem.

Definition 2 (Interactive Constraint Acquisition Problem) *Given a constraint bias* B *and an unknown user classification function f, the* Interactive Constraint Acquisition Problem *is to find a converging sequence $\mathcal{Q} = q_1, \ldots, q_m$ of queries, that is, a sequence such that: q_{i+1} is a query relative to* B *and $V_{\mathsf{B}}(E_i^f)$ where $E_i = \{q_1, \ldots, q_i\}$, and $|V_{\mathsf{B}}(E_m^f)| = 1$.*

Note that the sequence of queries is built incrementally, that is, each query q_{i+1} is built according to the classification of $q_1, \ldots, q_i$. In practice, minimising the length of $\mathcal{Q}$ is impossible because we do not know in advance the answers from the user. However, in the remainder of the paper we propose techniques that are suitable for interactive learning.

4 Query Generation Strategies

4.1 Polynomial-time Query Generation

In practice, it can be the case that an example e from the training set does not bring any more information than that which has already been provided by the other examples that have been considered so far. If we allow for queries to be generated whose classification is already known based on the current representation of the version space, K, then we will ask the user to classify an excessive number of examples for no improvement in the quality of our representation of the version space of the target network. We exemplify this problem with a short example.

Example 2 (A Redundant Query) *Consider an acquisition problem over the three variables x_1, x_2, x_3, with the domains $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3, 4\}$ using the same constraint library as in Example 1. Given the positive example $e_1^+ = \langle (x_1, 1), (x_2, 2), (x_3, 3) \rangle$, $\mathsf{K} = \neg\, \geq_{12} \wedge \neg\, \geq_{13} \wedge \neg\, \geq_{23}$. Asking the user to classify $e_2 = \langle (x_1, 1), (x_2, 2), (x_3, 4) \rangle$ is redundant, since all constraints rejecting it are already forbidden by* K. *Then any constraint network in the version space accepts e_2.* ▲

We propose a simple (poly-time) technique that avoids proposing such redundant queries to the user. This *irredundant queries* technique seeks a classification only for an example e that cannot be classified, given the current representation K of the version space. An example e can be classified by $V_{\mathsf{B}}(E^f)$ if it is either a solution in all networks in $V_{\mathsf{B}}(E^f)$ or a non-solution in all networks in $V_{\mathsf{B}}(E^f)$. e is a solution in all networks in $V_{\mathsf{B}}(E^f)$ iff the subset $\kappa(e)_{[\mathsf{K}]}$ of $\kappa(e)$, obtained by removing from $\kappa(e)$ all constraints that appear as negated literals in K, is empty. Alternatively, e is a non-solution in all networks in $V_{\mathsf{B}}(E^f)$, if $\kappa(e)_{[\mathsf{K}]}$ is a superset of an existing clause of K.

Example 3 (An Irredundant Query) *Consider again Example 2 in which the positive example e_1^+ has been considered. The query $e = \langle (x_1, 1), (x_2, 2), (x_3, 2) \rangle$ is irredundant. This can be seen by considering the literals that would be added to* K *by this query. If the query is classified as positive, the clauses $(\neg \geq_{12}), (\neg \geq_{13})$ and $(\neg \neq_{23})$ will be added to* K*, otherwise the clause $(\geq_{12} \vee \geq_{13} \vee \neq_{23})$ will be added. Since we know from example e_1^+ that both $\geq_{12}$ and $\geq_{13}$ must be set to false, the only extra literal this new example adds is either $(\neg \neq_{23})$ or $(\neq_{23})$ (indeed $\kappa(e)_{[\mathsf{K}]} = \{\neq_{23}\}$). Regardless of the classification of e, something new is learned, so this is an irredundant query.* ▲

4.2 Towards Optimal Query Generation

The technique presented in Section 4.1 guarantees that each newly classified query e adds something new to K. However, different irredundant examples give us different gains in knowledge. In fact, the gain for a query q is directly related to the size k of $\kappa(q)_{[\mathsf{K}]}$ and its classification $f(q)$. If $f(q) = 1$, k unary negative clauses will be added to K, then k literals will be fixed to 0. In terms of CONACQ, we do not have direct access to the size of the version space, unless we wish to perform very expensive computation through the clausal representation K. But assuming that the models of K are uniformly distributed, fixing k literals divides the number of models by 2^k. If $f(q) = 0$, a positive clause of size k is added to K, thus removing $1/2^k$ models. We can distinguish between queries that can be regarded as *optimistic*, or as *optimal-in-expectation*.

An *optimistic query* is one that gives us a large gain in knowledge when it is classified "in our favour", but which tells us very little when it is classified otherwise. More specifically, in CONACQ the larger the $\kappa(q)_{[\mathsf{K}]}$ of a query q, the more optimistic it is. When classified as positive, such a query allows us to set $|\kappa(q)_{[\mathsf{K}]}|$ literals to 0. If the query is classified as negative we just add a clause of size $|\kappa(q)_{[\mathsf{K}]}|$. Therefore, an optimistic query is maximally informative – sets all literals it introduces to 0 – if it is classified as positive, but is minimally informative if it is classified as negative.

The optimal query strategy is one that involves proposing a query that will reduce the size of the version space in half *regardless of how the user classifies it*. We define a query as being *optimal-in-expectation* if we are guaranteed that one literal will be fixed to either a 0 or a 1 regardless of the classification provided by the user. Formally, such a query will have a $\kappa(q)_{[\mathsf{K}]}$ of size 1, therefore, if it is classified as positive, we can set the literal in $\kappa(q)_{[\mathsf{K}]}$ to 0, otherwise it is set to a 1.

We illustrate a sequence of queries that are sufficient for the version space of the problem presented as Example 1 to converge using queries that are optimal-in-expectation.

Example 4 (Optimal-in-Expectation Queries) *We want to converge on the target network from Example 1 (i.e., the only constraint $x_1 \neq x_4$ in a network with four variables and the complete bias of constraints $\{\leq, \neq, \geq\}$). Recall that having processed the set of examples $E = \{e_1^+, e_2^+, e_3^-\}$, the unique positive clause in* K *is $Cl = (\neq_{12} \vee \neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$. All other atoms in* K *are fixed to 0 because of e_1^+ and e_2^+. In the following $\mathsf{K}_{\{e_1^+, e_2^+\}}$ refers to $(\neg \geq_{12}) \wedge \ldots (\neg \geq_{34}) \wedge (\neg \leq_{12}) \wedge \ldots \wedge (\neg \leq_{34})$. According to this notation, the clausal theory* K *built by* CONACQ *having processed E is $\mathsf{K} = \mathsf{K}_{\{e_1^+, e_2^+\}} \wedge Cl$. Table 2 shows*

Table 2: *Optimal-in-expectation* query generation strategy on Example 4.

e	$\kappa(e)_{[\mathsf{K}]}$	$f(e)$	K
$e_4 = (1,1,2,3)$	$\{\neq_{12}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$
$e_5 = (2,1,1,3)$	$\{\neq_{23}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{24} \vee \neq_{34})$
$e_6 = (2,3,1,1)$	$\{\neq_{34}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{24})$
$e_7 = (1,3,1,2)$	$\{\neq_{13}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \wedge (\neg \neq_{13}) \quad \wedge \quad (\neq_{14} \vee \neq_{24})$
$e_8 = (2,1,3,1)$	$\{\neq_{24}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \wedge (\neg \neq_{13}) \wedge (\neg \neq_{24}) \wedge (\neq_{14})$

a sequence of queries that are optimal-in-expectation on the version space obtained after the three first examples are processed. The goal is to reduce $V_{\mathsf{B}}(E)$ to contain a single hypothesis. The first column is a query e generated according to the optimal-in-expectation strategy. The second column gives the set $\kappa(e)_{[\mathsf{K}]}$ of constraints still possible in a network of the version space that could reject e. The third column is the classification of e by the user, and the fourth column is the update of K. The query e_4 is such that $\neq_{12}$ is the only constraint still possible in the version space that can reject it. Because it is classified as positive, we are sure $\neq_{12}$ cannot belong to a network in the version space. CONACQ *adds $(\neg \neq_{12})$ to K and the literal $\neq_{12}$ is removed from Cl by unit propagation. The process repeats with e_5, e_6 and e_7, decreasing the size of Cl by one literal at a time, and thus reducing the version space by half. Finally, e_8 is the last example required to ensure that the version space converges on the target network, which contains the single constraint $x_1 \neq x_4$.*

Note that at the beginning of this example, the version space $V_{\mathsf{B}}(E)$ contained 2^6 possible constraint networks, and we could converge using $\mathcal{O}(log_2|V_{\mathsf{B}}(E)|)$ queries, which is an optimal worst-case [Mitchell, 1982]. ▲

In Example 4, we always found an example e with $|\kappa(e)_{[\mathsf{K}]}| = 1$, as the optimal-in-expectation strategy requires. However, redundancy can prevent us from being able to generate an example e with a given size for its $\kappa(e)_{[\mathsf{K}]}$. For instance, consider the acquisition problem, using a complete and uniform bias, with $L = \{\leq, \neq, \geq\}$ as a library, and with $x_1 = x_2 = x_3$ as a target network. After processing an initial positive example (for instance $e_1^+ = (2,2,2)$), the possible constraints in the version space are $\leq_{12}, \leq_{13}, \leq_{23}, \geq_{12}, \geq_{13}, \geq_{23}$. Hence, every further negative example e has either a $\kappa(e)_{[\mathsf{K}]}$ of size 3 (if no variables equal) or a $\kappa(e)_{[\mathsf{K}]}$ of size 2 (if two variables equal). Therefore, no example with a $\kappa(e)_{[\mathsf{K}]}$ of size 1 can be generated. Redundancy prevents us from generating such examples.

5 Implementing our Strategies

In Section 4.2, we presented two strategies for generating queries: optimal-in-expectation and optimistic. These two strategies are characterised by the target number t of constraints still possible in the version space that reject the instances q they try to produce. However, it may be the case that, due to redundancy between constraints, there does not exist any network in the version space that has a solution s with $|\kappa(s)_{[\mathsf{K}]}| = t$. (And it is useless to ask classification of an instance if it is not a solution of some network in the version space – see Section 4.1). We then must allow for some uncertainty in the number of constraints rejecting an instance.

We implement the query generation problem as a two step process. First, Algorithm 1 tries to find an interpretation I on B such that any solution s of $\phi(I)$ is such that $t - \epsilon \leq |\kappa(s)_{[\mathsf{K}]}| \leq t + \epsilon$, where ϵ is the variation accepted on the size of the $\kappa(q)_{[\mathsf{K}]}$ of the query q we want to generate. This algorithm takes another input parameter which is the set L of constraints in which $\kappa(q)_{[\mathsf{K}]}$ must be included. We will explain later that this is a way to monitor the 'direction' in which we want to improve our knowledge of the target network of the user. Second, once I has been found, we take a solution of $\phi(I)$ as a query. We first present the algorithm, then we will discuss its complexity and describe how we can use it to implement our strategies (by choosing the values t and ϵ).

Algorithm 1: QUERY GENERATION PROBLEM

input : B the bias, K the clausal theory, L a set of literals, t a target size and ϵ the variation
output: An interpretation I

1 $\mathsf{F} \leftarrow \mathsf{K}$
2 **foreach** $\mathsf{b}_{ij} \in \mathsf{B} \setminus \{\mathsf{b}_{ij} \mid (\neg \mathsf{b}_{ij}) \in \mathsf{K}\}$ **do**
3 $\quad$ **if** $\mathsf{b}_{ij} \notin \mathsf{L}$ **then** $\mathsf{F} \leftarrow \mathsf{F} \wedge (\mathsf{b}_{ij})$
$\quad$ **else** $\mathsf{F} \leftarrow \mathsf{F} \wedge (\mathsf{b}_{ij} \vee \overline{\mathsf{b}_{ij}})$
4 $lower \leftarrow max(|\mathsf{L}| - t - \epsilon, 1)$
5 $upper \leftarrow min(|\mathsf{L}| - t + \epsilon, |\mathsf{L}|)$
6 $\mathsf{F} \leftarrow \mathsf{F} \wedge atLeast(lower, \mathsf{L}) \wedge atMost(upper, \mathsf{L})$
7 **if** $Models(\mathsf{F}) \neq \emptyset$ **then** return a model of F
8 **else** return "inconsistency"

Algorithm 1 works as follows. It takes as input the target size t, the allowed variation ϵ and the set L of literals on which to concentrate. The idea is to build a formula F for which every model I will satisfy the requirements listed above. F is initialised to K to guarantee that any model will correspond to a network in the version space (line 1). For each literal b_{ij} not already negated in K (line 2), if b_{ij} does not belong to L, we add the clause (b_{ij}) to F to enforce the constraint b_{ij} to belong to the network $\phi(I)$ for all models I of F ('**then**' instruction of line 3). Hence, any solution s of $\phi(I)$ will be rejected either by a constraint in L or a constraint b_{ij} already negated in K (so no longer in the version space). Thus, $\kappa(s)_{[\mathsf{K}]} \subseteq \mathsf{L}$. We now have to force the size of $\kappa(s)_{[\mathsf{K}]}$ to be in the right interval. If b_{ij} belongs to L ('**else**' instruction of line 3), we add the clause $(\mathsf{b}_{ij} \vee \overline{\mathsf{b}_{ij}})$ to F to ensure that either b_{ij} or its complementary constraint $\overline{\mathsf{b}_{ij}}$ is in the re-

sulting network.[1] $\overline{\mathsf{b}_{ij}}$ is required because $\neg\mathsf{b}_{ij}$ only expresses the absence of the constraint b_{ij}. $\neg\mathsf{b}_{ij}$ is not sufficient to enforce b_{ij} to be violated. We now just add two pseudo-Boolean constraints that enforce the number of constraints from L violated by solutions of $\phi(I)$ to be in the interval $[t-\epsilon\,..\,t+\epsilon]$. This is done by forcing at most $|\mathsf{L}|-t+\epsilon$ constraints and at least $|\mathsf{L}|-t-\epsilon$ constraints to be satisfied (lines 4-6). The 'min' and 'max' ensure we avoid trivial cases (no constraint from L is violated) and to remain under the size of L. Line 7 searches for a model of F and returns it. But remember that redundancy may prevent us from computing a query q with a given $\kappa(q)_{[\mathsf{K}]}$ size (Section 4.2). So, if ϵ is too small, F can be unsatisfiable and an inconsistency is returned (line 8).

The following property tells us when the output of Algorithm 1 is guaranteed to lead to a query.

Property 1 (Satisfiability) *Given a bias* B*, a clausal theory* K*, and a model* I *of* K*. If* K *contains all existing redundancy rules over* B*, then* $\phi(I)$ *has solutions.*

If not all redundancy rules belong to K, Algorithm 1 can return I such that $\phi(I)$ is inconsistent. In such a case, we extract a conflict set of constraints S from $\phi(I)$ and add the clause $\bigvee_{\mathsf{b}_{ij}\in S}\neg\mathsf{b}_{ij}$ to K to avoid repeatedly generating models I' with this hidden inconsistency in $\phi(I')$.

The next property tells us that generating a given type of query can be hard.

Property 2 *Given a bias* B*, a theory* K*, a set* L *of constraints, a target size* t *and a variation* ϵ*, generating a query* q *such that:* $\kappa(q)_{[\mathsf{K}]}\subset\mathsf{L}$ *and* $t-\epsilon\leq|\kappa(q)_{[\mathsf{K}]}|\leq t+\epsilon$ *is* NP*-hard.*

The experimental section will show that despite its complexity, this problem is handled very efficiently by the technique presented in Algorithm 1. The algorithm can be used to check if there exists a query rejected by a set of constraints from the version space of size $t\pm\epsilon$ included in a given set L. The optimal-in-expectation strategy requires $t=1$ and optimistic requires a larger t. In the following, we chose to be "half-way" optimistic and to fix t to $|\mathsf{L}|/2$. There still remains the issue of which set L to use and which values of ϵ to try. ϵ is always initialised to 0. Concerning L, we take the smallest non-unary positive clause of K. A positive clause represents the set of constraints that reject a negative example already processed by CONACQ. So, we are sure that at least one of the constraints in such a set L rejects an instance. Choosing the smallest one increases the chances to quickly converge on a unary clause. If K does not contain any such non-unary clauses we take the set containing all non-fixed literals in K.

Since Algorithm 1 can return an inconsistency when called for a query, we have to find another set of input parameters on which to call the algorithm. t is fixed by the strategy, so we can change L or ϵ. If there are several non-unary clauses in K, we set L to the next positive clause in K (ordered by size). If we have tried all the clauses without success, we have to increase ϵ. We have two options. The first one, called *closest*, will look for a query generated with a set L instantiated to the clause that permits the smallest ϵ. The second one, called *approximate*, increases ϵ by fixed steps. It first tries to find a set L where a query exists with $\epsilon=0.25\cdot|\mathsf{L}|$. If not found, it looks (repeatedly) with $0.50\cdot|\mathsf{L}|$, $0.75\cdot|\mathsf{L}|$ and then $|\mathsf{L}|$.

We thus have four policies to generate queries: optimistic and optimal-in-expectation combined with closest and approximate: optimistic means $t=\mathsf{L}/2$ whereas optimal-in-expectation means $t=1$; closest finds the smallest ϵ whereas approximate increases ϵ by steps of 25%.

6 Experimental Results

We implemented CONACQ using `SAT4J`[2] and `Choco`[3]. In our implementation we exploit redundancy to the largest extent possible, using both redundancy rules and backbone detection [Bessiere *et al.*, 2005].

Problem Classes. We used a mix of binary and non-binary problem classes in our experiments. We studied random binary problems, with and without structure, as well as acquiring a CSP defining the rules of the logic puzzle Sudoku. CONACQ used a learning bias defined as the set of all edges in each problem using the library $\{\leq,\geq,\neq\}$. The random binary problems comprised 14 variables, with a uniform domain of size 20. We generated target constraint networks by randomly selecting a specified number of constraints from $\{<,\leq,=,\geq,>,\neq\}$, retaining only those that were soluble. We also considered instances in which we forced some constraint *patterns* in the constraint graph to assess the effect of structure [Bessiere *et al.*, 2005]. We did this by selecting the same constraint relation to form a path in the target network. Finally, we used a 4×4 Sudoku as the target network. The acquisition problem in this case was to learn the rules of Sudoku from (counter)examples of grid configurations.

As an example of a non-binary problem, we considered the Schur's lemma, which is Problem 15 from the CSPLIB[4]. In this case, CONACQ used the library of ternary constraints $\{$ALLDIFF, ALLEQUAL, NOTALLDIFF, NOTALLEQUAL$\}$.

Results. In Table 3 we report averaged results for 100 experiments of each query generation approach on each of the problem classes we studied. In each case the initial training set contained a single positive example. In the table the first column contains a description of the target networks in terms of number of variables and constraints. We report results for each of the query generation approaches we studied. *Random* is a baseline approach, generating queries entirely at random, which may produce queries that are redundant with respect to each other. The *Irredundant* approach generates queries at random, but only uses those that can provide new information to refine the version space. Finally, *Optimistic* and *Optimal-in-expectation* refer to approaches described in Section 5 and

[1] Not all libraries of constraints contain the complement of each constraint. However, the complements may be expressed by a conjunction of other constraints. For instance, in library $\leq,\neq,\geq$, $\overline{\leq}$ does not exist but it can be expressed by $(\geq\wedge\neq)$. If no conjunction can express the complement of a constraint, we can post an approximation of the negation (or nothing). We just lose the guarantee on the number of constraints in L that will reject the generated query.

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

[3] Available from: `http://choco.sourceforge.net`.

[4] Available from: `http://www.csplib.org`.

Table 3: Comparison of the various queries generation approaches on different classes of problems. Time is measured in milliseconds on a Pentium IV 1.8 GHz processor. We highlight the smallest number of queries for each problem class in bold.

Target Network		*Random*		*Irredundant*		*Optimistic*				*Optimal-in-expectation*			
						approximate		*closest*		*approximate*		*closest*	
$\|X\|$	$\|C\|$	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time
Random Binary Problem													
14	1	48	1	36	1	24	19	**24**	46	106	12	99	57
14	2	118	1	71	1	55	87	**50**	204	102	13	97	58
14	4	> 1000	1	729	1	101	237	94	573	81	19	**75**	63
14	14	> 1000	1	> 1000	1	235	412	219	918	72	23	**58**	67
14	40	> 1000	1	> 1000	1	298	1314	273	3048	71	27	**44**	66
Pattern Binary Problem													
14	14	> 1000	1	> 1000	1	220	17	197	34	42	45	**32**	76
Sudoku 4×4													
16	72	> 1000	1	> 1000	1	178	154	168	186	69	31	**57**	82
Schur's lemma													
6	6	88	1	27	1	21	167	**19**	382	24	198	23	432
8	12	298	1	66	1	56	274	51	772	46	218	**44**	563

for both we consider the *approximate* and the *closest* variants. Each column is divided in two parts. The left part is the number of queries needed to converge on the target network; a limit was set 1000 queries. The right part measures the average time needed to compute a query.

With the exception of very sparse random problems and Schur's Lemma, generating queries with *Random* is never able to converge on the target hypothesis, even with a large number of queries. The *Irredundant* approach is strictly better than *Random* and successfully converged in a number of cases. However, when the density of the target network increases, *Irredundant* begins to struggle to converge.

Optimistic and *Optimal-in-expectation* are more accurate, since they always enable us to converge, regardless of the target network used. Their *closest* variants require an average computation time between 2 and 5 times longer than the *approximate* ones, as to be expected. However, the closest strategies have the advantage of being able to converge on the target network by asking up to 40% fewer queries than the approximate strategies. *Optimistic* is the best approach on very sparse networks, but as the number of constraints in the target network grows, *Optimal-in-expectation* becomes the best strategy, since it requires both fewer queries to converge and less computation time. The number of queries for *Optimal-in-expectation* decreases when density increases because redundancy rules apply more frequently, deriving more constraints. Despite this, *Optimistic* performance decays when density increases because the probability that a query is classified negative (unlucky case) grows with density.

7 Related Work

Recently, researchers have become interested in techniques that can be used to acquire constraint networks in situations where a precise statement of the constraints of the problem is not available [Freuder and Wallace, 1998; Rossi and Sperduti, 2004]. The use of version space learning [Mitchell, 1982] as a basis for constraint acquisition has received most attention from the constraints community [O'Connell *et al.*, 2003], but the problem of query generation for acquiring constraint networks has not been studied.

8 Conclusion

In this paper we have tackled the question of how a constraint acquisition system, based on CONACQ, can help improve the interactive acquisition process by seeking fewer, but better selected, examples to be proposed as queries for classification by a user. We have provided a theoretical and empirical evaluation of query generation strategies for interactive constraint acquisition, with very positive results.

Acknowledgments

The authors would like to thank Frederic Koriche for very useful discussions and comments. This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

References

[Angluin, 2004] D. Angluin. Queries revisited. *Theoretical Computer Science*, 313:175–194, 2004.

[Bessiere *et al.*, 2005] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. Acquiring constraint networks using a SAT-based version space algorithm. In *ECML*, pages 23–34, 2005.

[Freuder and Wallace, 1998] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of CP-1998*, pages 192–204, 1998.

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

[O'Connell *et al.*, 2003] S. O'Connell, B. O'Sullivan, and E.C. Freuder. A study of query generation strategies for interactive constraint acquisition. In *Applications and Science in Soft Computing*, pages 225–232. 2003.

[Rossi and Sperduti, 2004] F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.