Constraint Acquisition via Partial Queries*

Christian Bessiere[1] **Remi Coletta**[1] **Emmanuel Hebrard**[2] **George Katsirelos**[3]
Nadjib Lazaar[1] **Nina Narodytska**[4] **Claude-Guy Quimper**[5] **Toby Walsh**[4]
[1]CNRS, U. Montpellier, France [2]LAAS-CNRS, Toulouse, France [3]INRA Toulouse, France
[4]NICTA, UNSW, Sydney, Australia [5]U. Laval, Quebec City, Canada

Abstract

We learn constraint networks by asking the user partial queries. That is, we ask the user to classify assignments to subsets of the variables as positive or negative. We provide an algorithm that, given a negative example, focuses onto a constraint of the target network in a number of queries logarithmic in the size of the example. We give information theoretic lower bounds for learning some simple classes of constraint networks and show that our generic algorithm is optimal in some cases. Finally we evaluate our algorithm on some benchmarks.

1 Introduction

A major bottleneck in the use of constraint solvers is modelling. How does the user write down the constraints of a problem? Several techniques have been proposed to tackle this bottleneck. For example, the matchmaker agent [Freuder and Wallace, 1998] interactively asks the user to provide one of the constraints of the target problem each time the system proposes an incorrect solution. In *Conacq.1* [Bessiere *et al.*, 2004; 2005], the user provides examples of solutions and non-solutions. Based on these examples, the system learns a set of constraints that correctly classifies all examples given so far. This is a form *passive* learning. In [Lallouet *et al.*, 2010], a system based on inductive logic programming uses background knowledge on the structure of the problem to learn a representation of the problem correctly classifying the examples. A last passive learner is *ModelSeeker* [Beldiceanu and Simonis, 2012]. Positive examples are provided by the user to the system which arranges each of them as a matrix and identifies constraints in the global constraints catalog that are satisfied by rows or columns of all examples. One weakness of passive learning is that the user needs to provide diverse examples for the target set of constraints to be learned.

By contrast, in an *active learner* like *Conacq.2* [Bessiere *et al.*, 2007], the system proposes examples to the user to classify as solutions or non solutions. Such questions are called *membership queries* [Angluin, 1987]. Such active learning has several advantages. It can decrease the number of examples necessary to converge to the target set of constraints. Another advantage is that the user need not be a human. It might be a previous system developed to solve the problem. For instance, the Normind company has recently hired a constraint programming specialist to transform their expert system for detecting failures in electric circuits in Airbus airplanes in to a constraint model in order to make it more efficient and easier to maintain. As another example, active learning is used to build a constraint model that encodes non-atomic actions of a robot (e.g., catch a ball) by asking queries of the simulator of the robot in [Paulin *et al.*, 2008]. Such active learning introduces two computational challenges. First, how does the system generate a useful query? Second, how many queries are needed for the system to converge to the target set of constraints? It has been shown that the number of membership queries required to converge to the target set of constraints can be exponentially large [Bessiere and Koriche, 2012].

In this paper, we propose QUACQ (for QuickAcquisition), an active learner that asks the user to classify *partial* queries. Given a negative example, QUACQ is able to learn a constraint of the target constraint network in a number of queries logarithmic in the number of variables. In fact, we identify information theoretic lower bounds on the complexity of learning constraint networks which show that QUACQ is optimal on some simple languages. We demonstrate the promise of this approach in practice with some experimental results.

One application for QUACQ would be to learn a general purpose model. In constraint programming, a distinction is made between model and data. For example, in a sudoku puzzle, the model contains generic constraints like each subsquare contains a permutation of the numbers. The data, on the other hand, gives the pre-filled squares for a specific puzzle. As a second example, in a time-tabling problem, the model specifies generic constraints like no teacher can teach multiple classes at the same time. The data, on the other hand, specifies particular room sizes, and teacher availability for a particular time-tabling problem instance. The cost of learning the model can then be amortized over the lifetime of the model. Another advantage of this approach is that it provides less of a burden on the user. First, it often converges quicker than other methods. Second, partial queries will be easier to answer than complete queries. Third, as opposed to existing techniques, the user does not need to give positive examples. This might be useful if the problem has not yet been solved,

*This work has been partially funded by the ANR project BR4CP (ANR-11-BS02-008) and by the EU project ICON (FP7-284715).

so there are no examples of past solutions.

2 Background

The learner and the user need to share some common knowledge to communicate. We suppose this common knowledge, called the *vocabulary*, is a (finite) set of n variables X and a domain $D = \{D(X_i)\}_{X_i \in X}$, where $D(X_i) \subset \mathbb{Z}$ is the finite set of values for X_i. Given a sequence of variables $S \subseteq X$, a *constraint* is a pair (c, S) (also written c_S), where c is a relation over $\mathbb{Z}$ specifying which sequences of $|S|$ values are allowed for the variables S. S is called the *scope* of c_S. A *constraint network* is a set C of constraints on the vocabulary (X, D). An assignment e_Y on a set of variables $Y \subseteq X$ *is rejected by* a constraint c_S if $S \subseteq Y$ and the projection $e_Y[S]$ of e on the variables in S is not in c. An assignment on X is a *solution* of C iff it is not rejected by any constraint in C. We write $sol(C)$ for the set of solutions of C, and $C[Y]$ for the set of constraints from C whose scope is included in Y.

Adapting terms from machine learning, the *constraint bias*, denoted by B, is a set of constraints built from the constraint language Γ on the vocabulary (X, D) from which the learner builds the constraint network. A *concept* is a Boolean function over $D^X = \Pi_{X_i \in X} D(X_i)$, that is, a map that assigns to each $e \in D^X$ a value in $\{0, 1\}$. A *target concept* is a concept f_T that returns 1 for e if and only if e is a solution of the problem the user has in mind. A *membership query* $ASK(e)$ is a classification question asked of the user whether e is a *complete* assignment in D^X. The answer to $ASK(e)$ is "yes" if and only if $f_T(e) = 1$.

To be able to use *partial queries*, we have an extra condition on the capabilities of the user. Even if she is not able to articulate the constraints of her problem, she is able to decide if partial assignments of X violate some requirements or not. The *target model* or *target constraint network* is a network C_T such that $sol(C_T) = \{e \in D^X \mid f_T(e) = 1\}$. A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, is a classification question asked of the user, where e_Y is a *partial* assignment in $D^Y = \Pi_{X_i \in Y} D(X_i)$. A set of constraints C *accepts* a partial query e_Y if and only if there does not exist any constraint c_S in C rejecting $e_Y[S]$. The answer to $ASK(e_Y)$ is "yes" if and only if C_T accepts e_Y. For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y. A classified assignment e_Y is called positive or negative *example* depending on whether $ASK(e_Y)$ is "yes" or "no".

We now define *convergence*, which is the constraint acquisition problem we are interested in. We are given a set E of (partial) examples labelled by the user 0 or 1. We say that a constraint network C agrees with E if C accepts all examples labelled 1 in E and does not accept those labelled 0. The learning process has *converged* on the network $C_L \subseteq B$ if C_L agrees with E and for every other network $C' \subseteq B$ agreeing with E, we have $sol(C') = sol(C_L)$. If there does not exist any $C_L \subseteq B$ such that C_L agrees with E, we say that we have *collapsed*. This happens when $C_T \not\subseteq B$.

3 Generic Constraint Acquisition Algorithm

We propose QUACQ, a novel active learning algorithm. QUACQ takes as input a bias B on a vocabulary (X, D). It asks partial queries of the user until it has converged on a constraint network C_L equivalent to the target network C_T, or collapses. When a query is answered yes, constraints rejecting it are removed from B. When a query is answered no, QUACQ enters a loop (functions `FindScope` and `FindC`) that will end by the addition of a constraint to C_L.

3.1 Description of QUACQ

QUACQ (see Algorithm 1) initializes the network C_L it will learn to the empty set (line 1). If C_L is unsatisfiable (line 3), the space of possible networks collapses because there does not exist any subset of the given bias B that is able to correctly classify the examples already asked of the user. In line 4, QUACQ computes a complete assignment e satisfying C_L but violating at least one constraint from B. If such an example does not exist (line 5), then all constraints in B are implied by C_L, and we have converged. If we have not converged, we propose the example e to the user, who will answer by yes or no. If the answer is yes, we can remove from B the set $\kappa_B(e)$ of all constraints in B that reject e (line 6). If the answer is no, we are sure that e violates at least one constraint of the target network C_T. We then call the function `FindScope` to discover the scope of one of these violated constraints. `FindC` will select which one with the given scope is violated by e (line 8). If no constraint is returned (line 9), this is again a condition for collapsing as we could not find in B a constraint rejecting one of the negative examples. Otherwise, the constraint returned by `FindC` is added to the learned network C_L (line 10).

Algorithm 1: QUACQ: Acquiring a constraint network C_T with partial queries

```
1  C_L ← ∅;
2  while true do
3      if sol(C_L) = ∅ then return "collapse";
4      choose e in D^X accepted by C_L and rejected by B;
5      if e = nil then return "convergence on C_L";
6      if ASK(e) = yes then B ← B \ κ_B(e) ;
7      else
8          c ← FindC(e, FindScope(e, ∅, X, false));
9          if c = nil then return "collapse";
10         else C_L ← C_L ∪ {c};
```

The recursive function `FindScope` takes as parameters an example e, two sets R and Y of variables, and a Boolean ask_query. An invariant of `FindScope` is that e violates at least one constraint whose scope is a subset of $R \cup Y$. When `FindScope` is called with $ask\_qery =$ **false**, we already know whether R contains the scope of a constraint that rejects e (line 1). If $ask\_qery =$ **true** we ask the user whether $e[R]$ is positive or not (line 2). If yes, we can remove all the constraints that reject $e[R]$ from the bias, otherwise we return the empty set (line 3). We reach line 4 only in case $e[R]$ does not violate any constraint. We know that $e[R \cup Y]$ violates a constraint. Hence, as Y is a singleton, the variable it contains necessarily belongs to the scope of a constraint that violates

Algorithm 2: Function `FindScope`: returns the scope of a constraint in C_T

```
function FindScope(in e, R, Y, ask_query): scope;
begin
1   if ask_query then
2       if ASK(e[R]) = yes then B ← B \ κ_B(e[R]);
3       else return ∅;
4   if |Y| = 1 then return Y;
5   split Y into < Y_1, Y_2 > such that |Y_1| = ⌈|Y|/2⌉ ;
6   S_1 ← FindScope(e, R ∪ Y_1, Y_2, true);
7   S_2 ← FindScope(e, R ∪ S_1, Y_1, (S_1 ≠ ∅));
8   return S_1 ∪ S_2;
end
```

Algorithm 3: Function `FindC`: returns a constraint of C_T with scope Y

```
function FindC(in e, Y): constraint;
begin
1   B ← B \ {c_Y | C_L ⊨ c_Y};
2   Δ ← {c_Y ∈ B[Y] | e ⊭ c_Y};
3   if Δ = ∅ then return ∅;
4   while true do
5       choose e' in sol(C_L[Y]) such that
        ∃c, c' ∈ Δ, e' ⊨ c and e' ⊭ c';
6       if e' = nil then
7           if Δ = ∅ then return nil;
8           else pick c in Δ; return c;
9       if ASK(e') = yes then
10          B ← B \ κ_B(e'); Δ ← Δ \ κ_B(e');
11      else Δ ← Δ ∩ κ_B(e');
end
```

$e[R \cup Y]$. The function returns Y. If none of the return conditions are satisfied, the set Y is split in two balanced parts (line 5) and we apply a technique similar[1] to QUICKXPLAIN ([Junker, 2004]) to elucidate the variables of a constraint violating $e[R \cup Y]$ in a logarithmic number of steps (lines 6–8).

The function `FindC` takes as parameter e and Y, e being the negative example that led `FindScope` to find that there is a constraint from the target network C_T over the scope Y. `FindC` first removes from B all constraints with scope Y that are implied by C_L because there is no need to learn them (line 1).[2] The set Δ is initialized to all candidate constraints violated by e (line 2). If Δ no longer contains constraints with scope Y (line 3), we return $\varnothing$, which will provoke a collapse in QUACQ. In line 5, an example e' is chosen in such a way that Δ contains both constraints rejecting e' and constraints satisfying e'. If no such example exists (line 6), this means that all constraints in Δ are equivalent wrt $C_L[Y]$. Any of them is returned except if Δ is empty (lines 7-8). If a suitable example was found, it is proposed to the user for classification (line 9). If classified positive, all constraints rejecting it are removed from B and Δ (line 10), otherwise we remove from Δ all constraints accepting that example (line 11).

3.2 Example

We illustrate the behavior of QUACQ on a simple example. Consider the set of variables $X_1, \ldots, X_5$ with domains $\{1..5\}$, a language $\Gamma = \{=, \neq\}$, a bias $B = \{=_{ij}, \neq_{ij} | i, j \in 1..5, i < j\}$, and a target network $C_T = \{=_{15}, \neq_{34}\}$. Suppose the first example generated in line 4 of QUACQ is $e_1 = (1, 1, 1, 1, 1)$. The trace of the execution of `FindScope`$(e_1, \varnothing, X_1 \ldots X_5,$ **false**$)$ is in the table below. Each line corresponds to a call to `FindScope`. Queries are always on the variables in R. '$\times$' in the column ASK means that the previous call returned $\varnothing$, so the question is skipped. The queries in lines 1 and 2.1 in the table permit `FindScope` to remove the constraints $\neq_{12}, \neq_{13}, \neq_{23}$ and $\neq_{14}, \neq_{24}$ from B. Once the scope (X_3, X_4) is returned, `FindC` requires a single example to return $\neq_{34}$ and prune $=_{34}$ from B. Suppose the next example generated by QUACQ is $e_2 = (1, 2, 3, 4, 5)$. `FindScope` will find the scope (X_1, X_5) and `FindC` will return $=_{15}$ in a way similar to the processing of e_1. The constraints $=_{12}, =_{13}, =_{14}, =_{23}, =_{24}$ are removed from B by a partial positive query on $X_1, \ldots, X_4$ and $\neq_{15}$ by `FindC`. Finally, examples $e_3 = (1, 1, 1, 2, 1)$ and $e_4 = (3, 2, 2, 3, 3)$, both positive, will prune $\neq_{25}, \neq_{35}, =_{45}$ and $=_{25}, =_{35}, \neq_{45}$ from B respectively, leading to convergence.

call	R	Y	ASK	*return*
0	$\varnothing$	X_1, X_2, X_3, X_4, X_5	$\times$	X_3, X_4
1	X_1, X_2, X_3	X_4, X_5	yes	X_4
1.1	X_1, X_2, X_3, X_4	X_5	no	$\varnothing$
1.2	X_1, X_2, X_3	X_4	$\times$	X_4
2	X_4	X_1, X_2, X_3	yes	X_3
2.1	X_4, X_1, X_2	X_3	yes	X_3
2.2	X_4, X_3	X_1, X_2	no	$\varnothing$

3.3 Analysis

We analyse the complexity of QUACQ in terms of the number of queries it can ask of the user. Queries are proposed to the user in lines 6 of QUACQ, 2 of `FindScope` and 9 of `FindC`.

Proposition 1. *Given a bias B built from a language Γ, a target network C_T, a scope Y, `FindC` uses $O(|\Gamma|)$ queries to return a constraint c_Y from C_T if it exists.*

Proof. Each time `FindC` asks a query, whatever the answer of the user, the size of Δ strictly decreases. Thus the total number of queries asked in `FindC` is bounded above by $|\Delta|$, which itself, by construction in line 2, is bounded above by the number of constraints from Γ of arity $|Y|$. □

Proposition 2. *Given a bias B, a target network C_T, an example $e \in D^X \setminus sol(C_T)$, `FindScope` uses $O(|S| \cdot \log |X|)$ queries to return the scope S of one of the constraints of C_T violated by e.*

Proof. `FindScope` is a recursive algorithm that asks at most one query per call (line 2). Hence, the number of queries is bounded above by the number of nodes of the tree of recursive calls to `FindScope`. We will show that a leaf node is

[1] The main difference is that QUACQ splits the set of variables whereas QUICKXPLAIN splits the set of constraints.

[2] This operation could proactively be done in QUACQ, just after line 10, but we preferred the lazy mode as this is a computationally expensive operation.

either on a branch that leads to the elucidation of a variable in the scope S that will be returned, or is a child of a node of such a branch. When a branch does not lead to the elucidation of a variable in the scope S that will be returned, that branch necessarily only leads to leaves that correspond to calls to `FindScope` that returned the empty set. The only way for a leaf call to `FindScope` to return the empty set is to have received a no answer to its query (line 3). Let R_{child}, Y_{child} be the values of the parameters R and Y for a leaf call with a no answer, and R_{parent}, Y_{parent} be the values of the parameters R and Y for its parent call in the recursive tree. From the no answer to the query $ASK(e[R_{child}])$, we know that $S \subseteq R_{child}$ but $S \nsubseteq R_{parent}$ because the parent call received a yes answer. Consider first the case where the leaf is the left child of the parent node. By construction, $R_{parent} \subsetneq R_{child} \subsetneq R_{parent} \cup Y_{parent}$. As a result, Y_{parent} intersects S, and the parent node is on a branch that leads to the elucidation of a variable in S. Consider now the case where the leaf is the right child of the parent node. As we are on a leaf, if the ask_query Boolean is false, we have necessarily exited from `FindScope` through line 4, which means that this node is the end of a branch leading to a variable in S. Thus, we are guaranteed that the ask_query Boolean is true, which means that the left child of the parent node returned a non empty set and that the parent node is on a branch to a leaf that elucidates a variable in S.

We have proved that every leaf is either on a branch that elucidates a variable in S or is a child of a node on such a branch. Hence the number of nodes in the tree is at most twice the number of nodes in branches that lead to the elucidation of a variable from S. Branches can be at most $\log |X|$ long. Therefore the total number of queries `FindScope` asks is at most $2 \cdot |S| \cdot \log |X|$, which is in $O(|S| \cdot \log |X|)$. □

Theorem 1. *Given a bias B built from a language Γ of bounded arity constraints, and a target network C_T,* QUACQ *uses $O(|C_T| \cdot (\log |X| + |\Gamma|))$ queries to find the target network or to collapse and $O(|B|)$ queries to prove convergence.*

Proof. Each time line 6 of QUACQ classifies an example as negative, the scope of a constraint c_S from C_T is found in at most $|S| \cdot \log |X|$ queries (Proposition 2). As Γ only contains constraints of bounded arity, either $|S|$ is bounded and c_S is found in $O(|\Gamma|)$ or we collapse (Proposition 1). Hence, the number of queries necessary for finding C_T or collapsing is in $O(|C_T| \cdot (\log |X| + |\Gamma|))$. Convergence is obtained once B is wiped out thanks to the examples that are classified positive in line 6 of QUACQ. Each of these examples necessarily leads to at least one constraint removal from B because of the way the example is built in line 4. This gives a total in $O(|B|)$. □

4 Learning Simple Languages

In order to gain a theoretical insight into the "efficiency" of QUACQ, we look at some simple languages, and analyze the number of queries required to learn networks on these languages. In some cases, we show that QUACQ will learn problems of a given language with an asymptotically optimal number of queries. However, for some other languages, a suboptimal number of queries can be necessary in the worst case. Our analysis assumes that when generating a complete example in line 4 of QUACQ, the solution of C_L *maximizing* the number of violated constraints in the bias B is chosen.

4.1 Languages for which QUACQ is optimal

Theorem 2. QUACQ *learns Boolean networks on the language $\{=, \neq\}$ in an asymptotically optimal number of queries.*

Proof. (Sketch.) First, we give a lower bound on the number of queries required to learn a constraint network in this language. Consider the restriction to equalities only. In an instance of this language, all variables of a connected component must be equal. This is isomorphic to the set of partitions of n objects, whose size is given by *Bell's Number*:

$$C(n+1) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=1}^{n} \binom{n}{i} C(n-i) & \text{if } n > 0 \end{cases} \quad (1)$$

By an information theoretic argument, at least $\log C(n)$ queries are required to learn such a problem. This entails a lower bound of $\Omega(n \log n)$ since $\log C(n) \in \Omega(n \log n)$ (see [De Bruijn, 1970] for the proof). The language $\{=, \neq\}$ is richer and thus requires at least as many queries.

Second, we consider the query submitted to the user in line 6 of QUACQ and count how many times it can receive the answer yes and no. The key observation is that an instance of this language contains at most $O(n)$ non-redundant constraints. For each no answer in line 6 of QUACQ, a new constraint will eventually be added to C_L. Only non-redundant constraints are discovered in this way because the query must satisfy C_L. It follows that at most $O(n)$ such queries are answered no, each one entailing $O(\log n)$ more queries through the procedure `FindScope`.

Now we bound the number of yes answers in line 6 of QUACQ. The same observation on the structure of this language is useful here as well. We show in the complete proof that a query maximizing the number of violations of constraints in the bias B while satisfying the constraints in C_L violates at least $\lceil |B|/2 \rceil$ constraints in B. Thus, each query answered yes at least halves the number of constraints in B. It follows that the query submitted in line 6 of QUACQ cannot receive more than $O(\log n)$ yes answers. The total number of queries is therefore bounded by $O(n \log n)$. □

The same argument holds for simpler languages ($\{=\}$ and $\{\neq\}$ on Boolean domains). Moreover, this is still true for $\{=\}$ on arbitrary domains.

Corollary 1. QUACQ *can learn constraint networks with unbounded domains on the language $\{=\}$ in an asymptotically optimal number of queries.*

4.2 Languages for which QUACQ is not optimal

First, we show that a Boolean constraint network on the language $\{<\}$ can be learnt with $O(n)$ queries. Then, we show that QUACQ requires $\Omega(n \log n)$ queries.

Theorem 3. *Boolean constraint networks on the language $\{<\}$ can be learned in $O(n)$ queries.*

Proof. Observe that in order to describe such a problem, the variables can be partionned into three sets, one for variables that must take the value 0 (i.e., on the left side of a $<$ constraint), a second for variables that must take the value 1 (i.e., on the right side of a $<$ constraint), and the third for unconstrained variables. In the first phase, we greedily partition variables into three sets, L, R, U initially empty and standing respectively for *Left*, *Right* and *Unknown*. During this phase, we have three invariants:

1. There is no $x, y \in U$ such that $x < y$ belongs to the target network
2. $x \in L$ iff there exists $y \in U$ and a constraint $x < y$ in the target network
3. $x \in R$ iff there exists $y \in U$ and a constraint $y < x$ in the target network

We go through all variables of the problem, one at a time. Let x be the last variable picked. We query the user with an assignment where x, as well as all variables in U are set to 0, and all variables in R are set to 1 (variables in L are left unassigned). If the answer is yes, then there is no constraints between x and any variable in $y \in U$, hence we add x to the set of undecided variables U without breaking any invariant. Otherwise we know that x is either involved in a constraint $y < x$ with $y \in U$, or a constraint $x < y$ with $y \in U$. In order to decide which way is correct, we make a second query, where the value of x is flipped to 1 and all other variables are left unchanged. If this second query receives a yes answer, then the former hypothesis is true and we add x to R, otherwise, we add it to L. Here again, the invariants still hold.

At the end of the first phase, we therefore know that variables in U have no constraints between them. However, they might be involved in constraints with variables in L or in R. In the second phase, we go over each undecided variable $x \in U$, and query the user with an assignment where all variables in L are set to 0, all variables in R are set to 1 and x is set to 0. If the answer is no, we conclude that there is a constraint $y < x$ with $y \in L$ and therefore x is added to R (and removed from U). Otherwise, we ask the same query, but with the value of x flipped to 1. If the answer is no, there must exists $y \in R$ such that $x < y$ belongs to the network, hence x is added to R (and removed from U). Last, if both queries get the answer yes, we conclude that x is not constrained. When every variable has been examined in this way, variables remaining in U are not constrained. □

Theorem 4. QUACQ *does not learn Boolean networks on the language* $\{<\}$ *with a minimal number of queries.*

Proof. By Theorem 3, we know that these networks can be learned in $O(n)$ queries. Such networks can contain up to $n-1$ non redundant constraints. QUACQ learns constraints one at a time, and each call to `FindScope` takes $\Omega(\log n)$ queries. Therefore, QUACQ requires $\Omega(n \log n)$ queries. □

5 Experimental Evaluation

To test the practicality of such active learning, we ran some experiments with QUACQ. on an Intel Xeon E5462 @ 2.80GHz with 16 Gb of RAM. We used several benchmarks.

Random. We generated binary random target networks with 50 variables, domains of size 10, and m binary constraints. The binary constraints are selected from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. QUACQ is initialized with the bias B containing the complete graph of 7350 binary constraints taken from Γ. For densities $m = 12$ (under-constrained) and $m = 122$ (phase transition) we launched QUACQ on 100 instances and report averages.

Golomb rulers. (prob006 in [Gent and Walsh, 1999]) This is encoded as a target network with m variables corresponding to the m marks, and constraints of varying arity. We learned the target network encoding the 8 mark ruler. We initialized QUACQ with a bias of 770 constraints using the language $\Gamma = \{|x_i - x_j| \neq |x_k - x_l|, |x_i - x_j| = |x_k - x_l|, x_i < x_j, x_i \geq x_j\}$ including binary, ternary[3] and quaternary constraints.

Zebra problem. Lewis Carroll's zebra problem has a single solution. The target network is formulated using 25 variables of domain size of 5 with 5 cliques of $\neq$ constraints and 11 additional constraints given in the description of the problem. To check QUACQ on this problem, we fed it with a bias B of 4450 unary and binary constraints taken from a language with 24 basic arithmetic and distance constraints.

Sudoku. The target network of the Sudoku has 81 variables with domains of size 9 and 810 binary $\neq$ constraints on rows, columns and squares. Here we fed QUACQ with a bias B of 6480 binary constraints from the language $\Gamma = \{=, \neq\}$.

For all our experiments we report the size $|C_L|$ of the learned network (which can be smaller than the target network due to redundant constraints), the total number $\#q$ of queries, the number $\#q_c$ of complete queries (i.e., generated in line 6 of QUACQ), the average size $\bar{q}$ of all queries, and the average time needed to compute a query (in seconds).

5.1 QUACQ and convergence

To ensure quick converge, we want a query answered yes to prune B as much as possible. This is best achieved when the query generated in line 4 of QUACQ is an assignment violating a large number of constraints in B. We implemented the *max* heuristic to generate a solution of C_L that violates a maximum number of constraints from B. However, this heuristic can be time consuming as it solves an optimisation problem. We then added a cutoff of 1 or 10 seconds to the solver using *max*, which gives the two heuristics *max-1* and *max-10* respectively. We also implemented a cheaper heuristic that we call *sol*. It simply solves C_L and stops at its first solution violating at least one constraint from B.

Our first experiment was to compare *max-1* and *max-10* on large problems. We observed that the performance when using *max-1* is not significantly worse in number of queries than when using *max-10*. For instance, on the rand_50_10_122, $\#q = 1074$ for *max-1* and $\#q = 1005$ for *max-10*. The average time for generating a query is 0.14 seconds for *max-1* and 0.86 for *max-10* with a maximum of 1 and 10 seconds respectively. We then chose not to report results for *max-10*.

Table 1 reports the results obtained with QUACQ to reach convergence using the heuristics *max-1* and *sol*. A first obser-

[3]The ternary constraints are obtained when $i = k$ or $j = l$ in $|x_i - x_j| \neq |x_k - x_l|$.

Table 1: Results of QUACQ learning until convergence.

		$\|C_L\|$	$\#q$	$\#q_c$	$\bar{q}$	time
rand_50_10_12	*max-1*	12	196	34	24.04	0.23
	sol	12	286	133	33.22	0.09
rand_50_10_122	*max-1*	86	1074	94	13.90	0.14
	sol	83	1062	120	15.64	0.06
Golomb-8	*max-1*	91	488	101	5.12	0.32
	sol	138	709	153	5.31	0.25
Zebra	*max-1*	62	656	63	8.00	0.10
	sol	62	641	63	8.26	0.02
Sudoku 9×9	*max-1*	810	8645	821	20.58	0.16
	sol	810	9593	815	20.84	0.06

vation is that *max-1* generally requires less queries than *sol* to reach convergence. This is especially true for rand_50_10_12, which is very sparse, and Golomb-8, which contains many redundant constraints. If we have a closer look, these differences are mainly explained by the fact that *max-1* requires significantly less complete positive queries than *sol* to prune B totally and prove convergence (22 complete positive queries for *max-1* and 121 for *sol* on rand_50_10_12). But in general, *sol* is not as bad as we could have expected. The reason is that, except on very sparse networks, the number of constraints from B violated 'by chance' with *sol* is large enough. The second observation is that when the network contains a lot of redundancies, *max-1* converges on a smaller network than *sol*. We observed this on Golomb-8, and other problems not reported here. The third observation is that the average size of queries is always significantly smaller than the number of variables in the problem. A last observation is that *sol* is very fast for all its queries (see the time column). We can expect it to be usable on even larger problems.

As a second experiment we evaluated the effect of the size of the bias on the number of queries. On the zebra problem we fed QUACQ with biases of different sizes and stored the number of queries for each run. Figure 1 shows that when $|B|$ grows, the number of queries follows a logarithmic scale. This is very good news as it means that learning problems with expressive biases will scale well.

QUACQ has two main advantages over learning with membership queries, as in CONACQ. One is the small average size of queries $\bar{q}$, which are probably easier to answer by the user. The second advantage is the time to generate queries. *Conacq.2* needs to find examples that violate exactly one constraint of the bias to make progress towards convergence. This can be expensive to compute, preventing the use of *Conacq.2* on large problems. QUACQ, on the other hand, can use cheap heuristics like *max-1* and *sol* to generate queries.

5.2 QUACQ as a solver

Conacq.2 and *ModelSeeker* need complete positive examples to learn a constraint network. By comparison, QUACQ can learn without any complete positive example. This extends its use to solving a single instance of a problem never solved before. We simply need to exit QUACQ as soon as a complete example is classified yes by the user. We assessed this feature by solving a sequence of 5 *instances* of Sudoku, that

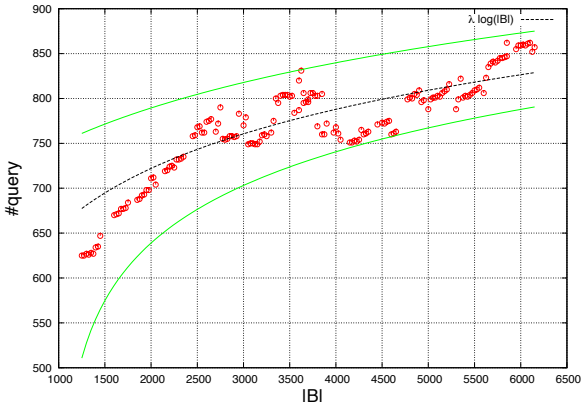

Figure 1: QUACQ behavior on different bias sizes for Zebra

is, Sudoku grids with their pre-filled cells. For each of them, QUACQ exits when the solution is found. As the goal is not to converge we replaced *max-1* by a *min-1* heuristic, that is, a heuristic that tries to satisfy as much as possible the constraints in B, with a cutoff at 1 second. Each run takes as input the C_L and B in the state they were at the end of the previous run because the partially learned network is a valid starting point for further learning. The number of queries required to solve each of the 5 instances in the sequence was 3859, 1521, 687, 135, and 34 respectively. The size of C_L after each run was 340, 482, 547, 558, and 561, respectively. We see that for the first grid, where QUACQ starts with a complete bias, we find the solution in only 44% of the queries needed to QUACQ to converge (See Table 1). On each run, QUACQ needs fewer queries to find a solution as C_L becomes closer to the target network.

6 Conclusion

We have proposed QUACQ, an algorithm that learns constraint networks by asking the user to classify partial assignments as positive or negative. Each time it receives a negative example, the algorithm converges on a constraint of the target network in a logarithmic number of queries. We have shown that QUACQ is optimal on certain constraint languages. Our approach has several advantages over existing work. First, it always converges on the target constraint network in a polynomial number of queries. Second, the queries are often much shorter than membership queries, so are easier to handle for the user. Third, as opposed to existing techniques, the user does not need to provide positive examples to converge. This last feature can be very useful when the problem has not been previously solved. Our experimental evaluation shows that generating good queries in QUACQ is not computationally difficult and that when the bias increases in size, the increase in number of queries follows a logarithmic shape. These results are promising for the use of QUACQ on real problems. However, problems with dense constraint networks (as Sudoku) require a number of queries that could be too large for a human user. An interesting direction would be to combine *ModelSeeker* and QUACQ to quickly learn global constraints and use QUACQ to finalize the model.

References

[Angluin, 1987] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[Beldiceanu and Simonis, 2012] N. Beldiceanu and H. Simonis. A model seeker: Extracting global constraint models from positive examples. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'12), LNCS 7514, Springer–Verlag*, pages 141–157, Quebec City, Canada, 2012.

[Bessiere and Koriche, 2012] C. Bessiere and F. Koriche. Non learnability of constraint networks with membership queries. Technical report, Coconut, Montpellier, France, February 2012.

[Bessiere *et al.*, 2004] C. Bessiere, R. Coletta, E. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04), LNCS 3258, Springer–Verlag*, pages 123–137, Toronto, Canada, 2004.

[Bessiere *et al.*, 2005] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of the European Conference on Machine Learning (ECML'05), LNAI 3720, Springer–Verlag*, pages 23–34, Porto, Portugal, 2005.

[Bessiere *et al.*, 2007] C. Bessiere, R. Coletta, B O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 44–49, Hyderabad, India, 2007.

[De Bruijn, 1970] N.G. De Bruijn. *Asymptotic Methods in Analysis*. Dover Books on Mathematics. Dover Publications, 1970.

[Freuder and Wallace, 1998] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98), LNCS 1520, Springer–Verlag*, pages 192–204, Pisa, Italy, 1998.

[Gent and Walsh, 1999] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. http://www.csplib.org/, 1999.

[Junker, 2004] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, San Jose CA, 2004.

[Lallouet *et al.*, 2010] A. Lallouet, M. Lopez, L. Martin, and C. Vrain. On learning constraint problems. In *Proceedings of the 22nd IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'10)*, pages 45–52, Arras, France, 2010.

[Paulin *et al.*, 2008] M. Paulin, C. Bessiere, and J. Sallantin. Automatic design of robot behaviors through constraint network acquisition. In *Proceedings of the 20th IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'08)*, pages 275–282, Dayton OH, 2008.