New Approaches to Constraint Acquisition

Christian Bessiere, Abderrazak Daoudi, Emmanuel Hébrard, George Katsirelos, Nadjib Lazaar, Younes Mechqrane, Nina Narodytska, Claude-Guy Quimper, Toby Walsh

New Approaches to Constraint Acquisition*

Christian Bessiere[1], Abderrazak Daoudi[1,2], Emmanuel Hebrard[3], George Katsirelos[4], Nadjib Lazaar[1], Younes Mechqrane[2], Nina Narodytska[5], Claude-Guy Quimper[6] and Toby Walsh[7]

[1]CNRS, University of Montpellier, France
[2]University Mohammed V of Rabat, Morocco
[3]LAAS-CNRS, Toulouse, France
[4]INRA Toulouse, France
[5]Samsung Research America, USA
[6]University Laval, Quebec City, Canada
[7]NICTA, UNSW, Sydney, Australia

Abstract. In this chapter we present the recent results on constraint acquisition obtained by the Coconut team and their collaborators. In a first part we show how to 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, called QUACQ, that, given a negative example, finds a constraint of the target network in a number of queries logarithmic in the size of the example. In a second part, we show that using some background knowledge may improve the acquisition process a lot. We introduce the concept of generalization query based on an aggregation of variables into types. We propose a generalization algorithm together with several strategies that we incorporate in QUACQ. Finally we evaluate our algorithms 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 [13] 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* [4, 5], 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 of *passive* learning. In [16], 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* [3]. 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 ([2]) that are satisfied by particular subsets of variables in all the examples. Such particular subsets are for instance rows, columns, diagonals, etc.

* Sections 3 and 4 of this paper describe material published in [9], Section 5 describes material published in [8], and Section 6 describes results coming from both of these two papers. The whole work has been funded by the European Community project FP7-284715 ICON.

An efficient ranking technique combined with a representation of solutions as matrices allows *ModelSeeker* to find quickly a good model when a problem has an underlying matrix structure.

By contrast, in an *active learner* like *Conacq.2* [6], the system proposes examples to the user to classify as solutions or non solutions. Such questions are called *membership queries* [1]. 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 needs not be a human. It might be a previous system developed to solve the problem. For instance, the Normind company has hired a constraint programming specialist to transform their expert system for detecting failures in electric circuits in Airbus airplanes into 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 [18]. 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 [7].

In this chapter, 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. However, even that good theoretical bounds can be hard to put in practice. For instance, QUACQ requires the user to classify more than 8000 examples to get the complete Sudoku model. We then propose a new technique to make constraint acquisition more efficient in practice by using variable types. In real problems, variables often represent components of the problem that can be classified in various types. For instance, in a school time-tabling problem, variables can represent teachers, students, rooms, courses, or time-slots. Such types are often known by the user. To deal with types of variables, we introduce a new kind of query, namely, *generalization query*. We expect the user to be able to decide if a learned constraint can be generalized to other scopes of variables of the same type as those in the learned constraint. We propose an algorithm, GENACQ for *generalized acquisition*, that asks such generalization queries each time a new constraint is learned. We propose several strategies and heuristics to select the good candidate generalization query. We plugged our generalization functionality into the QUACQ constraint acquisition system, leading to the G-QUACQ algorithm. We experimentally evaluate the benefits of our algorithms on several benchmark problems. The results show the polynomial behavior of QUACQ. They also show that G-QUACQ dramatically improves the basic QUACQ algorithm in terms of number of queries.

One application for QUACQ and G-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, so there are no examples of past solutions.

The rest of the paper is organized as follows. Section 2 gives the necessary definitions to understand the technical presentation. Section 3 presents QUACQ, the algorithm that learns constraint networks by asking partial queries. In Section 4, we show how QUACQ behaves on some simple languages. Section 5 describes the generalization algorithm G-QUACQ. In Section 5.5, several strategies are presented to make G-QUACQ more efficient. Section 6 presents the experimental results we obtained when comparing G-QUACQ to the basic QUACQ and when comparing the different strategies in G-QUACQ. Section 7 concludes the paper and gives some directions for future research.

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 $D = \{D(X_1), \ldots, D(X_n)\}$, 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.

In addition to the vocabulary, the learner owns a *language* Γ of bounded arity relations from which it can build constraints on specified sets of variables. To simplify the descriptions, we only consider languages closed by conjunction, that is, languages Γ such that if relations c_1 and c_2 belong to Γ, then relation $c_1 \cap c_2$ also belongs to Γ. Adapting terms from machine learning, the *constraint basis*, 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. Formally speaking, $B = \{c_S \mid (S \subseteq X) \wedge (c \in \Gamma) \wedge arity(c) = |S|)\}$.

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. e is called a positive or negative *example* depending on whether $f_T(e) = 1$ or $f_T(e) = 0$. A

membership query $ASK(e)$ takes as input a *complete* assignment e in D^X and asks the user to classify it. 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. More formally, we consider that the user has in mind her problem in the form of a target constraint network. A *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. It is important to observe that "$ASK(e_Y) = yes$" does not mean that e_Y extends to a solution of C_T, which would put an NP-complete problem on the shoulders of the user. 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 $C_T \not\subseteq B$.

Finally, we define types of variables to be used to generalize constraints. A *type* T_i is a subset of variables defined by the user as having a common property. A variable X_i is of type T_i iff $X_i \in T_i$. A scope $S = (X_1, \ldots, X_k)$ of variables *belongs* to a sequence of types $T = (T_1, \ldots, T_k)$ (denoted by $S \in T$) if and only if $X_i \in T_i$ for all $i \in 1..k$. Consider $T = (T_1, \ldots, T_k)$ and $T' = (T'_1, \ldots, T'_k)$ two sequences of types. We say that T' *covers* T (denoted by $T \sqsubseteq T'$) iff $T_i \subseteq T'_i$ for all $i \in 1..k$. A relation c *holds* on a sequence of types T if and only if $c_S \in C_T$ for all $S \in T$. A sequence of types T is *maximal* with respect to a relation c if and only if c holds on T and there does not exist T' covering T on which c holds.

3 Constraint Acquisition with Partial Queries

We propose QUACQ, a novel active learning algorithm. QUACQ takes as input a basis 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

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          S ← FindScope(e, ∅, X, false);
9          c_S ← FindC(S);
10         if c_S = nil then return "collapse";
11         else C_L ← C_L ∪ {c_S};
```

there does not exist any subset of the given basis B that is able to correctly classify the examples the user has already been asked. In line 4, QUACQ computes a complete assignment e satisfying C_L but violating at least one constraint from B. (Observe that for this task, the constraint solver needs to be able to express the negation of the constraints in B. This is not a problem as we have only bounded arity constraints in 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 S of one of these violated constraints (line 8). `FindC` will return a constraint of C_T whose scope is in S (line 9). If no constraint is returned (line 10), 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 11).

The recursive function `FindScope` (see Algorithm 2) 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 $R \cup Y$. A second invariant is that `FindScope` always returns a subset of Y that is also the subset of the scope of a constraint violated by e. When `FindScope` is called with $ask\_query =$ **false**, we already know that R does not contain the scope of a constraint that rejects e (line 3). If $ask\_qery =$ **true** we ask the user whether $e[R]$ is positive or not (line 4). If yes, we can remove all the constraints that reject $e[R]$ from the basis, otherwise we return the empty set because we have no guarantee that any variable of Y belongs to a scope (line 5). We reach line 6 only in case $e[R]$ does not violate any constraint. We know that $e[R \cup Y]$ violates a constraint. Hence, if Y is a singleton, the variable it contains necessarily belongs to the scope of a constraint that violates $e[R \cup Y]$. The function returns Y. If none of the return conditions are satisfied, the set Y is split in two balanced parts (line 7) and we apply a technique similar to QUICKXPLAIN ([15]) to elucidate the variables of a constraint violating $e[R \cup Y]$ in a logarithmic number of steps (lines 8–10). The rationale of lines 8 and 9 is to quickly find sets R and Y so that

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

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

$e[R]$ is positive and $e[R \cup Y]$ is negative. If Y is a singleton this ensures that the variable in Y belongs to the scope of a constraint we are looking for. This variable is returned and forced to be in R in all subsequent calls to `FindScope`.

The function `FindC` (see Algorithm 3) takes as parameter Y, the scope on which `FindScope` has found that there is a constraint from the target network C_T. `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 3).[1] The set Δ is initialized to all candidate constraints (line 4). In line 6, 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 7), this means that all constraints in Δ are equivalent wrt $C_L[Y]$. Any of them is returned except if Δ is empty (lines 8-9). If a suitable example was found, it is proposed to the user for classification (line 10). If classified positive, all constraints rejecting it are removed from B and Δ (line 11). Otherwise we test whether the example e does not violate constraints with scope strictly included in Y (line 13). If yes, we recursively call `FindScope` and `FindC` to find that smaller arity constraint before the one having scope Y (line 14). If no, we remove from Δ all constraints accepting that example e (line 15) and we continue the loop of line 5.

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 basis $B = \{=_{ij}, \neq_{ij} | i, j \in 1..5, i < j\}$, 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 Table 1. 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 initial call (call-0 in Table 1) does not ask the question because the initial call to `FindScope` has the Boolean ask_query set to **false**. Y is split in two sets $Y_1 = \{X_1, X_2, X_3\}$ and $Y_2 =$

[1] This operation could proactively be done in QUACQ, just after line 11, but we preferred the lazy mode as this is a computationally expensive operation.

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

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

$\{X_4, X_5\}$ and the recursive call-1 is performed with $R = Y_1$ and $Y = Y_2$. As $e_1[R]$ is classified as positive, line 4 of `FindScope` removes the constraints $\neq_{12}, \neq_{13}$ and $\neq_{23}$ from B. A new split of Y leads to the call-1.1 with $R = \{X_1, X_2, X_3, X_4\}$ and $Y = \{X_5\}$. As $e_1[R]$ is negative, the empty set is returned in line 5. Call-1.2 (line 9) is performed with $R = \{X_1, X_2, X_3\}$, $Y = \{X_4\}$ and ask_query = **true**. It merely detects that Y is a singleton and thus returns $\{X_4\}$. Call-1 finishes by returning $\{X_4\}$ one level above in the recursivity (line 10). Call-2 classifies $e_1[X_4]$ as positive and goes down to call-2.1 with $R = \{X_4, X_1, X_2\}$ and $Y = \{X_3\}$. In call-2.1, $e_1[R]$ is classified positive. `FindScope` thus removes constraints $\neq_{14}$ and $\neq_{24}$ from B and returns the singleton $\{X_3\}$. In call-2.2, $e_1[R]$ is classified as negative with $R = \{X_4, X_3\}$ and $Y = \{X_1, X_2\}$. This proves that $\{X_3, X_4\}$ is the scope of a constraint rejecting e_1. Empty set is returned by call-2.2. As a result, call-2 returns $\{X_3\}$, and call-0 returns $\{X_3, X_4\}$. 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

Table 1. The example

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	$\varnothing$

$e_2 = (1, 2, 3, 4, 5)$. `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.

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, 4 of `FindScope` and 10 of `FindC`.

Proposition 1. *Given a basis 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 4, is bounded above by the number of constraints from Γ of arity $|Y|$.

Proposition 2. *Given a basis 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 4). 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 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 5). 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 6, 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 basis 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 basis 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 [12] 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 basis 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 Constraint Acquisition with Generalization Queries

In this section we present GENACQ, a *generalized acquisition* algorithm, The idea behind this algorithm is, given a constraint c_S learned on S, to generalize this constraint to sequences of types T covering S by asking generalization queries $AskGen(T, c)$. A generalization query $AskGen(T, c)$ is answered *yes* by the user if and only if for every sequence S of variables covered by T the relation c holds on S in the target constraint network C_T.

5.1 Description of GENACQ

The algorithm GENACQ (see Algorithm 4) takes as input a target constraint c_S that has already been learned and a set $NonTarget$ of constraints that are known not to belong to the target network. GENACQ returns the set of all sequences of scopes that are maximal with respect to the relation c. GENACQ uses the global data structure $NegativeQ$, which is a set of pairs (T, c) for which we know that c does not hold on all sequences of variables covered by T. c_S and $NonTarget$ can come from any constraint acquisition mechanism or as background knowledge. $NegativeQ$ is built incrementally by each call to GENACQ. GENACQ also uses the set $Table$ as local data structure. $Table$ will contain all sequences of types that are candidates for generalizing c_S.

In line 2, GENACQ initializes the set $Table$ to all possible sequences T of types that contain the scope S of the constraint c_S. In line 3, GENACQ initializes the set G to the sequence S. G will contain the output of GENACQ, that is, the set of maximal sequences from $Table$ on which c holds. The counter $\#NoAnswers$ counts the number of consecutive times generalization queries have been classified negative by the user. It is initialized to zero (line 4). $\#NoAnswers$ is not used in the basic version of GENACQ

Algorithm 4: GENACQ: returns the maximum generalizations of a constraint c_S

```
1  function GENACQ(in c_S: constraint, NonTarget: set): Generalizations;
2  Table ← {T | S ∈ T} \ {S};
3  G ← {S} ;
4  #NoAnswers ← 0 ;
5  foreach T ∈ Table do
6      if ∃(T', c') ∈ NegativeQ | c ⊆ c' ∧ T' ⊑ T then Table ← Table \ {T} ;
7      if ∃c_S' ∈ NonTarget | S' ∈ T then Table ← Table \ {T} ;
8  while Table ≠ ∅ ∧ #NoAnswers < cutoffNo do
9      pick T in Table ;
10     if AskGen(T, c) = yes then
11         G ← G ∪ {T} \ {T' ∈ G | T' ⊑ T};
12         Table ← Table \ {T' ∈ Table | T' ⊑ T};
13         #NoAnswers ← 0;
14     else
15         Table ← Table \ {T' ∈ Table | T ⊑ T'} ;
16         NegativeQ ← NegativeQ ∪ {(T, c)} ;
17         #NoAnswers + +;
18 return G;
```

but it will be used in the version with cutoffs. (In other words, the basic version uses $\mathsf{cutoffNo} = +\infty$ in line 8).

The first loop in GENACQ (line 5) eliminates from $Table$ all sequences T for which we already know the answer to the query $AskGen(T, c)$. In line 6, GENACQ eliminates from $Table$ all sequences T such that a relation c' entailed by c is already known not to hold on a sequence T' covered by T (i.e., (T', c') is in $NegativeQ$). We can remove such sequences because the absence of c' on some scope in T' implies the absence of c on some scope in T (see Lemma 1). In line 7, GENACQ eliminates from $Table$ all sequences T such that we know from $NonTarget$ that there exists a scope S' in T such that $c_{S'}$ does not belong to C_T.

In the main loop of GENACQ (line 8), we pick a sequence T from $Table$ at each iteration and we ask a generalization query to the user (line 10). If the user says *yes*, T is a sequence on which c holds. We put T in G and remove from G all sequences covered by T, so as to keep only the maximal ones (line 11). We also remove from $Table$ all sequences T' covered by T (line 12) to avoid asking redundant questions later. If the user says *no*, we remove from $Table$ all sequences T' that cover T (line 15) because we know they are no longer candidate for generalization of c and we store in $NegativeQ$ the fact that (T, c) has been answered *no*. The loop finishes when $Table$ is empty and we return G (line 18).

5.2 Completeness and Complexity

We analyze the completeness and complexity of GENACQ in terms of number of generalization queries it ask of the user.

Lemma 1. *If $AskGen(T, c) = no$ then for any (T', c') such that $T \sqsubseteq T'$ and $c' \subseteq c$, we have $AskGen(T', c') = no$.*

Proof. Assume that $AskGen(T, c) = no$. Hence, there exists a sequence $S \in T$ such that $c_S \notin C_T$. As $T \sqsubseteq T'$ we have $S \in T'$ and then we know that $c_S \notin C_T$. As $c' \subseteq c$, we also have $c'_S \notin C_T$. As a result, $AskGen(T', c') = no$.

Lemma 2. *If $AskGen(T, c) = yes$ then for any T' such that $T' \sqsubseteq T$, we have $AskGen(T', c) = yes$.*

Proof. Assume that $AskGen(T, c) = yes$. As $T' \sqsubseteq T$, for all $S \in T'$ we have $S \in T$ and then we know that $c_S \in C_T$. As a result, $AskGen(T', c) = yes$.

Proposition 3 (Completeness). *When called with constraint c_S as input, the algorithm* GENACQ *returns all maximal sequences of types covering S on which the relation c holds.*

Proof. All sequences covering S are put in $Table$. A sequence in $Table$ is either asked for generalization or removed from $Table$ in lines 6, 7, 12, or 15. We know from Lemma 1 that a sequence removed in line 6, 7, or 15 would necessarily lead to a *no* answer. We know from Lemma 2 that a sequence removed in line 12 is subsumed and less general than another one just added to G.

Proposition 4. *Given a learned constraint c_S and its associated $Table$,* GENACQ *uses $O(|Table|)$ generalization queries to return all maximal sequences of types covering S on which the relation c holds.*

Proof. For each query on $T \in Table$ asked by GENACQ, the size of $Table$ strictly decreases regardless of the answer. As a result, the total number of queries is bounded above by $|Table|$.

5.3 Illustrative Example

Let us take the Lewis Carroll's Zebra problem to illustrate our generalization approach. The problem is to find where the zebra lives, given five houses of five different colors, owned by five men of five different nationalities, having five different drinks, cigarets, and pets. (See for instance [10] for a complete description of the Zebra problem.) The Zebra problem has a single solution. The target network is formulated using 25 variables, partitioned in 5 types of 5 variables each. The ith variable of a given type represents the number of the house where the ith element of the given type is located. The types are *color*, *nationality*, *drink*, *cigaret*, *pet*, and the trivial type X of all variables. There is a clique of $\neq$ constraints on all pairs of variables of the same non trivial type and 14 additional constraints given in the description of the problem.

Figure 1 shows the variables of the Zebra problem and their types. In this example, the constraint $X_2 \neq X_5$ has been learned between the two color variables X_2 and X_5. This constraint is given as input of the GENACQ algorithm. GENACQ computes the $Table$ of all sequences of types covering the scope (X_2, X_5). $Table = \{(X_2, color), (X_2, X), (color, X_5), (color, color), (color, X), (X, X_5), (X, color), (X, X)\}$. Suppose

Fig. 1. Variables and types for the Zebra problem.

we pick $T = (X, X_5)$ at line 9 of GENACQ. According to the user's answer (*no* in this case), the $Table$ is reduced to $Table = \{(X_2, color), (X_2, X), (color, X_5), (color, color), (color, X)\}$. As next iteration, let us pick $T = (color, color)$. The user will answer *yes* because there is indeed a clique of $\neq$ on the $color$ variables. Hence, $(color, color)$ is added to G and the $Table$ is reduced to $Table = \{(X_2, X), (color, X)\}$. If we pick (X_2, X), the user answers *no* and we reduce the $Table$ to the empty set and return $G = \{(color, color)\}$, which means that the constraint $X_2 \neq X_5$ can be generalized to all pairs of variables in the sequence $(color, color)$, that is, $(X_i \neq X_j) \in C_T$ for all $(X_i, X_j) \in (color, color)$.

5.4 Using Generalization in QUACQ

GENACQ is a generic technique that can be plugged into any constraint acquisition system. In this section we present G-QUACQ, a constraint acquisition algorithm obtained by plugging GENACQ into QUACQ, G-QUACQ is presented in Algorithm 5.

G-QUACQ has a structure very similar to QUACQ. It initializes the set $NonTarget$ and 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 basis B that is able to correctly classify the examples the user has already been asked. 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* (line 6). If the answer is *yes*, we can remove from B the set $\kappa_B(e)$ of all constraints in B that reject e (line 7) and we add all these ruled out constraints to the set $NonTarget$ to be used in GENACQ (line 8). 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 10). If no constraint is returned (line 11),

Algorithm 5: G-QUACQ = QUACQ + GENACQ

```
1  C_L ← ∅, NonTarget ← ∅;
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
7          B ← B \ κ_B(e);
8          NonTarget ← NonTarget ∪ κ_B(e);
9      else
10         c_S ← FindC(e, FindScope(e, ∅, X, false));
11         if c_S = nil then return "collapse";
12         else
13             G ← GENACQ(c_S, NonTarget);
14             foreach T ∈ G do C_L ← C_L ∪ {c_S' | S' ∈ T} ;
```

this is again a condition for collapsing as we could not find in B a constraint rejecting one of the negative examples. Otherwise, we know that the constraint c_S returned by `FindC` belongs to the target network C_T. This is here that the algorithm differs from QUACQ as we call GENACQ to find all the maximal sequences of types covering S on which c holds. They are returned in G (line 13). Then, for every sequence of variables S' belonging to one of these sequences in G, we add the constraint $c_{S'}$ to the learned network C_L (line 14).

5.5 Strategies

GENACQ learns the maximal sequences of types on which a constraint can be generalized. The order in which sequences are picked from $Table$ in line 9 of Algorithm 4 is not specified by the algorithm. As shown on the following example, different orderings can lead more or less quickly to the good (maximal) sequences on which a relation c holds. Let us come back to our example on the Zebra problem (Section 5.3). In the way we developed the example, we needed only 3 generalization queries to empty the set $Table$ and converge on the maximal sequence $(color, color)$ on which $\neq$ holds:

1. $AskGen((X, X_5), \neq) = no$
2. $AskGen((color, color), \neq) = yes$
3. $AskGen((X_2, X), \neq) = no$

Using another ordering, GENACQ needs 8 generalization queries:

1. $AskGen((X, X), \neq) = no$
2. $AskGen((X, color), \neq) = no$
3. $AskGen((color, X), \neq) = no$
4. $AskGen((X, X_5), \neq) = no$
5. $AskGen((X_2, X), \neq) = no$
6. $AskGen((X_2, color), \neq) = yes$
7. $AskGen((color, X_5), \neq) = yes$

8. $AskGen((color, color), \neq) = yes$

If we want to reduce the number of generalization queries, we may wonder which strategy to use. In this section we propose two techniques. The first idea is to pick sequences in the set $Table$ following an order given by a heuristic that will try to minimize the number of queries. The second idea is to put a cutoff on the number of consecutive negative queries we accept to face, leading to a non complete generalization startegy: the output of GENACQ will no longer be guaranteed to be the *maximal* sequences.

Query Selection Heuristics We propose some query selection heuristics to decide which sequence to pick next from $Table$. We first propose *optimistic* heuristics, which try to take the best from positive answers:

- **`max_CST`:** This heuristic selects a sequence T maximizing the number of possible constraints c_S in the basis such that S is in T and c is the relation we try to generalize. The intuition is that if the user answers *yes*, the generalization will be maximal in terms of number of constraints.
- **`max_VAR`:** This heuristic selects a sequence T involving a maximum number of variables, that is, maximizing $|\bigcup_{S \in T} S|$. The intuition is that if the user answers *yes*, the generalization will involve many variables.

Dually, we propose *pessimistic* heuristics, which try to take the best from negative answers:

- **`min_CST`:** This heuristic selects a sequence T minimizing the number of possible constraints c_S in the basis such that S is in T and c is the relation we try to generalize. The intuition is to maximize the chances to receive a *yes* answer. If, despite this, the user answers *no*, a great number of sequences are removed from $Table$ (see Lemma 1).
- **`min_VAR`:** This heuristic selects a sequence T involving a minimum number of variables, that is, minimizing $|\bigcup_{S \in T} S|$. The intuition is to maximize the chances of a *yes* answer while focusing on smaller sets of variables than `min_CST`. Again, a *no* answer leads to a great number of sequences removed from $Table$.

As a baseline for comparison, we define a random selector.

- **`random`:** It picks randomly a sequence T in $Table$.

Using Cutoffs The idea here is to exit GENACQ before having proved the maximality of the sequences returned. We put a threshold cutoffNo on the number of consecutive negative answers to avoid using queries to check unpromising sequences. The hope is that GENACQ will return near-maximal sequences of types despite not proving maximality. This cutoff strategy is implemented by setting the variable cutoffNo to a predefined value. In lines 13 and 17 of GENACQ, a counter of consecutive negative answers is respectively reset and incremented depending on the answer from the user. In line 8, that counter is compared to cutoffNo to decide to exit or not.

6 Experimental Evaluation

We made some experiments to evaluate the behavior of active learning with partial queries (QUACQ) and to test the impact of using generalization (GENACQ). GENACQ was plugged in QUACQ, leading to G-QUACQ. All the experiments were done on an Intel Xeon E5462 @ 2.80GHz with 16 Gb of RAM.

We first present the benchmark problems we used for our experiments. Then, we report the results of several experiments. The first experiment presents the performance of QUACQ to learn our benchmark problems. The second one compares the performance of G-QUACQ to the basic QUACQ. The third reports experiments evaluating the different strategies we proposed (query selection heuristics and cutoffs) on G-QUACQ. Finally, we evaluate the performance of G-QUACQ when our knowledge of the types of variables is incomplete.

6.1 Benchmark Problems

Problems without types

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 basis 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 [14]) A Golomb ruler is a set of m marks to put on a ruler so that the distances between marks are all distinct. 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 and G-QUACQ with a basis of 55,484 constraints taken from a language with 24 basic arithmetic and distance constraints with unary, binary, ternary and quaternary scopes.

Problems with types

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. We initialized QUACQ and G-QUACQ with a basis of 6,850 constraints taken from a language with 24 basic arithmetic and distance constraints with unary, binary, ternary and quaternary scopes. In G-QUACQ, the variables are given as the 5 types of 5 variables that naturally occur from the problem description (color, nationality, pet, cigaret, drink).

Sudoku. The Sudoku logic puzzle is a 9×9 grid pre-filled with some numbers. It must be completed in such a way that all the rows, all the columns and the 9 non overlapping 3×3 squares contain the numbers 1 to 9. The target network of the Sudoku has 81 variables with domains of size 9 and 810 binary $\neq$ constraints on rows, columns and squares. QUACQ and G-QUACQ are initialized with a basis B of 6,480 binary constraints from the language $\Gamma = \{=, \neq\}$. In this problem, the types are the 9 rows, 9 columns and 9 squares, of 9 variables each.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. For this problem, we use 25 variables with domains of size 5 and 100 binary $\neq$ constraints on rows and columns. Rows and columns are the types of variables (10 types). QUACQ and G-QUACQ are initialized with a basis of constraints based on the language $\Gamma = \{=, \neq\}$.

Radio Link Frequency Assignment Problem. The RLFAP problem is to provide communication channels from limited spectral resources [11]. The constraint model of the instance we selected has 25 variables with domains of size 25 and 125 binary constraints. We have five stations of five terminals (transmitters/receivers), which form five types. We initialized QUACQ and G-QUACQ with a basis of 1,800 binary constraints taken from a language of 6 arithmetic and distance constraints

Purdey [17]. Like Zebra, this problem has a single solution. Four families have stopped by Purdeys general store, each to buy a different item and paying differently. Under a set of additional constraints given in the description, the problem is how can we match family with the item they bought and how they paid for it. The target network of Purdey has 12 variables with domains of size 4 and 30 binary constraints. Here we have three types of variables, which are *family*, *bought* and *paid*, each of them contains four variables. We initialized QUACQ and G-QUACQ with a basis of constraints based on the language $\Gamma = \{=, \neq\}$.

6.2 QUACQ evaluation

To ensure rapid 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 optimization 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, $\#Ask = 1074$ for *max-1* and $\#Ask = 1005$ for *max-10*. The average time for generating a query

Table 2. Results of QUACQ learning until convergence.

		$\lvert C_L \rvert$	$\#Ask$	$\#Ask_c$	$\overline{Ask}$	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*	83	438	83	5.12	0.85
	sol	127	645	132	5.34	0.46
Zebra	*max-1*	60	764	61	10.99	0.29
	sol	60	752	61	11.17	0.06
Sudoku 9×9	*max-1*	810	8645	821	20.58	0.16
	sol	810	9593	815	20.84	0.06

is 0.14 seconds for *max-1* and 0.86 seconds for *max-10* with a maximum of 1 and 10 seconds respectively. We then chose not to report results for *max-10*.

Table 2 reports the results obtained with QUACQ to reach convergence using the heuristics *max-1* and *sol*. 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 $\#Ask$ of queries, the number $\#Ask_c$ of complete queries (i.e., generated in line 6 of QUACQ), the average size $\overline{Ask}$ of all queries, and the average time needed to compute a query (in seconds). A first observation 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 totally prune B 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 basis on the number of queries. On the zebra problem we initialized QUACQ with bases of different sizes and stored the number of queries for each run. Figure 2 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 bases will scale well.

QUACQ has two main advantages over learning with membership queries, as in CONACQ. One is the small average size of queries $\overline{Ask}$, 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 basis to make progress

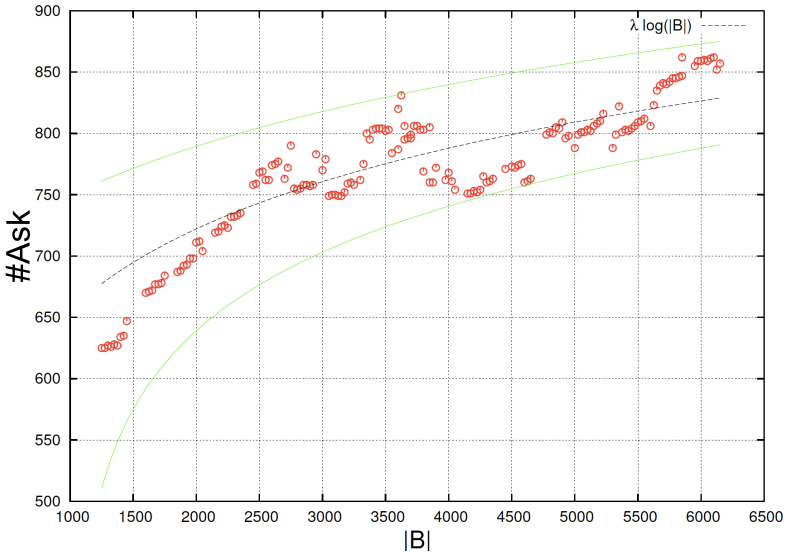

Fig. 2. QUACQ behavior on different basis sizes for Zebra

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.

6.3 Using Generalization Queries

For all our experiments we report, the total number $\#Ask$ of standard queries asked by the basic QUACQ, the total number $\#AskGen$ of generalization queries, and the numbers $\#no$ and $\#yes$ of negative and positive generalization queries, respectively, where $\#AskGen = \#no + \#yes$. The time overhead of using G-QUACQ rather than QUACQ is not reported. Computing a generalization query takes a few milliseconds.

Table 3. QUACQ vs G-QUACQ.

	QUACQ	G-QUACQ +`random`	
	$\#Ask$	$\#Ask$	$\#AskGen$
Zebra	764	257	67
Sudoku	8645	260	166
Latin square	1129	117	60
RFLAP	1653	151	37
Purdey	173	82	31

Our first experiment compares QUACQ and G-QUACQ in its baseline version, G-QUACQ + `random`, on our benchmark problems. Table 3 reports the results. We observe that the number of queries asked by G-QUACQ is dramatically reduced compared to QUACQ. This is especially true on problems with many types involving many variables, such as Sudoku or Latin square. G-QUACQ acquires the Sudoku with 260 standard queries plus 166 generalization queries, when QUACQ acquires it in 8645 standard queries.

Table 4. G-QUACQ with heuristics and cutoff strategy on Sudoku.

	cutoff	$\#Ask$	$\#AskGen$	$\#yes$	$\#no$
random	$+\infty$	260	166	42	124
min_VAR			90	21	69
min_CST			132	63	69
max_VAR			263	63	200
max_CST			247	21	226
min_VAR	3	260	75	21	54
	2		57	21	36
	1		**39**	21	18
min_CST	3	626	238	112	126
	2	679	231	132	99
	1	837	213	153	60

Let us now focus on the behavior of our different heuristics in G-QUACQ. The upper part of Table 4 reports the results obtained with G-QUACQ using `min_VAR`, `min_CST`, `max_VAR`, and `max_CST` to acquire the Sudoku model. (Other problems showed similar trends.) The results clearly show that `max_VAR`, and `max_CST` are very bad heuristics. They are worse than the baseline `random`. On the contrary, `min_VAR` and `min_CST` significantly outperform `random`. They respectively require 90 and 132 generalization queries instead of 166 for `random`. Notice that they all ask the same number of standard queries (260) as they all find the same maximal sets of sequences for each learned constraint.

The lower part of Table 4 we compare the behavior of our two best heuristics (`min_VAR` and `min_CST`) when combined with the cutoff strategy. We tried all values of the cutoff from 1 to 3. A first observation is that `min_VAR` remains the best whatever the value of the cutoff is. Interestingly, even with a cutoff equal to 1, `min_VAR` requires the same number of standard queries as the versions of G-QUACQ without cutoff. This means that using `min_VAR` as selection heuristic in $Table$, G-QUACQ is able to return the maximal sequences despite being stopped after the first negative generalization answer. We also observe that the number of generalization queries with `min_VAR` decreases when the cutoff becomes smaller (from 90 to 39 when the cutoff goes from $+\infty$ to 1). By looking at the last two columns we see that this is the number $\#no$ of negative answers which decreases. The good performance of `min_VAR` + cutoff=1 can thus be explained by the fact that `min_VAR` selects first queries that cover a minimum number

of variables, which increases the chances to have a *yes* answer. Finally, we observe that the heuristic `min_CST` does not have the same nice characteristics as `min_VAR`. The smaller the cutoff, the more standard queries are needed, not compensating for the saving in number of generalization queries (from 260 to 837 standard queries for `min_CST` when the cutoff goes from $+\infty$ to 1). This means that with `min_CST`, when the cutoff becomes too small, GENACQ does not return the maximal sequences of types where the learned constraint holds.

Table 5. G-QUACQ with `random`, `min_VAR`, and cutoff=1 on Zebra, Latin square, RLFAP, and Purdey.

		$\#Ask$	$\#AskGen$	$\#yes$	$\#no$
Zebra	**Random**	257	67	10	57
	min_VAR		48	5	43
	min_VAR +cutoff=1		**23**	5	18
L.Square	**Random**	117	60	16	44
	min_VAR		34	10	24
	min_VAR +cutoff=1		**20**	10	10
RLFAP	**Random**	151	37	16	21
	min_VAR		41	14	27
	min_VAR +cutoff=1		**22**	14	8
Purdey	**Random**	82	31	5	26
	min_VAR		24	3	21
	min_VAR +cutoff=1		**12**	3	9

In Table 5, we report the performance of G-QUACQ with `random`, `min_VAR` and `min_VAR` +cutoff=1 on all the other problems. We see that `min_VAR` +cutoff=1 significantly improve the performance of G-QUACQ on all problems. As in the case of Sudoku, we observe that `min_VAR` +cutoff=1 does not lead to an increase in the number of standard queries. This means that on all these problems `min_VAR` +cutoff=1 always returns the maximal sequences while asking less generalization queries with negative answers.

From these experiments we see that G-QUACQ with `min_VAR` +cutoff=1 leads to tremendous savings in number of queries compared to QUACQ: 257+23 instead of 764 on Zebra, 260+39 instead of 8645 on Sudoku, 117+20 instead of 1129 on Latin square, 151+22 instead of 1653 on RLFAP, 82+12 instead of 173 on Purdey.

In our last experiment, we show the effect on the performance of G-QUACQ of a lack of knowledge on some variable types. We took again our 5 benchmark problems in which we have varied the amount of types known by the algorithm. This simulates a situation where the user does not know that some variables are from the same type. For instance, in Sudoku, the user could not have noticed that variables are arranged in columns. Figure 3 shows the number of standard queries and generalization queries asked by G-QUACQ with `min_VAR` +cutoff=1 to learn the RLFAP model when fed with an increasingly more accurate knowledge of types. We observe that as soon as a

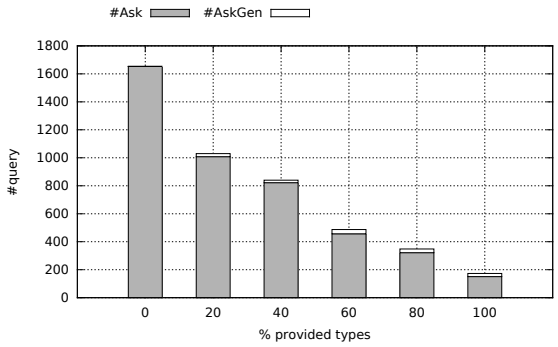

Fig. 3. G-QUACQ on RLFAP when the percentage of provided types increases.

small percentage of types is known (20%), G-QUACQ reduces drastically its number of queries. Table 6 gives the same information for all other problems.

7 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. Asking the user to classify partial assignments allows to converge on the target constraint network in a polynomial number of queries. Furthermore, 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. We have also proposed GENACQ, a technique to make constraint acquisition more efficient in practice by using information on the types of components the variables in the problem represent. We have introduced generalization queries. They are asked to the user to generalize a constraint to other scopes of variables of the same type where this constraint possibly applies. GENACQ can be called to generalize each new constraint that is learned. We have proposed several heuristics and strategies to select the good candidate generalization query. We have plugged GENACQ into the QUACQ constraint acquisition system, leading to the G-QUACQ version. Our experimental evaluation shows that generating good queries in QUACQ is not computationally difficult and that when the basis 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 require a number of queries that could be too large for a human user. We have then evaluated the benefit of generalization queries, with and without complete knowledge on the types of variables. The results show that G-QUACQ dramatically improves the basic QUACQ algorithm in terms of number of queries.

Table 6. G-QUACQ when the percentage of provided types increases.

	% of types	$\#Ask$	$\#AskGen$
Zebra	0	764	0
	20	619	12
	40	529	20
	60	417	27
	80	332	40
	100	257	48
Sudoku 9×9	0	8645	0
	33	3583	232
	66	610	60
	100	260	39
Latin Square	0	1129	0
	50	469	49
	100	117	20
Purdey	0	173	0
	33	111	8
	66	100	10
	100	82	12

References

1. Angluin, D.: Queries and concept learning. Machine Learning 2(4), 319–342 (1987)
2. Beldiceanu, N., Carlsson, M., Rampon, J.: Global constraint catalog. Tech. Rep. T2005:08, Swedish Institute of Computer Science, Kista, Sweden (May 2005)
3. Beldiceanu, N., Simonis, H.: 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. pp. 141–157. Quebec City, Canada (2012)
4. Bessiere, C., Coletta, R., Freuder, E., O'Sullivan, B.: 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. pp. 123–137. Toronto, Canada (2004)
5. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: 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. pp. 23–34. Porto, Portugal (2005)
6. Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M.: Query-driven constraint acquisition. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 44–49. Hyderabad, India (2007)
7. Bessiere, C., Koriche, F.: Non learnability of constraint networks with membership queries. Tech. rep., Coconut, Montpellier, France (February 2012)
8. Bessiere, C., Coletta, R., Daoudi, A., Lazaar, N., Mechqrane, Y., Bouyakhf, E.: Boosting constraint acquisition via generalization queries. In: Proceedings of the 21st European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 263, pp. 99–104. IOS Press, Prague, Czech Republic (2014)

9. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Walsh, T.: Constraint acquisition via partial queries. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence. pp. 475–481. IJCAI/AAAI, Beijing, China (2013)
10. Bessiere, C., Cordier, M.: Arc-consistency and arc-consistency again. In: Proceedings of the 11th National Conference on Artificial Intelligence. pp. 108–113. AAAI Press / The MIT Press, Washington DC (1993)
11. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.P.: Radio link frequency assignment. Constraints 4(1), 79–89 (1999)
12. De Bruijn, N.: Asymptotic Methods in Analysis. Dover Books on Mathematics, Dover Publications (1970)
13. Freuder, E., Wallace, R.: 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. pp. 192–204. Pisa, Italy (1998)
14. Gent, I., Walsh, T.: Csplib: a benchmark library for constraints. http://www.csplib.org/ (1999)
15. Junker, U.: Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04). pp. 167–172. San Jose CA (2004)
16. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: Proceedings of the 22nd IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'10). pp. 45–52. Arras, France (2010)
17. Mason, J.: Purdey's general store. Dell Magazine 54, 10–10 (April 1997)
18. Paulin, M., Bessiere, C., Sallantin, J.: 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). pp. 275–282. Dayton OH (2008)