Constraint Acquisition via Partial Queries

keywords: Constraints, Modeling

Abstract

We propose to learn constraint networks by asking the user partial queries, that is, to classify assignments on subsets of the variables as positive or negative. We provide an algorithm that given a negative example is able to focus on a constraint of the target network in a number of queries logarithmic in the size of the example. We give lower bounds for learning some simple classes of constraint networks and we show that our generic algorithm is optimal on some of them. Finally we evaluate our algorithm on some benchmark problems.

1 Introduction

Constraint programming has been used in many application areas, such as resource allocation or scheduling. The success of constraint programming comes from its efficiency to solve combinatorial problems, but also from the ease to consider an instance of a problem as a combination of two separate aspects: the model and the data. For example, in a time-tabling problem, the model specifies all generic constraints about non-ubiquity of teachers and capacity of rooms. This remains available over the weeks/years. The data identifies specific teachers availability and unexpected repairs in a room for a particular instance. In a Sudoku problem, the model contains all constraints specifying that two cells in the same row, column, or 3x3 square, should take different numbers. The data are the pre-filled cells of the current instance to solve. Thanks to this separation between model and data, the user simply specifies the new data each time she wants to solve another instance of the same problem. However, the design of the original model, that is, the construction of the set of constraints defining the problem, remains a delicate task that requires some expertise in constraint programming. This bottleneck has a negative effect on the uptake of constraint technology by non-experts.

There already exist several techniques to tackle this bottleneck. [Freuder and Wallace, 1998] proposed the matchmaker agent, an interactive process where the user is able to provide one of the constraints of her target problem each time the system proposes a wrong solution. In [Bessiere *et al.*, 2004; 2005], the assumption is made that the only thing the user is able to provide is examples of solutions (e.g., past time-tables designed by hand) and non-solutions of the target problem. Based on these examples, the CONACQ system learns a set of constraints that correctly classifies all examples given so far. This type of learning is called *passive* learning. The set of constraints which are learned are taken from a given language of constraints that only contains constraints of bounded arity. [Beldiceanu and Simonis, 2012] have proposed *Model Seeker*, another passive learning approach. Positive examples are provided by the user to the learner, which looks for global constraints from the global constraints catalog ([Beldiceanu *et al.*, 2005]) that accept these examples. An efficient ranking technique combined with a representation of solutions as matrices allows Model Seeker to quickly find a good model when a problem has an underlying matrix structure. The weakness of the passive learning approach is that the user is not always able to provide enough informative examples (i.e., sufficiently heterogeneous) to let the learner learn the target set of constraints.

In [Bessiere *et al.*, 2007], the learning process is made *active* by allowing the learner to propose examples to the user and to ask her to classify them as solutions or non solutions of her problem. Such questions are called *membership queries* [Angluin, 1987]. An advantage is to decrease the number of examples necessary to converge to the target set of constraints. Another advantage is the possibility to have a non-human user in front of the learner. This happens when a problem is expressed in a computer format that for some reason is no longer appropriate for the task to be solved. For instance, in [Paulin *et al.*, 2008], the learner builds a CP model that encodes non-atomic actions of a robot (e.g., catch a ball) by asking queries to the simulator of the robot. The active learning approach with membership queries has two main weaknesses. First, it can be computationally hard to generate a query that will allow the learner to learn useful information from the answer of the user. Second, it has recently been shown in [Bessiere and Koriche, 2012] that the number of queries required to converge to the target set of constraints can be exponentially large in the worst case.

In this paper, we propose QUACQ (for QuickAcquisition), a learner that is able to ask the user to classify partial queries. As opposed to membership queries, partial queries are assignments of only part of the variables of the problem. We describe the algorithm QUACQ. Given a negative example QUACQ is able to detect a constraint of the target constraint

network in a number of queries logarithmic in the number of variables. We give lower bounds to the complexity of learning constraint networks in some simple languages and we prove whether QUACQ is optimal or not. Finally, we give some experimental results.

Our approach has several advantages over existing work. First, it allows the learner to converge 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, it is not necessary that the user provides positive examples to converge. This last feature can be very useful when this is the first time the user faces her problem and does not have examples of past solutions.

2 Background

The learner and the user need to share some common knowledge to communicate. We consider that this common knowledge, called the *vocabulary*, is composed of 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 possible values for variable X_i. Given a sequence of variables $S \subseteq X$, a *constraint* is a pair (c, S) (also denoted by c_S), where c is a relation defined on $\mathbb{Z}$ that specifies which sequences of $|S|$ values are allowed for the variables S. The sequence of variables S is called the *scope* of c_S. A *constraint network* is a set C of constraints on a given 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 e on X is a *solution* of C iff it is not rejected by any constraint in C. We denote by $sol(C)$ the set of solutions of C. We denote by $C[Y]$ the set of constraints from C whose scope is included in Y, that is, $C[Y] = \{c_S \in C \mid S \subseteq Y\}$. .

In addition to the vocabulary, the learner has at its disposal a language Γ of bounded arity relations from which it can build constraints. A *constraint bias* is a collection B of constraints built from the constraint language Γ on the vocabulary (X, D).

Using terms from the machine learning world, 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 $\{0, 1\}$. We call *target concept* the 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 to the user, where e is a *complete* assignment in D^X. The answer to $ASK(e)$ is "yes" if and only if $e \in f_T$.

To be able to use what will be called *partial queries*, we have to put an extra condition on the capabilities of the user. Even if she is not able to articulate the constraints of her problem to design a CP model, 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 C_T such that $sol(C_T) = \{e \in D^X \mid f(e) = 1\}$. C_T is called the *target model* or *target constraint network*.

A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, is a classification question asked to 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. Given a labelled set E of (partial) examples labelled 1 or 0 depending on whether the user classified them 1 or 0 respectively. 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 can happen when $C_T \not\subseteq B$.

3 Generic Constraint Acquisition Algorithm

The QUACQ algorithm we present in this section is a learning algorithm that takes as input a bias B on a vocabulary (X, D). It asks queries to the user until it has converged on a constraint network C_L equivalent to the target network C_T or it has collapsed.

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 to 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. We have converged. If 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).

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**, this means that 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 from the bias all the constraints that reject $e[R]$, otherwise we return the emptyset (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,

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 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 conditions to return were satisfied, the set Y is split in two balanced parts (line 5) and we apply a technique close[1] to what is done in QUICKXPLAIN ([Junker, 2004]) to go logarithmically to the elucidation of the variables of a constraint violating $e[R \cup Y]$ (lines 6–8).

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
```

The function `FindC` takes as parameter e and Y, e being the negative example that led `FindScope` to find that there is for sure a constraint from the target network C_T on 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).

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
```

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.

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*	$\varnothing$

Queries asked in lines 1 and 2.1 in the table allow `FindScope` to remove from B the constraints $\neq_{12}, \neq_{13}, \neq_{23}$ and $\neq_{14}, \neq_{24}$ respectively. 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. 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, which leads to convergence.

[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.

3.3 Analysis

We analyse the complexity of QUACQ in terms of the number of queries it can ask to 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, given a target network C_T, given a scope Y, `FindC` requires $O(|\Gamma|)$ 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, given a target network C_T, given an example $e \in D^X \setminus sol(C_T)$, `FindScope` requires $O(|S| \cdot log|X|)$ 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 either is 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. Therefore, 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. By construction, branches can be at most $log|X|$ long, therefore, the total number of queries `FindScope` can ask 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, given a target network C_T,* QUACQ *requires $O(|C_T| \cdot (log|X| + |\Gamma|))$ 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 $O(|B|)$. □

A desirable feature of a learning algorithm is to be attribute-efficient, that is, to be linear in the size of the representation of the concept to learn, but logarithmic in the size of the bias. In the next section we consider several languages with different kinds of simpple arithmetic relations. We analyse the complexity of learning constraint networks on these languages and compare them to the complexity of QUACQ. In some cases it is attribute-efficient and optimal. On others it is not.

4 Complexity of Learning in Some Simple Languages

In order to get a theoretical insight on the "efficiency" of QUACQ, we look at simple languages, and analyze the number of queries required to learn networks on this language. In a number of cases, we are able to 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 might be necessary in the worst case. In this analysis, we assume that when generating a complete example in line 4 of QUACQ, the assignment *maximizing* the number of violated constraints in the bias B is chosen.

4.1 Languages for which QUACQ is optimal

Lemma 1. QUACQ *can learn Boolean a CSP 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. Therefore it is isomorphic to the set of partitions of n objects and its 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 a basic information theory argument, we know that at least $\log C(n)$ queries are required to learn such a problem. This entails a lower bound of $n \log n$ since $\log C(n) \in \Omega(n \log n)$ (see [De Bruijn, 1970] for the proof). Moreover, the language $\{=, \neq\}$ is richer and thus requires at least as many queries.

Second, we show that QUACQ can learn networks of this language in $O(n \log n)$ queries, hence being optimal, if the heuristic used in line 4 of QUACQ is to generate solutions of C_L that maximize the number of violations in B. 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. Variables may be clustered into equal components if they are linked by equality constraints, and two such components may be linked by an inequality constraint. Indeed, if there is a chain of more than two such components with inequalities, the k^{th} and $k + 2^{th}$ components can be merged because domains are Boolean.

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 the problem of computing a query maximizing the number of violations of constraints in the bias B while satisfying the constraints in C_L corresponds to the problem of partitioning a set of numbers in two such that the product of pairs of numbers of different partitions is maximized. The set of numbers is given by the set of sizes of components in C_L, and the product corresponds to the number of constraints that can be removed from B if components have not been assigned the same value. However, let $S = \{n_1, \ldots, n_k\}$ be a set of numbers in decreasing order. The partition int S_1, S_2 such that $S_{=}\{n_{2p+1} \mid 0 \leq p < \lceil k/2 \rceil\}$ and $S_2 = S \setminus S_1$ is such that: $\sum_{x \in S_1, y \in S_2} xy > \sum_{x<y \in S_1} xy + \sum_{x<y \in S_2} xy$. In other words, each query answered by a yes reduces the number of constraints in B by a factor at least 2. 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 $O(n \log n)$. □

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

Lemma 2. QUACQ *can learn a CSP on the language* $\{=\}$ *in an asymptotically optimal number of queries.*

Proof. On the language $\{=\}$, the proof above can be lifted to any domain size. Indeed, the linear bound on the number of non-redundant constraint is still true, and the analysis of the count of no answers does not change.

Moreover, consider the analysis of yes in the proof of Theorem 2, and assume that in the first query all variables are assigned the same value. It follows that all $\neq$ constraints are removed, and the rest of the proof is unchanged, except that the number partitioning problem has as many partitions as available values, hence making it easier to reduce the size of B. □

Lemma 3. QUACQ *can learn a CSP with unbounded domains on the language* $\{\neq\}$ *in an asymptotically optimal number of queries.*

Proof. With unbounded domains, it is not possible to infer a redundant constraints from a set of inequalities. Therefore, no two constraint graphs of inequalities are equivalent. The number of possible networks is therefore $2^{\binom{n}{2}}$. Consequently $\Omega(n^2)$ *yes*/*no* queries are required to learn such a network.

However, QUACQ can learn such a CSP in $O(|B|) = O(n^2)$ in the worst case, hence it is optimal. □

Theorem 2. QUACQ *can learn a CSP with on the language* $\{=, \neq\}$ *or any sublanguage in an asymptotically optimal number of queries both for Boolean and unbounded domains.*

Proof. Lemma 1 shows that this is true on $\{=, \neq\}$ for Boolean domains. Moreover, the lower bound argument is valid even if the language is restricted to either $\{=\}$ or $\{\neq\}$, hence the Theorem holds for Boolean domains.

Lemma 2 show that this is true on $\{=\}$ for any domain, and Lemma 3 for $\{\neq\}$ on unbounded domains. Now learning a CSP on $\{=, \neq\}$ is at least as hard as on $\{\neq\}$, hence the lower bound of $\Omega(n^2)$ is valid in this case as well.

However, QUACQ can learn such a CSP in $O(|B|) = O(n^2)$ in the worst case, hence it is optimal. □

Theorem 3. QUACQ *can learn a CSP with unbounded domains on the language* $\{<\}$ *in an asymptotically optimal number of queries.*

Proof. A CSP in this language will not be satisfiable unless the constraint graph is a directed acyclic graph (DAG). Moreover, all transitive closures of a given DAG will yield an equivalent problem. In other words there are as many possible CSP in this language as transitively closed DAG (or partial orders). However, logarithm of this number is in $\Theta(n^2)$ [Kleitman and Rothschild, 1970]. Therefore, we need $\Omega(n^2)$ *yes*/*no* queries to discover such a CSP.

However, QUACQ can learn such networks in $O(|B|) = O(n^2)$ in the worst case, hence it is optimal. □

4.2 Languages for which QUACQ is not optimal

First, we show that it is possible to learn a Boolean constraint network on the language $\{<\}$ with a linear number of queries. Then, we show that QUACQ requires at least $O(nlogn)$ queries.

Theorem 4. *Boolean constraint networks on the language* $\{<\}$ *can be learned in* $O(n)$ *queries.*

Proof. Observe that in order to describe such a problem, one can partition the variables into three sets, one for the variables that must take the value 0 (i.e., on the left side of a $<$ constraint), a second for the 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:

- There is no $x, y \in U$ such that $x < y$ belongs to the target network
- $x \in L$ iff there exists $y \in U$ and a constraint $x < y$ in the target network
- $x \in R$ 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 that we 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 do not have 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 can 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 can conclude that x is not constrained. When every variable has been examined in this way, the only variables remaining in U are those that are not constrained. □

Theorem 5. QUACQ *does not learn Boolean networks on the language* $\{<\}$ *with a minimal number of queries.*

Proof. By Theorem 4, 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. □

Table 1 gives a summary of the results of this section.

Table 1: Results summary

Langu.	D	LB	UB	QUACQ
$\{=\}$	$\{0,1\}$	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$
	$\mathbb{Z}$	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$
$\{\neq\}$	$\{0,1\}$	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$
	$\mathbb{Z}$	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$
$\{<\}$	$\{0,1\}$	$\Omega(n)$	$O(n)$	$O(n \log n)$
	$\mathbb{Z}$	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$
$\{=, \neq\}$	$\{0,1\}$	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$
	$\mathbb{Z}$	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$

5 Experimental Evaluation

In this Section, we present some experimental results obtained with QUACQ. Our experiments were conducted on Intel(R) Xeon(R) CPU E5462 @ 2.80GHz with 16 Go of RAM.

5.1 Benchmark problems

Random. We generated binary random target networks with 50 variables, domains of size 10, and m binary constraints. The binary constraints are selected in 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]) A Golomb ruler is a set of m marks to put on a ruler so that the distances between marks are all distinct. We encode it as a target network with m variables corresponding to the m marks. This problem formulation requires constraints of various arities. In our experiment, we learned the target network encoding the ruler with 8 marks. 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. The zebra problem (from Lewis Carroll 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 2650 unary and binary constraints. The bias is built from the language $\Gamma = \{=, \neq, neighbour(x, y, i), notNeighbour(x, y, i), next(x, y), notNext(x, y), prev(x, y), notPrev(x, y), At(x, i), notAt(x, i)\}$, where $i \in -4..4$.

Sudoku. The target network of the Sudoku problem 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 $\Gamma = \{=, \neq\}$.

5.2 Experimental protocol

We tried to assess several features of QUACQ. We first tried to see the effect of the heuristic we use to generate a complete example in line 4 of QUACQ. We propose three different heuristics:

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

Table 2: Results of QUACQ learning until convergence.

		$\|C_L\|$	$\#q$	$\#q_c$	$\bar{q}$	time
rand_50_10_12	*max*	11,96	195,93	34,36	24,04	0.23
	min	11,96	1280,97	1131,82	46,39	0.38
	sol	12	286,41	132,88	33,22	0.09
rand_50_10_122	*max*	86,39	1073,91	93,68	13,9	0.14
	min	82,91	1129,23	201,22	18,22	0.18
	sol	83,41	1061,9	119,6	15,64	0.06
Golomb-8	*max*	91	488	101	5.12	0.32
	min	106	547	116	5.36	0.29
	sol	138	709	153	5.31	0.25
Zebra	*max*	60	638	64	8.22	0.15
	min	59	626	54	8.72	0.11
	sol	60	634	63	8.20	0.02
Sudoku 9×9	*max*	810	8645	821	20.58	0.16
	min	810	9298	1472	27.36	0.16
	sol	810	9593	815	20.84	0.06

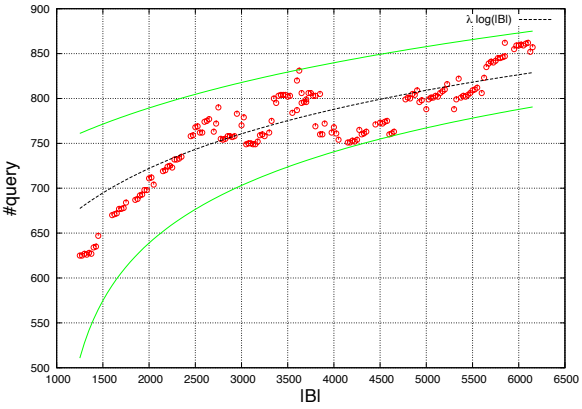

Figure 1: QUACQ behavior on different bias size of Zebra problem

max returns a complete example that violates a maximum number of the constraints in the bias B. This is expected to greatly reduce B when the example is classified as positive.

min returns a complete example that violates a minimum number of the constraints in the bias B, but at least one.

sol can be seen as a compromise between the *min* and the *max* heuristics. It returns a complete example that violates at least one constraint of the bias. It has the advantage to be very fast to compute.

In the following tables, we report the size $|C_L|$ of the learned network (which can be smaller than the target network if it contains redundant constraints), the total number $\#q$ of queries asked to the user, the number $\#q_c$ of queries that were complete (i.e., generated in line 6 of QUACQ), the average size $\bar{q}$ of all queries, and the average time needed to compute one query (in seconds). After a few preliminary tests, we set for minimization (resp. maximization) a cutoff of $1s$.

5.3 Results

Table 2 reports results for each class of problem using the three heuristics *max*, *min*, and *sol*.

On random problems, we see that the different heuristics do not behave the same. In terms of time consumption, the *sol* heuristic is much faster as it does not solve an optimization problem at each loop. In terms of number of queries, the *max* heuristic works well on sparse and dense networks. This heuristic asks less complete queries and, therefore, leads us to good average size. Let us take the first random instance, to reach a convergence state, the *max* heuristic needs 22 complete positive query (out-of 34) where the *min* needs more than 1120 (out-of 1131) . Here, a *max* query is fifty times more powerful than a *min* query in terms of bias reduction. We recall that QUACQ, regardless the used heuristic, learn a constraint from each negative complete query. In the light of this result, QUACQ can ensure convergence using *min*, *max* or *sol* heuristic.

In CONACQ, the convergence can be got iff we have a query-generator that violate, at each time, exactly one constraint of the bias. This allows CONACQto be faster on the convergence of small instances (bias and/or N_T), but it remains a strong assumption on big instances.

The Golomb ruler has the characteristic of containing a great number of redundant constraints. Here also the *max* heuristic returns the smallest network that formulates the problem and it requires significantly less queries than *min* and *sol*.

On the zebra problem, *min* is the one that performs the best in terms of queries, though the differences are tiny. Here, the three heuristics generate more or less the same complete queries. The negative ones allow QUACQ to learns zebra constraints, once the zebra single solution is captured by N_L, the three heuristics will have the same behaviour on constraint acquisition (same behaviour on bias reduction). It is important to stress that though a complete query is of size 25, the a verage size using QUACQ is only around 8. On the zebra problem we also tested how much the number of queries grows when the size of the bias increases. We fed QUACQ with biases of different sizes and stored the number of queries for each run. Figure 5.3 shows that when the bias grows, the number of queries follows a logarithmic scale, which is very good news for learning problems with expressive biases.

Finally, on the Sudoku, we see that *max* is the best in terms of queries, though the differences are not that big. *sol* is again the best heuristic in terms of time.

5.4 QUACQ as a solver

The existing techniques to learn constraint networks (CONACQ and *Model Seeker*) need complete positive examples. On the contrary, QUACQ can learn without any complete positive example. This extends its use to not only learn the network of a problem class for which we repeatedly solve instances and already solved some by another mean. QUACQ can learn and solve at the same time an instance of a problem we never solved before. To let it do that we simply exit as

soon as a complete example is classified *yes* by the user.

We assessed this feature by solving a sequence of *instances* of Sudoku, that is, Sudoku grids with their pre-filled cells. QUACQ is stopped when the solution is found. Table 3 gives the results for a sequence of 5 grids of various difficulties that are solved in sequence. Each run takes as input the C_L and B in the state they were at the end of the previous run. The partially learned network is indeed a valid starting point for further learning. For the first grid where QUACQ starts with a complete bias, we needs only 40% of queries comparing to QUACQ convergence (i.e., the last part in table 2). Then again, QUACQ needs less and less of queries to solve the next grids (only 20% , 6%, 1% and $0,3\%$ for the 2^{nd}, 3^{rd}, 4^{th} and 5^{th} grid).

Table 3: A sequence of Sudoku problem solved by QUACQ.

	$\|C_L\|$	$\#q$	$\#q_c$	$\bar{q}$	time
grid#1	340	3859	341	20,71	0,12
grid#2	482	1521	143	23,43	0,11
grid#3	547	687	66	26,04	0,11
grid#4	558	135	12	29,24	0,11
grid#5	561	34	4	34,09	0,17

6 Conclusion

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, 2012.

[Beldiceanu *et al.*, 2005] N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, Swedish Institute of Computer Science, Kista, Sweden, May 2005.

[Bessiere and Koriche, 2012] C. Bessiere and F. Koriche. Non learnability of constraint networks with membership queries. Technical report, Coconut Report, 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, 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 ECML'05*, 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, 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. pages 192–204, 1998.

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

[Junker, 2004] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, San Jose CA, 2004.

[Kleitman and Rothschild, 1970] D.J. Kleitman and B.L. Rothschild. The number of finite topologies. *Proc. Amer. Math. Soc.*, (25):276–282, 1970.

[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.