Partial Queries for Constraint Acquisition*†

Christian Bessiere
CNRS, University of Montpellier
France

Clément Carbonnel
CNRS, University of Montpellier
France

Anton Dries
DTAI, KU Leuven
Belgium

Emmanuel Hebrard
LAAS-CNRS
Toulouse, France

George Katsirelos
INRA Toulouse
France

Nadjib Lazaar
University of Montpellier
France

Nina Narodytska
Samsung Research America
USA

Claude-Guy Quimper
University Laval
Quebec City, Canada

Kostas Stergiou
University of Western Macedonia
Kozani, Greece

Dimosthenis C. Tsouros
University of Western Macedonia
Kozani, Greece

Toby Walsh
NICTA, UNSW
Sydney, Australia

October 13, 2021

Abstract

Learning constraint networks is known to require a number of membership queries exponential in the number of variables. In this paper, we learn constraint networks by asking the user partial queries. That is, we ask the user to classify assignments to subsets of the variables as positive or negative. We provide an algorithm, called QuAcq, that, given a negative example, focuses onto a constraint of the target network in a number of queries logarithmic in the size of the example. The whole constraint network can then be learned with a polynomial number of partial queries. We give information theoretic lower bounds for learning some

*This paper extends and corrects the work published in [8].
†This work has been funded by the European Community project FP7-284715 ICON.
simple classes of constraint networks and show that our generic algorithm
is optimal in some cases. Finally we evaluate our algorithm on some
benchmarks.

1 Introduction

Constraint programming (CP) has been more and more used to solve combina-
torial problems in industrial applications. One of the strengths of CP is that it is
declarative, which means that the user specifies the problem as a CP model, and
the solver finds solutions. However, it appears that specifying the CP model
is not that easy for non-specialists. Hence, the modeling phase constitutes a
major bottleneck in the use of CP. Several techniques have been proposed to
tackle this bottleneck. In Conacq.1 [7, 9, 12], the user provides examples of
solutions (positive) and non-solutions (negative). Based on these positive and
negative examples, the system learns a set of constraints that correctly classifies
all examples given so far. This is a form of passive learning. A passive learner
based on inductive logic programming is presented in [21]. This system requires
background knowledge on the structure of the problem to learn a representa-
tion of the problem correctly classifying the examples. In ModelSeeker [5], the
user provides positive examples to the system, which arranges each of them as
a matrix and identifies constraints in the global constraints catalog ([4]) that
are satisfied by particular subsets of variables in all the examples. Such par-
ticular subsets are for instance rows or columns. The candidate constraints are
ranked and proposed to the user for selection. This ranking/selection combined
with the representation of examples as matrices allows ModelSeeker to quickly
find a good model when the problem has an underlying matrix structure. More
recently, a passive learner called Arnold has been proposed [20]. Arnold takes
positive examples as input and returns an integer program that accepts these
examples as solutions. Arnold relies on a tensor-based language for describing
polynomial constraints between multi-dimensional vectors. As in ModelSeeker,
the problem needs to have an underlying matrix structure. Conacq.1 is thus
the only passive learner that can learn constraints when the problem does not
have a specific structure.

By contrast, in an active learner like Conacq.2 [10, 12], the system proposes
examples to the user to classify as solutions or non solutions. Such questions are
called membership queries [2]. In applications where we need a proof that the
learning system has converged to the target set of constraints, active learning is
a good candidate because it can significantly decrease the number of examples
necessary to converge. For instance, a few years ago, the Normind company
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. An active learner
can do this by automatically interacting with the expert system. As another
example, active learning was 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 [23]. 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 [11, 12].

In this paper, we propose QUACQ (for Quick Acquisition), an active learner that asks the user to classify partial queries as positive or negative. 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. As a result, QUACQ converges on the target constraint network in a polynomial number of queries. In fact, we identify information theoretic lower bounds on the complexity of learning constraint networks that show that QUACQ is optimal on some simple languages and close to optimal on others. One application for QUACQ would be to learn a general purpose model. In constraint programming, a distinction is made between model and data. For example, in a sudoku puzzle, the model contains generic constraints like each subsquare contains a permutation of the numbers. The data, on the other hand, gives the pre-filled squares for a specific puzzle. As a second example, in a time-tableting 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-tableting problem instance. The cost of learning the model can then be amortized over the lifetime of the model.

QUACQ has several advantage. First, it is the only approach ensuring the property of convergence in a polynomial number of queries. Second, 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. Third, QUACQ learns any kind of network, whatever the constraints are organized in a specific structure or not. Fourth, QUACQ can be used when part of the network is already known from the user or from another learning technique. Experiments show that the larger the amount of known constraints, the fewer the queries required to converge on the target network.

The rest of the paper is organized as follows. Section 2 gives the necessary definitions to understand the technical presentation. Section 3 describes the differences between the algorithm presented in this paper and the version in [8]. Section 4 presents QUACQ, the algorithm that learns constraint networks by asking partial queries. In Section 5, we show that QUACQ behaves optimally on some simple languages. Section 6 presents an experimental evaluation of QUACQ. Section 7 concludes the paper.

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 $X_i$. A constraint $c$ is defined by a sequence of
variables \( scp(c) \subseteq X \), called the constraint scope, and a relation \( rel(c) \) over \( \mathbb{Z} \) specifying which sequences of \( |scp(c)| \) values are allowed for the variables \( scp(c) \). We will use the notation \( \text{var}(c) \) to refer to the set of variables in \( scp(c) \), and we abusively call it ‘scope’ too when no confusion is possible. A constraint network (or simply network) is a set \( C \) of constraints on the vocabulary \((X, D)\). An assignment \( e_Y \in D^Y \), where \( D^Y = \Pi_{X_i \in Y} D(X_i) \), is called a partial assignment when \( Y \subseteq X \) and a complete assignment when \( Y = X \). An assignment \( e_Y \) on a set of variables \( Y \subseteq X \) is rejected by a constraint \( c \) (or \( e_Y \) violates \( c \)) if \( \text{var}(c) \subseteq Y \) and the projection \( e_Y[scp(c)] \) of \( e_Y \) on the variables \( scp(c) \) is not in \( rel(c) \). If \( e_Y \) does not violate \( c \), it satisfies it. An assignment \( e_Y \) on \( Y \) is accepted by \( C \) if and only if it does not violate any constraint in \( C \). An assignment on \( X \) that is accepted by \( C \) is a solution of \( C \). We write \( \text{sol}(C) \) for the set of solutions of \( C \). We write \( C[Y] \) for the set of constraints in \( C \) whose scope is included in \( Y \), and \( C_Y \) for the set of constraints in \( C \) whose scope is exactly \( Y \). We say that two networks \( C \) and \( C' \) are equivalent if \( \text{sol}(C) = \text{sol}(C') \).

In addition to the vocabulary, the learner owns a language \( \Gamma \) of bounded arity relations from which it can build constraints on specified sets of variables. Adapting terms from machine learning, the constraint basis, denoted by \( B \), is a set of constraints built from the constraint language \( \Gamma \) on the vocabulary \((X, D)\) from which the learner builds a constraint network.

The target network is a network \( T \) such that \( T \subseteq B \) and for any example \( e \in D^X \), \( e \) is a solution of \( T \) if and only if \( e \) is a solution of the problem that the user has in mind. A membership query \( \text{ASK}(e) \) takes as input a complete assignment \( e \) in \( D^X \) and asks the user to classify it. The answer to \( \text{ASK}(e) \) is yes if and only if \( e \in \text{sol}(T) \). A partial query \( \text{ASK}(e_Y) \), with \( Y \subseteq X \), takes as input a partial assignment \( e \) in \( D^Y \) and asks the user to classify it. The answer to \( \text{ASK}(e_Y) \) is yes if and only if \( e_Y \) does not violate any constraint in \( T \). It is important to observe that "\( \text{ASK}(e_Y) = \text{yes} \)" does not mean that \( e_Y \) extends to a solution of \( T \), which would put an NP-complete problem on the shoulders of the user. A classified assignment \( e_Y \) is called positive or negative example depending on whether \( \text{ASK}(e_Y) \) is yes or no. For any assignment \( e_Y \) on \( Y \), \( \kappa_B(e_Y) \) denotes the set of all constraints in \( B \) that reject \( e_Y \). We will also use \( \kappa_{\Delta}(e_Y) \) to denote the set of constraints in a given set \( \Delta \) that reject \( e_Y \).

We now define convergence, which is the constraint acquisition problem we are interested in. Given a set \( E \) of (partial) examples labeled by the user yes or no, we say that a network \( C \) agrees with \( E \) if \( C \) accepts all examples labeled yes in \( E \) and does not accept those labeled no. The learning process has converged on the network \( L \subseteq B \) if \( L \) agrees with \( E \) and for every other network \( L' \subseteq B \) agreeing with \( E \), we have \( \text{sol}(L') = \text{sol}(L) \). We are thus guaranteed that \( L \) is equivalent to \( T \). It is important to note that \( L \) is not necessarily unique and equal to \( T \). This is because of redundant constraints. Given a set \( C \) of constraints, a constraint \( c \notin C \) is redundant wrt \( C \) if \( \text{sol}(C) = \text{sol}(C \cup \{ e \}) \). If a constraint \( c \) from \( B \) is redundant wrt \( T \), the network \( T \cup \{ e \} \) is equivalent to \( T \).

In the algorithms presented in the rest of the paper we will use the join operation, denoted by \( \bowtie \). Given two sets of constraints \( S \) and \( S' \), the join of \( S \) with \( S' \) is the set of non-empty constraints obtained by pairwise conjunction.
of a constraint in $S$ with a constraint in $S'$. That is, $S \bowtie S' = \{ c \land c' \mid c \in S, c' \in S', c \land c' \neq \bot \}$. A constraint belonging to the basis $B$ will be called elementary in contrast to a constraint composed of the conjunction of several elementary constraints, which will be called conjunction. A conjunction will also sometimes be referred to as a set of elementary constraints. Given a set $S$ of conjunctions, we will use the notation $S_p$ to refer to the subset of $S$ containing only the conjunctions composed of at most $p$ elementary constraints. Finally, a normalized network is a network that does not contain conjunctions of constraints on any scope, that is, all its constraints are elementary.

3 QuAcq2 versus QuAcq1

A first version of QuACQ was published in [8]. From now on let us call it QuACQ1. That version was devoted to normalized constraint networks, that is, networks for which there does not exist any pair of constraints with scopes included one in the other. In addition, QuACQ1 was not taking as assumption that the target network is a subset of constraints from the basis. As a consequence, when the target network was not a subset of the basis, QuACQ1 was either learning a wrong network or was subject to a "collapse" state. When the target network was a subset of the basis, QuACQ1 was asking redundant (i.e., useless) queries. In QuACQ2, the problem of constraint acquisition is formulated in a way that is more in line with standard concept learning [3, 24]. The target network is a subset of the constraints in the basis. As a consequence, an active learner such as QuACQ2 will always return the last possible constraint network given a set of examples already classified. It will never collapse. The second difference with QuACQ1 is that QuACQ2 does not require that the target network is normalized. QuACQ2 can learn any type of constraint network.

4 Constraint Acquisition with Partial Queries

We propose QuACQ2, a novel active learning algorithm. QuACQ2 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 $L$ equivalent to the target network $T$. When a query is answered yes, constraints rejecting it are removed from $B$. When a query is answered no, QuACQ2 enters a loop (functions FindScope and FindC) that will end by the addition of a constraint to $L$.

4.1 Description of QuAcq2

QuACQ2 (see Algorithm 1) initializes the network $L$ it will learn to the empty set (line 2). In line 4, QuACQ2 calls function GenerateExample that computes an assignment $e_Y$ on a subset of variables $Y$ satisfying the constraints of $L$ that have a scope included in $Y$, but violating at least one constraint from $B$. We

\footnote{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 $B$.}
Algorithm 1: QUAcq2

**In:** A basis $B$

**Out:** A learned network $L$

1. begin
   2. $L \leftarrow \emptyset$;
   3. while true do
      4. $e_Y \leftarrow \text{GenerateExample}(X, L, B)$;
      5. if $e_Y = \bot$ then return “convergence on $L$”;
      6. if $\text{ASK}(e_Y) = \text{yes}$ then
         7. $B \leftarrow B \setminus \kappa_B(e_Y)$;
      8. else $\text{FindC}(e_Y, \text{FindScope}(e_Y, \emptyset, Y), L)$;

will see later that there are multiple ways to design function $\text{GenerateExample}$. If there does not exist any pair $(Y, e_Y)$ accepted by $L$ and rejected by $B$ (i.e., $\text{GenerateExample}$ returns $\bot$), then all constraints in $B$ are implied by $L$, and we have converged (line 5). If we have not converged, we propose the example $e_Y$ to the user, who will answer by yes or no (line 6). If the answer is yes, we can remove from $B$ the set $\kappa_B(e_Y)$ of all constraints in $B$ that reject $e_Y$ (line 7). If the answer is no, we are sure that $e_Y$ violates at least one constraint of the target network $T$. We then call the function $\text{FindScope}$ to discover the scope $S$ of one of these violated constraints, and the procedure $\text{FindC}$ will learn (that is, put in $L$) at least one constraint of $T$ whose scope is in $S$ (line 8).

The recursive function $\text{FindScope}$ (see Algorithm 2) takes as parameters an example $e$ and two sets $R$ and $Y$ of variables. An invariant of $\text{FindScope}$ is that $e$ violates at least one constraint whose scope is a subset of $R \cup Y$. A second invariant is that $\text{FindScope}$ always returns a subset of $Y$ that is also the subset of the scope of a constraint violated by $e$. If there is at least one constraint in $B$ rejecting $e[R]$ (i.e., $\kappa_B(e[R]) \neq \emptyset$, line 2), we ask the user whether $e[R]$ is positive or not (line 3). If the answer is yes, we can remove all the constraints that reject $e[R]$ from $B$. If the answer is no, we are sure that $R$ itself contains the scope of a constraint of $T$ rejecting $e$. As $Y$ is not needed to cover that scope, we return the empty set (line 4). We reach line 5 only in case $e[R]$ does not contain any scope $S$ of constraint rejecting $e$. 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 $Y_1$ and $Y_2$ (line 6) and we apply a technique similar to QUICKXPLAIN ([19]) to elucidate the variables of a constraint violating $e[R \cup Y]$ in a logarithmic number of steps (lines 8 and 10). In the first recursive call, if $R \cup Y_1$ does not contain any scope $S$ of constraint rejecting $e$, $\text{FindScope}$ returns a subset $S_1$ of such a scope such that $S_1 = S \cap Y_2$ and $S \subseteq R \cup Y$. In the second recursive call, the variables returned in $S_1$ are added to $R$. If $R \cup S_1$ does not contain any scope $S$ of constraint rejecting $e$, $\text{FindScope}$ returns a subset $S_2$ of such a scope such that...
Algorithm 2: Function FindScope

In : An example $e$; Two scopes $R, Y$
Out : The scope of a constraint in $T$

1 begin
2 if $\kappa_B(e[R]) \neq \emptyset$ then
3 if $\text{ASK}(e[R]) = \text{yes}$ then
4 $B \leftarrow B \setminus \kappa_B(e[R])$;
5 else return $\emptyset$;
6 if $|Y| = 1$ then return $Y$;
7 split $Y$ into $< Y_1, Y_2 >$ such that $|Y_1| = \lfloor |Y|/2 \rfloor$;
8 if $\kappa_B(e[R \cup Y]) = \kappa_B(e[R \cup Y_1])$ then $S_1 \leftarrow \emptyset$;
9 else $S_1 \leftarrow \text{FindScope}(e, R \cup Y_1, Y_2)$;
10 if $\kappa_B(e[R \cup S_1]) = \kappa_B(e[R \cup Y])$ then $S_2 \leftarrow \emptyset$;
11 else $S_2 \leftarrow \text{FindScope}(e, R \cup S_1, Y_1)$;
12 return $S_1 \cup S_2$;

$S_2 = S \cap Y_1$ and $S \subseteq R \cup Y$. The rationale of lines 7 and 9 is to avoid entering a recursive call to FindScope when we know the answer to the query in line 3 of that call will necessarily be no. It happens when all the constraints rejecting $e[R \cup Y]$ have a scope included in the set of variables that will be $R$ inside that call (that is, $R \cup Y_1$ for the call in line 8 and $R$ union the output of line 8 for the call in line 10). Finally, line 11 of FindScope returns the union of the two subsets of variables returned by the two recursive calls, as we know they all belong to the same scope of a constraint of $T$ rejecting $e$.

The function FindC (see Algorithm 3) takes as parameter $e$ and $Y$, $e$ being the negative example that led FindScope to find that there is a constraint from the target network $T$ over the scope $Y$. The set $\Delta$ is initialized to all candidate constraints, that is, the set $B_Y$ of all constraints from $B$ with scope exactly $Y$ (line 2). As we know from FindScope that there will be a constraint with scope $Y$ in $T$, we join $\Delta$ with the set of constraints of scope $Y$ rejecting $e$ (line 3). In line 5 an example $e'$ is chosen in such a way that $\Delta$ contains both constraints satisfied by $e'$ and constraints violated by $e'$. If no such example exists (line 6), this means that all constraints in $\Delta$ are equivalent wrt $L[Y]$. Any of them is added to $L$ and $B$ is emptied of all its constraints with scope $Y$ (line 8). If a suitable example $e'$ was found, it is proposed to the user for classification (line 10). If $e'$ is classified positive, all constraints rejecting it are removed from $\Delta$ and $B$ (line 11). Otherwise we call FindScope to seek constraints with scope strictly included in $Y$ that violate $e'$ (line 13). If FindScope returns the scope of such a constraint, we recursively call FindC to find that smaller arity constraint before the one having scope $Y$ (line 14). If FindScope has not found such a scope (that is, it returned $Y$ itself), we do the same join as in line 3 to keep in $\Delta$ only constraints rejecting the example $e'$ (line 15). Then, we continue the loop of line 4.

At this point we can make an observation on the kind of response the user
Algorithm 3: Procedure FindC

In : An example \( e \); A scope \( Y \)
In Out: The network \( L \)

begin
\[ \Delta \leftarrow B_Y; \]
\[ \Delta \leftarrow \Delta \times \kappa_\Delta(e); \]
while true do
\[ \text{choose } e'_Y \text{ in } \text{sol}(L[Y]) \text{ such that } \emptyset \subsetneq \kappa_\Delta(e'_Y) \subsetneq \Delta; \]
if \( e'_Y = \perp \) then
\[ \text{pick } c \text{ in } \Delta; \]
\[ L \leftarrow L \cup \{c\}; B \leftarrow B \setminus B_Y; \text{ exit}; \]
else
if \( \text{ASK}(e'_Y) = \text{yes} \) then
\[ \Delta \leftarrow \Delta \setminus \kappa_\Delta(e'_Y); B \leftarrow B \setminus \kappa_B(e'_Y); \]
else
\[ S \leftarrow \text{FindScope}(e'_Y, \emptyset, Y); \]
if \( S \subsetneq Y \) then \( \text{FindC}(e'_Y, S, L); \)
else
\[ \Delta \leftarrow \Delta \times \kappa_\Delta(e'_Y); \]
end
end

is able to give. QUAcq2 is designed to communicate with users who are not able to provide any more hint than "Yes, this example works" or "No, this example doesn't work". We can imagine cases where the user is a bit more skilled than that and can provide answers such as (a) "This example \( e \) doesn't work because there is something wrong on the variables in this set \( Y \)" or (b) "This example \( e \) doesn't work because it violates this constraint \( c \)" or (c) "This example \( e \) doesn't work: here is the set of all the constraints that it violates". QUAcq2 can easily be adapted to these more informative types of answers. In the case of (a) we just have to skip the call to FindScope. In the case of (b), we can both skip FindScope and FindC. The case (c) corresponds to the matchmaker agent described in [17]. The more informative the query, the more dramatic the decrease in number of queries needed to find the right constraint network.

4.2 Illustration example

We illustrate the behavior of QUAcq2 and its two sub-procedures FindScope and FindC on a simple example. Consider the variables \( X_1, \ldots, X_5 \) with domains \([-10,10]\), the language \( \Gamma = \{\leq, \neq, \sum \neq\} \), and the basis \( B = \{\leq_{ij}, \geq_{ij}, \neq_{ij}\} \) \( i, j \in 1..5, i < j \} \cup \{\sum_{ij}^{\neq k} \mid i, j, k \in 1..5, i < j \neq k \neq i\} \), where \( \leq_{ij} \) is the constraint \( X_i \leq X_j, \geq_{ij} \text{ is } X_i \geq X_j, \neq_{ij} \text{ is } X_i \neq X_j, \) and \( \sum_{ij}^{\neq k} \text{ is } X_i + X_j \neq X_k \).

The target network is \( T = \{=_{15}, <_{23}, \sum_{23}^{\neq 4}\}. \)

\[ \text{Note that } \geq_{ij} \text{ denotes } \leq_{ji}. \]
Table 1: FindScope on the example (0, 1, 2, 3, 4)

| call | $R$ | $Y$ | ASK | return |
|------|-----|-----|-----|--------|
| 0    | ∅   | $X_1, X_2, X_3, X_4, X_5$ | ×    | $X_2, X_3, X_4$ |
| 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  | ∅     |
| 1.2  | $X_1, X_2, X_3$ | $X_4$ | ×   | $X_4$ |
| 2    | $X_4$ | $X_1, X_2, X_3$ | ×   | $X_2, X_3$ |
| 2.1  | $X_1, X_2, X_4$ | $X_3$ | yes | $X_3$ |
| 2.2  | $X_3, X_4$ | $X_1, X_2$ | yes | $X_2$ |
| 2.2.1| $X_1, X_3, X_4$ | $X_2$ | ×   | $X_2$ |

Suppose that the first example $e_1$ generated in line 4 of QuAcq2 is $\{X_1 = 0, X_2 = 1, X_3 = 2, X_4 = 3, X_5 = 4\}$, denoted by $(0, 1, 2, 3, 4)$. The query is proposed to the user in line 6 of QuAcq2 and the user replies no because the constraints $= 15$ and $\sum_{x=4}^{23}$ are violated. As a result, FindScope($e_1, ∅, \{X_1, \ldots, X_5\}$) is called in line 8 of QuAcq2.

**Running FindScope**

The trace of the execution of FindScope($e_1, ∅, \{X_1, \ldots, X_5\}$) is displayed in Table 1. Each row corresponds to a call to FindScope. Queries are always on the variables in $R$. “×” in the column ASK means that the question is skipped because $κ_B(e_1[R]) = ∅$. This happens when $R$ is of size less than 2 (the smallest constraints in $B$ are binary) or because a (positive) query has already been asked on $e_1[R]$ and $κ_B(e_1[R])$ has been emptied.

- **The initial call** (call-0 in Table 1) does not ask the query because $R = ∅$ and $κ_B(e_1[∅]) = ∅$. $Y$ is split in two sets $Y_1 = \{X_1, X_2, X_3\}$ and $Y_2 = \{X_4, X_5\}$. Line 2 detects that $κ_B(e_1[X_1, X_2, X_3])$ and $κ_B(e_1[X_1, X_2, X_3, X_4, X_5])$ are different (e.g., $\sum_{x=25}$ is still in $B$), so the recursive call call-1 is performed.

- **Call-1:** $R = \{X_1, X_2, X_3\}$ (i.e., the $R \cup Y_1$ of call-0) and $Y = \{X_4, X_5\}$ (i.e., the $Y_2$ of call-0). $e_1[X_1, X_2, X_3]$ is classified positive. Hence, line 9 of FindScope removes all constraints in $κ_B(e_1[X_1, X_2, X_3])$ (i.e., $\geq 12, \geq 13, \geq 23$) from $B$. $Y$ is split in two sets $Y_1 = \{X_4\}$ and $Y_2 = \{X_5\}$. Again, $κ_B(e_1[X_1, X_2, X_3, X_4])$ and $κ_B(e_1[X_1, X_2, X_3, X_4, X_5])$ are different in line 7 ($\sum_{x=25}$ is still in $B$), so call-1.1 is performed.

- **Call-1.1:** $e_1[R]$ is classified negative. The empty set is returned in line 4 of call-1.1. We are back to call-1. Line 9 of call-1 detects that $κ_B(e_1[X_1, X_2, X_3])$ and $κ_B(e_1[X_1, X_2, X_3, X_4, X_5])$ are different, so call-1.2 is performed in line 10 of call-1.

- **Call-1.2:** $R = \{X_1, X_2, X_3\}$ (i.e., the $S_1 \cup Y_1$ of call-1) and $Y = \{X_4\}$ (i.e., the $Y_2$ of call-1). Call-1.2 does not ask the query because $κ_B(e_1[X_1, X_2, X_3])$
is already empty (see call-1). In line 8 call-1.2 detects that \( Y \) is a singleton and returns \( \{ X_4 \} \). We are back to call-1. In line 11 call-1 returns \( \{ X_4 \} \) one level above in the recursion. We are back to call-0. As \( \kappa_B(e_1[X_4]) \) and \( \kappa_B(e_1[X_1, X_2, X_3, X_4, X_5]) \) are different, we go to call-2.

- **Call-2**: The query \( \text{ASK}(e_1[X_4]) \) is not asked because \( \kappa_B(e_1[X_4]) \) is empty. \( Y \) is split in two sets \( Y_1 = \{ X_1, X_2 \} \) and \( Y_2 = \{ X_3 \} \). As \( \kappa_B(e_1[X_1, X_2, X_4]) \) and \( \kappa_B(e_1[X_1, X_2, X_3, X_4]) \) are different (\( \geq 44 \) is still in \( B \)), we go to call-2.1.

- **Call-2.1**: \( e_1[X_1, X_2, X_4] \) is classified positive. \text{FindScope} removes the constraints in \( \kappa_B(e_1[X_1, X_2, X_4]) \) from \( B \) and returns the singleton \( \{ X_4 \} \). We are back to call-2. As \( \kappa_B(e_1[X_3, X_4]) \) and \( \kappa_B(e_1[X_1, X_2, X_3, X_4]) \) are different, we go to call-2.2.

- **Call-2.2**: \( e_1[X_3, X_4] \) is classified positive. \text{FindScope} removes the constraints in \( \kappa_B(e_1[X_3, X_4]) \) from \( B \). \( Y \) is split in two sets \( Y_1 = \{ X_1 \} \) and \( Y_2 = \{ X_2 \} \). As \( \kappa_B(e_1[X_1, X_3, X_4]) \) and \( \kappa_B(e_1[X_1, X_2, X_3, X_4]) \) are different (\( \sum_{23}^{24} \) is still in \( B \)), we go to call-2.2.1.

- **Call 2.2.1** does not ask the query because \( \kappa_B(e_1[X_1, X_3, X_4]) \) is empty. (Binary constraints have been removed by former yes answers and there is no ternary constraint on \( \{ X_1, X_3, X_4 \} \) that is violated by \( e_1 \).) As \( Y \) is a singleton. Call-2.2.1 returns \( \{ X_2 \} \). We are back to call-2.2.

- **Line 9** of call-2.2 detects that \( \kappa_B(e_1[X_2, X_3, X_4]) = \kappa_B(e_1[X_1, X_2, X_3, X_4]) \) because all constraints between \( X_1 \) and \( X_2, X_3, X_4 \) that were in \( \kappa_B(e_1) \) have been removed by yes answers. Call-2.2.2 is skipped and \( \emptyset \) is added to \( \{ X_2 \} \) in line 11 of call-2.2. \( \{ X_2 \} \) is returned to call-2. Call-2 returns \( \{ X_2, X_3 \} \), and call-0 returns \( \{ X_2, X_3, X_4 \} \). Line 8 of \text{QUACq2} then calls \text{FindC} with \( e_1[X_2, X_3, X_4] = (1, 2, 3) \) and \( Y = \{ X_2, X_3, X_4 \} \).

**Running \text{FindC}**

The trace of the execution of \text{FindC}((1, 2, 3), \{ X_2, X_3, X_4 \}) is displayed in Table 2. Each row reports the results of the actions performed after generating a new example in line 5 of \text{FindC}. For each of these examples, we report the example generated, its classification, and the new state of \( \Delta \), \( L \), and \( B[X_2, X_3, X_4] \). We also specify in which lines of \text{FindC} these changes occur.

- **Row-0**: The example (1,2,3) was not generated in \text{FindC} but inherited from \text{FindScope}. By definition of \text{FindScope}, we know that it is a negative example (denoted by "(no)" in the table). In line 2 of \text{FindC}, \( \Delta \) is initialized to the set of constraints from \( B \) having scope \( \{ X_2, X_3, X_4 \} \), that is \( \{ \Sigma_{23}, \Sigma_{24}, \Sigma_{34} \} \), and then in line 3 these constraints are joined with \( \Sigma_{23} \), the only constraint in \( \kappa_\Delta((1,2,3)) \). At this point the learned network \( L \) is still empty because \text{FindScope} did not modify it. \( B[X_2, X_3, X_4] \) contains all the constraints from the original \( B \) with scope included in
Row-1: In line 5 \texttt{FindC} generates the example (2, 3, 1), satisfying some constraints from $\Delta$ but not all. (2, 3, 1) is classified positive in line 10. Hence, the violated conjunction $\sum_{23}^{34} \land \sum_{24}^{34} \land \sum_{24}^{34}$ is removed from $\Delta$ and all violated constraints in $B$ (i.e., $\sum_{23}^{34}, \leq_{24}, \text{and } \leq_{34}$) are removed (line 11). $L$ remains unchanged.

Row-2: \texttt{FindC} generates the example (3, 2, 1), which is classified negative. The call to \texttt{FindScope} in line 13 returns $S = \{ X_2, X_3 \}$. Line 14 then recursively calls \texttt{FindC} on the scope $\{ X_2, X_3 \}$.

Row-3: The example (3, 2) is known to be negative without asking. Lines 15 initializes \( \Delta \) to the set of constraints in $B_{\{X_2, X_3\}}$ and then join them to those rejecting the example. (Note that $\leq_{23}$ is a shortcut for $\leq_{23}$ \text{and } $\not\leq_{23}$.) $L$ and $B$ remain unchanged.

Row-4: \texttt{FindC} generates the example (1, 1), which is classified negative. The call to \texttt{FindScope} in line 13 returns the same scope $S = \{ X_2, X_3 \}$ because $B$ does not contain any smaller arity constraints. Line 15 reduces $\Delta$ to the singleton $\leq_{23}$. As a result, the next loop of \texttt{FindC} cannot generate any new example in line 5. Line 8 adds $\leq_{23}$ to $L$ and removes all the constraints with scope $\{ X_2, X_3 \}$ from $B$. This subcall to \texttt{FindC} exits.

Row-5: We are back to the original call to \texttt{FindC} with the same $\Delta$ as in row-2. Line 5 must generate an example accepted by $L$ and violating part

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\text{example} & \text{ASK} & $\Delta$ & $L$ & $B_{\{X_2, X_3, X_4\}}$ \\
(line 3) & (line 10) & (lines 2, 3, 11 and 15) & (line 8) & (lines 8 and 11) \\
\hline
\texttt{FindC}($\{1, 2, 3\}, \{X_2, X_3, X_4\}$) & & & & & \\
\hline
0. & (1, 2, 3) & (no) & $\sum_{23}^{34} \land \sum_{24}^{34} \land \sum_{24}^{34}$ & $\emptyset$ & $\sum_{23}^{34} \land \sum_{24}^{34} \land \sum_{24}^{34}$ \leq_{23}, \leq_{24}, \leq_{34}, \not\leq_{23}, \not\leq_{24}, \not\leq_{34} \\
\hline
1. & (2, 3, 1) & yes & $\sum_{23}^{34} \land \sum_{24}^{34} \land \sum_{24}^{34}$ & $\emptyset$ & $\sum_{23}^{34} \land \sum_{24}^{34} \land \sum_{24}^{34}$ \leq_{23}, \not\leq_{23}, \not\leq_{24}, \not\leq_{34} \\
\hline
2. & (3, 2, 1) & no & unchanged & $\emptyset$ & unchanged \\
\hline
\texttt{FindC}($\{3, 2\}, \{X_2, X_3\}$) & & & & & \\
\hline
3. & (3, 2) & (no) & $\leq_{23}, \leq_{23}$ & $\emptyset$ & unchanged \\
\hline
4. & (1, 1) & no & $\leq_{23}$ & $<_{23}$ & $\sum_{23}^{34} \land \sum_{34}^{34} \neq_{24}, \neq_{34}$ \\
\hline
& & & & back to \texttt{FindC}($\{1, 2, 3\}, \{X_2, X_3, X_4\}$) & & \\
\hline
5. & (1, 2, -1) & yes & $\sum_{23}^{34}$ & $<_{23}, \sum_{23}^{34}$ & $\neq_{24}, \neq_{34}$ \\
\hline
\end{tabular}
\caption{FindC}
\end{table}
of $\Delta$. It generates $(1, 2, -1)$, which is positive. The violated conjunction
$\sum_{23}^{28} \wedge \sum_{29}^{32}$ is removed from $\Delta$ and $\sum_{34}^{37}$ is removed from $B$ (line 11). The
next loop of $\text{FindC}$ cannot generate any new example in line 5 because $\Delta$
is now a singleton. Line 8 adds $\sum_{23}^{28}$ to $L$ and removes all the constraints
with scope $\{X_2, X_3, X_4\}$ from $B$. $\text{FindC}$ exits.

4.3 Theoretical analysis

We first show that $\text{QuAcq2}$ is a correct algorithm to learn a constraint network
equivalent to a target network that can be specified within a given basis. We
prove that $\text{QuAcq2}$ is sound, complete, and terminates.

**Proposition 1 (Soundness)** Given a basis $B$ and a target network $T \subseteq B$, the network $L$ returned by $\text{QuAcq2}$ is such that $\text{sol}(T) \subseteq \text{sol}(L)$.

**Proof.** Suppose there exists $e_1 \in \text{sol}(L) \setminus \text{sol}(T)$. Hence, there exists at least
one scope on which $\text{QuAcq2}$ has learned a conjunction of constraints rejecting $e_1$. Let us consider the first such conjunction $c^*$ learned by $\text{QuAcq2}$, and let us
denote its scope by $Y$. By assumption, $c^*$ contains an elementary constraint $c_1$
rejecting $e_1$. The only place where we add a conjunction of constraints to $L$ is
line 5 of $\text{FindC}$. This conjunction has been built by join operations in lines 3 and
15 of $\text{FindC}$. By construction of $\text{FindScope}$, $e_1[Y]$ is rejected by a constraint
with scope $Y$ in $T$ and by none of the constraints on subscopes of $Y$ in $T$ when
the join operation in line 3 of $\text{FindC}$ is executed. By construction of $\text{FindC}$, the
join operations in line 15 of $\text{FindC}$ are executed for and only for $e'_Y$ generated
in this call to $\text{FindC}$ that are rejected by a constraint with scope $Y$ in $T$ and by
none of the constraints on subscopes of $Y$. As a result, $\Delta$ contains all minimal
conjunctions of elementary constraints from $B_Y$ that reject $e_1[Y]$ and all $e'_Y$
generated in this call to $\text{FindC}$ that are rejected by a constraint of scope $Y$ in $T$
and by none of the constraints on subscopes of $Y$. One of those minimal
conjunctions is necessarily a subset of the conjunction in $T$. In line 5 when
we put one of these conjunctions in $L$, they are all equivalent wrt $L$ because
line 5 could not produce an example $e'_Y$ violating some conjunctions from $\Delta$
and satisfying the others. As scope $Y$ is, by assumption, the first scope on
which $\text{QuAcq2}$ learns a wrong conjunction of constraints, we deduce that all
conjunctions in $\Delta$ are equivalent wrt to $T$. As a consequence, none can contain
$c_1$. Therefore, adding one of them to $L$ cannot reject $e_1$. □

**Proposition 2 (Completeness)** Given a basis $B$ and a target network $T \subseteq B$, the network $L$ returned by $\text{QuAcq2}$ is such that $\text{sol}(L) \subseteq \text{sol}(T)$.

**Proof.** Suppose there exists $e_1 \in \text{sol}(L) \setminus \text{sol}(T)$ when $\text{QuAcq2}$ terminates.
Hence, there exists an elementary constraint $c_1$ in $B$ that rejects $e_1$, and $c_1$
belongs to $c^*$, the conjunction of the constraints in $T$ with same scope as $c_1$.
The only way for $\text{QuAcq2}$ to terminate is line 3 of $\text{QuAcq2}$. This means that
in line 4 $\text{GenerateExample}$ was not able to generate an example $e_Y$ accepted
by $L[Y]$ and rejected by $B[Y]$. Thus, $c_1$ is not in $B$ when $\text{QuAcq2}$ terminates,
otherwise the projection $e_1[Y]$ of $e_1$ on any $Y$ containing $\text{var}(c_1)$ would have been such an example. We know that $c_1 \in T$, so $c_1$ was in $B$ before starting QUACQ2. Constraints can be removed from $B$ in line 7 of QUACQ2, line 3 of FindScope, and lines 8 and 11 of FindC. In line 7 of QUACQ2, line 3 of FindScope, and line 11 of FindC, a constraint $c_2$ is removed from $B$ because it rejects a positive example. This removed constraint $c_2$ cannot be $c_1$ because $c_1$ belongs to $T$, so it cannot reject a positive example. In line 8 of FindC, all (elementary) constraints with scope $Y$ are removed from $B$. Let us see if one of them could be our $c_1$. Given an elementary constraint $c_2$ with scope $Y$ that is removed from $B$ in line 8 of FindC, either $c_2$ is still appearing in one conjunction of $\Delta$ when FindC terminates, or not. Thanks to lines 9 and 10 we know that $L \cup \{c_Y\} \models \Delta$. Thus, if $c_2$ is in one of the conjunctions of $\Delta$, then $L \models c_2$ after the execution of line 8 the only line where FindC can terminate. Thus, $c_2$ cannot be $c_1$ because by assumption $c_1$ rejects $c_1$, which itself is accepted by $L$. If $c_2$ is not in any of the conjunctions of $\Delta$ when FindC terminates, these conjunctions must have been removed in line 11 or in line 15, the two places where $\Delta$ is modified. Let us denote by $\hat{c}_2$ a conjunction in $\Delta$ composed of $c_2$ and a subset of $c^*$. It necessarily exists at the first execution of the loop in line 4 because $c_2 \in B$ and line 3 either keeps $c_2$ (if $c_2$ is violated by $e$), or joins it with elements of $c^*$ (if $c_2$ is satisfied by $e$). Line 15 is executed after a negative query $e_{c_Y}$. If $c_2$ rejects $e_{c_Y}$, all the conjunctions containing it remain in $\Delta$. If $c_2$ is satisfied by $e_{c_Y}$, there necessarily exists a conjunction in $\kappa_\Delta(e_{c_Y})$ which is a subset of the conjunction $c^*$ because QUACQ2 is sound (Proposition 1). $\hat{c}_2$ is joined with this subset. Thus, $\Delta$ still contains a conjunction composed of $c_2$ and a subset of $c^*$. Each time a negative example will be generated, this subset will either stay in $\Delta$ or be joined with another subset of $c^*$. As a result, line 15 cannot remove all conjunctions composed of $c_2$ and a subset of $c^*$. These conjunctions $\hat{c}_2$ must then have been removed in line 11 because they were rejecting the example $e_{c_Y}$ classified positive in line 10. These conjunctions can be removed only if $c_2$ rejects $e_{c_Y}$ because the rest of the conjunction is a subset of $c^*$. Again $c_2$ cannot be $c_1$ because $c_1$ cannot reject positive examples. Therefore, $c_1$ cannot reject an example accepted by $L$, which proves that $\text{sol}(L) \subseteq \text{sol}(T)$. \hfill \Box

**Proposition 3 (Termination)** Given a basis $B$ and a target network $T \subseteq B$, QUACQ2 terminates.

**Proof.** Each execution of the loop in line 8 of QUACQ2 either executes line 7 of QUACQ2 or enters FindC. By construction of $e_Y$ in line 1 of QUACQ2 we know that $\kappa_B(e_Y)$ is not empty. Hence, in line 7 of QUACQ2, $B$ strictly decreases in size. By definition of FindScope, the set $Z$ returned by FindScope is such that there exists a constraint $c$ with $\text{var}(c) = Z$ in $B$ rejecting $e_Y$. Thus, $\kappa_B(e_Y[Z])$ is not empty. As a result, each time FindC is called, $B$ strictly decreases in size because FindC always executes line 8 before exiting. Therefore, at each execution of the loop in line 8 of QUACQ2, $B$ strictly decreases in size. As $B$ has finite size, we have termination. \hfill \Box

13
Theorem 1 (Correctness) Given a basis $B$ and a target network $T \subseteq B$, QUACQ2 returns a network $L$ such that $\text{sol}(L) = \text{sol}(T)$.

Proof. Correctness immediately follows from Propositions 1, 2, and 3. □

We analyze the complexity of QUACQ2 in terms of the number of queries it can ask of the user. Queries are proposed to the user in line 6 of QUACQ2, line 3 of FindScope and line 10 of FindC.

Proposition 4 Given a vocabulary $(X, D)$, a basis $B$, a target network $T$, and an example $e_Y \in D^Y$ rejected by $T$, FindScope uses $O(|S| \cdot \log |Y|)$ queries to return the scope $S$ of one of the constraints of $T$ violated by $e_Y$.

Proof. Let us first consider a version of FindScope that would execute lines 8 and 10 unconditionally. That is, a version without the tests in lines 7 and 9. FindScope is a recursive algorithm that asks at most one query per call (line 3). Hence, the number of queries is bounded above by the number of nodes of the tree of recursive calls to FindScope. We 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. By construction of FindScope, we observe that no answers to the query in line 3 always occur in leaf calls and that the only way for a leaf call to return the empty set is to have received a no answer to its query (line 4). 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. We know that $S \subseteq R_{parent}$ because the parent call necessarily received a yes answer. Furthermore, from the no answer to the query $ASK(e[R_{child}])$, we know that $S \subseteq R_{child}$. Consider first the case where the leaf is the left child of the parent node. By construction, $R_{child} \subseteq R_{parent} \cup Y_{parent}$. 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 test of line 2 is false (i.e., $\kappa_B(e[R_{child}]) = \emptyset$), we necessarily exit from FindScope through line 3, which means that this node is the end of a branch leading to a variable in $S$. If the test of line 2 is true (i.e., $\kappa_B(e[R_{child}]) \neq \emptyset$), we are guaranteed that the left child of the parent node returned a non-empty set, otherwise $R_{child}$ would be equal to $R_{parent}$ and we know that $\kappa_B(e[R_{parent}])$ has been emptied in line 3 as it received a yes answer. Thus, 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 $\log |Y|$ long. Therefore the total number of queries FindScope can ask is at most $2 \cdot |S| \cdot \log |Y|$, which is in $O(|S| \cdot \log |Y|)$.

Let us come back to the complete version of FindScope, where lines 7 and 9 are active. The purpose of lines 7 and 9 is only to avoid useless calls to
FindScope that would return $\emptyset$ anyway. These lines do not affect anything else in the algorithm. Hence, by adding lines 7 and 9 we can only decrease the number of recursive calls to FindScope. As a result, we cannot increase the number of queries. □

Theorem 2 (Complexity) Let $\Gamma$ be a language of bounded-arity relations. QuAcq2 learns constraint networks over $\Gamma$ in $O(m \log n + b)$ queries, where $n$ and $m$ are respectively the number of variables and the number of constraints of the target network, and $b$ is the size of the basis.

Proof. Each time line 6 of QuAcq2 classifies an example as negative, the scope $\text{var}(c)$ of a constraint $c$ from the target network is found in $O(|\text{var}(c)| \cdot \log n)$ queries (Proposition 4). As the basis only contains constraints of bounded arity, $\text{var}(c)$ is found in $O(\log n)$ queries. Finding $c$ with FindC requires a number of queries in $O(1)$ because the size of $\Gamma$ does not depend on the size of the target network. Hence, the number of queries necessary for finding the target network is in $O(m \log n)$. Convergence is obtained once the basis is wiped out of all its constraints or those remaining are implied by the learned network $L$. Each time an example is classified positive in line 6 of QuAcq2 or line 3 of FindScope, this leads to at least one constraint removal from the basis because, by construction of QuAcq2 and FindScope, this example violates at least one constraint from the basis. Concerning queries asked in FindC, their number is in $O(1)$ at each call to FindC, and there are no more calls to FindC than constraints in the target network because FindC always adds at least one constraint to $L$ during its execution (line 8). This gives a total number of queries required for convergence that is bounded above by the size $b$ of the basis. □

The complexities stated in Theorem 2 are based on the size of the target network and size of the basis. The size of the language $\Gamma$ is not considered because it has a fixed size, independent on the number of variables in the target network. Nevertheless, line 15 of FindC can lead to an increase in the size of $\Delta$ up to $2^{|I|}$. By reformulating line 5 of FindC as shown below, we can bound the increase in size of $\Delta$. In the following, we use the notation $\Delta_p$ as defined at the very end of Section 2.

... 5bis choose $e'_Y$ in $\text{sol}(L[Y])$ and $\emptyset \subseteq \kappa_\Delta(e'_Y) \subseteq \Delta$, minimizing $p$ such that $\emptyset \subseteq \kappa_{\Delta_p}(e'_Y) \subseteq \Delta_p$ if possible, $\kappa_{\Delta_p}(e'_Y) \subseteq \Delta_p$ otherwise;

...  

Proposition 5 Given a basis $B$, a target network $T$, and a scope $Y$, the number of queries required by FindC to learn a subset of $B_Y$ equivalent to the conjunction of constraints of $T$ with scope $Y$ in $T[Y]$ is in $O(|B_Y| + 2^{\max(|c^*|,|I_{c^*}|)})$, where $c^*$ is the smallest such conjunction and $I_{c^*} = \{c_i \in B_Y \mid c^* \rightarrow c_i\}$.  

15
Proof. We first compute the number of queries required to generate \(c^*\) in \(\Delta\), and then the number of queries required to remove all conjunctions of constraints not equivalent to \(c^*\) from \(\Delta\).

Let us first prove that line 5bis of FindC will not stop generating examples before \(c^*\) is one of the conjunctions in \(\Delta\). Let us take as induction hypothesis that when entering a new execution of the loop in line 4 if \(c^*\) is not in \(\Delta\), then the set of the conjunctions in \(\Delta\) that are included in \(c^*\) covers the whole set of elementary constraints from \(c^*\). That is, \(\bigcup\{\text{sub} \in \Delta \mid \text{sub} \subset c^*\} = c^*\). The only way to modify \(\Delta\) is to ask a query \(e_Y\). If \(e_Y\) is positive, this means that \(c^*\) is satisfied and all its subsets remain in \(\Delta\). If \(e_Y\) is negative, either this is due to a constraint of \(T\) on a subscope of \(Y\) or not. If it is due to a constraint on a subscope, line 14 is executed and not line 15, so \(\Delta\) remains unchanged. If it is not due to a constraint on a subscope, this guarantees that at least one elementary constraint of \(c^*\) is violated, and according to our induction hypothesis, at least one subset of \(c^*\), call it \(sub_1\), is in \(\kappa_{\Delta}(e_Y)\). Hence, line 15 generates a conjunction of \(sub_1\) with each of the other subsets of \(c^*\) that are in \(\Delta\). As a result, every elementary constraint in \(c^*\) belongs to at least one of these conjunctions with \(sub_1\) that are uniquely composed of elementary constraints from \(c^*\). Furthermore, before line 10 by construction, all elementary constraints composing \(c^*\) are in \(\Delta\) and line 10 is similar to line 15. As a consequence, our induction hypothesis is true. We prove now that as long as \(c^*\) is not in \(\Delta\), line 5bis is able to generate a query \(e_Y\). By definition, we know that \(c^*\) is the smallest conjunction equivalent to the constraint of \(T\) with scope \(Y\). Thus, no subset of \(c^*\) can be implied by any other subset of \(c^*\). This guarantees that there exists an example \(e_Y\) such that one subset \(sub_1\) of \(c^*\) is in \(\kappa_{\Delta}(e_Y)\) and another subset, \(sub_2\), is in \(\Delta \setminus \kappa_{\Delta}(e_Y)\). \(e_Y\) is a valid query to be generated in line 5bis and to be asked in line 10. As a consequence, we cannot exit FindC as long as \(c^*\) is not in \(\Delta\).

We now prove that \(c^*\) is in \(\Delta\) after a number of queries linear in \(|BY|\). We first count the number of positive queries. Thanks to the condition in line 5bis of FindC, we know that at least one elementary constraint \(c_i\) of \(BY\) is violated by the query. Thus, all the conjunctions containing \(c_i\) are removed from \(\Delta\) in line 11 and no conjunction containing \(c_i\) will be able to come again in \(\Delta\). As a result, the number of positive queries is bounded above by \(|BY|\). Let us now count the number of negative queries. A query can be negative because of a constraint on a subscope of \(Y\) or because of \(c^*\). If because of a subscope we do not count it in the cost of learning \(c^*\). If because of \(c^*\), we saw that there exists a subset \(sub_1\) of \(c^*\) in \(\kappa_{\Delta}(e_Y)\). Line 15 generates a conjunction of \(sub_1\) with each of the other subsets of \(c^*\) that are in \(\Delta\). Before the joining operation, either \(sub_1\) is included in the largest subset \(maxsub\) or not. If \(sub_1\) is included in \(maxsub\), then \(maxsub\) also belongs to \(\kappa_{\Delta}(e_Y)\) and it produces a larger subset by joining with any other non-included subset of \(c^*\). If \(sub_1\) is not included in \(maxsub\), they are necessarily joined together, generating again a subset strictly larger than \(maxsub\). Thus, the number of queries that are negative because of \(c^*\) is bounded above by \(|c^*|\). Therefore, the number of queries necessary to have \(c^*\) in \(\Delta\) is in \(O(|BY|)\).
Once $c^*$ has been generated, it will remain in $\Delta$ until the end of this call to \texttt{FindC} because it can be removed neither by a positive query (it would not be in $\kappa_\Delta(c'_Y)$) nor by a negative (either it is in the $\kappa_\Delta(c'_Y)$ or a subconstraint is found and $\Delta$ is not modified).

We now show that the number of queries required to remove all conjunctions of constraints not equivalent to $c^*$ from $\Delta$ is in $O(|B_Y| + 2^{\max(|c^*|, |I_{c^*}|)})$. We first have to prove that once a conjunction $rem$ has been removed from $\Delta$, it will never come back in $\Delta$ by some join operation. The conjunction $rem$ can come back in $\Delta$ if and only if there exist $a$ and $b$ in $\Delta$ such that $rem = a \land b$. If $rem$ was removed due to a positive query $c'_Y$, then $rem$ was in $\kappa_\Delta(c'_Y)$ and then, either $a$ or $b$ was in $\kappa_\Delta(c'_Y)$ too. Thus, $a$ or $b$ has been removed from $\Delta$ at the same time as $rem$, which contradicts the assumption that $rem$ came back due to the join of $a$ and $b$. If $rem$ was removed due to a negative query $c'_Y$, then $rem$ was not in $\kappa_\Delta(c'_Y)$ and then, none of $a$ and $b$ were in $\kappa_\Delta(c'_Y)$. $a$ and $b$ have thus both been joined with other elements of $\kappa_\Delta(c'_Y)$ and have disappeared from $\Delta$ at the same time as $rem$. This again contradicts the assumption.

We are now ready to show that all conjunctions not equivalent to $c^*$ are removed from $\Delta$ in $O(|B_Y| + 2^{\max(|c^*|, |I_{c^*}|)})$ queries. For that, we first prove that all conjunctions not implied by $c^*$ are removed from $\Delta$ in $O(|B_Y| + 2^{|c^*|})$ queries. As long as there exists a conjunction $nimp$ in $\Delta$ such that $c^* \neq nimp$, line 5bis can generate a query $c'_Y$ with $p \leq |c^*|$. If $\emptyset \subseteq \kappa_\Delta(c'_Y) \subseteq \Delta_p$ cannot be satisfied for any $p \leq |c^*|$, then there necessarily exists an $c'_Y$ (satisfying $c^*$ and violating $nimp$) with $\kappa_{\Delta|_{c^*}}(c'_Y) = \emptyset \subseteq \Delta_{|c^*|}$ and $nimp \in \kappa_\Delta(c'_Y)$, otherwise we would have $c^* \models nimp$. As a result, line 5bis can never return a query $c'_Y$ with $p > |c^*|$ if there exists $nimp$ in $\Delta$ such that $c^* \neq nimp$. Suppose first that $\text{ASK}(c'_Y) = \text{yes}$. By construction of $c'_Y$, we know that at least one elementary constraint $c_i$ of the initial $B_Y$ (line 2) is violated by $c'_Y$. Thus, all the conjunctions containing $c_i$ are removed from $\Delta$ and the number of positive queries is bounded above by $|B_Y|$. Suppose now that $\text{ASK}(c'_Y) = \text{no}$. By construction of $c'_Y$, we know that $\Delta_p \setminus \kappa_{\Delta_p}(c'_Y)$ is not empty for some $p \leq |c^*|$, and all these conjunctions in $\Delta_p \setminus \kappa_{\Delta_p}(c'_Y)$ disappear from $\Delta_p$ in line 15 because they are joined with other conjunctions of $\kappa_\Delta(c'_Y)$. Hence, the number of negative queries is bounded above by the number of possible conjunctions in $\Delta_{|c^*|}$, which is in $O(2^{|c^*|})$.

Once all the conjunctions not implied by $c^*$ have been removed from $\Delta$, $\Delta$ only contains $c^*$ and conjunctions included in the set $I_{c^*}$ of elementary constraints implied by $c^*$. We show that removing from $\Delta$ all conjunctions implied by $c^*$ is performed in $O(2^{|I_{c^*}|})$ queries. As all conjunctions remaining in $\Delta$ are implied by $c^*$, all queries will be negative. By construction of such a negative query $c'_Y$, we know that $\Delta \setminus \kappa_\Delta(c'_Y)$ is not empty. All these conjunctions in $\Delta \setminus \kappa_\Delta(c'_Y)$ disappear from $\Delta$ in line 15 because they are joined with other conjunctions of $\kappa_\Delta(c'_Y)$. Thus, each query removes at least one element from $\Delta$, which is a subset of $\{c^*\} \cup 2^{I_{c^*}}$. As a result, the number of such queries is in $O(2^{|I_{c^*}|})$. □

Corollary 1 Given a basis $B$, a target network $T$, and a scope $Y$ such that $B_Y$
contains a constraint \( c^* \) equivalent to the conjunction of constraints of \( T \) with scope \( Y \) and there does not exist any \( c \) in \( B_Y \) such that \( c^* \rightarrow c \), \textbf{FindC} returns \( c^* \) in \( O(|B_Y|) \) queries, which is included in \( O(|\Gamma|) \).

The good news brought by Corollary 1 are that despite the join operation required in \textbf{FindC} to deal with non-normalized networks, QUACQ2 is linear in the size of the language \( \Gamma \) when the target network is normalized and \( \Gamma \) does not contain constraints subsuming others.

5 Learning Simple Languages

The performance of QUACQ2 (in terms of the number of queries submitted to the user) crucially depends on the nature of the relations in the language \( \Gamma \). Some constraint languages are intrinsically harder to learn than others, and there may exist languages that are easy to learn using a specialized algorithm but difficult to learn using QUACQ2.

Determining precisely how QUACQ2 fares when compared with an optimal learning algorithm (that uses partial queries) on a given language \( \Gamma \) is in general a very difficult question. However, if \( \Gamma \) is simple enough then a complete analysis of the efficiency of QUACQ2 is possible. In this section, we focus on constraint languages built from the elementary relations \( \{=, \neq, >\} \) and systematically compare QUACQ2 with optimal learning algorithms. We will measure the number of queries as a function of the number \( n \) of variables; our analysis only assumes that the example \( e_Y \) generated in line 4 of QUACQ2 is complete (i.e., \( Y = X \)) and is a solution of \( L \) that maximizes the number of violated constraints in the basis \( B \).

The next Theorem summarizes our findings. For the sake of readability, its proof is delayed at the end of the section.

\textbf{Theorem 3} Let \( \Gamma \subseteq \{=, \neq, >\} \) be a non-empty constraint language over a finite domain \( D \subseteq \mathbb{Z}, |D| > 1 \). The following holds:

- If \( |D| = 2 \), then QUACQ2 learns networks over \( \Gamma \) in \( \Theta(n \log n) \) queries in the worst case. This is asymptotically optimal, except for \( \Gamma = \{>\} \) for which the optimum is \( \Theta(n) \).

- If \( |D| > 2 \), then in the worst case QUACQ2 learns networks over \( \Gamma \) in
  
  (i) \( \Theta(n \log n) \) queries if \( \Gamma = \{=\} \), which is asymptotically optimal, and
  
  (ii) \( \Theta(n^2 \log n) \) queries otherwise, while the optimum is \( \Theta(n^2) \).

Note that for all these languages, the asymptotic number of queries made by QUACQ2 differs from the best possible by a factor that is at most logarithmic.

The proof of Theorem 3 is based on the following six lemmas. The first three (Lemmas 1, 2 and 3) derive unconditional lower bounds on the number of queries necessary to learn certain constraint languages from a simple counting argument. Lemmas 4, 5 and 6 (together with Theorem 2) will then establish matching upper bounds.
Lemma 1 Let $\Gamma$ be a constraint language over a finite domain $D \subset \mathbb{Z}$, $|D| > 2$, such that $\{>,\neq\} \cap \Gamma \neq \emptyset$. Then, learning constraint networks over $\Gamma$ requires $\Omega(n^2)$ partial queries in the worst case.

Proof. Let $d_1, d_2, d_3$ be three values in $D$ such that $d_1 > d_2 > d_3$ and $(X, D)$ be a vocabulary with an even number $n$ of variables. Let $\mathcal{C}_n$ denote the set of all possible solution sets of constraint networks over $\Gamma$ with vocabulary $(X, D)$. For any $(i, j) \in [1, \ldots, n/2] \times [n/2 + 1, \ldots, n]$ we define the assignment $\phi_{ij} : X \rightarrow D$ as follows:

$$
\phi_{ij}(X_q) = \begin{cases} 
  d_1 & \text{if } q \in [1, \ldots, n/2]\{i\} \\
  d_2 & \text{if } q \in \{i, j\} \\
  d_3 & \text{if } q \in [n/2 + 1, \ldots, n]\{j\}
\end{cases}
$$

Now, let $R$ denote a relation in $\{>,\neq\} \cap \Gamma$ and observe that $R$ contains the three tuples $(d_1, d_2), (d_1, d_3), (d_2, d_3)$ but not the tuple $(d_2, d_2)$. Then, for any subset $S \subseteq S = \{\phi_{ij} \mid (i, j) \in [1, \ldots, n/2] \times [n/2 + 1, \ldots, n]\}$ the constraint network $C^S = \{R(X_i, X_j) : \phi_{ij} \notin S\}$ over $\Gamma$ has the property that $\text{sol}(C^S) \cap S = S$. In particular, for any two distinct sets $S_1, S_2 \subseteq S$ we have $\text{sol}(C^{S_1}) \neq \text{sol}(C^{S_2})$ and hence

$$
|\mathcal{C}_n| \geq |\{C^S \mid S \subseteq S\}| = 2^{|S|} = 2^{(n/2)^2}.
$$

It follows that learning constraint networks over $\Gamma$ requires $\Omega(n^2)$ partial queries since each query only provides a single bit of information on the target network.

Lemma 2 Let $\Gamma$ be a constraint language such that $\{=\} \subseteq \Gamma$. Then, learning constraint networks over $\Gamma$ requires $\Omega(n \log n)$ partial queries in the worst case.

Proof. In a constraint network over $\{=\}$, all variables of a connected component must be equal. In particular, two constraint networks over $\{=\}$ with the same variable set $X$ are equivalent (i.e. have the same solution set) if and only if the partitions of $X$ induced by the connected components are identical. The number of possible partitions of $n$ objects is known as the $n$th Bell Number $C(n)$. It is known that $\log C(n) = \Omega(n \log n)$ [10], so this entails a lower bound of $\Omega(n \log n)$ queries to learn constraint networks over $\Gamma$.

Lemma 3 Let $\Gamma$ be a constraint language over a domain $D \subset \mathbb{Z}$, $|D| = 2$, such that $\{\neq\} \subseteq \Gamma$. Then, learning constraint networks over $\Gamma$ requires $\Omega(n \log n)$ partial queries in the worst case.

Proof. Since $|D| = 2$ and $\{\neq\} \subseteq \Gamma$, we can simulate an equality constraint $X_i = X_j$ over $D$ by introducing one fresh variable $X_{ij}$ and two constraints $X_i \neq X_{ij}, X_j \neq X_{ij}$. It follows that for every set $S^\neq$ of non-equivalent constraint networks over $\{=\}$ with domain $D$, $n$ variables and $O(n)$ constraints, we can construct a set $S^\neq$ of non-equivalent constraint networks over $\Gamma$ with $n^* = O(n)$ variables and such that $|S^\neq| = |S^\neq|$. As we have seen in the proof of Lemma 2 $|S^\neq|$ can be chosen such that $\log |S^\neq| = \Omega(n \log n)$. In that case, we have $\log |S^\neq| = \Omega(n^* \log n^*)$ and the desired lower bound follows.
Lemma 4 For any finite domain $D \subset \mathbb{Z}$ with $|D| \geq 2$, QuACQ2 learns constraint networks over the constraint language $\{=\}$ in $O(n \log n)$ partial queries.

Proof. We consider the queries submitted to the user in line 6 of QuACQ2 and count how many times they can receive the answers yes and no.

For each no answer in line 6 of QuACQ2, a new constraint will eventually be added to $L$. This new constraint $c$ cannot be entailed by $L$ because the (complete) query generated in line 4 of QuACQ2 must be accepted by $L$ and rejected by $c$. In particular, $c$ cannot induce a cycle in $L$. It follows that at most $n - 1$ queries in line 6 are answered no, each one entailing $O(\log n)$ more queries through the function FindScope and $O(1)$ through the function FindC.

Now we bound the number of yes answers in line 6 of QuACQ2. Let $e_Y$ be an example generated by QuACQ2 in line 4. Let $B^{L^Y}$ denote the set of constraints in $B$ that are not entailed by $L$. In order to obtain a lower bound on the number of constraints in $B^{L^Y}$ that $e_Y$ violates, we consider an assignment $\phi$ to $X$ that maps each connected component of $L$ to a value in $D$ drawn uniformly at random. We will show that the expected number of constraints that $\phi$ violates is $|B^{L^Y}|/2$. Since QuACQ2 selects the assignment that maximizes the number of violated constraints, it will follow that $e_Y$ violates at least half of $B^{L^Y}$.

By construction, the random assignment $\phi$ is accepted by $L$. Furthermore, each constraint $c$ in $B^{L^Y}$ involves two variables belonging to distinct connected components of $L$ so the probability that $\phi$ satisfies $c$ is $|\text{rel}(c)|/|D|^2 = 1/|D|$, where $|\text{rel}(c)|$ denotes the number of tuples in $|D|^2$ that belong to the equality relation (the relation of the constraint $c$). By linearity of expectation, the expected number of constraints that $\phi$ violates is therefore $|B^{L^Y}|(1 - 1/|D|) \geq |B^{L^Y}|/2$. As discussed in the previous paragraph, this implies in particular that $e_Y$ violates at least half the constraints in $B^{L^Y}$. It follows that throughout its execution QuACQ2 will receive at most $\lceil \log |B| \rceil = \lceil \log n^2 \rceil$ yes answers at line 6.

Putting everything together, the total number of queries that QuACQ2 may submit before it converges is bounded by $O(n \log n)$, as claimed. □

Lemma 5 If $|D| = 2$, then QuACQ2 learns constraint networks over the constraint language $\{=, \neq, >\}$ in $O(n \log^2 n)$ partial queries.

Proof. The proof follows the same strategy as that of Lemma 4 although the details are a little more involved. Again, we will count how many queries can be submitted to the user in line 6 of QuACQ2.

Each (complete) query submitted in line 6 that receives a negative answer will eventually add a new, non-redundant constraint to $L$. Observe that if $(L_\leq, L_\geq, L_>)$ denotes the partition of $L$ into sub-networks containing only constraints $\leq, \neq$ and $>$ respectively, then neither $L_\leq$ nor $L_\geq$ may contain a cycle; if $L_\geq$ does then the solution set of $L$ is empty and QuACQ2 will halt at line 5 the next time it goes through the main loop. Therefore, at most $3n$ queries may receive a negative answer in line 6, each entailing $O(\log n)$ additional queries through the function FindScope and $O(1)$ through the function FindC.

In order to bound the number of yes answers in line 6 of QuACQ2, consider an example $e_Y$ generated by QuACQ2 at line 4. Let $B^{L^Y}$ denote the set of
constraints in $B$ that are not entailed by $L$. Again, we claim that $e_Y$ violates at least half the constraints in $B^{L \neq \phi}$.

We assume without loss of generality that $D = \{0, 1\}$, interpreted as the Boolean values true and false. Let $\mathcal{S}$ denote the set of connected components in the constraint network $L_{= \neq}$ (the restriction of $L$ to constraints that are either equalities or disequalities). We say that a connected component $S \in \mathcal{S}$ is free if there does not exist a constraint in $L$ of the form $X_i > X_j$ with either $X_i$ or $X_j$ in $S$. Because $L$ is satisfiable, free connected components $S$ have exactly two satisfying assignments $s, \bar{s}$, where $\bar{s}$ is the logical negation of $s$. All other components have exactly one satisfying assignment $s$.

We construct a random assignment $\phi$ to $X$ as follows. For each connected component $S \in \mathcal{S}$, the restriction of $\phi$ to $S$ is either $s$ or $\bar{s}$ (chosen uniformly at random) if $S$ is free, and $s$ otherwise. By construction $\phi$ is accepted by $L$, and for each variable $X_k \in X$ that belongs to a free component, the probability that $\phi$ assigns $X_k$ to 1 is exactly 1/2. It follows that, for each constraint $c$ in $B^{L \neq \phi}$, the probability that $\phi$ violates $c$ is either 1/2 (if $c$ is an equality or disequality, or a constraint $X_i > X_j$ involving exactly one free component) or 3/4 (if $c$ is a constraint $X_i > X_j$ involving two free components). Overall, the expected number of constraints in $B^{L \neq \phi}$ that $\phi$ violates is at least 1/2 $\cdot |B^{L \neq \phi}|$. In particular, there exists an assignment that violates at least half the constraints in $B^{L \neq \phi}$, and by the way QuAcq2 generates examples in line 4, $e_Y$ does as well.

In conclusion, QuAcq2 will receive $\lceil \log |B| \rceil = O(\log n)$ yes answers and $O(n)$ no answers at line 6, plus $O(n \log n)$ answers within FindScope and FindC. The total number of queries made by QuAcq2 is therefore bounded by $O(n \log n)$. □

Lemma 6 If $|D| = 2$, then constraint networks on the language $\{>\}$ can be learned in $O(n)$ partial queries.

Proof. Suppose that the constraint network we are trying to learn has at least one solution. Observe that in order to describe such a problem, the variables can be partitioned into three sets: one for variables that must take the value 1 (i.e., on the left side of a $>$ constraint), a second for variables that must take the value 0 (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, $\mathcal{L}, \mathcal{R}, \mathcal{U}$ initially empty and standing respectively for Left, Right and Unknown. During this phase, we have three invariants:

1. There is no $X_i, X_j \in \mathcal{U}$ such that $X_i > X_j$ belongs to the target network
2. $X_i \in \mathcal{L}$ iff there exists $X_j \in \mathcal{U}$ and a constraint $X_i > X_j$ in the target network
3. $X_i \in \mathcal{R}$ iff there exists $X_j \in \mathcal{U}$ and a constraint $X_j > X_i$ in the target network

We go through all variables of the problem, one at a time. Let $X_i$ be the last variable picked. We query the user with an assignment where $X_i$, as well
as all variables in $U$ are set to 1, and all variables in $R$ are set to 0 (variables in $L$ are left unassigned). If the answer is yes, then there are no constraints between $X_i$ and any variable in $U$, hence we add $X_i$ to $U$ without breaking any invariant. Otherwise we know that $X_i$ is either involved in a constraint $X_j > X_i$ with $X_j \in U$, or a constraint $X_j > X_i$ with $X_j \in U$. In order to decide which way is correct, we make a second query, where the value of $X_i$ is flipped to 0 and all other variables are left unchanged. If this second query receives a yes answer, then the former hypothesis is true and we add $X_i$ 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 variable $X_i \in U$, and query the user with an assignment where all variables in $L$ are set to 1, all variables in $R$ are set to 0 and $X_i$ is set to 1. If the answer is no, we conclude that there is a constraint $X_j > X_i$ with $X_j \in L$ and therefore $X_i$ is added to $R$ (and removed from $U$). Otherwise, we ask the same query, but with the value of $X_i$ flipped to 0. If the answer is no, there must exist $X_j \in R$ such that $X_i > X_j$ belongs to the network, hence $X_i$ is added to $L$ (and removed from $U$). Last, if both queries get the answer yes, we conclude that $X_i$ is not constrained. When every variable has been examined in this way, variables remaining in $U$ are not constrained.

Once $L, R, U$ are computed, we construct an arbitrary constraint network $C$ over $\{>\}$ that is consistent with these sets. At this point, either $C$ is equivalent to the target network or our only assumption (the target network has at least one solution) was incorrect. We resolve this last possibility by submitting an arbitrary solution to $C$ to the user. If the answer is yes, then we return $C$. Otherwise, the target network has no solution and we return an arbitrary unsatisfiable network over $\{>\}$. □

We are now ready to prove Theorem 3.

Proof. [of Theorem 3] We first consider the case $|D| = 2$. By Lemma 5, QUACQ2 learns constraint networks over any language $\Gamma \subseteq \{=, \neq, >\}$ in $O(n \log n)$ queries. Furthermore, if $\Gamma$ contains either $\{=\}$ or $\{\neq\}$ then this bound is optimal by Lemma 2 and Lemma 3. This leaves the case of $\Gamma = \{>\}$. By Lemma 6 this language is learnable in $O(n)$ queries; this upper bound is tight since there are $\Omega(2^{n/2})$ non-equivalent constraint networks over $\{>\}$ on $n$ variables. (Take, for instance, the $2^{n/2}$ sub-networks of $C = \{(X_i > X_{n/2+i}) \mid 1 \leq i \leq n/2\}$ for $n$ even.) On the other hand, such constraint networks can have $\Omega(n)$ non-redundant constraints and QUACQ2 learns $O(1)$ constraints per call to FindScope. Each of these calls to FindScope takes $\Omega(\log n)$ queries, so in the worst case QUACQ2 requires $\Omega(n \log n)$ queries. Combining this observation with Lemma 5 we obtain that QUACQ2 learns networks over $\{>\}$ (with domain size 2) in $\Theta(n \log n)$ queries in the worst case.

Now, assume that $|D| > 2$. If $\Gamma = \{=\}$ then by Lemma 2 and Lemma 4 QUACQ2 learns networks over $\Gamma$ in $\Theta(n \log n)$ queries in the worst case and this
bound is optimal. For every other language, Lemma 1 establishes a universal worst-case lower bound of $\Omega(n^2)$ queries. A straightforward learning algorithm that examines all possible ordered pairs of variables and uses partial queries to determine the constraints of the target network on each pair will converge after $O(n^2)$ partial queries. Such constraints networks can have $\Omega(n^2)$ non-redundant constraints, so in the worst case QUACQ2 submits $\Omega(n^2 \log n)$ queries. This matches the general upper bound from Theorem 2 since the basis has size $O(n^2)$.

\[\square\]

6 Experimental Evaluation

In this section, we experimentally evaluate QUACQ2. The purpose of our evaluation is to answer the following questions:

[Q1] How does QUACQ2 behave in its basic setting?

[Q2] How to make QUACQ2 faster to generate queries?

[Q3] How effective is QUACQ2 when a background knowledge is provided?

In the following subsections, we first describe the benchmark instances. Second, we evaluate QUACQ2 in its basic setting. This baseline version allows us to observe that QUACQ2 may be subject to long query-generation times. We then propose a strategy to make QUACQ2 faster in generating queries. We validate this strategy on our benchmark problems. Finally we evaluate the efficiency of QUACQ2 when a background knowledge is provided. This last experiment shows us that the number of queries required by QUACQ2 to converge can dramatically decrease when the user is able to provide some background knowledge about the problem to acquire.

For each of our experiments, QUACQ2 was run ten times on each problem and the reported results are the averages of the ten runs. For each run, we have set a time limit of one hour on the time to generate a query, after which a time out (TO) was reported. All the results reported in this section were obtained with the version of FindC that uses line 5bis described in Section 4.3. We also tried the basic version that uses line 5 described in Algorithm 3. The results did not make any significant difference. All experiments were conducted using C++ platform\(^3\) on an Intel(R) Xeon(R) E5-2667 CPU, 2.9 GHz with 8 Gb of RAM.

The performance of QUACQ2 is measured according to the following criteria:

\[\begin{align*}
|T| & \text{ size (i.e., number of constraints) of the target network } T, \\
|L| & \text{ size of the learned network } L, \\
\#Q & \text{ total number of queries to learn a network } L \text{ equivalent to } T,
\end{align*}\]

\(^3\)gite.lirmm.fr/constraint-acquisition-team/quacq-cpp

23
The total number of queries to converge (i.e., until it is proved that $L$ is equivalent to $T$),

the average size of all queries,

the cumulated waiting time until a network $L$ equivalent to $T$ is learned, that is, time needed to generate all the queries until this network $L$ equivalent to $T$ is found,

the cumulated waiting time until convergence is reported,

the average time needed to compute a query,

the maximum waiting time between two queries, and

the number of runs that finished without triggering the 1-hour cutoff.

6.1 Benchmark Problems

We evaluated QuAcq2 on a variety of benchmark problems whose characteristics are the following.

**Problem Purdey** [22]. Four families stopped by Purdey’s general store, each to buy a different item. They all paid with different means. Under a set of additional constraints given in the description, the problem is to match each family with the item they bought and how they paid for it. This problem has a single solution. The target network of Purdey has 12 variables with domains of size 4 and 27 binary constraints. There are three types of variables, family, bought and paid, each of them containing four variables. We initialized QuAcq2 with a basis of constraints of size 396 from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

**Problem Zebra**. The target network of the well-known Lewis Carroll’s zebra problem is formulated using 25 variables of domain size of 5 with 5 cliques of $\neq$ constraints and 14 additional constraints given in the description of the problem. The problem has a single solution. We initialized QuAcq2 with a basis of 2700 unary and binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =, \circ vale, i_{11}, i_{v1}\}$, where $\circ vale$ denotes the unary relation $(x \circ vale) \circ o \in \{\geq, \leq, <, >, \neq, =\}$ and $\circ vale \in 1..5$, and where $i_{11}$ and $i_{v1}$ respectively denote the distance relations $|x - y| = 1$ and $|x - y| \neq 1$.

**Problem Golomb** [18 prob006]. A Golomb ruler problem is to put a set of $n$ marks on a ruler so that the distances between marks are all distinct. This is encoded as a target network with $n$ variables corresponding to the $n$ marks, and constraints of varying arity. We learned the target network of 350 constraints encoding the 8-marks ruler. We initialized QuAcq2 with a basis of 1680 binary, ternary and quaternary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =, i_{zt}^{xy}, i_{zt}^{y}x\}$, where $i_{zt}^{xy}$ and $i_{zt}^{y}x$ respectively denote the distance relations $|x - y| = |z - t|$.
and $|x - y| \neq |z - t|$. Observe that when $x$ and $z$, or $y$ and $t$ represent the same variable, $u_{xy}^z$ and $u_{xy}^z$ yield ternary constraints.

**Problem Random.** We generated a binary random target network with 50 variables, domains of size 10, and 122 binary constraints. The 122 binary constraints are iteratively and randomly selected from the complete graph of binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. When a constraint is randomly selected it is inserted in the target network only if this pair of variables is not already linked by a constraint and if the new constraint is not implied by the already selected constraints. QuAcq2 is initialized with a basis of constraints containing the complete graph of 7350 binary constraints from $\Gamma$.

**Problem RLFAP.** The Radio Link Frequency Assignment Problem is to provide communication channels from limited spectral resources so as to avoid interferences between channels [14]. The constraint network of the instance we selected has 50 variables with domains of size 40 and 125 binary constraints (arithmetic and distance constraints). We initialized QuAcq2 with a basis of 12,250 constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =_{val}, >_{val}\}$, where $=_{val}$ and $>_{val}$ respectively denote the distance relations $|x - y| = val$ and $|x - y| > val$, and val $\in \{12, 14, 28, 35, 56, 84, 238\}$.

**Problem Sudoku.** The Sudoku logic puzzle is a 9 \times 9 grid. It must be filled in such a way that all the rows, all the columns and the 9 non-overlapping 3 \times 3 squares contain the numbers 1 to 9. The target network of Sudoku has 81 variables with domains of size 9 and 810 binary $\neq$ constraints on rows, columns and squares. QuAcq2 is initialized with a basis $B$ of 19,440 binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

**Problem Jigsaw.** The Jigsaw Sudoku is a variant of Sudoku in which the 3 \times 3 squares are replaced by irregular shapes. We used the instance of Jigsaw Sudoku displayed in Figure 1. The target network has 81 variables with domains of size 9 and 811 binary $\neq$ constraints on rows, columns and shapes. QuAcq2 is initialized with a basis $B$ of 19,440 binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

### 6.2 [Q1] QuAcq2 in its basic setting

When QuAcq2 is used in its basic setting, we denote it by QuAcq2 \_basic. What we call the basic setting is when, in line 4 of Algorithm 1, QuAcq2 uses the function GenerateExample \_basic described in Algorithm 2. GenerateExample \_basic computes a complete assignment on $X$ satisfying the constraints in $L$ and violating at least one constraint from $B$. We build a network $C$ that contains the constraints from the network $L$ already learned (line 1), plus a reification of the constraints in $B$. A Boolean $b_i$ is introduced for each $c_i \in B$. This Boolean is forced to be true if and only if the constraint $c_i$ is satisfied.
Figure 1: Our instance of Jigsaw problem.

Algorithm 4: GenerateExample.basic\((X, B, L)\)

1. \(C \leftarrow L\);
2. \(\text{foreach } c_i \in B \text{ do } C \leftarrow C \cup \{b_i \leftrightarrow c_i\}\);
3. \(C \leftarrow C \cup \{\sum b_i \neq |B|\}\);
4. \(e \leftarrow \text{solve}(C)\);
5. \(\text{return } e[X]\);

(line 2). We then force the sum of \(b_i\)'s not to be equal to \(|B|\) (line 3). Function \text{solve} is called on \(C\) (line 4) and returns a solution of \(C\), or \(\perp\) if no solution exists. Finally, the projection on \(X\) of the solution is returned (line 5). The constraint solver inside \text{solve} uses the \text{dom/wdeg} variable ordering heuristic \[13\] and a random value selection.

Table 3 reports the results of running QUACQ2.basic on all our benchmark problems. The first observation we can make by looking at the table is that there are only four problems on which QUACQ2.basic has been able to converge in all of the ten runs (Purdey, Zebra, Golomb) or in some of them (Random). For Random, on which QUACQ2.basic converges 7 times out of 10, Table 3 reports the averages of these 7 runs.

Table 3: QUACQ2.basic. All results are averages of ten runs (time in seconds).

| Instance | \(|T|\) | \(|L|\) | \(|Q_A|\) | \(|Q_C|\) | \(|Q|/|X|\) | \(t_{\text{A}}\) | \(t_{\text{C}}\) | \(\bar{t}\) | \(t_{\text{max}}\) | \(|C|\) |
|----------|----------|----------|----------|----------|----------|---------|---------|--------|---------|--------|
| Purdey   | 27       | 26.2     | 175.3    | 177.1    | 5.0/12   | 0.08    | 0.09    | 0.00   | 0.01    | 10     |
| Zebra    | 64       | 61.1     | 555.6    | 555.8    | 8.1/25   | 2.54    | 2.54    | 0.00   | 1.50    | 10     |
| Golomb   | 350      | 96.4     | 351.5    | 351.5    | 4.8/8    | 116.39  | 217.34  | 0.33   | 8.70    | 10     |
| Random   | 122      | 122.0    | 1 082.2  | 1 092.0  | 20.8/50  | 2.08    | 85.94   | 0.08   | 83.80   | 7      |
| RLFAP    | 125      | 98.5     | 1 103.6  | –        | –        | 43.35   | –       | –      | \(\perp\) | 0      |
| Sudoku   | 810      | 775.7    | 6 849.9  | –        | –        | 214.16  | –       | –      | \(\perp\) | 0      |
| Jigsaw   | 811      | 764.0    | 6 749.6  | –        | –        | 224.18  | –       | –      | \(\perp\) | 0      |

\(\perp = 1\) hour
We first focus our attention on these four problems: Purdey, Zebra, Golomb, and Random. Let us first compare the columns \(|T|\) and \(|L|\). On Purdey and Zebra, we observe that the size of \(L\) is slightly smaller than the size of the target network \(T\). This is due to a few constraints that are redundant wrt to some subsets of \(T\). On Golomb, \(|L| \ll |T|\) (96 and 350 respectively) because our target network with all quaternary constraints \(|X_i - X_j| \neq |X_k - X_l|\) contains a lot of redundancies QUAcq2.basic detects convergence before learning them. Finally, as Random does not have any structure, it does not contain any redundant constraint, and \(|L| = |T|\). The column \(|\overline{Q}|/|X|\) shows us that the queries asked by QUAcq2.basic are often much shorter than \(|X|\). The average size \(|\overline{Q}|\) of queries varies from one third to one half of \(|X|\). The the number of queries \#Qc is two to seven times smaller than the size of the basis \(B\). This means that each positive query leads to the removal of several constraints from \(B\). Let us now compare the costs of finding the right network (columns \#Qa and timea) and the costs of converging (columns \#Qc and timec). This tells us a lot about the end of the learning process. On Purdey and Zebra, \#Qa and timea are similar to \#Qc and timec (respectively), which means that QUAcq2.basic learns constraints until the very end of the process. On Golomb, \#Qa and \#Qc are again similar, but timec is much larger than timea. The reason is that after having learned all the constraints necessary to have a \(L\) equivalent to \(T\), GenerateExample.basic spends 100 seconds to show that \(L \models B\), which proves convergence. On Random, we observe yet another behavior. As on Golomb, timec is much larger than timea (almost two orders of magnitude larger), but \#Qc is also larger than \#Qa. The reason is that QUAcq2.basic has found a network \(L\) equivalent to \(T\) ten queries before the end and spends the end of the learning process generating complete queries that are positive and that allow QUAcq2.basic to remove useless constraints from \(B\) and finally prove convergence. This last phenomenon is probably due to the sparseness of \(T\) in Random. The columns \(\overline{\mathfrak{t}}\) and \(t_{max}\) tell us that most queries are very easy to generate (from milliseconds to one third of a second in average) and that most of the time is in fact consumed by generating the last positive queries. Random is an extreme case where the very last query consumes forty times the time needed for the whole process of learning \(T\).

Let us now move our attention to the last three problems in Table 3, namely, RLFAP, Sudoku, and Jigsaw. On these problems, on each of the ten runs, QUAcq2.basic reaches the 1-hour cutoff on the time to generate a query. However we see that \#Qa and timea, which represent the cost of learning a network equivalent to \(T\) without having proved convergence, are reported in the table. For all the runs and all problems, QUAcq2.basic has found a network \(L\) equivalent to \(T\) before reaching the cutoff. This is the proof of convergence that leads QUAcq2.basic to the time-out. \#Qa and timea represent the cost of learning a network equivalent to \(T\) but QUAcq2.basic does not know it is the target. Similarly to the first four problems, the number of queries required to learn a network equivalent to \(T\) is significantly smaller than the size of \(B\) (from three to eleven times smaller).

From this first experiment we conclude that QUAcq2.basic learns small
constraint networks in a number of queries always significantly smaller than the size of the basis and generates queries in very short times. However, as soon as the size of the target network increases, the time to generate the last queries becomes prohibitive for an interactive learning process.

6.3 [Q2] Faster query generation

6.3.1 GenerateExample.cutoff: Generating (partial) queries with a time limit

The experiments in Section 6.2 have shown that QUACQ2.basic can be subject to excessive waiting time between two queries. This prevents its use in an interactive process where a human is in the loop. In this section we propose a new version of GenerateExample that fixes this weakness. We start from the observation that the example generated by GenerateExample in line 4 of Algorithm 1 does not need to be an assignment on $X$. Any partial assignment is satisfactory as long as it does not violate any constraint from $L$ and violates at least one constraint from $B$. We propose thus GenerateExample.cutoff, a new version of GenerateExample that quickly returns a partial assignment on a subset $Y$ of $X$ accepted by $L$ and violating at least one constraint from $B$. The main idea is to modify the function solve so that it can be called with a cutoff.

Function $solve(C, S, obj, ub)$ takes as input a set $C$ of constraints to satisfy, a set $S$ of variables that must be included in the assignment, a parameter $obj$ to maximize, and an upper bound $ub$ on the time allocated. $solve$ returns a pair $(e_Y, t)$ where $e_Y$ is an assignment on a set $Y$ of variables containing $S$, and $t$ is the time consumed by $solve$. If $solve$ proves that $C$ is inconsistent (that is, it found a set $Y$ containing $S$ for which every assignment on $Y$ either violates $C[Y]$ or leads to arc inconsistency on $C$), it returns the pair $(⊥, t)$ where $t$ is the time needed to prove that $C$ is inconsistent. If the allocated time $ub$ is not sufficient to find a satisfying assignment or prove an inconsistency, $solve$ returns the pair $(nil, ub)$. Otherwise, $solve$ returns a pair $(e_Y, t)$ where $e_Y$ is an assignment accepted by $C$ and with highest value of $obj$ found during the allocated time $ub$, and $t$ is the time consumed. When $solve$ is called with $obj = nil$, there is nothing to maximize and the first satisfying assignment (on $S$) is returned. The function $solve$ uses the $bdeg$ variable ordering heuristic [25]. $bdeg$ selects the variable involved in a maximum number of constraints from $B$. By following $bdeg$, $solve$ tends to generate assignments that violate more constraints from $B$, so that in case of yes answer, the size of $B$ decreases faster.

Algorithm 5 describes GenerateExample.cutoff. GenerateExample.cutoff takes as input the set of variables $X$, a current basis of constraints $B$, a current learned network $L$ and a timeout parameter cutoff. GenerateExample.cutoff iteratively picks a constraint $c$ from $B$ until a satisfying assignment is returned or $B$ is exhausted (line 2). The call to $solve$ in line 3 computes an assignment $e_{var(c)}$ on $var(c)$ violating $c$ and accepted by $L$. The time $t$ needed to compute $e_{var(c)}$ is added to the time counter (line 4). If $solve$ returns $e_{var(c)} = ⊥$ (i.e., $L ∪ \{¬c\}$ is inconsistent), $c$ is marked as redundant because it is implied by
Algorithm 5: GenerateExample.cutoff(X, B, L)

1 time ← 0;
2 foreach c ∈ B do
3 (e\textsubscript{var(c)}, t) ← solve(L ∪ {¬c}, var(c), nil, +∞);
4 time ← time + t;
5 if e\textsubscript{var(c)} = ⊥ then
6 mark c as redundant; L ← L ∪ {c}; B ← B \{c\};
7 else
8 (e\textsubscript{Y}', t') ← solve(L ∪ {¬c}, var(c), |Y|, cutoff − time);
9 time ← time + t';
10 if e\textsubscript{Y}' = nil then return e\textsubscript{var(c)};
11 if e\textsubscript{Y}' = ⊥ then
12 mark c as redundant; L ← L ∪ {c}; B ← B \{c\};
13 else return e\textsubscript{Y}';
14 remove all constraints marked as redundant from L;
15 return ⊥;

L. c is then removed from B and added to L (line 6). Adding c to L is required to avoid that QUACQ2 will later try to learn this constraint which is no longer in B. If solve returns an assignment e\textsubscript{var(c)} different from ⊥, GenerateExample.cutoff enters a second phase during which a second call to solve will use the remaining amount of time, cutoff − time, to compute an assignment e\textsubscript{Y}' violating c and accepted by L, whilst maximizing |Y| (line 8). If no such assignment is found in the remaining time, solve returns an e\textsubscript{Y}' equal to nil and GenerateExample.cutoff returns the e\textsubscript{var(c)} found by the first call to solve (line 10). If solve proved the inconsistency of L ∪ {¬c} over a given scope Y (i.e., e\textsubscript{Y}' = ⊥), c is marked as redundant, removed from B, and added to L (line 12), exactly like line 6. GenerateExample.cutoff then goes back to line 2 to select a new constraint from B. Otherwise (i.e., e\textsubscript{Y}' ∈ {nil, ⊥}), GenerateExample.cutoff returns the assignment e\textsubscript{Y}' with the largest size of Y that has been found in the allocated time (line 13). Finally, if all constraints in B have been processed without finding a suitable assignment (line 2), this means that all the constraints that were in B were implied by L. The learning process has thus converged. We just need to remove all constraints marked as redundant from L (line 14) and GenerateExample.cutoff returns ⊥ (line 15).

It is not necessary to remove the redundant constraints but it usually makes the learned network more compact and easier to understand.

6.3.2 Evaluation of GenerateExample.cutoff

We made the same experiments as in Section 6.2 but instead of using QUACQ2.basic, we used QUACQ2.cutoff, that is, QUACQ2 calling GenerateExample.cutoff.
We have set the cutoff to one second so that the acquisition process remains
comfortable in the case where the learner interacts with a human. Table 4 reports the results for the same measures as in Table 3.

The main information that we extract from Table 4 is that the use of GenerateExample.cutoff has a dramatic impact on the time consumption of generating queries. The cumulated generation time for all queries until convergence never exceeds five minutes on any run on any problem whereas QuAcq2.basic was reaching the one-hour cutoff for a query on all runs on three of the problems. Even on the problems where QuAcq2.basic was converging, QuAcq2.cutoff can show a significant speed up. For instance, on Random, QuAcq2.basic was taking a long time to prove that the learned network was equivalent to the target one (timeA, timeC). With QuAcq2.cutoff, timeC and timeA are almost equal and are close to the value of timeA of QuAcq2.basic.

We could have expected that the generation of shorter queries at the end of the learning process leads to an increase in the overall number of queries for QuAcq2.cutoff because shorter positive queries lead to fewer redundant constraints detected. But, when comparing #QA and #QC in Tables 3 and 4, we see that the increase is negligible. There is a -2% to +2% difference on most problems. The exceptions are Golomb and RLFAP, on which QuAcq2.cutoff exhibits an increase of 6% and 7% respectively. Similarly, |Q|/|X| is essentially the same for QuAcq2.basic and QuAcq2.cutoff. As a last observation on Table 4 it can seem surprising that tmax is more than eight seconds on Golomb despite the cutoff of one second in GenerateExample.cutoff. This is because during the learning process, GenerateExample.cutoff repetitively finds redundant constraints without asking a question to the user (line 6 in Algorithm 5).

This experiment shows us that the introduction of a cutoff in the generation of examples completely solves the issue of extremely long waiting times at the end of the process. QuAcq2.cutoff learned all our benchmarks problems with extremely fast query generation. The only price to pay is a slight increase in number of queries in two of the problems.

### 6.4 [Q3] QuAcq2 with background knowledge

In practical applications, it is often the case that the user already knows some of the constraints of her problem. These constraints can be known because
Table 5: QUACQ2.cutoff when the cliques of disequalities are already given in Purdey, Zebra, and Jigsaw, and the symmetry-breaking constraints in Golomb.

| Instance | \(|T|\) | \(|K|\) | \(|L| \setminus |K|\) | \#Q_A | \#Q_C | Q | time_A | time_C | \(\xi\) | \(t_{\text{max}}\) |
|----------|--------|--------|------------------|-------|-------|-----|--------|--------|------|--------|
| Purdey   | 27     | 18     | 9.0              | 70.8  | 81.5  | 5.6/12 | 0.05  | 0.24  | 0.00 | 0.03  |
| Zebra    | 64     | 50     | 14.0             | 185.6 | 199.2 | 7.1/25 | 3.91  | 4.87  | 0.02 | 2.92  |
| Golomb   | 350    | 28     | 70.0             | 246.5 | 246.5 | 5.1/8  | 132.10 | 139.51 | 0.53 | 6.00  |
| Jigsaw   | 811    | 648    | 163.0            | 1 688.8 | 1 715.0 | 20.5/81 | 175.37 | 201.29 | 0.12 | 1.03  |

they are easy to express, or because they are implied by the structure of the problem, or because they have been learned by another tool. For instance, given a solution to a sudoku or to a jigsaw sudoku, ModelSeeker would be able to learn that all the cells in a row must take different values. We can also inherit constraints from a past/obsolete model that needs to be updated because some changes have occurred in the problem. For instance, if new workers have joined the company, we need to learn constraints on them, but the rest of the problem remains unchanged. This set of already known constraints will be called the background knowledge.

QUACQ2 can easily be adapted to handle the case of a background knowledge. In the following, a background knowledge will be a set \(K\) of constraints, where \(K\) is the part of the target problem that we already know, that is, \(K \subseteq T\). Instead of calling QUACQ2 with an empty network \(L\) and a basis \(B\), QUACQ2 is called with \(L\) initialized to \(K\) and the basis initialized to \(B \setminus \{K \cup \bar{K}\}\), where \(\bar{K} = \{c \mid \neg c \in K\}\).

We performed a first experiment with Purdey, Zebra, Golomb, and Jigsaw. In Purdey, we assume that the user was able to express that if four different families buy four different items with four different paying means, then there is a clique of dis-equalities on the four variables representing families, a clique of dis-equalities on the variables representing items to buy, and also a clique on the variables representing paying means. Similarly, in Zebra, we assume that the user was able to express that if there are five people of five different nationalities, there is a clique of dis-equalities on the five variables representing nationalities. Idem on colors of houses, drinks, cigarettes and pets. In Golomb, we assume the user was able to express the symmetry breaking constraint \(X_i < X_j\) for all pairs of marks \(i, j, i < j\). Finally, in Jigsaw, we assume that the user ran ModelSeeker on the solutions of a few instances of these problems and learned that there is a clique of dis-equalities on all the rows and all the columns.

Table 5 reports the results when running QUACQ2.cutoff with an \(L\) initialized to \(K\) as described above for the four problems. The main observation is that the number of queries asked by QUACQ2.cutoff significantly decreases. This decrease in number of queries goes from a factor 1.5 for Golomb, where \(|K|/|T| \approx 0.08\), 2.2 for Purdey, where \(|K|/|T| \approx 0.67\), 2.8 for Zebra, where \(|K|/|T| \approx 0.78\), to 4.0 for Jigsaw, where \(|K|/|T| \approx 0.80\). This shows that the larger the number of constraints already known, the greater the decrease in number of queries. These are good news because the number of queries is
a critical criterion when the user is a human. The second interesting information we learn from this experiment is that most of the other characteristics of QuAcq2.cutoff are essentially the same whatever QuAcq2.cutoff is provided with an initial background knowledge or not. The only exception is the average time to generate a query, $\bar{t}$, that tends to increase in the presence of a background knowledge. This is not surprising because we know that this is when we are close to the end of the acquisition process — when $L$ is large — that query generation costs the most. But this increase only occurs because our queries are very fast to generate, faster than the cutoff of one second. If queries were becoming too long to generate, the cutoff would force shorter queries.

We performed a second experiment on Random, RLFAP, Sudoku, and Jigsaw, that are the problems on which QuAcq2.cutoff asks the more queries. Similarly to the previous experiment, we called QuAcq2.cutoff with a learned network $L$ partially filled with a background knowledge $K$ and a basis initialized to $B \cup \bar{K}$. We varied the size of $K$ by randomly picking from 0% to 90% of the constraints in the target network.

Figure 2 reports the ratio $\#q_{c-wK}/\#q_{c-woK}$ of the number of queries that QuAcq2.cutoff requires to converge with a given $K$ on the number of queries required to converge without any $K$. These results show that when the size of $K$ increases, the number of queries decreases. On RLFAP, QuAcq2.cutoff drops from 1168 queries without $K$ to only 13 queries when $K$ contains 90% of the target network. Importantly, on all problems the decrease is strongly correlated to the amount of background knowledge provided (slope almost linear). This is very good news because this means that it always deserves to add more background knowledge.

We do not report any result on QuAcq2.basic with background knowledge. Whatever the amount of background knowledge provided, QuAcq2.basic suffers from the same drawback as QuAcq2.basic without background knowledge:
the last few queries are prohibitively expensive to generate. On RLFAP, Sudoku, and Jigsaw, QUACQ2\texttt{basic} cannot converge on any run of any size of $K$ within the one-hour time limit on query generation time.

6.5 Discussion

These experiments tell us several important features of QUACQ2. These experiments show us that QUACQ2 can learn any kind of network, whatever all their constraints are organized with a specific structure (such as Sudoku), some of their constraints have a structure (Purdey, Zebra, Golomb, Jigsaw), or they have no structure at all (Random, RLFAP).

A second general observation is that QUACQ2 learns a network in a number of queries always significantly smaller than the size of the basis. This confirms that QUACQ2 is able to select the queries in a way that makes them very informative for the learning process. However, the experiment in Section 6.2 shows that when QUACQ2 is used in its basic version, the time to generate queries can be prohibitive, especially when interacting with a human. We indeed observe that when we are close to the end of the learning process, it can be extremely time consuming to generate a complete example at the start of each loop of acquisition of QUACQ2.

The experiment in Section 6.3.2 shows that a simple adaptation of the way examples are generated (see function \texttt{GenerateExample}.1 in Section 6.3.1), allow us to monitor the CPU time needed to generate an example with a cutoff. In the experiment we see that a cutoff of one second leads to a very smooth interaction between the learner and the user. It is important to bear in mind that even with a cutoff, QUACQ2 ensures the property of convergence.

All experiments in Sections 6.2 and 6.3.2 were performed in the scenario where we do not have any background knowledge about the problem. In these experiments, QUACQ2 is always initialized with an empty learned network $L$. As a result, the number of queries necessary on some of the benchmark problems can seem unrealistic for a real use, especially in the presence of a human user. But it is often the case that the user is able to express some of the constraints of the problem, or that an initial subset of the constraints can be learned with a tool such as ModelSeeker or Arnold. Such tools are able to learn constraints with very few examples when these constraints follow some specific structure. In such scenarios, QUACQ2 can be used as a complement that will learn the missing constraints, that is, the constraint that cannot be captured in a specific structure recognized by these tools. Our experiment in Section 6.4 shows that QUACQ2 is extremely good when it is provided with a background knowledge in the form of a set of constraints. The larger the set of constraints given as a background knowledge, the fewer queries needed to learn the network.
7 Conclusion

We have proposed QuAcq2, 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. Asking the user to classify partial assignments allows QuAcq2 to converge on the target constraint network in a polynomial number of queries (as opposed to the exponential number of queries required when learning with membership queries only). We have shown that QuAcq2 is optimal on certain constraint languages and that it is close to optimal (up to log n worse) on others. Furthermore, as opposed to other techniques, the user does not need to provide positive examples to learn the target network. This can be very useful when the problem has never been solved before. Our experiments show that QuAcq2 in its basic version can be time consuming but they also show that QuAcq2 can be parameterized with a cutoff on the waiting time that allows it to generate queries quickly. These experiments also show that QuAcq2 can learn any kind of network, whatever its constraints follow a specific structure (such as matrices) or not. More, we observed that QuAcq2 behaves very well in the presence of a background knowledge. The larger the background knowledge, the fewer the queries required to converge on the target network. This last feature makes QuAcq2 a perfect candidate to learn missing constraints in a partially filled constraint model. As a last comment, we should bear in mind that all the improvements of QuAcq1 already published in the literature (for instance [26, 6, 15, 1, 25]), can be used with QuAcq2.

References

[1] H.A. Addi, C. Bessiere, R. Ezzahir, and N. Lazaar. Time-bounded query generator for constraint acquisition. In Proceedings of the 15th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018), pages 1–17, Delft, The Netherlands, 2018. Springer.

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

[3] D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of horn clauses. Mach. Learn., 9:147–164, 1992.

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

[5] N. Beldiceanu and H. Simonis. A model seeker: Extracting global constraint models from positive examples. In Proceedings of the Seventeenth Interna-
[6] C. Bessiere, R. Coletta, A. Daoudi, N. Lazaar, Y. Mechqrane, and E.H. Bouyakhf. Boosting constraint acquisition via generalization queries. In Proceedings of the 21st European Conference on Artificial Intelligence, pages 99–104, Prague, Czech Republic, 2014. IOS Press.

[7] 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), pages 123–137, Toronto, Canada, 2004. Springer.

[8] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, and T. Walsh. Constraint acquisition via partial queries. In Proceedings of the 23rd International Joint Conference on Artificial Intelligence, pages 475–481, Beijing, China, 2013.

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

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

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

[12] C. Bessiere, F. Koriche, N. Lazaar, and B. O’Sullivan. Constraint acquisition. Artif. Intell., 244:315–342, 2017.

[13] F. Boussenart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI’04), pages 146–150, Valencia, Spain, 2004.

[14] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. Constraints, 4(1):79–89, 1999.

[15] A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar, and E.H. Bouyakhf. Constraint acquisition using recommendation queries. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI’16), pages 720–726, New York City, NY, 2016.
[16] N.G. De Bruijn. *Asymptotic Methods in Analysis*. Dover Books on Mathematics. Dover Publications, 1970.

[17] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In Michael J. Maher and Jean-Francois Puget, editors, *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, pages 192–204, Pisa, Italy, 1998. Springer.

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

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

[20] M. Kumar, S. Teso, and L. De Raedt. Acquiring integer programs from data. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pages 1130–1136, Macao, China, 2019. ijcai.org.

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

[22] J. Mason. Purdey's general store. *Dell Magazine*, 54:10–10, April 1997.

[23] 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.

[24] L. De Raedt, A. Passerini, and S. Teso. Learning constraints from examples. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 7965–7970, New Orleans, Louisiana, USA, 2018. AAAI Press.

[25] D.C. Tsouros, K. Stergiou, and C. Bessiere. Structure-driven multiple constraint acquisition. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, pages 709–725, Stamford, CT, 2019. Springer.

[26] D.C. Tsouros, K. Stergiou, and P.G. Sarigiannidis. Efficient methods for constraint acquisition. In John N. Hooker, editor, *Proceedings of the 24th International Conference Principles and Practice of Constraint Programming (CP 2018)*, pages 373–388, Lille, France, 2018. Springer.