Learning Constraints through Partial Queries

Christian Bessiere [1], Clément Carbonnel [1], Anton Dries [2], Emmanuel Hebrard [3], George Katsirelos [4,5], Nina Narodytska [6], Claude-Guy Quimper [7], Kostas Stergiou [8], Dimosthenis C. Tsouros [8,9], and Toby Walsh [10]

[1]CNRS, University of Montpellier, France
[2]Nokia Bell Labs, Belgium
[3]LAAS-CNRS, Toulouse, France
[4]Université Paris-Saclay, INRAE, AgroParisTech, UMR MIA Paris-Saclay, France
[5]Université Fédérale de Toulouse, ANITI, France
[6]VMware Research, USA
[7]Université Laval, Quebec City, Canada
[8]University of Western Macedonia, Kozani, Greece
[9]KU Leuven, Belgium
[10]UNSW Sydney and CSIRO Data61, Australia

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 QuAcq2, that, given a negative example, elucidates 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 simple classes of constraint networks and show that our generic algorithm is optimal in some cases. We provide a version of QuAcq2 with a cutoff mechanism that controls the time to generate a query. Our experiments illustrate the good behavior of QuAcq2 in practice, especially in the case where QuAcq2 is executed to learn the missing constraints in a partially filled constraint model. Our experiments also show that QuAcq2 requires significantly fewer queries to learn a network than its predecessor QuAcq1.

Keywords. Constraint programming; Constraint acquisition; Active learning

1 Introduction

Constraint programming (CP) has been used more and more frequently to solve combinatorial 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 constraint network (or constraint model), and the solver finds solutions. However, it appears that specifying the constraints of a problem is not always easy for non-specialists. Hence, the modeling phase constitutes a major bottleneck in the use of CP. Research on *constraint acquisition* tackles this bottleneck. In most of the constraint acquisition systems (e.g., *Conacq.1* [10, 12, 17], *ModelSeeker* [7], *Arnold* [29], *SeqAcq* [36]), the user provides examples of solutions (positive) and non-solutions (negative). Positive examples can come from historical data whereas negative examples can come from failed executions of attempted solutions. The system is also provided with a *learning bias* (i.e., a space of possible constraints).

For instance, in *Conacq.1*, the bias is the set of all constraints that can be specified by using a given language of bounded-arity relations. In *ModelSeeker*, variables are arranged as a matrix, the language is composed of global constraints, and the bias is the set of all possible occurrences of these global constraints on specific subsets of variables such as rows or columns. Given the bias and the set of examples, all these constraint acquisition systems learn sets of constraints that correctly classify all examples (or most of the examples) given so far. This is a form of *passive* learning.

In some cases, learning the constraints with passive learning is not sufficient to obtain a satisfactory constraint model. For instance, *ModelSeeker* or *Arnold* cannot learn constraints that fall outside of the kind of structures (rows, columns, etc.) that those systems use to build their bias. *Conacq.1* does not have such strong restrictions on the kind of structures on which the constraints of its bias can occur. But if we do not have enough training examples, or if they are not diverse enough, *Conacq.1* may return a set of possible constraint networks that is too large to be useful to the user. Active learning can address this issue. An active learner asks queries of the user to elucidate the uncertain parts of the network until it has converged on the unique constraint network (up to equivalence) correctly classifying all examples. (See Section 2 for more details.) Active learning may also be useful when the user, because of cognitive limitations, is not able to formulate the whole set of constraints representing the problem. For instance, Fumagalli et al. use active learning in the context of conceptual modeling to assist the user in acquiring the missing constraints [25]. The user is simply asked to classify model interpretations as allowed or forbidden.

Conacq.2, proposed by Bessiere et al. [14, 17], was the first constraint acquisition system in the literature using active learning. *Conacq.2* proposes complete assignments of the variables of the problem to the user to classify as solutions (positive examples) or non solutions (negative examples). Such questions are called *membership queries* by Angluin [2]. For instance, Menguy et al. used *Conacq.2* in the system PreCa to automatically infer program preconditions from input-output observations of the behavior of the program to analyze [33]. Interestingly, PreCa does not require any human action in the learning process because the classification of examples is obtained by running the program on the given assignment of variables. However, when this is a human who classifies examples, the concern with active learning with membership queries is the large number of queries that may be required to converge on the target set of constraints. Bessiere et al. proved that the number of membership queries required to converge can be exponentially large [16, 17].

In this paper, we propose QuAcq2 (for Quick Acquisition, version 2), a new member in the family of QuAcq algorithms, whose first version –called QuAcq1 from now on– was proposed by Bessiere et al. in [11]. QuAcq algorithms are active learners that ask the user to classify *partial* queries as positive or negative. As opposed to membership queries, partial queries only involve a subset of the variables of the problem. Given a negative example, QuAcq1 and QuAcq2 are able to learn a constraint of the target constraint network in a number of queries logarithmic in the number of variables. As a result, QuAcq1 and QuAcq2 converge on the target constraint network in a polynomial number of queries. In addition, most of these partial queries are easier to classify for a human user than membership queries. Many of those partial queries are either subsets of queries already asked, or they are extremely short (i.e., containing a bounded number of variables). QuAcq algorithms have other advantages. We identify information theoretic lower bounds on the complexity of learning constraint networks that show that QuAcq2 is optimal on some simple languages and close to optimal on others. We provide a complete complexity map for languages composed of the relations $<$, $=$, and/or $\neq$. Another advantage of QuAcq algorithms is that as opposed to most existing techniques, the user does not need to give positive examples. This might be useful if the problem has never been solved before, so that there are no examples of past solutions. However, QuAcq1 has several limitations. QuAcq1 cannot learn constraint networks in which two constraints involve exactly the same variables. Another limitation of QuAcq1 is that some queries –especially at the end of the learning process– can require so much time to generate that it makes difficult to use QuAcq1 in an interactive learning process. QuAcq2 has several new features compared to QuAcq1, and in particular, it addresses the two issues above.

QuAcq2 learns any kind of constraint network, regardless of whether the constraints are organized in a specific structure or not (like QuAcq1) and regardless of whether the network has several constraints defined on the same variables or not (unlike QuAcq1). QuAcq2 can be used when part of the network is already known from the user or from another learning technique. The larger the number of known constraints, the fewer the queries required to converge on the target network. Finally, thanks to partial queries, we are able to propose a version of QuAcq2 with a cutoff mechanism that controls the time to generate a query. This solves the issue of prohibitive generation time of queries in QuAcq1. We provide several experiments on various benchmark problems to validate these advantages in practice.

The rest of the paper is organized as follows. Section 2 positions our approach with respect to the existing literature. Section 3 gives the necessary definitions to understand the technical presentation. Section 4 describes and analyses QuAcq2, an algorithm that learns constraint networks by asking partial queries. In Section 5, we show that QuAcq2 behaves optimally on some simple languages and close to optimally on others. Section 6 reports the results of an experimental evaluation of QuAcq2. Section 7 concludes the paper.

2 Related Work

Constraint acquisition has already been addressed in the literature. The two main approaches are passive learning, where the system learns constraints from a given set of examples, and active learning, where the system asks queries of the user in order to elucidate the constraints.

2.1 Passive constraint acquisition

An early approach to passive constraint acquisition is the algorithm *Conacq.1*, proposed by Bessiere et al. [10, 12, 17]. *Conacq.1* uses the version space learning paradigm. Based on the examples of solutions (positive) and non-solutions (negative) that are provided by the user, *Conacq.1* builds a version space that only contains those constraint networks that accept all positive examples and reject all negative examples. The version space is compactly represented by a clausal formula. *Conacq.1* is a generic algorithm in the sense that it does not require the problem to have any specific structure.

Recently, several other generic passive learners have been proposed. Prestwich et al. proposed *BayesAcq*, a technique that trains a Naive Bayes classifier to discriminate between solutions and non-solutions. A constraint network is then derived from the trained classifier [37]. In the standard setting proposed by Prestwich et al., *BayesAcq* generally returns the network containing all constraints violated by at least one negative example and not violated by any positive example. This network belongs to the version space returned by *Conacq.1*. However, a noticeable difference is that if there exists a constraint rejecting one positive example e^+ and also rejecting many (i.e., more than 100 in the standard setting) negative examples, *Conacq.1* discards all networks containing this constraint from the version space whereas *BayesAcq* considers e^+ as an error. Prestwich also proposed *SeqAcq*, another passive learner robust to errors [36]. *SeqAcq* uses a statistical approach based on sequential analysis. Given two parameters A and R and a set of examples, *SeqAcq* returns a network that contains all constraints from the bias that reject fewer than R positive example and at least A negative examples. In its standard setting ($R = 1$, $A = 50$), *SeqAcq* returns a network that belongs to the version space of *Conacq.1*.

Lallouet et al. proposed a passive learning method based on inductive logic programming [30]. Their system uses background knowledge describing the structure of the problem and, given a set of examples, learns a representation of the problem that correctly classifies the examples. A limitation of this approach is that the user has to be able to provide the whole structure of the problem.

A successful approach to passive learning is *ModelSeeker*, proposed by Beldiceanu and Simonis [7]. The user provides positive examples to the system, which arranges their variables as a matrix. Then *ModelSeeker* uses the global constraints catalog ([5]) to identify global constraints that are

satisfied by particular subsets of variables in all the examples. Such particular 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 find a good model with few examples when the problem has an underlying matrix structure. In *ModelSeeker*, we assume that the user has enough expertise in CP to select the right global constraints among the candidate constraints that the system proposes. *ModelSeeker* cannot find constraints other than global constraints from the catalog holding on specific structures it can recognize. Beldiceanu et al. have successfully used *ModelSeeker* in conjunction with *Conacq.1* to extract a constraint model for computing the power output for power plants at EDF, the national electricity company in France [6].

Pawlak and Krawiec proposed a passive learner in the context of mathematical programming [35]. Given a set of positive and negative examples, their system formulates a mixed integer linear program that will return a set of constraints that correctly classifies all examples. The constraints are picked from a predefined bias containing linear, quadratic, and trigonometric constraints. The returned set of constraints is the one that minimizes a simplicity criterion that can be parameterized. More recently, Kumar et al. have proposed a passive learner called *Arnold* [29]. *Arnold* takes positive examples as input and learns an integer program that accepts these examples as solutions. *Arnold* relies on a tensor-based language for describing polynomial constraints between multi-dimensional vectors. Like *ModelSeeker*, *Arnold* requires that the problem has an underlying matrix structure, and it has to be given as input.

There exist other works related to constraint acquisition, though a bit more distant from our current contribution. In [34], Paramonov et al. apply passive constraint learning to the specific case of Excel sheets. Their system, *TaCLe*, takes an Excel sheet as input and returns a set of constraints in the form of formulas that hold on regions of the sheet (e.g., columns, rows, rectangles). The system *Hassle* proposed by Kumar et al. learns MAX-SAT formulas encoding combinatorial optimizations problems [28]. The originality of the approach is that the system learns both hard and soft clauses and that it works when provided with examples labeled by a context. A context is a partial assignment under which the example holds. Even more remotely related is the *Empirical Model Learning* (*EML*) system from Lombardi et al. [31]. *EML* learns a single constraint that will be embedded into a constraint model. Given real data or a simulator, the constraint is learned with a pure machine learning technique such as decision tree or neural network classifiers. The tuples of the constraint are those classified positive by the classifier. Interestingly, the classifier is transformed into a propagator that can be used inside a CP solver exactly like a standard constraint propagator.

2.2 Active constraint acquisition

Active constraint acquisition is a special case of query-directed learning known as *exact learning* (see Angluin [2], or Bshouty [19]). In the learning formalism introduced by Angluin [2] and used in most approaches to exact learning, two types of queries are used. A membership query requests the user to classify a given example as positive or negative whereas an *equivalence* query asks the user to decide whether the given concept is equivalent to the target concept. In case of a negative answer to an equivalence query, the user must provide a counter example that proves that the two concepts are not equivalent. In applications where we need a proof that the learning system has converged on the target set of constraints, active learning is a good candidate because it can select the queries that will lead to convergence. Nevertheless, in the context of constraint acquisition, Bessiere et al. observed that requiring the user to answer equivalence queries is far too difficult because we assume that the users do not have the skills to express the target network by themselves [17].

An early work in active constraint acquisition is the *Matchmaker* agent proposed by Freuder and Wallace [24]. The matchmaker suggests examples to the user as potential solutions of the problem. A negative answer from the user comes with a "correction" that indicates why the suggestion (i.e., the example) fails. The correction is composed of one (or more) of the constraints that are violated. This approach requires enough expertise from the user to be able to express

constraints that are violated.

Bessiere et al. proposed another active learner called *Conacq.2* [14, 17]. In *Conacq.2*, membership queries are asked of the user. As *Conacq.2* works with membership queries only, it does not require any other expertise from the user than being able to classify examples as solutions or non solutions. An important drawback though, is that the number of membership queries required to converge on the target network can be exponentially large [16, 17].

In [41], Shchekotykhin and Friedrich extend *Conacq.2* to allow the user to attach arguments to the classification of examples. An argument is a set of constraints and/or variable assignments labeled as positive, necessary, sufficient, etc. Such arguments allow the system to converge with fewer examples than *Conacq.2*. However, it puts more effort on the user (some arguments are NP-hard to generate) and again requires more expertise in CP.

Recently, Tsouros and Stergiou have proposed the *PrefAcq* algorithm [43]. *PrefAcq* is able to learn soft constraints in Max-CSPs via active constraint acquisition. A new type of queries is introduced, namely *(partial) preference queries*, inspired by works on preference elicitation. Driven by the user's answers to preference queries, *PrefAcq* is able to learn all the soft constraints appearing in a Max-CSP.

2.3 QuAcq1 and related acquisition techniques

The first version of QuAcq, that we call QuAcq1 in this paper, was published by Bessiere et al. [11]. It was the first time partial queries were used to learn constraint networks. As opposed to QuAcq2, QuAcq1 is devoted to constraint networks for which there does not exist any constraint involving a set of variables included in the set of variables of another constraint. This limitation was partially fixed in the version published in [15] but the network is still required to be normalized (i.e., two constraints cannot be defined on the same set of variables). A second weakness of QuAcq1 is that it is not built on the neat framework of exact learning. This leads to a behavior that is not well-founded. When the target network is not a subset of the bias, QuAcq1 can either converge on a wrong network or return a "collapse" state. Another consequence of QuAcq1 being ill-defined is that QuAcq1 can ask useless queries, that is, queries that are redundant wrt the learning process. In QuAcq2, the problem of constraint acquisition is formulated in a way that is more in line with the exact learning setting used in active concept learning [3, 38]: The target network is a subset of the constraints in the bias. As a consequence, QuAcq2 never asks redundant queries and never collapses.[1]

All the contributions that we present in the rest of this subsection are extensions of QuAcq1. As such, they suffer from the same weaknesses as QuAcq1. They would directly benefit from moving to a QuAcq2-based architecture.

The *MultiAcq* algorithm, proposed by Arcangioli et al. [4], comes from the observation that a negative complete example may violate several constraints from the target network whereas QuAcq1 learns only one constraint from this example. *MultiAcq* learns all the constraints of the target network that are violated by a complete example that is classified as negative. The downside is that MultiAcq needs a number of queries linear in the size of the example to learn a constraint, instead of logarithmic for QuAcq1. In the algorithm *MQuAcq*, Tsouros et al. combine ideas from *MultiAcq* and QuAcq1 [46]. *MQuAcq* learns a maximum number of constraints, like *MultiAcq*, but keeps the logarithmic complexity from QuAcq1 to learn each constraint. Tsouros et al. extended *MQuAcq* to *MQuAcq-2*, an algorithm that exploits the structure of the learned network to focus its queries on specific parts of the problem [45]. When no more constraint can be learned by exploiting the structure, *MQuAcq-2* looks for non-overlapping constraints of the target network, alleviating the high run time that *MQuAcq* incurs when searching for all the constraints violated by a negative example.

Interestingly, QuAcq1 does not require complete positive examples in order to learn a constraint network. This observation was used by Bessiere et al. to propose Ask&Solve, a QuAcq1-

[1] If QuAcq2 is called to learn a target network that is not in the bias, QuAcq2 will stop asking queries when there remains a unique network (not considering implied constraints) consistent with all examples classified so far. QuAcq2 returns that network.

based solver that can find a solution to a problem without modeling it [13]. Ask&Solve tries to find the best trade-off between learning and solving to converge as soon as possible on a solution of the target network rather than learning the whole target network.

One of the drawbacks of active constraint acquisition in general, and QuAcq1 in particular, is that generating membership queries can be very time-consuming. Attempts to reduce the time needed to generate a query were presented by Addi et al. [1] and by Tsouros and Stergiou [42]. In [1], *T-Quacq* generates queries in a bounded amount of time. Unfortunately, this bounded time comes at the risk of failing to learn the target network (called premature convergence). In [42], several heuristics are proposed to speed up query generation. In QuAcq2, query generation does not suffer from premature convergence and the time to generate a query can be controlled by the use of a predefined time cutoff.

In [9], Bessiere et al. observe that in many problems, variables can be of different types (e.g., nurses, doctors, patients), and that variables of the same type are often involved in constraints with the same relation. A new type of queries is introduced, namely *generalization queries*. Generalization queries ask the user whether a constraint learned on variables of a given type can be applied to all variables of this type. Such queries are shown to speed up convergence when plugged into QuAcq1. In the paper of Daoudi et al. [21], the user no longer needs to provide the types in advance because they are learned by analyzing the structure of the currently learned constraint network. In [22], Daoudi et al. introduce *recommendation queries*. By using techniques from the link prediction field, the system *Predict&Ask* asks the user whether or not a predicted constraint belongs to the target network. It is shown that when plugged into QuAcq1, *Predict&Ask* reduces the number of queries. All these techniques require enough expertise from the user to be able to answer these new types of queries.

Finally, Tsouros et al. release the assumption that the user is always able to answer queries by *yes* or *no* [44]. The *MQuAcq-2* algorithm is modified so that it can accommodate from omissions, that is, when the user's answer is *'I don't know'*. It is shown that when the omissions are related to a gap in the user's knowledge, the proposed algorithm can find that gap.

3 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 $scp(c)$ of variables from 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 $scp(c)$. The notation $var(c)$ will refer to the *set* of variables occurring 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 $Y \subset X$ 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 $var(c) \subseteq Y$ 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 satisfies a set C of constraints (or e_Y *is accepted by* C) if and only if e_Y satisfies all constraints in C. An assignment on X that is accepted by C is a *solution* of C. We write $sol(C)$ 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 c in C such that $var(c) = Y$, that is, constraints whose scope exactly fits Y. We say that two networks C and C' are *equivalent* if $sol(C) = sol(C')$.

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

The *target network* is the subset T of B that the user has in mind to represent her problem, without necessarily being able to formulate it. In the following, we assume that T does not

contain any pair of constraints that *directly contradict* each other. By 'directly contradicts', we mean two constraints c and c' such that $var(c) \subseteq var(c')$ and $sol(\{c, c'\}) = \varnothing$.[2] A *membership query* $ASK(e)$ takes as input a complete assignment e in D^X and asks the user to classify it. The answer to $ASK(e)$ is *yes* if $e \in sol(T)$, otherwise the answer is *no*. A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, takes as input a *partial* assignment e_Y in D^Y and asks the user to classify it. The answer to $ASK(e_Y)$ is *yes* if and only if e_Y does not violate any constraint in T. It is important to observe that "$ASK(e_Y)$=*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 $ASK(e_Y)$ 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 Δ 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 *yes* or *no* by the user, 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 $sol(L') = sol(L)$. 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 $sol(C) = sol(C \cup \{c\})$. We also say that C *implies* c (denoted by $C \models c$).

In the algorithms presented in the rest of the paper we will use the *join* operation, denoted by $\bowtie$, which itself uses the *conjunction* of constraints, denoted by $\wedge$. Given two constraints c and c' defined on the same set $var(c)$ $(= var(c'))$ of variables, the conjunction $c \wedge c'$ of c and c' is the constraint whose relation is the intersection of $rel(c)$ and $rel(c')$.[3] Given two sets of constraints S and S' defined on same the set of variables, the join of S with S' is the set of constraints with non-empty relation obtained by pairwise conjunction of a constraint in S with a constraint in S'. That is, $S \bowtie S' = \{c \wedge c' \mid c \in S, c' \in S', c \wedge c' \not\models \bot\}$. A constraint belonging to the bias 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 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. When a network does not contain any conjunction of constraints on any scope, that is, all its constraints are elementary, we call it a *normalized* network.

4 Constraint Acquisition with Partial Queries

In this Section, we describe QuAcq2, a novel active learning algorithm. QuAcq2 takes as input a bias B on a vocabulary (X, D). QuAcq2 asks partial queries of the user until it has converged on a constraint network L equivalent to the target network T, which is expressed with constraints from B.

4.1 Preamble

Before entering in the algorithmic description of QuAcq2, let us give the intuition behind its basic principles. QuAcq2 can be seen as a kind of successor to *Conacq.2*, the active version of *Conacq*. However, there are significant differences with *Conacq.2*. *Conacq.2* is a version space learning algorithm. *Conacq.2* only uses membership queries, that is, it asks the user to classify *complete* assignments of the variables as positive or negative examples. Each example is used to remove networks from the version space. To avoid storing a version space of exponential size, *Conacq.2* compactly represents the version space through a SAT formula, whose solutions represent networks in the version space. When an example e^+ is classified as positive by the user, we know that the set

[2] This minor restriction could easily be released but would lead to slightly more work for the learning algorithm we describe in this paper.

[3] By convention, we will assume the scope is lexicographically ordered to avoid duplicates.

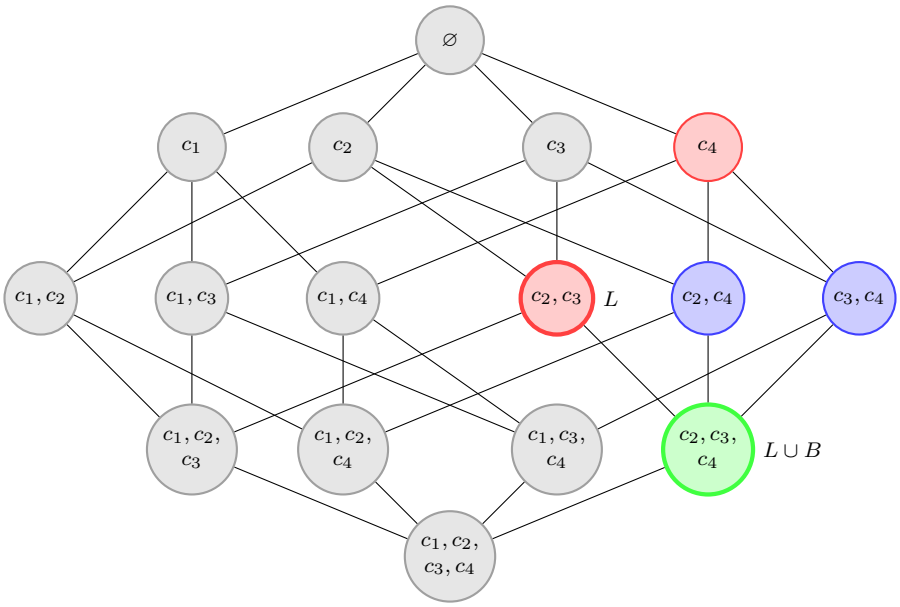

Figure 1: Comparison of the version spaces of *Conacq.2* and QuAcq2 after processing a positive example e_1^+ and two negative examples e_2^- and e_3^-, violating constraints $\{c_1\}$, $\{c_2, c_4\}$, and $\{c_3, c_4\}$, respectively. After processing e_1^+, *Conacq.2* removes all networks containing c_1 from its version space (grey nodes) because they would reject e_1^+. Hence, the most specific bound of the version space becomes the green node $\{c_2, c_3, c_4\}$. When processing the two negative examples e_2^- and e_3^-, *Conacq.2* stores two clauses stating that at least one constraint among $\{c_2, c_4\}$ and at least one constraint among $\{c_3, c_4\}$ must belong to the learned network. The most general bound of the version space then becomes the red nodes $\{c_4\}$ and $\{c_2, c_3\}$, that is, the two solutions satisfying all clauses. Concerning QuAcq2, e_1^+ leads to the removal of c_1 from its bias B, leading to the most specific bound $L \cup B = \{c_2, c_3, c_4\}$, as in *Conacq.2*. When processing e_2^- and e_3^-, thanks to `FindScope` and `FindC`, QuAcq2 learns a unique constraint for each of these two examples, respectively c_2 and c_3 in the figure. This leads to a unique most general bound: the bold red node $L = \{c_2, c_3\}$ in the version space.

$\kappa_B(e^+)$ of constraints violated by that example cannot be part of the learned network (otherwise the example would be rejected). A set of negative unary clauses (one per constraint in $\kappa_B(e^+)$) is added to the SAT formula to prevent the use of these constraints. When an example e^- is classified as negative, we know that at least one of the constraints in $\kappa_B(e^-)$ must belong to the learned network (otherwise the example would not be rejected). A single positive clause (with all constraints in $\kappa_B(e^-)$ as atoms) is added to the SAT formula to enforce the presence of at least one of these constraints. There is a one-to-one mapping between the set of solutions of the SAT formula and the networks belonging to the version space. Any solution to the SAT formula with a minimal set of atoms assigned true corresponds to a most general bound of the version space, that is, a network with a minimal set of constraints. There can be exponentially many such networks. The unique solution to the SAT formula with a maximal set of atoms assigned true corresponds to the network that is the unique most specific bound of the version space. (See Figure 1 for an illustrative example.)

Even if QuAcq2 has similarities with *Conacq.2*, it is a very degenerated case of version space learning. QuAcq2 does not only ask membership queries. QuAcq2 is able to ask partial queries. When processing a positive example e^+, QuAcq2 behaves in a way similar to *Conacq.2*, except that QuAcq2 works directly on the constraints, not on a SAT formula. That is, QuAcq2 directly rules out the set $\kappa_B(e^+)$ of constraints violated by the positive example from the bias B.

When processing a negative example e^-, the behavior of QuAcq2 is quite different from that of *Conacq.2*. Instead of storing a clause containing the set $\kappa_B(e^-)$ of constraints from B that are candidate to reject e^-, QuAcq2 calls the subprocedures `FindScope` and `FindC` to learn a constraint. `FindScope` and `FindC` ask a sequence of queries of the user until one of the constraints of the target network rejecting e^- is elucidated and added to the learned network L. (See again Figure 1 for an illustrative example.)

Whereas the main procedure QuAcq2 generates its queries in a way that has similarities with the way *Conacq.2* does, the queries asked by the subprocedures `FindScope` and `FindC` are generated in a very different way. Given a query e^-_Y generated in the main procedure QuAcq2 and classified as negative by the user, the queries that `FindScope` asks have the form $e^-_Y[R]$ for some subset R of Y. That is, `FindScope` does not generate any new assignment of the variables. It just projects e^-_Y on subsets of Y. Once `FindScope` has found a scope S that contains a constraint from the target network, `FindC` generates queries on scope S until the constraint is found and added to the learned network. By construction of the bias B, S has bounded size.

Thanks to partial queries, QuAcq2 can isolate the scope of one of the constraints from the target network (`FindScope`) and elucidate the constraint that appears on this scope in the target network (`FindC`) in a number of queries logarithmic in the number of variables in the target network. In comparison, *Conacq.2* can require exponentially many queries. A second benefit of QuAcq2 is that it does not need to maintain any kind of compact representation of the version space. The version space of QuAcq2 is a degenerated one. The most specific network is simply the union of the current learned network L and of the constraints remaining in B. The most general network (which is unique, as opposed to standard version spaces) is the current learned network L itself. Therefore, the version space is the set of networks $\{C \mid L \subseteq C \subseteq L \cup B\}$.

4.2 Technical description of QuAcq2

In our description of QuAcq2, we make explicit the possible use of *background knowledge* in the form of already known constraints. In practical applications, it is indeed often the case that the user already knows some of the constraints of her problem. These constraints can be known because 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. 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. It is this set of already known constraints that we will call the background knowledge in the rest of the paper. That is, the background knowledge is a set K of constraints, where K is the part that we already know of the target network T (i.e., $K \subseteq T$).

QuAcq2 (see Algorithm 1) initializes the network L that it will learn to the set K of already known constraints (line 2). If K is not empty, B probably contains constraints that are either directly contradicted by L or implied by L. Constraints that are directly contradicted can safely be removed from B (line 3). Implied constraints can safely be transferred from B to L. As it may be very time consuming to check whether a constraint c is such that $L \models c$, we propose a lighter version, which, for each constraint $c \in B$, simply checks whether there exists a constraint c' in L such that $var(c) \subseteq var(c')$ and $c' \models c$. If there exists such a c', we add c to L (line 4). Line 5 removes all constraints (implied or coming from K) already in L from B. It is important to note that lines 3 and 4 are optional. Skipping them does not hurt correctness of QuAcq2. Nevertheless, it can lead to additional queries asked of the user just to understand that a constraint is contradicted or implied. Another point to bear in mind is that the (redundant) constraints added to L in line 4 can be removed from the returned network once QuAcq2 has converged.[4] After this initialization step, QuAcq2 enters its main loop (line 6). In line 7, QuAcq2 calls function `GenerateQuery` that computes an assignment e_Y on a subset of variables Y satisfying the constraints of L that have a scope included in Y, and violating at least one constraint from

[4]These constraints were added to L just to guarantee correctness of QuAcq2, as we will see later.

Algorithm 1: QuAcq2

```
   In    : bias B
   In    : background knowledge K ⊆ B
   Out   : learned network L
1  begin
2      L ← K;
3      B ← B \ {c | c is directly contradicted by c' ∈ L};                    // optional
4      L ← L ∪ {c ∈ B | ∃c' ∈ L, var(c) ⊆ var(c') ∧ c' ⊭ c};                   // optional
5      B ← B \ L;
6      while true do
7          e_Y ← GenerateQuery(L, B);
8          if e_Y = ⊥ then return "convergence on L";
9          if ASK(e_Y) = yes then
10             B ← B \ κ_B(e_Y);
11         else FindC(e_Y, FindScope(e_Y, ∅, Y), L);
```

B.[5] We will see later that there are multiple ways to design function `GenerateQuery`. If there does not exist any e_Y accepted by L and rejected by B (i.e., `GenerateQuery` returns $\bot$), then all constraints in B are implied by L, and QuAcq2 has converged (line 8). If QuAcq2 has not converged, we ask the user the classification of the assignment e_Y (line 9). If the answer is *yes* (i.e., e_Y does not violate any constraint of the target network T), we remove from B the set $\kappa_B(e_Y)$ of all constraints in B that reject e_Y (line 10). If the answer is *no*, we are sure that e_Y violates at least one constraint of T. We then call function `FindScope` to discover the scope S of one of these violated constraints. Procedure `FindC` will learn (that is, put in L) all constraints of T whose scope is S (line 11).

The structure of the recursive function `FindScope` (see Algorithm 2) is inspired from the QuickXplain algorithm of Junker [27]. QuickXplain was proposed to extract a minimal inconsistent subset of an inconsistent set of constraints. As shown by Rodler [39], the principle of QuickXplain can be applied to any problem in which the property of the subset we are looking for is monotone. In `FindScope`, given an assignment e on variables Y, we are looking for a minimal subset S of Y such that $e[S]$ violates some constraint from the target network T. This is a monotone property because for any S' superset of S, $e[S']$ violates a constraint from T. The queries to an NP-complete oracle in QuickXplain become simple checks of constraint satisfaction, where the oracle is the user.

`FindScope` takes as parameters an example e and two sets R and Y of variables. An invariant of `FindScope` is that e violates at least one constraint of the target network T whose scope is a subset of $R \cup Y$. A second invariant is that any recursive call to `FindScope` returns a subset of Y that is also a subset of the scope of a constraint in T violated by e. If there is at least one constraint in B rejecting $e[R]$ (i.e., $\kappa_B(e[R]) \neq \varnothing$, 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 violate any constraint. We know that $e[R \cup Y]$ violates a constraint. Hence, if Y is a singleton, the variable it contains necessarily belongs to the scope of a constraint that violates $e[R \cup Y]$. `FindScope` returns Y. If none of the return conditions are satisfied, the set Y is split into two balanced parts Y_1 and Y_2 (line 6) and we recursively call `FindScope` 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 (line 8), if $R \cup Y_1$ does not contain any scope S of constraint rejecting e, `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

[5] 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 2: Function `FindScope`

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

second recursive call (line 10), 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, `FindScope` returns a subset S_2 of such a scope such that $S_2 = S \cap Y_1$ 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.

Procedure `FindC` (see Algorithm 3) is designed to learn the constraint of the target network T that has the given scope Y. (By 'the constraint of T' we mean a conjunction of elementary constraints from B that is equivalent to the conjunction of the constraints in T with scope Y.) `FindC` maintains a set Δ of constraints that are candidate to be learned on the scope Y. An invariant of `FindC` is that at any time during its execution, each elementary constraint composing the constraint c to be learned is contained in a subset of c that belongs to Δ. `FindC` learns the constraint on Y by generating assignments on Y that violate some but not all constraints from Δ. When an assignment is classified as positive example, all violated constraints are removed from Δ, and when classified negative, new constraints are generated by joining Δ with its subset of constraints violated by the assignment.

`FindC` takes as parameters e, Y, and L, 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 Δ is initialized to all candidate elementary constraints, that is, the set B_Y of all constraints from B whose scope exactly fits Y (line 2). As we know from `FindScope` that T contains a constraint with scope Y rejecting e, we join Δ with the set of constraints with scope Y rejecting e (line 3). In line 5, an assignment e' is chosen in such a way that Δ contains both constraints satisfied by e' and constraints violated by e'. If no such assignment exists (line 6), this means that all constraints in Δ 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 assignment e' was found, the user is asked to classify e' (line 10). If e' is classified positive, all constraints rejecting it are removed from Δ 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 Δ 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 is able to give.

Algorithm 3: Procedure `FindC`

```
   In      : negative example e
   In      : scope Y
   In Out: network L
1  begin
2      Δ ← B_Y;
3      Δ ← Δ ⋈ κ_Δ(e);
4      while true do
5          choose e'_Y in sol(L[Y]) such that ∅ ⊊ κ_Δ(e'_Y) ⊊ Δ;
6          if e'_Y = ⊥ then
7              pick c in Δ;
8              L ← L ∪ {c}; B ← B \ B_Y; exit;
9          else
10             if ASK(e'_Y) = yes then
11                 Δ ← Δ \ κ_Δ(e'_Y); B ← B \ κ_B(e'_Y);
12             else
13                 S ← FindScope(e'_Y, ∅, Y);
14                 if S ⊊ Y then FindC(e'_Y, S, L);
15                 else  Δ ← Δ ⋈ κ_Δ(e'_Y);
```

QuAcq2 is designed to communicate with users who are not able to provide any more hint than "*Yes, I'm ok: Nothing is wrong on this example*" or "*No, this example does not satisfy my requirements*". We can imagine cases where the user is a bit more skilled than that and can provide answers such as (a): "*This example does not work because there is something wrong on this subset of variables*", or (b): "*This example does not work because it violates this constraint*", or (c): "*This example does not 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 [24]. The more informative the query, the more dramatic the decrease in number of queries needed to find the target constraint network.

4.3 Illustrative examples

We illustrate the behavior of QuAcq2 and its two sub-procedures `FindScope` and `FindC` on two simple illustrative examples. In the first illustrative example, we focus on how `FindScope` and `FindC` work. In the second illustrative example, we show that even if the target network is inconsistent, QuAcq2 is able to work and then returns an inconsistent set of constraints. The reason is that QuAcq2 never requires complete positive examples, as opposed to most other constraint acquisition systems.

4.3.1 Illustrative example showing the details of `FindScope` and `FindC`

Consider the variables $X_1, \ldots, X_5$ with domains $\{-10..10\}$, the language $\Gamma = \{\leq, \neq, \sum^{\neq}\}$, and the bias $B = \{\leq_{ij}, \geq_{ij}, \neq_{ij} | \ i, j \in 1..5, i < j\} \cup \{\sum^{\neq \mathtt{k}}_{\mathtt{ij}} \mid i, j, k \in 1..5, i < j \neq k \neq i\}$, where $\leq_{ij}$ is the constraint $X_i \leq X_j$, $\geq_{ij}$ is $X_j \leq X_i$, $\neq_{ij}$ is $X_i \neq X_j$, and $\sum^{\neq \mathtt{k}}_{\mathtt{ij}}$ is $X_i + X_j \neq X_k$. The target network is $T = \{=_{15}, <_{23}, \sum^{\neq 4}_{23}\}$, where $<_{ij}$ denotes the conjunction $\leq_{ij} \wedge \neq_{ij}$.

Suppose that the first query e_1 generated in line 7 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 9 of QuAcq2 and the user replies *no* because the constraints $=_{15}$ and $\sum^{\neq 4}_{23}$ are violated. As a result, `FindScope`$(e_1, \varnothing, \{X_1 \ldots X_5\})$ is called in line 11 of QuAcq2.

Table 1: `FindScope` on the negative example $(0, 1, 2, 3, 4)$

call	R	Y	ASK	return
0	$\varnothing$	X_1, X_2, X_3, X_4, X_5	$\times$	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*	$\varnothing$
1.2	X_1, X_2, X_3	X_4	$\times$	X_4
2	X_4	X_1, X_2, X_3	$\times$	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	$\times$	X_2

Running `FindScope`

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

- The initial call (call-0 in Table 1) does not ask the query because $R = \varnothing$ and $\kappa_B(e_1[\varnothing]) = \varnothing$. Y is split into two sets $Y_1 = \{X_1, X_2, X_3\}$ and $Y_2 = \{X_4, X_5\}$. Line 7 detects that $\kappa_B(e_1[X_1, X_2, X_3])$ and $\kappa_B(e_1[X_1, X_2, X_3, X_4, X_5])$ are different (e.g., $\geq_{25}$ 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 3 of `FindScope` removes all constraints in $\kappa_B(e_1[X_1, X_2, X_3])$ (i.e., $\geq_{12}$, $\geq_{13}$, $\geq_{23}$) from B. Y is split into two sets $Y_1 = \{X_4\}$ and $Y_2 = \{X_5\}$. Again, $\kappa_B(e_1[X_1, X_2, X_3, X_4])$ and $\kappa_B(e_1[X_1, X_2, X_3, X_4, X_5])$ are different in line 7 ($\geq_{25}$ is still in B), so call-1.1 is performed.

- Call-1.1: $e_1[X_1, X_2, X_3, X_4]$ 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 $\kappa_B(e_1[X_1, X_2, X_3])$ and $\kappa_B(e_1[X_1, X_2, X_3, X_4, X_5])$ 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 $\kappa_B(e_1[X_1, X_2, X_3])$ is already empty (see call-1). In line 5, 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 does not ask the query because $\kappa_B(e_1[X_4])$ is empty. Y is split into 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 ($\geq_{34}$ is still in B), we go to call-2.1.

- Call-2.1: $e_1[X_1, X_2, X_4]$ is classified positive. `FindScope` removes the constraints in $\kappa_B(e_1[X_1, X_2, X_4])$ from B and returns the singleton $\{X_3\}$. 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. `FindScope` removes the constraints in $\kappa_B(e_1[X_3, X_4])$ from B. Y is split into 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 ($\sum_{23}^{\neq 4}$ 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.

Table 2: `FindC`

	assignment (line 5)	ASK (line 10)	Δ (lines 2-3, 11, and 15)	L (line 8)	$B[X_2, X_3, X_4]$ (lines 8 and 11)
`FindC`$((1,2,3), \{X_2, X_3, X_4\})$					
0.	$(1,2,3)$	(no)	$\sum_{23}^{\neq 4}$ $\sum_{23}^{\neq 4} \wedge \sum_{24}^{\neq 3}, \sum_{23}^{\neq 4} \wedge \sum_{34}^{\neq 2}$	$\varnothing$	$\sum_{23}^{\neq 4}, \sum_{24}^{\neq 3}, \sum_{34}^{\neq 2}$ $\leq_{23}, \leq_{24}, \leq_{34}, \neq_{23}, \neq_{24}, \neq_{34}$
1.	$(2,3,1)$	*yes*	$\sum_{23}^{\neq 4}, \sum_{23}^{\neq 4} \wedge \sum_{34}^{\neq 2}$	$\varnothing$	$\sum_{23}^{\neq 4}, \sum_{34}^{\neq 2}$ $\leq_{23}, \neq_{23}, \neq_{24}, \neq_{34}$
2.	$(3,2,1)$	*no*	unchanged	$\varnothing$	unchanged
`FindC`$((3,2), \{X_2, X_3\})$					
3.	$(3,2)$	(no)	$\leq_{23}, <_{23}$	$\varnothing$	unchanged
4.	$(1,1)$	*no*	$<_{23}$	$<_{23}$	$\sum_{23}^{\neq 4}, \sum_{34}^{\neq 1}, \neq_{24}, \neq_{34}$
back to `FindC`$((1,2,3), \{X_2, X_3, X_4\})$					
5.	$(1,2,-1)$	*yes*	$\sum_{23}^{\neq 4}$	$<_{23}, \sum_{23}^{\neq 4}$	$\neq_{24}, \neq_{34}$

- 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 $\varnothing$ 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 11 of QuAcq2 then calls `FindC` with $e_1[X_2, X_3, X_4] = (1,2,3)$ and $Y = \{X_2, X_3, X_4\}$.

Running `FindC`

The trace of the execution of `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 assignment in line 5 of `FindC`. For each of these assignments, we report the assignment generated, its classification, and the new state of Δ, L, and $B[X_2, X_3, X_4]$. We also specify in which lines of `FindC` these changes occur.

- Row-0: The assignment $(1,2,3)$ was not generated in `FindC` but inherited from `FindScope`. By definition of `FindScope`, we know that it is a negative example (denoted by "(no)" in the table). In line 2 of `FindC`, Δ is initialized to the set of constraints from B having scope $\{X_2, X_3, X_4\}$, that is $\{\sum_{23}^{\neq 4}, \sum_{24}^{\neq 3}, \sum_{34}^{\neq 2}\}$, and then in line 3 these constraints are joined with $\sum_{23}^{\neq 4}$, the only constraint in $\kappa_\Delta((1,2,3))$. At this point the learned network L is still empty because `FindScope` did not modify it. $B[X_2, X_3, X_4]$ contains all the constraints from the original B with scope included in $\{X_2, X_3, X_4\}$, except $\geq_{23}, \geq_{24}$, and $\geq_{34}$, which were discarded during call-1, call-2.1 and call-2.2 of `FindScope`, respectively.

- Row-1: In line 5, `FindC` generates the assignment $(2,3,1)$, satisfying some constraints from Δ but not all. The query on $(2,3,1)$ is classified as positive example in line 10. Hence, the violated conjunction $\sum_{23}^{\neq 4} \wedge \sum_{24}^{\neq 3}$ is removed from Δ and all violated constraints in B (i.e., $\sum_{24}^{\neq 3}$, $\leq_{24}$, and $\leq_{34}$) are removed (line 11). L remains unchanged.

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

- Row-3: The assignment $(3,2)$ is known to be a negative example without asking. Lines 2-3 initialize Δ to the set of constraints in $B_{\{X_2, X_3\}}$ and then join them to those rejecting the example. (Note that $<_{23}$ is a shortcut for $\leq_{23} \wedge \neq_{23}$.) L and B remain unchanged.

- Row-4: `FindC` generates the assignment $(1, 1)$, which is classified as negative example. The call to `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 Δ to a the singleton $<_{23}$. As a result, the next loop of `FindC` cannot generate any new assignment in line 5. Line 8 adds $<_{23}$ to L and removes all the constraints with scope $\{X_2, X_3\}$ from B. This subcall to `FindC` exits.

- Row-5: We are back to the original call to `FindC` with the same Δ as in row-2. Line 5 must generate an assignment accepted by L and violating part of Δ. It generates $(1, 2, -1)$, which is classified as positive example. The violated conjunction $\sum_{23}^{\neq 4} \wedge \sum_{34}^{\neq 2}$ is removed from Δ and $\sum_{34}^{\neq 2}$ is removed from B (line 11). The next loop of `FindC` cannot generate any new assignment in line 5 because Δ is now a singleton. Line 8 adds $\sum_{23}^{\neq 4}$ to L and removes all the constraints with scope $\{X_2, X_3, X_4\}$ from B. `FindC` exits.

4.3.2 Learning an inconsistent network

In the previous illustrative example, we can observe that no complete positive example is necessary to learn the constraint $\sum_{23}^{\neq 4}$. The same could happen for learning the two remaining constraints. Indeed, as opposed to other techniques, QuAcq2 does not need examples of solutions to learn the target network. In the extreme case of a target network that does not have any solution, QuAcq2 is able to learn an inconsistent set of constraints.[6] Here is a simple illustrative example.

Consider the variables X_1, X_2, X_3 with domains $\{0..10\}$, the language $\Gamma = \{\neq, =\}$, the bias $B = \{\neq_{ij}, =_{ij} | \ i, j \in 1..3, i < j\}$, and the inconsistent target network $T = \{=_{12}, =_{23}, \neq_{13}\}$.

A compact representation of the trace of the execution of QuAcq2 is displayed in Table 3. Each row reports the results of the actions performed after a query asked of the user. For each of these queries, we report the assignment generated, its classification by the user, the procedure that asked the query (main QuAcq2, `FindScope`, or `FindC`), and the new state of B and L after that query.

- Row-0. L is empty and B contains all possible constraints.

- Row-1. QuAcq2 asks $(1, 1, 1)$, classified as negative, which leads to a call to `FindScope`.

- Row-1.1. `FindScope` asks $(1, 1, -)$, classified as positive, leading to the removal of $\neq_{12}$ from B. As a result of this query, `FindScope` knows that X_3 belongs to the scope it is looking for.

- Row-1.2. Another call to `FindScope` asks $(1, -, 1)$, classified as negative. It is enough to elucidate the scope $\{X_1, X_3\}$ because B does not contain any unary constraint, so $\{X_3\}$ cannot be the scope of a constraint of T. Interestingly, `FindC` does not need any query to learn $\neq_{13}$ because $(1, -, 1)$ satisfies the only other constraint in $B_{\{X_1, X_3\}}$ (i.e., $=_{13}$), which cannot be joined with $\neq_{13}$ as it is its complement.

- Row-2. QuAcq2 asks $(1, 1, 2)$, classified as negative. Interestingly again, `FindScope` returns the scope $\{X_2, X_3\}$ without asking any query because $=_{23}$ is the only constraint in B violated by $(1, 1, 2)$. For the same reason as above, `FindC` does not need any query to learn $=_{23}$.

- Row-3. QuAcq2 asks $(1, 2, 2)$, classified as negative. As $=_{12}$ is the only constraint remaining in B, it is learned without any query from `FindScope` or `FindC`. B is now empty. QuAcq2 returns the constraint network $\{=_{12}, =_{23}, \neq_{13}\}$, on which it has converged.

QuAcq2 has learned the target network T with five queries. None of these queries is a solution of T because T is inconsistent.

[6] Bear in mind though, that by definition of join operation presented in Section 3, QuAcq2 cannot learn an empty constraint. It thus cannot learn a network that is inconsistent for such a trivial reason.

Table 3: QuAcq2 with an inconsistent target network

	assignment	ASK	by whom	B	L
0.				$=_{12}, \neq_{12}, =_{23}, \neq_{23}, =_{13}, \neq_{13}$	$\varnothing$
1.	$(1,1,1)$	*no*	QuAcq2	$=_{12}, \neq_{12}, =_{23}, \neq_{23}, =_{13}, \neq_{13}$	$\varnothing$
1.1.	$(1,1,-)$	*yes*	`FindScope`	$=_{12}$, $=_{23}, \neq_{23}, =_{13}, \neq_{13}$	$\varnothing$
1.2.	$(1,-,1)$	*no*	`FindScope`	$=_{12}$, $=_{23}, \neq_{23}$	$\neq_{13}$
2.	$(1,1,2)$	*no*	QuAcq2	$=_{12}$	$=_{23}, \neq_{13}$
3.	$(1,2,2)$	*no*	QuAcq2	$\varnothing$	$=_{12}, =_{23}, \neq_{13}$

4.4 Theoretical analysis

We first show that QuAcq2 is a correct algorithm to converge on a constraint network equivalent to a target network that can be specified within a given bias. We then analyze the complexity of running QuAcq2, both in terms of computational complexity and number of queries. We finally provide a small change in function `FindC` that allows us to bound its complexity when the target network is normalized. This bound holds even if QuAcq2 does not know that the target network is normalized.

4.4.1 Correctness of QuAcq2

We first prove that QuAcq2 is sound, complete, and terminates.

Proposition 1 (Soundness) *Given a bias B, a target network $T \subseteq B$, and optionally background knowledge $K \subseteq T$, the network L returned by* QuAcq2 *is such that $sol(T) \subseteq sol(L)$.*

Proof. Suppose $sol(T) \setminus sol(L) \neq \varnothing$ when QuAcq2 terminates. This means that there exists a constraint in L that rejects a solution of T. That constraint cannot be added to L in line 2 of QuAcq2 because, by definition, $K \subseteq T$. That constraint cannot be added to L in line 4 of QuAcq2 because constraints added in line 4 are implied by K, and thus cannot reject a solution of T. Hence, there exists at least one scope on which QuAcq2 has learned a conjunction of constraints rejecting a solution of T. Let us consider the first conjunction c^* learned by QuAcq2 that rejects an assignment $e_1 \in sol(T) \setminus sol(L)$, 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 8 of `FindC`. This conjunction has been built by join operations in lines 3 and 15 of `FindC`. By construction of `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 `FindC` is executed. By construction of `FindC`, the join operations in line 15 of `FindC` are executed for and only for e'_Y generated in this call to `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, Δ contains all minimal conjunctions of elementary constraints from B_Y rejecting both $e_1[Y]$ and all e'_Y generated in this call to `FindC` that violate a constraint of scope Y in T and satisfy all the constraints on subscopes of Y. One of those minimal conjunctions is necessarily a subset of the conjunction in T. In line 8, when we put one of these conjunctions in L, they are all equivalent wrt L because line 5 was not able to produce an assignment e'_Y violating some conjunctions from Δ and satisfying the others. As scope Y is, by assumption, the first scope on which QuAcq2 learns a wrong conjunction of constraints, we know that we still have $T \models L$ at this point and we can deduce that all conjunctions in Δ are equivalent wrt to T. We already saw that one of the conjunctions in Δ is a subset of the conjunction on scope Y in T. As a consequence, none of these conjunctions can contain c_1 because c_1 rejects e_1 that T does not reject. Therefore, adding one of them to L cannot reject e_1. □

Proposition 2 (Completeness) *Given a bias B, a target network $T \subseteq B$, and optionally background knowledge $K \subseteq T$, the network L returned by* QuAcq2 *is such that $sol(L) \subseteq sol(T)$.*

Proof. Suppose there exists $e_1 \in sol(L) \setminus sol(T)$ when 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. We now prove that it is not possible for QuAcq2 to terminate with L accepting e_1 as solution. There is only one way for QuAcq2 to terminate, it is line 8 of QuAcq2. This means that in line 7 of QuAcq2, `GenerateQuery` was not able to generate an assignment e_Y accepted by $L[Y]$ and rejected by $B[Y]$. Thus, c_1 is not in B when QuAcq2 terminates, otherwise the projection $e_1[Y]$ of e_1 on any Y containing $var(c_1)$ would have been such an assignment. We know that $c_1 \in T$, so c_1 was in B before starting QuAcq2. Constraints can be removed from B in lines 3, 5, and 10 of QuAcq2, line 3 of `FindScope`, and lines 8 and 11 of `FindC`. In line 3 of QuAcq2, a constraint c_2 can be removed from B if there exists a constraint c' in L that directly contradicts c_2. c_2 cannot be c_1 because c' is in L, which is equal to K, K is included in T and, by convention, T does not contain any pair of constraints that contradict each other. In line 5 of QuAcq2, a constraint c_2 is removed from B if c_2 is in L. c_2 cannot be c_1 because $e_1 \in sol(L)$ when QuAcq2 terminates. In line 10 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 Δ when `FindC` terminates, or not. By construction of e'_Y in line 5 of `FindC`, lines 6 and 7 of `FindC` tell us that $L \cup \{c\} \models \Delta$, where c is the conjunction selected in line 7. Thus, if c_2 is in one of the conjunctions of Δ, then $L \models c_2$ after the addition of c to L in line 8, the only line where `FindC` can terminate. Thus, c_2 cannot be c_1 because by assumption c_1 rejects e_1, which itself is accepted by L. If c_2 is not in any of the conjunctions of Δ when `FindC` terminates, these conjunctions must have been removed in line 11 or in line 15, the two places where Δ is modified. Let us denote by $\widehat{c_2}$ a conjunction in Δ 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'_Y. If c_2 rejects e'_Y, all the conjunctions containing it remain in Δ. If c_2 is satisfied by e'_Y, there necessarily exists a conjunction in $\kappa_\Delta(e'_Y)$ which is a subset of the conjunction c^* because QuAcq2 is sound (Proposition 1). $\widehat{c_2}$ is joined with this subset. Thus, Δ 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 Δ 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 $\widehat{c_2}$ must then have been removed in line 11 because they were rejecting the example e'_Y classified positive in line 10. These conjunctions can be removed only if c_2 rejects e'_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 $sol(L) \subseteq sol(T)$. □

Proposition 3 (Termination) *Given a bias B, a target network $T \subseteq B$, and optionally background knowledge $K \subseteq T$,* QuAcq2 *terminates.*

Proof. Let us first prove termination of the two subprocedures `FindScope` and `FindC`. The first part of `FindScope` is composed of two termination conditions that obviously terminate. The second part of `FindScope` contains two recursive calls. In these two recursive calls, the last parameter is either Y_2 (line 8), or Y_1 (line 10). Both are strict subsets of Y —the last parameter from the parent call. As line 5 terminates `FindScope` when this last parameter is a singleton, `FindScope` is guaranteed to terminate.

`FindC` is composed of a main loop (line 4) that terminates when the condition in line 6 is true. Each time `FindC` enters this loop, if the condition in line 6 is false, `FindC` executes either line 11, or line 14, or line 15. By construction of e'_Y in line 5 we know that there is at least one conjunction in Δ that rejects e'_Y and thus at least one elementary constraint from B_Y that rejects e'_Y. If line 11 is executed (because e'_Y is positive), all the conjunctions containing this elementary constraint are removed from Δ and will never reappear. Hence, line 11 cannot be executed an unbounded

number of times. In line 14, `FindC` is recursively called with a strict subset S of Y. This sequence of recursive calls will then necessarily terminate when $|S|$ will have reached the smallest arity of a constraint in the bias B. Finally, in line 15, by definition of the join operation, `FindC` cannot add any conjunction other than supersets of conjunctions already in Δ. By construction of e'_Y in line 5, we know that at least one conjunction is in Δ and not in $\kappa_\Delta(e'_Y)$. In line 15 this conjunction is replaced by a strict superset. As the total number of conjunctions that can enter Δ is bounded (i.e., the powerset of B_Y), line 15 cannot be executed an unbounded number of times.

Let us now prove termination of QuAcq2. Each execution of the loop in line 6 of QuAcq2 either executes line 10 of QuAcq2 or enters `FindC`. By construction of e_Y in line 7 of QuAcq2 we know that $\kappa_B(e_Y)$ is not empty. Hence, in line 10 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 $var(c) = Z$ 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 6 of QuAcq2, B strictly decreases in size. As B has finite size, we have termination. □

Theorem 1 (Correctness) *Given a bias B, a target network $T \subseteq B$, and optionally background knowledge $K \subseteq T$,* QuAcq2 *converges on a network L such that $sol(L) = sol(T)$.*

Proof. From Proposition 3 we know that QuAcq2 always returns a network L. If QuAcq2 had not yet converged when returning L there would exist a network L' such that $sol(L') \neq sol(L)$ and L' agrees with the set of queries asked of the user. Hence, T could equally be L or L'. But then, either $sol(L') \not\subseteq sol(L)$ and soundness would not hold, contradicting Proposition 1, or $sol(L) \not\subseteq sol(L')$ and completeness would not hold, contradicting Proposition 2. Therefore, QuAcq2 has converged and $sol(L) = sol(T)$. □

4.4.2 Complexity of running QuAcq2

We can wonder what is the cost of running QuAcq2. The resources needed to run QuAcq2 are both a computer and a user. We can then wonder what is the cost of running QuAcq2 in terms of the cpu time required (i.e., computational complexity) or in terms of the user's effort (i.e., number of queries). The latter is the most important as long as the former is kept under control.

Concerning computational complexity, the complexity of `FindScope` is dominated by its number of recursive calls. The depth of the recursion in `FindScope` is bounded above by a logarithm in the size of the initial set Y because Y is split evenly into two parts before each new recursive call (line 6). Hence, the total number of recursive calls to `FindScope` for a given set Y is linear in $|Y|$. `FindC` deals with a scope Y, which by definition of the bias B, is of bounded size. Thus, the complexity of one call to `FindC` is bounded. Finally, the main procedure QuAcq2 executes its main loop a number of times that is bounded by the size of the bias B because at each iteration, either line 10 of QuAcq2 or line 8 of `FindC` is executed, and both remove constraints from B. At each execution of its main loop, QuAcq2 calls `GenerateQuery`. As we will see in Section 6, there are several ways to design `GenerateQuery`. In its most basic version, `GenerateQuery` generates membership queries (i.e., queries on the full set X of variables). This task is similar to that in *Conacq.2* and it was proved NP-hard in Proposition 12 of [17]. In Section 6, we will see that if we allow `GenerateQuery` to produce partial queries, its time complexity is bounded.

Let us analyze now the complexity of QuAcq2 in terms of the number of queries it asks of the user. Queries are proposed to the user in line 9 of QuAcq2, line 3 of `FindScope` and line 10 of `FindC`.

Proposition 4 *Given a vocabulary (X, D), a bias 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 $S \not\subseteq R_{parent}$ 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, $R_{child} \subsetneq R_{parent} \cup Y_{parent}$. As a result, Y_{parent} intersects S, and the parent node is on a branch that leads to the elucidation of a variable in S. Consider now the case where the leaf is the right child of the parent node. As we are on a leaf, if the test of line 2 is false (i.e., $\kappa_B(e[R_{child}]) = \varnothing$), we necessarily exit from `FindScope` through line 5, 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., $\kappa_B(e[R_{child}]) \neq \varnothing$), 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 $\varnothing$ 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 in number of queries) *Let Γ be a language of bounded-arity relations.* QuAcq2 *learns constraint networks over Γ 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 bias.*

Proof. Each time line 9 of QuAcq2 classifies an example as negative, the scope $var(c)$ of a constraint c from the target network is found in $O(|var(c)| \cdot \log n)$ queries (Proposition 4). As the bias only contains constraints of bounded arity, $var(c)$ is found in $O(\log n)$ queries. Finding c with `FindC` requires a number of queries in $O(1)$ because the size of Γ 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 bias 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 9 of QuAcq2 or line 3 of `FindScope`, this leads to at least one constraint removal from the bias because, by construction of QuAcq2 and `FindScope`, this example violates at least one constraint from the bias. 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 bias. □

4.4.3 QuAcq2 on normalized networks

The complexities stated in Theorem 2 are based on the size of the target network and size of the bias. The size of the language Γ is not considered because it has a fixed size, independent of the number of variables in the target network. Nevertheless, line 15 of `FindC` can lead to a size of

Δ and a number of queries in $\Omega(2^{|\Gamma|})$. By reformulating line 5 of `FindC` as shown below, we can bound the size of Δ, and then the number of queries. In the following, we use the notation Δ_p as defined at the very end of Section 3.

...

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

...

We can first observe that soundness, completeness, and termination are robust to replacing line 5 by line 5bis. The only requirement used in the proofs of Propositions 1, 2, and 3 for ensuring soundness, completeness, and termination, is that the assignment e'_Y generated in line 5 of `FindC` satisfies L and violates at least one constraint from Δ but not all. Line 5bis above satisfies this requirement.

Proposition 5 and Corollary 1 prove the new bound on the number of queries that we obtain by replacing line 5 by line 5bis in `FindC`.

Proposition 5 *Given a bias 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 is in $O(|B_Y| + 2^{max(|c^*|,|I_{c^*}|)})$, where c^* is the smallest such conjunction and $I_{c^*} = \{c_i \in B_Y \mid c^* \rightarrow c_i\}$.*

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

Let us first prove that line 5bis of `FindC` will not stop generating assignments before c^* is one of the conjunctions in Δ. Let us take as induction hypothesis that when entering a new execution of the loop in line 4, if c^* is not in Δ, then the set of the conjunctions in Δ that are included in c^* covers the whole set of elementary constraints from c^*. That is, $\bigcup\{sub \in \Delta \mid sub \subset c^*\} = c^*$. The only way to modify Δ is to ask a query e'_Y. If e'_Y is positive, this means that c^* is satisfied and all its subsets remain in Δ. 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 Δ 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 Δ. 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 3, by construction, all elementary constraints composing c^* are in Δ and line 3 is similar to line 15. As a consequence, our induction hypothesis is true. We prove now that as long as c^* is not in Δ, 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 assignment 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 Δ.

We now prove that c^* is in Δ after a number of queries linear in $|B_Y|$. We first count the number of positive queries. Thanks to the condition in line 5/5bis of `FindC`, we know that at least one elementary constraint c_i of B_Y is violated by the query. Thus, all the conjunctions containing c_i are removed from Δ in line 11, and no conjunction containing c_i will be able to occur again in Δ. As a result, the number of positive queries is bounded above by $|B_Y|$. 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 Δ. 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 Δ is in $O(|B_Y|)$.

Once c^* has been generated, it will remain in Δ until the end of this call to `FindC` because it can be removed neither by a positive query (it would not be in $\kappa_\Delta(e'_Y)$) nor by a negative (either it is in the $\kappa_\Delta(e'_Y)$ or a subconstraint is found and Δ is not modified).

We now show that the number of queries required to remove all conjunctions of constraints not equivalent to c^* from Δ is in $O(|B_Y| + 2^{max(|c^*|,|I_{c^*}|)})$. We first have to prove that once a conjunction rem has been removed from Δ, it will never reappear in Δ by some join operation. The conjunction rem can reappear in Δ if and only if there exist a and b in Δ such that $rem = a \wedge b$. If rem was removed due to a positive query e'_Y, then rem was in $\kappa_\Delta(e'_Y)$ and then, either a or b was in $\kappa_\Delta(e'_Y)$ too. Thus, a or b has been removed from Δ at the same time as rem, which contradicts the assumption that rem reappeared due to the join of a and b. If rem was removed due to a negative query e'_Y, then rem was not in $\kappa_\Delta(e'_Y)$ and then, none of a and b were in $\kappa_\Delta(e'_Y)$. a and b have thus both been joined with other elements of $\kappa_\Delta(e'_Y)$ and have disappeared from Δ 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 Δ 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 Δ in $O(|B_Y| + 2^{|c^*|})$ queries. As long as there exists a conjunction $nimp$ in Δ such that $c^* \not\models nimp$, line 5bis can generate a query e'_Y with $p \leq |c^*|$. If $\varnothing \subsetneq \kappa_{\Delta_p}(e'_Y) \subsetneq \Delta_p$ cannot be satisfied for any $p \leq |c^*|$, then there necessarily exists an e'_Y (satisfying c^* and violating $nimp$) with $\kappa_{\Delta_{|c^*|}}(e'_Y) = \varnothing \subsetneq \Delta_{|c^*|}$ and $nimp \in \kappa_\Delta(e'_Y)$, otherwise we would have $c^* \models nimp$. As a result, line 5bis can never return a query e'_Y with $p > |c^*|$ if there exists $nimp$ in Δ such that $c^* \not\models nimp$. Suppose first that $ASK(e'_Y) = yes$. By construction of e'_Y, we know that at least one elementary constraint c_i of the initial B_Y (line 2) is violated by e'_Y. Thus, all the conjunctions containing c_i are removed from Δ and the number of positive queries is bounded above by $|B_Y|$. Suppose now that $ASK(e'_Y) = no$. By construction of e'_Y, we know that $\Delta_p \setminus \kappa_{\Delta_p}(e'_Y)$ is not empty for some $p \leq |c^*|$, and all these conjunctions in $\Delta_p \setminus \kappa_{\Delta_p}(e'_Y)$ disappear from Δ_p in line 15 because they are joined with other conjunctions of $\kappa_\Delta(e'_Y)$. 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 Δ, Δ only contains c^* and conjunctions included in the set I_{c^*} of elementary constraints implied by c^*. We show that removing from Δ all conjunctions implied by c^* is performed in $O(2^{|I_{c^*}|})$ queries. As all conjunctions remaining in Δ are implied by c^*, all queries will be negative. By construction of such a negative query e'_Y, we know that $\Delta \setminus \kappa_\Delta(e'_Y)$ is not empty. All these conjunctions in $\Delta \setminus \kappa_\Delta(e'_Y)$ disappear from Δ in line 15 because they are joined with other conjunctions of $\kappa_\Delta(e'_Y)$. Thus, each query removes at least one element from Δ, 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 (Normalized Networks) *Given a bias B, a normalized target network T, and a (elementary) constraint c^* in T with scope Y such that there does not exist any c in B_Y with $c^* \rightarrow c$, `FindC` requires $O(|B_Y|)$ queries, which is in $O(|\Gamma|)$.*

The good news brought by Corollary 1 is that despite the join operation required in `FindC` to be able to learn constraints that are conjunctions of several elementary constraints from the bias, QuAcq2 is linear in the size of the language Γ when the target network does not contain such conjunctions and Γ 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 Γ. 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 Γ is in general a very difficult question. However, if Γ 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 assignment e_Y generated in line 7 of QuAcq2 is *complete* (i.e., $Y = X$) and is a solution of L that *maximizes* the number of violated constraints in the bias B.[7]

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

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

- *If $|D| = 2$, then QuAcq2 learns networks over Γ 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 Γ 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 Γ be a constraint language over a finite domain $D \subset \mathbb{Z}$, $|D| > 2$, such that $\{>, \neq\} \cap \Gamma \neq \varnothing$. Then, learning constraint networks over Γ 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 Γ with vocabulary (X, D). For any $(i, j) \in [1, \ldots, n/2] \times [n/2 + 1, \ldots, n]$ we define the assignment $\phi_{ij} : X \to D$ as follows:

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

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 $S \subseteq \mathcal{S} = \{\phi_{ij} \mid (i, j) \in [1, \ldots, n/2] \times [n/2 + 1, \ldots, n]\}$ the constraint network $C^S = \{R_{(X_i, X_j)} : \phi_{ij} \notin S\}$ over Γ has the property that $sol(C^S) \cap \mathcal{S} = S$. In particular, for any two distinct sets $S_1, S_2 \subseteq \mathcal{S}$ we have $sol(C^{S_1}) \neq sol(C^{S_2})$ and hence

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

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

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

[7] Finding a complete assignment maximizing violation in B is NP-hard, but we are interested here in the complexity in terms of number of queries.

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 nth *Bell Number* $C(n)$. It is known that $\log C(n) = \Omega(n \log n)$ [23], so this entails a lower bound of $\Omega(n \log n)$ queries to learn constraints networks over Γ. □

Lemma 3 *Let Γ be a constraint language over a domain $D \subset \mathbb{Z}$, $|D| = 2$, such that $\{\neq\} \subseteq \Gamma$. Then, learning constraint networks over Γ 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 $X_i \neq X_{ij}$, $X_{ij} \neq X_j$. It follows that for every set $S^{=}$ 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 Γ with $n^* = O(n)$ variables and such that $|S^{\neq}| = |S^{=}|$. As we have seen in the proof of Lemma 2, $|S^{=}|$ can be chosen such that $\log |S^{=}| = \Omega(n \log n)$. 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 9 of QuAcq2 and count how many times they can receive the answers *yes* and *no*.

For each *no* answer in line 9 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 7 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 9 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 9 of QuAcq2. Let e_Y be an assignment generated by QuAcq2 in line 7. Let $B^{L \not\models}$ 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 $B^{L \not\models}$ that e_Y violates, we consider an assignment ϕ 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 ϕ violates is $|B^{L \not\models}|/2$. Since QuAcq2 selects the assignment that *maximizes* the number of violated constraints, it will follow that e_Y violates at least half of $B^{L \not\models}$.

By construction, the random assignment ϕ is accepted by L. Furthermore, each constraint c in $B^{L \not\models}$ involves two variables belonging to distinct connected components of L so the probability that ϕ satisfies c is $|rel(c)|/|D|^2 = 1/|D|$, where $|rel(c)|$ 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 ϕ violates is therefore $|B^{L \not\models}| \cdot (1 - 1/|D|) \geq |B^{L \not\models}| \cdot 1/2$. As discussed in the previous paragraph, this implies in particular that e_Y violates at least half the constraints in $B^{L \not\models}$. It follows that throughout its execution QuAcq2 will receive at most $\lceil \log |B| \rceil = \lceil \log n^2 \rceil$ yes answers at line 9.

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 $O(n \log n)$ 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 9 of QuAcq2.

Each (complete) query submitted in line 9 that receives a negative answer will eventually add a new, non-redundant constraint to L. Observe that if $(L_=, L_\neq, L_>)$ denotes the partition of L into sub-networks containing only constraints $=$, $\neq$ and $>$ respectively, then neither $L_=$ nor $L_\neq$

may contain a cycle; if $L_>$ does then the solution set of L is empty and QuAcq2 will halt at line 8 the next time it goes through the main loop. Therefore, at most $3n$ queries may receive a negative answer in line 9, 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 9 of QuAcq2, consider an assignment e_Y generated by QuAcq2 at line 7. Let $B^{L\not\models}$ 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 $B^{L\not\models}$.

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 $s, \overline{s}$, where $\overline{s}$ is the logical negation of s. All other components have exactly one satisfying assignment s.

We construct a random assignment ϕ to X as follows. For each connected component $S \in \mathcal{S}$, the restriction of ϕ to S is either s or $\overline{s}$ (chosen uniformly at random) if S is free, and s otherwise. By construction ϕ is accepted by L, and for each variable $X_k \in X$ that belongs to a free component, the probability that ϕ assigns X_k to 1 is exactly $1/2$. It follows that, for each constraint c in $B^{L\not\models}$, the probability that ϕ 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 $B^{L\not\models}$ that ϕ violates is at least $1/2 \cdot |B^{L\not\models}|$. In particular, there exists an assignment that violates at least half the constraints in $B^{L\not\models}$, and by the way QuAcq2 generates assignments in line 7, 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 9, 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 $\mathcal{U}$ are set to 1, and all variables in $\mathcal{R}$ are set to 0 (variables in $\mathcal{L}$ are left unassigned). If the answer is *yes*, then there are no constraints between X_i and any variable in $\mathcal{U}$, hence we add X_i to $\mathcal{U}$ without breaking any invariant. Otherwise we know that X_i is either involved in a constraint $X_j > X_i$ with $X_j \in \mathcal{U}$, or a constraint $X_i > X_j$ with $X_j \in \mathcal{U}$. 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 $\mathcal{R}$, otherwise, we add it to $\mathcal{L}$. Here again, the invariants still hold.

At the end of the first phase, we therefore know that variables in $\mathcal{U}$ have no constraints between them. However, they might be involved in constraints with variables in $\mathcal{L}$ or in $\mathcal{R}$. In the second phase, we go over each variable $X_i \in \mathcal{U}$, and query the user with an assignment where all variables

in $\mathcal{L}$ are set to 1, all variables in $\mathcal{R}$ 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 $X_j \in \mathcal{L}$ and therefore X_i is added to $\mathcal{R}$ (and removed from $\mathcal{U}$). Otherwise, we ask the same query, but with the value of X_i flipped to 0. If the answer is *no*, there must exist $X_j \in \mathcal{R}$ such that $X_i > X_j$ belongs to the network, hence X_i is added to $\mathcal{L}$ (and removed from $\mathcal{U}$). 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 $\mathcal{U}$ are not constrained.

Once $\mathcal{L}, \mathcal{R}, \mathcal{U}$ 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 Γ 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 Γ 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 bias has size $O(n^2)$. □

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] What do the queries look like?

[Q4] How does the size of the background knowledge impact the learning effort?

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 and analyze the characteristics of the queries. We evaluate the learning effort in terms of number of queries when

the size of the background knowledge varies. We report the behavior of QuAcq2 on an exam time-tabling use case. Finally, we provide a brief comparison with QuAcq1.

For each of our experiments, QuAcq2 was run ten times on each problem and the reported results are the averages of the ten runs. (Several runs on the same problem do not give the exact same results because, as described later, function `GenerateQuery` generates queries by selecting values in random order, and because function `FindScope` shuffles the variables to avoid a lexicographic bias in the order in which constraints are discovered.) For each run, we have set a time limit of one hour on the time to generate a query, after which a timeout (`TO`) was reported. All the results reported in this section were obtained with the version of `FindC` that uses line 5bis described in Section 4.4. 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 our C++ platform on an Intel(R) Xeon(R) E5-2667 CPU, 2.9GHz with 8Gb of RAM.

The tables and figures presented in this section report the following information:

- [$|T|$] size (i.e., number of constraints) of the target network T,
- [$|L|$] size of the learned network L,
- [$\#\mathtt{Q_A}$] total number of queries to find a network L equivalent to T,
- [$\#\mathtt{Q_C}$] total number of queries to converge (i.e., until it is proved that L is equivalent to T),
- [$\mathtt{time_A}$] cumulated time to generate all queries 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,
- [$\mathtt{time_C}$] cumulated time to generate all queries until convergence is reported,
- [$\overline{\mathtt{t}}$] average time needed to generate a query,
- [$\mathtt{t_{max}}$] maximum time needed to generate a query, and
- [$\#\mathtt{C}$] number of runs that finished without triggering the 1-hour cutoff.

6.1 Benchmark problems

In the following, the background knowledge is a set K of constraints, where K is the part of the target network that we already know, that is, $K \subseteq T$. In all experiments where background knowledge K is used, the two optional lines 3 and 4 in QuAcq2 are implemented in a light way. In line 3 we only remove a constraint c from B if there exists a constraint c' in L with $var(c) = var(c')$ and $\{c, c'\} \models \bot$. In line 4, we only add a constraint c to L if there already exists a constraint c' in L with $var(c) = var(c')$ and $c' \models c$. We evaluated QuAcq2 on a variety of benchmark problems whose characteristics are the following.

Problem `Purdey` [32]. 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 bias of 396 constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. Concerning the default background knowledge, 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 each of these types of variables. Hence, the default background knowledge contains the three 4-cliques of disequalities on the variables of each type (families, bought item, paying mean). After lines 3 to 5 of QuAcq2, $|B| = 324$.

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 bias of 2700 unary and binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =, \circ_{val}, \shortparallel_1, \nshortparallel_1\}$, where $\circ_{val}$ denotes the unary relation $(x \circ val)$ with $\circ \in \{\geq, \leq, <, >, \neq, =\}$ and $val \in 1..5$, and where $\shortparallel_1$ and $\nshortparallel_1$ respectively denote the distance relations $|x - y| = 1$ and $|x - y| \neq 1$. Concerning the default background knowledge, similarly to `Purdey`, 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. Hence, the default background knowledge contains the five 5-cliques of disequalities on the variables of each type (house, nationality, pet, cigarette, drink). After lines 3 to 5 of QuAcq2, $|B| = 2500$.

Problem `Golomb` **[26, 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 bias of 1680 binary, ternary and quaternary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =, \shortparallel^{zt}_{xy}, \nshortparallel^{zt}_{xy}\}$, where $\shortparallel^{zt}_{xy}$ and $\nshortparallel^{zt}_{xy}$ 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, $\shortparallel^{zt}_{xy}$ and $\nshortparallel^{zt}_{xy}$ yield ternary constraints. The default background knowledge contains the 28 symmetry-breaking inequalities on the sequence of variables. After lines 3 to 5 of QuAcq2, $|B| = 1512$.

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. We initialized QuAcq2 with a bias containing the complete graph of 7350 binary constraints from Γ. We did not use any default background knowledge on this benchmark.

Problem `RLFAP` **[20].** The Radio Link Frequency Assignment Problem is to provide communication channels from limited spectral resources so as to avoid interferences between channels. 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 bias of 24,500 constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =, =^{val}_{xy}, >^{val}_{xy}\}$, where $=^{val}_{xy}$ and $>^{val}_{xy}$ respectively denote the distance relations $|x - y| = val$ and $|x - y| > val$, and $val \in \{12, 14, 28, 35, 56, 84, 238\}$. The possible values for val come from the kinds of interferences and from technological constraints on transmitters. They are known in advance. The default background knowledge contains the 25 constraints of the duplex links to establish. The constraints to learn are those preventing interferences between neighboring transmitters. After lines 3 to 5 of QuAcq2, $|B| = 24,000$.

Problem `Jigsaw`**.** The Jigsaw Sudoku logic puzzle is a 9×9 grid. It must be filled with numbers in such a way that all the rows, all the columns, and all the predefined (irregular) shapes contain the numbers 1 to 9. We used the predefined shapes displayed in Figure 2. The target network has 81 variables with domains of size 9 and 811 binary $\neq$ constraints on rows, columns and shapes. We initialized QuAcq2 with a bias of 19,440 binary constraints from the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. The default background knowledge contains a 9-clique of disequalities on the variables of each row and each column. After lines 3 to 5 of QuAcq2, $|B| = 16,848$.

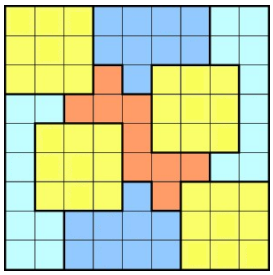

Figure 2: Our instance of `Jigsaw` problem.

Algorithm 4: Function `GenerateQuery.basic`

```
   In   : a bias B
   In   : a learned network L
   Out  : an assignment on X
1  begin
2      C ← L;
3      foreach c_i ∈ B do
4          introduce a Boolean variable b_i;
5          C ← C ∪ {b_i ↔ c_i}
6      C ← C ∪ {∑ b_i ≠ |B|};
7      e ← solve(C);
8      return e[X];
```

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 7 of Algorithm 1, QuAcq2 uses the function `GenerateQuery.basic` described in Algorithm 4. `GenerateQuery.basic` computes a complete assignment on X satisfying the constraints in L and violating at least one constraint from B. This is done the following way. We build a network C that contains the constraints from the network L already learned (line 2), plus a reification of the constraints in B. A Boolean b_i is introduced for each $c_i \in B$ (line 4). This Boolean is forced to be true if and only if the constraint c_i is satisfied (line 5). We then force the sum of b_i's not to be equal to $|B|$ (line 6). Function *solve* is called on C (line 7) and returns a solution of C, or $\perp$ if no solution exists. Finally, the solution is projected[8] on X to get rid of the reification variables and it is returned (line 8). The constraint solver inside *solve* maintains arc consistency during search [40], uses the `dom/wdeg` variable ordering heuristic [18], and selects values in random order.

Table 4 reports the results of running QuAcq2.`basic` on all our benchmark problems when provided with background knowledge as described in Section 6.1. The first observation we can make by looking at the table is that there are problems on which QuAcq2.`basic` is not able to converge on all of the ten runs (`Random`) and problems on which QuAcq2.`basic` is not able to converge on any run (`RLFAP`, `Jigsaw`). For `Random`, Table 4 reports the averages of the 7 runs on which QuAcq2.`basic` converges.

We first focus our attention on the four problems where QuAcq2.`basic` always or sometimes converges: `Purdey`, `Zebra`, `Golomb`, and `Random`. Let us first compare the size of the learned network to the size of the target network, that is, the sum of columns $|K|$ and $|L| \setminus |K|$ to column $|T|$. On `Golomb`, we observe that the size of the learned network L is much smaller than the size of the target network T (28+70=98 and 350 respectively). This is because the target network with

[8] We assume that $\perp[X] = \perp$.

Table 4: QuAcq2.basic when provided with background knowledge as described in Section 6.1. Averages of ten runs (time in seconds).

Problem	$\|T\|$	$\|K\|$	$\|L\| \setminus \|K\|$	$\#\mathtt{Q_A}$	$\#\mathtt{Q_C}$	$\mathtt{time_A}$	$\mathtt{time_C}$	$\overline{\mathtt{t}}$	$\mathtt{t_{max}}$	$\#\mathtt{C}$
Purdey	27	18	9	71.2	73.8	0.04	0.06	0.00	0.01	10
Zebra	64	50	14	187.1	188.0	1.76	1.78	0.01	1.09	10
Golomb	350	28	70	239.2	239.2	120.25	247.71	0.50	10.34	10
Random	122	0	122	1082.2	1092.0	2.08	85.94	0.08	83.80	7
RLFAP	125	25	71.1	752.9	–	151.13	–	–	TO	0
Jigsaw	811	648	117.0	–	–	–	–	–	TO	0

TO = 1 hour

all quaternary constraints $|X_i - X_j| \neq |X_k - X_l|$ contains a lot of redundancies. QuAcq2.basic detects convergence before learning them. On Purdey, Zebra, and Random, the learned network L has exactly the size as the target network T. For Random it is not surprising as the network was constructed not to contain redundant constraints. For Purdey and Zebra, it just means that the clues defining the problem were not found redundant wrt to the cliques of disequalities or among themselves. (Some of the disequalities are redundant but they were given as background knowledge.) If we compare now the number of queries $\#\mathtt{Q_C}$ to the size of B after lines 3 to 5 have reduced it (see Section 6.1), we observe that $\#\mathtt{Q_C}$ is 4 to 13 times smaller. This is an indication that QuAcq2 asks queries that are effective for quickly leading to convergence. Let us now compare the costs of finding the right network (columns $\#\mathtt{Q_A}$ and $\mathtt{time_A}$) and the costs of converging (columns $\#\mathtt{Q_C}$ and $\mathtt{time_C}$). This tells us a lot about the end of the learning process. On Purdey and Zebra, $\#\mathtt{Q_A}$ and $\mathtt{time_A}$ are very close to $\#\mathtt{Q_C}$ and $\mathtt{time_C}$ (respectively). This means that QuAcq2.basic learns constraints until the very end of the process. On Golomb, $\#\mathtt{Q_A}$ and $\#\mathtt{Q_C}$ are again similar, but $\mathtt{time_C}$ is much larger than $\mathtt{time_A}$. The reason is that after having learned all the constraints necessary to have a L equivalent to T, GenerateQuery.basic spends more than two more minutes to show that $L \models B$, which proves convergence. On Random, we observe yet another behavior. As on Golomb, $\mathtt{time_C}$ is much larger than $\mathtt{time_A}$ (more than one order of magnitude larger), but $\#\mathtt{Q_C}$ is also larger than $\#\mathtt{Q_A}$. The reason is that QuAcq2.basic has found a network L equivalent to T about 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 eventually to prove convergence. This last phenomenon is probably due to the sparseness of T in Random. The column $\overline{\mathtt{t}}$ tells us that most queries are very easy to generate (from milliseconds to half a second in average depending on the problem). However, we see in column $\mathtt{t_{max}}$ that the queries that are the hardest to generate can take orders of magnitude more time than the average. See Zebra, where the hardest query takes 100 times more time in average of the ten runs (most runs have a query requiring more than half a second) than the average time $\overline{\mathtt{t}}$ to generate a query, or Golomb, where the hardest query takes 20 times more time in average of the ten runs (all runs have a query requiring from 3 to 30 seconds) than the average time $\overline{\mathtt{t}}$. Random is an extreme case. In one of the ten runs, the query that is the hardest to generate requires almost ten minutes. That single query (over the ten runs) explains the very high value of $\mathtt{t_{max}}$. (Bear in mind that on Random there are three runs on which QuAcq2.basic was not able to converge because the one-hour timeout was reached.)

Let us now move our attention to the last two problems in Table 4, namely, RLFAP and Jigsaw. On these two problems, on each of the ten runs, QuAcq2.basic reaches the 1-hour cutoff. However, on RLFAP, on all the runs, QuAcq2.basic is able to find a network L equivalent to T in $\#\mathtt{Q_A}$ queries and $\mathtt{time_A}$ time. This is the proof of convergence that leads QuAcq2.basic to the timeout. The reason seems to be that when the bias B only contains constraints that are redundant with L, line 7 of GenerateQuery.basic becomes prohibitively expensive: it has to prove that C is inconsistent, which is coNP-hard. The presence of redundant constraints in RLFAP is confirmed by the difference between the size of the learned network $|L|$ ($= 25 + 71.1 = 96.1$) and the size of the target network T ($= 125$). This is a behavior similar to what we saw on Golomb. On Jigsaw, the

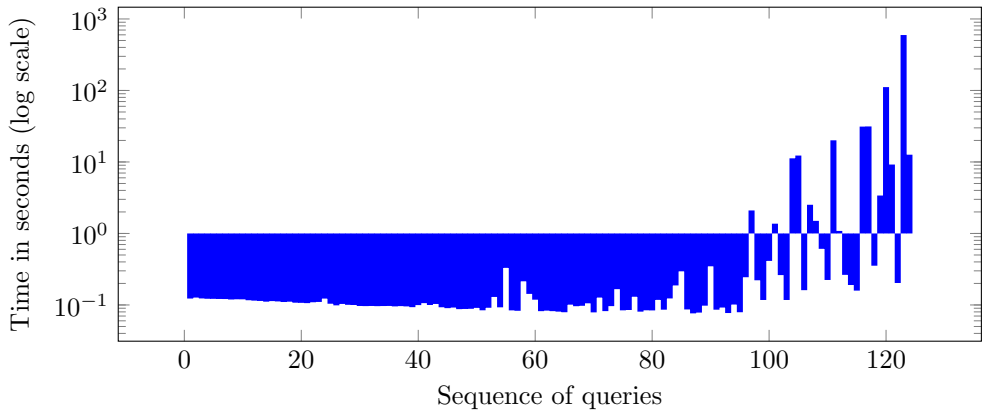

Figure 3: Time to generate (membership) queries in `GenerateQuery.basic` on `Jigsaw`.

scenario is different. There are runs for which QuAcq2.`basic` reaches the cutoff before having learned a network equivalent to T.

To illustrate a bit more the issue of the cost of generating queries in QuAcq2.`basic`, we selected one of the runs on `Jigsaw` on which QuAcq2.`basic` does not converge and we plot the cost of all the queries generated by `GenerateQuery.basic` in Figure 3. We observe that as long as B contains enough constraints that are non-redundant wrt L –approximately until the 100th query, the time needed by `GenerateQuery.basic` to generate a query is negligible (about 0.1 seconds). After the 100th query, L increased so much and B decreased so much that finding a query that violates some constraints from B while satisfying L is no longer a trivial problem. In the last 20 queries, half of them take more than 10 seconds to be generated with two extreme cases taking two minutes and ten minutes. Finally, after the 124th query, QuAcq2.`basic` gets stuck on a call to `GenerateQuery.basic` that is not able to generate a query before the one-hour cutoff.

The experiments presented in this subsection show us that QuAcq2.`basic` learns constraint networks in a number of queries always significantly smaller than the size of the bias. When the target network is small, QuAcq2.`basic` generates queries very fast. 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 Generating (partial) queries with a time cutoff

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 `GenerateQuery` that fixes this weakness. We start from the observation that the assignment generated by `GenerateQuery` in line 7 of Algorithm 1 does not need to be a complete 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 thus propose `GenerateQuery.cutoff`, a new version of `GenerateQuery` that quickly returns a partial assignment on a subset Y of X accepted by L and violating at least one constraint from B. The key 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 superset Y of S, and t is the time consumed by *solve*. If *solve* proves that C is inconsistent (that is, it found a set

Algorithm 5: Function `GenerateQuery.cutoff`

```
   In Out: a bias B
   In Out: a learned network L
   Out   : An assignment on a subset of X
1  begin
2      time ← 0;
3      foreach c ∈ B do
4          (e_var(c), t) ← solve(L ∪ {¬c}, var(c), nil, +∞);
5          time ← time + t;
6          if e_var(c) = ⊥ then
7              mark c as redundant; L ← L ∪ {c}; B ← B \ {c};
8          else
9              (e'_Y, t') ← solve(L ∪ {¬c}, var(c), |Y|, cutoff − time);
10             time ← time + t';
11             if e'_Y = nil then return e_var(c);
12             if e'_Y = ⊥ then
13                 mark c as redundant; L ← L ∪ {c}; B ← B \ {c};
14             else return e'_Y;
15     remove all constraints marked as redundant from L;          // optional
16     return ⊥;
```

Y containing S for which every assignment on Y either violates $C[Y]$ or leads to arc inconsistency[9] on C), it returns the pair $(\bot, t)$ 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. Function *solve* uses the `bdeg` variable ordering heuristic proposed by Tsouros et al. [45] and selects values in random order. `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 `GenerateQuery.cutoff`. Given a timeout parameter `cutoff`, `GenerateQuery.cutoff` computes a (partial) assignment satisfying the constraints in L, violating at least one constraint from B, and tries to maximize the size of that assignment as long as `cutoff` has not been reached. `GenerateQuery.cutoff` operates by iteratively picking a constraint c from B until an assignment satisfying L and violating c is returned, or B is exhausted (line 3). The call to *solve* in line 4 computes an assignment $e_{var(c)}$ on $var(c)$ satisfying L and violating c. The time t needed to compute $e_{var(c)}$ is added to the time counter (line 5). If *solve* returns $e_{var(c)} = \bot$ (i.e., $L \cup \{\neg c\}$ is inconsistent), c is marked as redundant because it is implied by L. c is then removed from B and added to L (line 7). 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 $e_{var(c)}$ different from $\bot$, `GenerateQuery.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'_Y satisfying L and violating c, whilst maximizing $|Y|$ as much as the remaining time permits (line 9).[10] If no assignment satisfying L and violating c is found in the remaining time, *solve* returns an e'_Y equal to nil and `GenerateQuery.cutoff` returns the $e_{var(c)}$ found by the first call to *solve* (line 11). If

[9] A network is said to be *arc inconsistent* if applying arc consistency leads to an empty domain [8].

[10] We have experimentally tried several criteria other than the size of Y. Size of Y appears to be a good compromise though maximizing the size of $\kappa_B(e'_Y)$ was also a good criterion in some cases. More elaborated heuristics were analyzed by Addi et al. [1] in the context of QuAcq1 but could be used in QuAcq2 as well.

Table 5: QuAcq2.cutoff on the same problems as Table 4. cutoff = 1 second. Averages of ten runs (time in seconds).

Problem	$\|T\|$	$\|K\|$	$\|L\| \setminus \|K\|$	$\#Q_A$	$\#Q_C$	$time_A$	$time_C$	$\overline{t}$	t_{max}	#C
Purdey	27	18	9	73.5	82.2	0.04	0.21	0.00	0.03	10
Zebra	64	50	14	183.0	197.9	1.05	2.09	0.01	0.58	10
Golomb	350	28	70	245.1	245.1	135.41	143.27	0.55	6.96	10
Random	122	0	122	1039.0	1041.8	2.14	2.16	0.00	0.07	10
RLFAP	125	25	100	818.4	818.4	42.61	42.61	0.05	1.05	10
Jigsaw	811	648	163	1 718.6	1 743.6	177.67	202.30	0.12	1.03	10

solve proved the inconsistency of $L \cup \{\neg c\}$ over a scope Y (i.e., $e'_Y = \bot$), c is marked as redundant, removed from B, and added to L (line 13), exactly as in line 7. GenerateQuery.cutoff then goes back to line 3 to select a new constraint from B. Otherwise (i.e., $e'_Y \notin \{nil, \bot\}$), GenerateQuery.cutoff returns the assignment e'_Y with the largest size of Y that has been found in the allocated time (line 14). Finally, if all constraints in B have been processed without finding a suitable assignment (line 3), this means that all the constraints that were in B were implied by L. The learning process has thus converged. We can remove all constraints marked as redundant from L (line 15) and GenerateQuery.cutoff returns $\bot$ (line 16). 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 GenerateQuery.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 GenerateQuery.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 user. Table 5 reports the results for the same problems and same measures as Table 4.

The main information that we extract from Table 5 is that the use of GenerateQuery.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 reached the one-hour cutoff for a query on some or all runs on three of the problems (Random, RLFAP and Jigsaw). Even on Golomb, where QuAcq2.basic was always converging, QuAcq2.cutoff shows a significant speed up.

To better understand the difference of behavior between QuAcq2.basic and QuAcq2.cutoff, we report an experiment with QuAcq2.cutoff similar to the one with QuAcq2.basic reported in Figure 3. Figure 4 reports the results of one run of QuAcq2.cutoff on Jigsaw. At the top of Figure 4, we display the time needed by GenerateQuery.cutoff to generate each of its queries, exactly as we did for GenerateQuery.basic in Figure 3. At the bottom of Figure 4 we report the length of each query. QuAcq2.basic was able to generate queries very fast until approximately the 100th query. QuAcq2.cutoff seems to have more difficulties in this early stage. Around the 70th query, GenerateQuery.cutoff starts to frequently reach its cutoff of 1 second. After the 90th query it almost always reaches the cutoff. The consequence, as we see at the bottom of the figure, is the generation of shorter queries (less than 81 variables long). This lower efficiency in generating queries is due to the efficient dom/wdeg heuristic used in GenerateQuery.basic. GenerateQuery.cutoff does not use the dom/wdeg heuristic. It uses bdeg because it tries to violate more constraints from B. (See Section 6.3.1). But the most interesting information that we learn from this experiment comes from the observation of the the second half of the learning process. After the 124th query, when QuAcq2.basic has failed to learn the Jigsaw problem, QuAcq2.cutoff continues generating queries, all taking one second to generate. As opposed to GenerateQuery.basic, GenerateQuery.cutoff is able to return queries on strict subsets of X when generating a query on X cannot be accomplished in one second. After the 124th query,

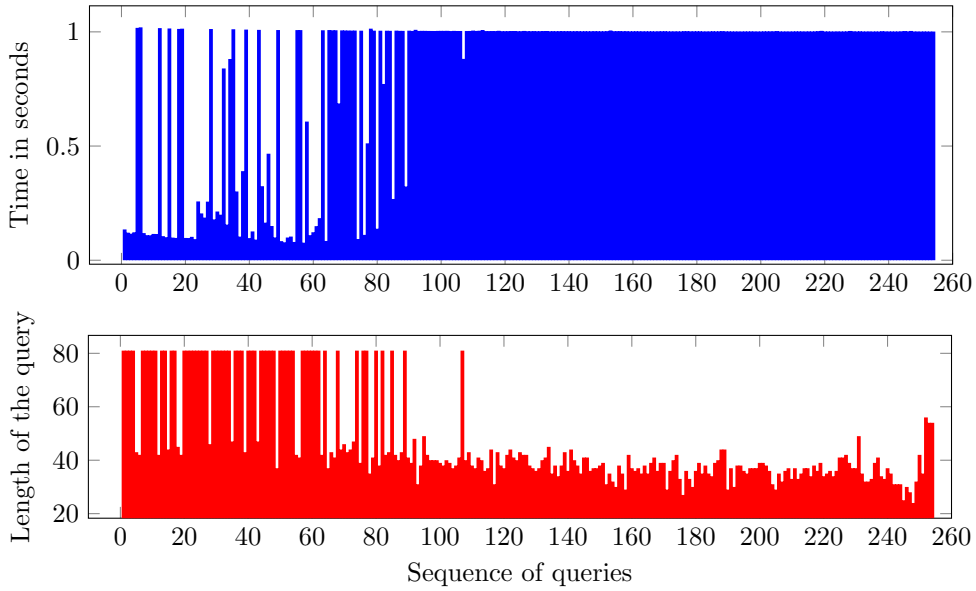

Figure 4: `GenerateQuery.cutoff` on `Jigsaw`: Time to generate queries (top) and their length (bottom).

most queries indeed have a size between 35 and 45 variables.

Coming back to Table 5, we observe that on `RLFAP`, QuAcq2.`cutoff` learns more constraints than QuAcq2.`basic` despite QuAcq2.`basic` had learned a network equivalent to T before being stopped by the 1-hour cutoff. The reason is that, when we are close to the end of the learning process, queries become hard to generate. `GenerateQuery.cutoff` then generates shorter queries. Such short queries can lead to the acquisition of constraints that are in fact redundant with L. `Golomb` also contains redundancies but it does not exhibit the same behavior because `Golomb` is an easy problem on which `GenerateQuery.cutoff` has enough time to generate large queries. Regarding the number of queries, we could have expected that the generation of shorter queries at the end of the learning process causes an increase in the overall number of queries asked by QuAcq2.`cutoff`. Shorter positive queries have indeed more chances to rule out fewer violated constraints. But, when comparing $\#\mathtt{Q_C}$ in Tables 4 and 5, we see that the increase is not dramatic. The largest increase for QuAcq2.`cutoff` is 11% more queries on `Purdey`. On the other problems on which QuAcq2.`basic` converged at least once, we either have a small increase (5% and 2% on `Zebra` and `Golomb`, respectively) or a small decrease (5% on `Random`). As a last observation on Table 5, it can seem surprising that $\mathtt{t_{max}}$ is more than six seconds on `Golomb` despite the cutoff of one second in `GenerateQuery.cutoff`. This is because on this problem, the bias contains a lot of redundant constraints. Hence, during the learning process, `GenerateQuery.cutoff` repetitively finds redundant constraints without asking a question to the user (lines 6-7 and 12-13 in Algorithm 5).

The experiments presented in this subsection show us that the introduction of a cutoff in the generation of queries completely solves the issue of extremely long query-generation times at the end of the process. QuAcq2.`cutoff` learned all our benchmark problems with fast query generation. The only price to pay is a slight increase in number of queries on the problems that are the easiest to learn.

6.4 [Q3] Properties of the queries in QuAcq2.`cutoff`

Let us now focus on the kind of queries the user has to answer when learning a network with QuAcq2.`cutoff`. In Table 5, we reported the number of queries $\#\mathtt{Q_A}$ and $\#\mathtt{Q_C}$ necessary to find

Table 6: QuAcq2.cutoff: The different kinds of queries on the same problems as Table 4 and Table 5. Averages of ten runs.

Problem	$\#\mathtt{Q}_{\mathtt{C}}$	$\#\mathtt{Q}_{\mathtt{GenerateQuery}}$	$\#\mathtt{Q}_{\mathtt{FindScope}}$	$\#\mathtt{Q}_{\mathtt{FindC}}$
Purdey	82.2	18.9	53.8	9.5
Zebra	197.9	29.2	138.2	30.5
Golomb	245.1	71.3	147.0	26.8
Random	1041.8	141.7	859.8	40.3
RLFAP	818.4	199.3	541.3	77.8
Jigsaw	1 743.6	244.8	1 410.3	88.5

the target network and to converge on it, respectively. However, the queries asked by QuAcq2 are not all of the same kind. QuAcq2 asks queries in three different places: in Line 9 of the main procedure (Algorithm 1), in Line 3 of `FindScope` (Algorithm 2), and in Line 10 of `FindC` (Algorithm 3). In Table 6, for the same problems as in Table 5, we recall the total number of queries $\#\mathtt{Q}_{\mathtt{C}}$ necessary to converge with QuAcq2.cutoff, but we also report the detail of how many queries were asked by the main procedure QuAcq2 ($\#\mathtt{Q}_{\mathtt{GenerateQuery}}$), by `FindScope` ($\#\mathtt{Q}_{\mathtt{FindScope}}$), or by `FindC` ($\#\mathtt{Q}_{\mathtt{FindC}}$).

The results reported in Table 6 contain interesting information that was not visible in the previous tables. First, we observe that the number of queries $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ generated by `GenerateQuery.cutoff` and asked by the main procedure is much smaller than the total number of queries $\#\mathtt{Q}_{\mathtt{C}}$. $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ goes from 3.5 to 14 times less than $\#\mathtt{Q}_{\mathtt{C}}$. Not surprisingly, $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ seems to depend on the number of constraints to learn. By comparing $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ to the column $|L|/|K|$ in Table 5, we see that the number of queries $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ is between 1.2 and 3 times larger than the number of constraints to learn. Second, on all the problems, the queries that are by far the most numerous are the queries asked by `FindScope` (column $\#\mathtt{Q}_{\mathtt{FindScope}}$). This is good news because such queries are much easier to answer than queries coming from `GenerateQuery.cutoff`. Queries generated by `GenerateQuery.cutoff` are new assignments to be classified. Queries from `FindScope` are just subsets of a query already asked of the user. They are thus much easier to answer. In addition, it is often the case that the user, even if not able to formulate the constraints, is able to circumscribe the subset of variables where the given assignment is problematic. (See the discussion at the end of Section 4.2.) In such a case, the total number of queries to learn a network drops from $\#\mathtt{Q}_{\mathtt{C}}$ to $\#\mathtt{Q}_{\mathtt{GenerateQuery}} + \#\mathtt{Q}_{\mathtt{FindC}}$, which is significantly smaller. Finally, concerning the number of queries $\#\mathtt{Q}_{\mathtt{FindC}}$ asked by `FindC`, it is not obvious to find a general trend. $\#\mathtt{Q}_{\mathtt{FindC}}$ is of the same order of magnitude as $\#\mathtt{Q}_{\mathtt{GenerateQuery}}$ on all problems. Like queries from `FindScope`, queries from `FindC` are very easy to answer: they involve the number of variables of the constraint to learn, which is bounded, by definition of Γ.

6.5 [Q4] Impact of background knowledge on the learning effort

In Section 6.2 and 6.3, we have presented experiments with fixed background knowledge K. We now want to assess the impact of the size of K on the cost of the learning process in terms of number of queries asked of the user.

We performed a new experiment. For each problem, QuAcq2.cutoff is called with background knowledge K whose size varies from 0% to 90% of the size of the target network. That is, for each percentage $p \in \{0, 10, \ldots, 90\}$, K is built by randomly picking p% of the constraints in the target network. Similarly to the previous experiments, the two optional lines 3 and 4 in QuAcq2 are implemented as described at the beginning of Section 6.1.

Figure 5 reports the number of queries $\#\mathtt{Q}_{\mathtt{C}}$ that QuAcq2.cutoff requires to converge depending on the percentage of constraints that were put in K. As the y-axis is logarithmic, Figure 5 also displays a linear slope to make it easier to see how much the other curves follow a linear slope. These results confirm that when the size of K increases, the number of queries decreases. To which amount? We observe that on all problems except `Golomb` and `RLFAP`, the decrease is

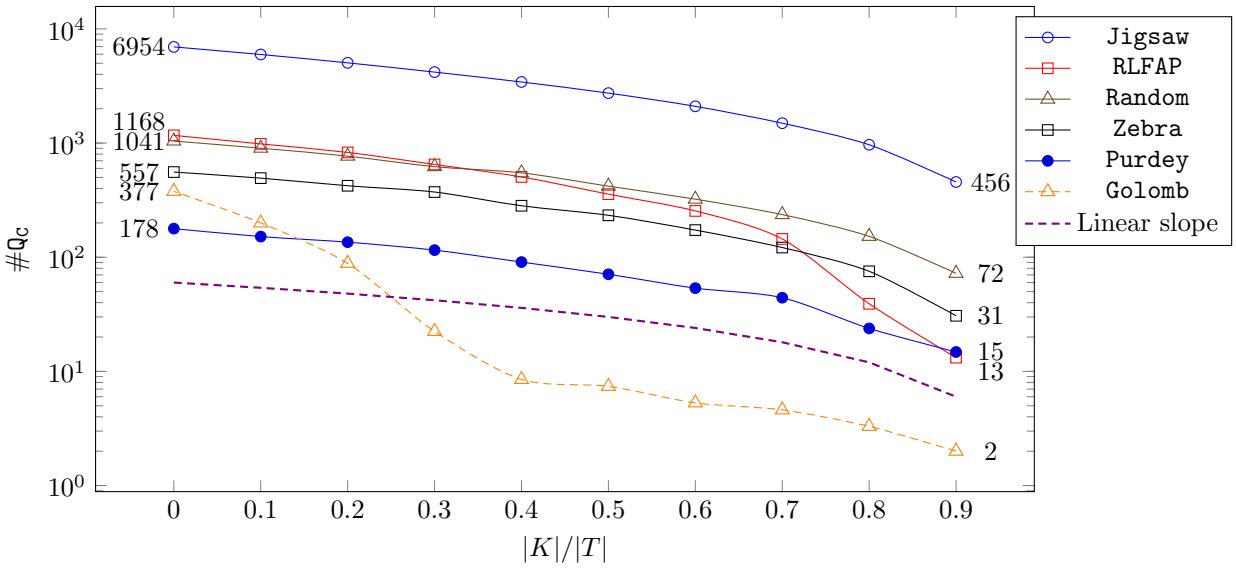

Figure 5: QuAcq2.cutoff: Number of queries #Qc when varying the size of the background knowledge K.

linearly correlated to the amount of background knowledge provided.[11] This is very good news because this means that it is always beneficial to add more background knowledge.

6.6 Our use case

We have also experimented QuAcq2 on a simplified version[12] of the exam time-tabling problem from the Department of Electrical and Computer Engineering of the University of Western Macedonia (UOWM), Greece. There are 24 courses to be examined and 2 weeks of exams, meaning that there are 10 possible days for each exam to be assigned. There are 3 time slots per day, i.e. 30 time slots in total. This resulted in a model with 24 variables and domains of size 30. There are $\neq$ constraints between any two exams because there is only one room for examination. Two courses of the same bachelor program cannot be examined on the same day. This is expressed by constraints $\lfloor x_i/3 \rfloor \neq \lfloor x_j/3 \rfloor, \forall i, j$ in the same program. The above constraints are fixed and do not vary from one year to the other. They are considered to be known and given as background knowledge to QuAcq2. The problem of this year contains 16 additional constraints from the language $\Gamma = \{\neq, =, <, >, \leq, \geq, |x_i - x_j| > k, |x_i - x_j| < k, ||\lfloor x_i/3 \rfloor - \lfloor x_j/3 \rfloor| > k, ||\lfloor x_i/3 \rfloor - \lfloor x_j/3 \rfloor| < k\}$, with $k \in 1..5$. These constraints capture the lecturers' restrictions about the examination of specific courses. An instance of such restriction is that of a lecturer who lives far from the campus: She wants all her courses to be examined in two consecutive days. This can be expressed by the constraint $||\lfloor x_i/3 \rfloor - \lfloor x_j/3 \rfloor| < 3$, x_i and x_j being courses of that lecturer. These additional constraints are the ones that the person in charge of the exam time-tabling has to formulate on top of the background knowledge. Until now the time-table has been constructed manually by the person in charge, resulting in rigid and often infeasible assignments. Hence, there is a wish to move to an automated (or semi-automated) process which takes into account all the existing constraints. However, the additional constraints have proved to be an obstacle as the time-tabling person found it difficult to formulate them. These could be the constraints to be learned by QuAcq2 if that

[11] On Golomb and RLFAP, we observe a decrease that is faster than linear. The reason is that there is a lot of redundancy among the constraints of the target network. Some constraints redundant with K can be inferred without the need of queries.

[12] The real version also contains preferences from the students.

Table 7: QuAcq2.basic and QuAcq2.cutoff on our use case exam time-tabling problem. Averages of ten runs (time in seconds).

Algorithm	$\|T\|$	$\|K\|$	$\|L\| \setminus \|K\|$	$\#\mathtt{Q_A}$	$\#\mathtt{Q_C}$	$\mathtt{time_A}$	$\mathtt{time_C}$	$\overline{\mathtt{t}}$	$\mathtt{t_{max}}$	$\#\mathtt{C}$
QuAcq2.basic	292	276	16	209.4	–	15.28	–	–	TO	0
QuAcq2.cutoff	292	276	16	203.2	259.9	12.89	27.07	0.10	6.77	10

TO = 1 hour

Algorithm	$\#\mathtt{Q_C}$	$\#\mathtt{Q_{GenerateQuery}}$	$\#\mathtt{Q_{FindScope}}$	$\#\mathtt{Q_{FindC}}$
QuAcq2.cutoff	259.9	81.4	134.6	43.9

person was let alone with her exam time-tabling problem. We initialized QuAcq2 with a bias of 14,352 constraints from Γ. After lines 3 to 5 of QuAcq2, $|B| = 9960$.

Table 7 reports the results for both QuAcq2.basic and QuAcq2.cutoff. Like RLFAP and Jigsaw, Exam-TT is difficult enough to learn so that QuAcq2.basic fails to converge because of the 1-hour cutoff on query generation. However, as on RLFAP, QuAcq2.basic has in fact already learned a network equivalent to T when it is interrupted by the cutoff. This is the proof of convergence that leads to the timeout. If we look at what happens with QuAcq2.cutoff, we see a behavior similar to what we observed for the other benchmark problems. QuAcq2.cutoff converges at all runs, and it has a cumulated generation time for all queries that is less than half a minute. We also observe that approximately half of this time is consumed by the proof of convergence. In average on the ten runs, 203.2 queries and 12.89 seconds were necessary to find a network equivalent to the target network. 56.7 more queries and 14.18 more seconds are necessary to prove convergence. This is again due to redundant constraints in the bias. It is confirmed by the high value of $\mathtt{t_{max}}$. As on Golomb, GenerateQuery.cutoff can spend six seconds repetitively proving the redundancy of constraints before being able to generate a query.

Concerning the kind of queries asked by QuAcq2.cutoff (bottom part of Table 7), we again observe that queries asked by FindScope are the most frequent. By comparing $\#\mathtt{Q_{GenerateQuery}}$ (81.4 queries) to the number of queries from GenerateQuery.cutoff used to prove convergence (i.e., $\#\mathtt{Q_C} - \#\mathtt{Q_A} = 56.7$ queries), we see that QuAcq2.cutoff has only asked $81.4 - 56.7 = 24.7$ queries generated by GenerateQuery.cutoff before finding the actual target network. Even more interestingly, by analyzing the sequence of queries asked by QuAcq2.cutoff (not reported here), we observe that, in average on the ten runs, GenerateQuery.cutoff returns a solution of the whole problem after having generated 12.2 queries only. At this point in the learning process, FindScope and FindC have asked 105.8 and 40.7 queries, respectively, for a total of 158.7 queries. As in this use case it is very easy to point out which variables violate a constraint (violations of constraints are only possible between courses of a same lecturer), the person in charge of the exam time-tabling would probably have been able to solve the instance of this year by answering 12.2 queries asked by the main procedure QuAcq2 plus 40.7 trivial queries from FindC. (On this problem, queries from FindC always involve two variables.)

6.7 Comparison of QuAcq2 with QuAcq1

We finish this experimental section with a comparison of the behaviors of QuAcq1 and QuAcq2. We cannot make an exact comparison of QuAcq1 with QuAcq2 because the function FindC in QuAcq1 is designed to learn networks that are normalized whereas QuAcq2 can learn networks that are not normalized. We then mimic QuAcq1 by running a QuAcq2.basic version in which function FindScope has been replaced by the FindScope in [11]. The algorithm obtained by this substitution of function FindScope in QuAcq2.basic is denoted by pseudoQuAcq1. Table 8 reports the results of pseudoQuAcq1 for the same problems as Table 4. For each measure, we give the percentage of difference with the results obtained by QuAcq2.basic.

The first information that we extract from the top part of Table 4 is that pseudoQuAcq1 fails

to converge on the target network on the same problems –RLFAP and Jigsaw– as QuAcq2.basic.[13] This is not a surprise because the reason why QuAcq2.basic does not converge on these problems is not related to the behavior of its FindScope. If we focus now on the number of queries required to converge on the four other problems (column $\#\mathtt{Q_C}$ in the top part of Table 8), we see that QuAcq2.basic is consistently better than pseudoQuAcq1 in terms of number of queries. The decay in performance when replacing the FindScope of QuAcq2.basic by the one of QuAcq1 goes from 13% to 242% depending on the problem. The bottom part of Table 8 displays the distribution of the number of queries asked by each of the three procedures: pseudoQuAcq1 (column $\#\mathtt{Q_{GenerateQuery}}$), FindScope (column $\#\mathtt{Q_{FindScope}}$), and FindC (column $\#\mathtt{Q_{FindC}}$). These results confirm that the increase of the number of queries in pseudoQuAcq1 is due to FindScope. The FindScope of pseudoQuAcq1 can ask from 20% to 417% more queries than the FindScope of QuAcq2.basic depending on the problem. The extremely bad performance of the FindScope of pseudoQuAcq1 on Golomb is due to the presence of constraints of various arities (two, three and four) in the bias and in the target network. When there are several arities involved, the risk of asking redundant queries is much higher. The small variations in $\#\mathtt{Q_{GenerateQuery}}$ and $\#\mathtt{Q_{FindC}}$ between pseudoQuAcq1 and QuAcq2.basic are due to the randomness introduced by the value selection heuristic and by the shuffling of the variables in FindScope. Different runs of QuAcq2.basic or pseudoQuAcq1 discover the constraints in a different order.

Let us now observe the time performance of pseudoQuAcq1 when compared to QuAcq2.basic (columns $\mathtt{time_C}$, $\overline{\mathtt{t}}$, and $\mathtt{t_{max}}$ in the top part of Table 8). The randomness introduced by the value selection heuristic and the shuffling is also probably at the origin of the apparently stochastic variations that we observe. On the four problems on which the learning process converges, the time to generate a query is in general so short that a single more expensive query significantly impacts the averages. For instance, on Random, on one of the seven runs in which QuAcq2.basic converged, one query required ten minutes to be generated whereas most of the other queries were generated in a few milliseconds. On the same Random problem, the maximum time required by pseudoQuAcq1 to generate a query on the six runs in which it converged was 'only' 49 seconds. This explains the almost one order of magnitude smaller $\mathtt{time_C}$, $\overline{\mathtt{t}}$ and $\mathtt{t_{max}}$, which are not negatively impacted by the fact that pseudoQuAcq1 failed to converge on one more run than QuAcq2.basic. (Averages are computed on the converging runs.) Concerning $\overline{\mathtt{t}}$, the lower values observed for pseudoQuAcq1 (except on Zebra) can also in part be explained by the fact that the proportion of queries asked by FindScope is higher in pseudoQuAcq1 than in QuAcq2.basic. (This is especially true on Golomb.) Remember that these queries are extremely fast to compute as they are just projections of an assignment on a subset of its variables.

6.8 Discussion

These experiments tell us several important features of QuAcq2. First of all, these experiments show us that QuAcq2 can learn any kind of network, independently of whether some of the constraints follow a specific structure (Purdey, Zebra, Golomb, Jigsaw, Exam-TT), or there is 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 bias. 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 if interacting with a human user. We indeed observe that when we are close to the end of the learning process, it can be extremely time consuming to generate complete queries.

The experiment in Section 6.3 shows that a simple adaptation of the way queries are generated (see function GenerateQuery.cutoff in Section 6.3.1), allows us to monitor the CPU time needed to generate a query thanks to a cutoff. In that experiment we see that a cutoff of one second leads

[13] For RLFAP and Jigsaw, the numbers reported (in italic) correspond to the number of queries asked by pseudoQuAcq1 and QuAcq2.basic when reaching the timeout.

Table 8: Performance of pseudoQuAcq1 (a version of QuAcq2.basic that uses the FindScope version of QuAcq1) on the same problems as Table 4. The percentage of difference with the performance of QuAcq2.basic is given in brackets. Averages of ten runs (time in seconds).

Problem	$\#Q_C$		$time_C$		$\bar{t}$		t_{max}		#C
Purdey	104.0	(+41 %)	0.04	(-41 %)	0.00	(-62 %)	0.00	(-56 %)	10
Zebra	212.8	(+13 %)	3.66	(+106 %)	0.02	(+80 %)	2.50	(+130 %)	10
Golomb	817.2	(+242 %)	353.72	(+43 %)	0.21	(-58 %)	14.27	(+38 %)	10
Random	1839.3	(+68 %)	11.84	(-86 %)	0.01	(-92 %)	9.04	(-89 %)	6
RLFAP*	*1223.8*	*(+61 %)*	–		–		–		0
Jigsaw*	*1902.4*	*(+40 %)*	–		–		–		0

TO = 1 hour

Problem	$\#Q_{GenerateQuery}$		$\#Q_{FindScope}$		$\#Q_{FindC}$	
Purdey	11.8	(-5 %)	81.9	(+64 %)	10.3	(-10 %)
Zebra	14.9	(+0 %)	168.2	(+20 %)	29.7	(-9 %)
Golomb	70.9	(-0 %)	716.9	(+417 %)	29.4	(-1 %)
Random	147.3	(-2 %)	1653.8	(+85 %)	38.2	(-15 %)
RLFAP*	*155.2*	*(-7 %)*	*993.6*	*(+90 %)*	*75.0*	*(+4 %)*
Jigsaw*	*120.8*	*(+3 %)*	*1699.2*	*(+46 %)*	*82.4*	*(+7 %)*

*The numbers in italic correspond to the number of queries asked when reaching the timeout.

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. Apart from the time to generate queries, another interesting information is the kind of queries asked of the user. We see in Section 6.4 that the most frequent queries are the ones asked by FindScope. This is good news because such queries are simply the projection of queries already asked of the user. They are thus easier to answer than brand new queries.

All experiments in Sections 6.2, 6.3, and 6.4 were performed with fixed background knowledge. The experiment in Section 6.5 shows that the amount of background knowledge provided at the start of the learning process has a strong impact on the number of queries asked by QuAcq2. The larger the set of constraints given as background knowledge, the fewer the queries needed to learn the network. On all our benchmark problems, the number of queries required to learn the network exhibits a decreases that is at least linear in the number of constraints given as background knowledge.

The experiment on our use case reported in Section 6.6 is very informative. A total of about 260 queries are required to completely learn the problem and to prove convergence. If we stop as soon as a solution to the problem has been generated, the number of queries falls to fewer than 160 queries, among which only 12 are queries asked by the main procedure QuAcq2. The rest is composed of about 106 queries from FindScope (that can be avoided if the user is able to spot the variables violating a constraint in the example), and about 40 from FindC that are trivial to answer (involving two variables).

Finally, in Section 6.7, we compare QuAcq2 to a version that uses the function FindScope of QuAcq1 as described in [11]. The results show that our new function FindScope asks significantly fewer queries than the version in QuAcq1.

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 QuAcq2 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 most 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 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 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 illustrated on our use case. A comparison with QuAcq1 demonstrates that QuAcq2 requires significantly fewer queries to learn a network than its predecessor. As a last comment, we should bear in mind that most of the improvements of QuAcq1 already published in the literature (see Section 2), can be applied to QuAcq2.

Acknowledgements

We thank Bruno Zanuttini for helpful discussions on this work. The first, second, fourth, and fifth authors were supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under grant agreement no. ANR-19-PI3A-0004. The first and second authors received funding from the European Union through the project FP7-284715 ICON and the Horizon 2020 research and innovation program under grant agreement No 952215.

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] R. Arcangioli, C. Bessiere, and N. Lazaar. Multiple constraint acquisition. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016), New York, NY*, pages 698–704. IJCAI/AAAI Press, 2016.

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

[6] N. Beldiceanu, G. Ifrim, A. Lenoir, and H. Simonis. Describing and generating solutions for the EDF unit commitment problem with the modelseeker. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 733–748, Uppsala, Sweden, 2013. Springer.

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

[8] C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] C. Bessiere, R. Coletta, and N. Lazaar. Solve a constraint problem without modeling it. In *26th IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2014), Limassol, Cyprus*, pages 1–7. IEEE Computer Society, 2014.

[14] 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.

[15] C. Bessiere, A. Daoudi, E. Hebrard, G. Katsirelos, N. Lazaar, Y. Mechqrane, N. Narodytska, C.-G. Quimper, and T. Walsh. New approaches to constraint acquisition. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, volume 10101 of *Lecture Notes in Computer Science*, pages 51–76. Springer, 2016.

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

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

[18] F. Boussemart, 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.

[19] N.H. Bshouty. Exact learning boolean function via the monotone theory. *Inf. Comput.*, 123(1):146–153, 1995.

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

[21] A. Daoudi, N. Lazaar, Y. Mechqrane, C. Bessiere, and E.-H. Bouyakhf. Detecting types of variables for generalization in constraint acquisition. In *27th IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2015), Vietri sul Mare, Italy*, pages 413–420. IEEE Computer Society, 2015.

[22] 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.

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

[24] 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.

[25] M. Fumagalli, T. Prince Sales, and G. Guizzardi. Mind the gap!: Learning missing constraints from annotated conceptual model simulations. In *The Practice of Enterprise Modeling - 14th IFIP WG 8.1 Working Conference, PoEM 2021, Riga, Latvia, November 24-26, 2021, Proceedings*, volume 432 of *Lecture Notes in Business Information Processing*, pages 64–79. Springer, 2021.

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

[27] 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.

[28] M. Kumar, S. Kolb, S. Teso, and L. De Raedt. Learning MAX-SAT from contextual examples for combinatorial optimisation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020), New York, NY*, pages 4493–4500. AAAI Press, 2020.

[29] 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.

[30] 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.

[31] M. Lombardi, M. Milano, and A. Bartolini. Empirical decision model learning. *Artif. Intell.*, 244:343–367, 2017.

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

[33] G. Menguy, S. Bardin, N. Lazaar, and A. Gotlieb. Automated program analysis: Revisiting precondition inference through constraint acquisition. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2022)*, Vienna, Austria, 2022. ijcai.org.

[34] S. Paramonov, S. Kolb, T. Guns, and L. De Raedt. Tacle: Learning constraints in tabular data. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM 2017), Singapore*, pages 2511–2514. ACM, 2017.

[35] T. P. Pawlak and K. Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *Eur. J. Oper. Res.*, 261(3):1141–1157, 2017.

[36] S. Prestwich. Robust constraint acquisition by sequential analysis. In *Proceedings of the 24th European Conference on Artificial Intelligence ((ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 355–362, Santiago de Compostela, Spain, 2020. IOS Press.

[37] S. D. Prestwich, E. C. Freuder, B. O'Sullivan, and D. Browne. Classifier-based constraint acquisition. *Ann. Math. Artif. Intell.*, 89(7):655–674, 2021.

[38] 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.

[39] P. Rodler. Understanding the quickxplain algorithm: Simple explanation and formal proof. *CoRR*, abs/2001.01835, 2020.

[40] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[41] K.M. Shchekotykhin and G. Friedrich. Argumentation based constraint acquisition. In *Proceedings of the Ninth IEEE International Conference on Data Mining (ICDM'09)*, pages 476–482, Miami, Florida, 2009. IEEE.

[42] D. C. Tsouros and K. Stergiou. Efficient multiple constraint acquisition. *Constraints An Int. J.*, 25(3-4):180–225, 2020.

[43] D. C. Tsouros and K. Stergiou. Learning max-csps via active constraint acquisition. In Laurent D. Michel, editor, *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP 2021), Montpellier, France (Virtual Conference)*, volume 210 of *LIPIcs*, pages 54:1–54:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[44] D. C. Tsouros, K. Stergiou, and C. Bessiere. Omissions in constraint acquisition. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP 2020), Louvain-la-Neuve, Belgium*, volume 12333 of *Lecture Notes in Computer Science*, pages 935–951. Springer, 2020.

[45] 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.

[46] 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.