Guided Bottom-Up Interactive Constraint Acquisition

Dimosthenis C. Tsouros ✉ 🔾
KU Leuven, Belgium

Senne Berden ✉ 🔾
KU Leuven, Belgium

Tias Guns ✉ 🔾
KU Leuven, Belgium

──────── **Abstract** ────────

Constraint Acquisition (CA) systems can be used to assist in the modeling of constraint satisfaction problems. In (inter)active CA, the system is given a set of candidate constraints and posts queries to the user with the goal of finding the right constraints among the candidates. Current interactive CA algorithms suffer from at least two major bottlenecks. First, in order to converge, they require a large number of queries to be asked to the user. Second, they cannot handle large sets of candidate constraints, since these lead to large waiting times for the user. For this reason, the user must have fairly precise knowledge about what constraints the system should consider. In this paper, we alleviate these bottlenecks by presenting two novel methods that improve the efficiency of CA. First, we introduce a bottom-up approach named GrowAcq that reduces the maximum waiting time for the user and allows the system to handle much larger sets of candidate constraints. It also reduces the total number of queries for problems in which the target constraint network is not sparse. Second, we propose a probability-based method to guide query generation and show that it can significantly reduce the number of queries required to converge. We also propose a new technique that allows the use of openly accessible CP solvers in query generation, removing the dependency of existing methods on less well-maintained custom solvers that are not publicly available. Experimental results show that our proposed methods outperform state-of-the-art CA methods, reducing the number of queries by up to 60%. Our methods work well even in cases where the set of candidate constraints is 50 times larger than the ones commonly used in the literature.

1 Introduction and related work

Constraint programming (CP) is considered one of the main paradigms for solving combinatorial problems, with many successful applications in a variety of domains. However, there are still challenges to be faced in order for CP technology to become even more widely used. One of the most important challenges is to ease the modeling process. The current assumption in CP is that the user first models the problem and that a solver is then used to

solve it. However, modeling is a non-trivial task. Expressing a combinatorial problem as a set of constraints over decision variables is not straightforward and requires substantial expertise [12]. As a result, modeling is considered a major bottleneck for the widespread adoption of CP [12, 13, 14].

This obstacle has led to research into a very different approach to modeling: that of *learning* the constraint problem from data, as opposed to manually constructing it. This is the focus of the research area of *constraint acquisition (CA)*, in which CP meets machine learning. In CA, the model of a constraint problem is acquired (i.e., learned) (semi-)automatically from a set of examples of solutions, and possibly non-solutions. CA methods can be categorized as *active* or *passive* on the basis of whether a user provides feedback during learning or not.

In *passive acquisition*, a dataset of examples of solutions and non-solutions is provided by the user upfront. Based on these examples, the system learns a set of constraints modeling the problem [4, 5, 7, 9, 11, 17, 18, 19, 21]. Approaches vary in the types of constraints they are able to learn and the methodologies they employ: Conacq.1 is a version space algorithm for learning fixed-arity constraints [7, 9, 11], ModelSeeker learns global constraints that are taken from a predefined constraint catalog [4], and COUNT-CP is a generate-and-aggregate approach that can learn expressive first-order constraints [18]. None of these approaches are robust to errors in the labeled data. To this end, SeqAcq and BayesAcq were introduced, being robust to noise in the training set. In SeqAcq, a statistical approach based on sequential analysis is used [22], while in BayesAcq, a naive Bayes classifier is trained, from which a constraint network is then derived [23].

In contrast to passive learning, *active* or *interactive acquisition* systems learn the constraints through interaction with the user, by asking queries. The main type of query used is the *membership query*, which asks the user to classify a given example (i.e., an assignment to the variables of the problem) as a solution or a non-solution. An early work in active CA is the Matchmaker agent [15], where users, when they answer a membership query negatively, also have to provide a violated constraint. In order to lower the expertise level required from the user, Bessiere et al. later proposed Conacq.2 [10, 11] – an active version of Conacq.1 that uses membership queries and does not require the user to provide any violated constraints. In [24], Conacq.2 was in turn extended to also accept arguments regarding *why* examples should be rejected or accepted.

As the number of membership queries needed can be exponentially large for these methods [11], a new family of interactive algorithms was proposed that use *partial queries* instead [3, 8, 20, 25, 26, 27, 28, 29]. A partial query asks the user to classify a partial assignment to the variables. Using partial queries, CA systems are able to converge faster. QuAcq was the first system to use partial queries [6, 8], and was later extended into MultiAcq [3]. MQuAcq was later introduced to reduce the number of queries needed per learned constraint [25, 29], and MQuAcq-2 further improved the performance by exploiting the structure of the constraints already learned [27].

Despite these advancements in active CA, there are still significant obstacles for the technology to become usable in practice. One of the main limitations is that it typically still requires asking a large number of queries to the user in order to find all constraints. In addition, existing systems cannot handle large sets of candidate constraints in reasonable run times, and thus require significant expertise from the user in limiting the constraints the system should consider (and thus the size of the candidate set) upfront. Finally, query generation – a highly important part of the CA process – currently requires the use of customized solvers that are not publicly available and are not as well-maintained as conventional solvers. Without the use of such customized solvers, current active CA algorithms can lead to very high query generation times or are sometimes unable to converge to the correct set of constraints when time limits are imposed [1, 25].

We focus on the above limitations, and contribute the following improvements:

- We present a novel query generation method named PQ-GEN that allows conventional constraint solvers to be used by CA algorithms while also ensuring convergence, removing the dependency on customized solvers.

- We propose a bottom-up learning approach named GROWACQ that uses any other CA algorithm to learn the constraints of an increasingly large problem. It starts learning with only a subset of variables and an associated subset of candidate constraints, and incrementally grows this set of variables and constraints. This allows it to handle significantly larger sets of candidate constraints and reduces the maximum waiting time for the user.

- Finally, we introduce a better way to *guide* the query generation process, with the goal of generating queries that learn the set of constraints faster. We propose an objective function for query generation that uses *probabilistic estimates* of whether constraints are likely to hold or not. We demonstrate the potential of this method by using a simple counting-based approach as probabilistic estimator.

The rest of the paper is structured as follows. Some background on CA is given in Section 2. Sections 3–5 present our proposed methods. An experimental evaluation is given in Section 6. Finally, Section 7 concludes the paper.

2 Background

We now introduce some basic notions regarding constraint satisfaction problems and interactive constraint acquisition.

2.1 Constraint satisfaction problems

A *constraint satisfaction problem* (*CSP*) is a triple $P = (X, D, C)$, consisting of:

- a set of n variables $X = \{x_1, x_2, ..., x_n\}$, representing the entities of the problem,
- a set of n domains $D = \{D_1, D_2, ..., D_n\}$, where $D_i \subset \mathbb{Z}$ is the finite set of values for x_i,
- a constraint set (also called constraint network) $C = \{c_1, c_2, ..., c_t\}$.

A *constraint* c is a pair $(rel(c), var(c))$, where $var(c) \subseteq X$ is the *scope* of the constraint and $rel(c)$ is a relation over the domains of the variables in $var(c)$ that specifies (implicitly or explicitly) what assignments are allowed. $|var(c)|$ is called the *arity* of the constraint. The constraint set $C[Y]$, where $Y \subseteq X$, denotes the set of constraints from C whose scope is a subset of Y. The set of solutions of a constraint set C is denoted by $sol(C)$. A *redundant* or *implied* constraint $c \in C$ is a constraint in C such that $sol(C) = sol(C \setminus \{c\})$.

A (partial) *assignment* e_Y is an assignment over a set of variables $Y \subseteq X$. e_Y is *rejected* by a constraint c iff $var(c) \subseteq Y$ and the projection $e_{var(c)}$ of e_Y on the variables in the scope $var(c)$, is not in $rel(c)$, that is, is not allowed by the constraint. $\kappa_C(e_Y)$ represents the subset of constraints from $C[Y]$ that reject e_Y, i.e., $\kappa_C(e_Y) = \{c \,|\, c \in C[Y] \wedge e_{var(c)} \notin rel(c) \}$.

A complete assignment e that is accepted by all the constraints in C is a *solution* to C, i.e., $e \in sol(C)$. A partial assignment e_Y is called a *partial solution* to C iff it is accepted by all the constraints in $C[Y]$. Note that a partial solution to C may not be extendable to a complete one, due to constraints not in $C[Y]$.

■ **Algorithm 1** Constraint Acquisition through partial queries.

Input: X, D, B, C_{in} (X: the set of variables, D: the set of domains, B: the bias, C_{in}: an optional set of known constraints)

Output: C_L : a learned constraint network

1: $C_L \leftarrow C_{in}$
2: **while** True **do**
3:     Generate an e accepted by C_L and rejected by B
4:     **if** $e =$ nil **then return** C_L ▷ Stopping condition
5:     **if** $ASK(e) =$ Yes **then**   ▷ Ask (partial) membership query e
6:         Remove the constraints rejecting e, namely $\kappa_B(e)$, from B
7:     **else**
8:         Find one (or more) minimal scopes S in e for which $|\kappa_B(e_S)| \geq 1$ and $ASK(e_S) =$ No
9:         Find all $\{c \in C_T \mid var(c) = S\}$ through partial queries; add to C_L, remove from B

2.2   Active constraint acquisition with partial membership queries

In CA, the pair (X, D) is called the *vocabulary* of the problem at hand and is common knowledge shared by the user and the system. Besides the vocabulary, the learner is also given a *language* Γ consisting of *fixed-arity* constraint relations. Using the vocabulary (X, D) and the constraint language Γ, the system generates the *constraint bias B*, which is the set of all possible candidate constraints for the problem.

Let C_T, the target constraint network, be an unknown set of constraints such that for every assignment e over X it holds that $e \in sol(C_T)$ iff e is a solution to the problem the user has in mind. The goal of CA is to learn a constraint set $C_L \subseteq B$ that is equivalent to C_T. Like other works, we assume that the bias B can represent C_T, i.e., there exists a $C \subseteq B$ s.t. $sol(C) = sol(C_T)$.

In active CA, the system interacts with the user while learning the constraints. A *membership query* [2] in this setting is a question $ASK(e_X)$, asking the user whether a complete assignment e_X is a solution to the problem that the user has in mind. A *partial query* $ASK(e_Y)$, with $Y \subset X$, asks the user to determine if e_Y, which is an assignment in D^Y, is a partial solution with respect to $C_T[Y]$. We use the notation $c \in C_T$ iff $\forall e \in D^Y$ with $var(c) \subseteq Y \subseteq X$, $ASK(e_Y) = True \implies e_{var(c)} \in sol(c)$.

While in passive acquisition there are methods that can handle noisy answers [22, 23], this is not the case for active acquisition. For this reason, in this work, we follow the assumption that the user answers all queries correctly.

A query $ASK(e_Y)$ is called *irredundant* iff the answer is not implied by any information already available to the system. That is, the query is irredundant iff e_Y is rejected by at least one constraint from the bias B and is not rejected by the network C_L learned thus far. The first condition captures that $\kappa_B(e_Y)$ cannot be empty, since if $\kappa_B(e_Y)$ would be empty, the answer to the query $ASK(e_Y)$ would have to be "yes", based on the assumption that C_T is representable by the constraints in B. The second condition captures that e_Y should not be rejected by any constraint in the learned network C_L, since otherwise the user would certainly answer "no" to the query.

Algorithm 1 presents the generic process followed by active CA methods with partial queries. The learned set C_L is first initialized either to the empty set or to a set of constraints given by the user that is known to be part of C_T (i.e., $C_{in} \subset C_T$) (line 1). Then the main

loop of the acquisition process begins, where, in every iteration, the system first generates an irredundant query (line 3) and posts it to the user (line 5). If the query is answered positively, then the candidate constraints from B that violate it are removed (line 6). Otherwise, the system has to find one or more constraints from C_T that violate the query. This is done in two steps. First, queries are asked to find the scope of a constraint in $\kappa_{C_T}(e)$ (line 8). Then, queries are asked to find all constraints $c \in C_T$ with that scope (line 9).

The acquisition process has *converged* on the learned network $C_L \subseteq B$ iff C_L agrees with the set of all labeled examples E, and for every other network $C \subseteq B$ that agrees with E, it holds that $sol(C) = sol(C_L)$. This is proved if no query could be generated at line 3, as in this case, all remaining constraints in B (if any) are redundant. If the first condition is true but the second condition has not been proved when the acquisition process finishes, *premature convergence* has occurred. This can happen when the query generation at line 3 returns $e = nil$, but without having proved that an irredundant query does not exist (e.g., because of a time limit).

Existing algorithms like QuAcq [6, 8], MQuAcq [25, 29] and MQuAcq-2 [27] follow this template, but differ mainly in how they implement lines 3, 8 and 9, and hence how many constraints they are able to learn in each iteration. Examples of functions used to locate the scope of a constraint (line 8) are *FindScope* [6, 8] or the more efficient *FindScope-2* [25]. To learn the constraints in the scope found (line 9), the *FindC* function is typically used [6, 8].

3 Using conventional solvers for query generation

Query generation (line 3 of Algorithm 1) is one of the most important parts of the CA process. It aims to find an *irredudant* membership query (i.e., a (partial) assignment that does not violate C_L but violates at least one $c \in B$) that will be asked to the user. Thus, it can be formalized as follows:

$$\text{find } e_Y \text{ s.t. } e_Y \in sol(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i),$$

which can be formulated as a CSP with variables Y and constraints $C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i$.

3.1 Problems when using conventional solvers

In principle, this CSP could be solved using any conventional CP solver. However, this can lead to issues for the following two reasons.

A large bias. At the start of the acquisition process, the set of candidate constraints B can be very large. This makes the propagation of the constraint $\bigvee_{c_i \in B[Y]} \neg c_i$ time-consuming, and severely slows down the query-generation process.

Indirectly implied constraints. At the end of the acquisition process, only constraints that are implied by C_L remain in B, if any. In this case, it will be impossible to generate a query that does not violate C_L and violates at least one constraint from B. However, propagation is often unable to prove such implications when they are indirect and involve multiple variables and constraints. For this reason, solvers internally end up enumerating all possible variable assignments satisfying C_L and checking if the constraint $\bigvee_{c_i \in B[Y]} \neg c_i$ can be satisfied. This can be very time-consuming, and a time limit is usually imposed on query generation, leading to *premature convergence*.

■ **Algorithm 2** PQ-GEN: Projection-based Query Generation.

Input: C_L, B, l, t (B: the set of candidate constraints (bias), C_L: set of known constraints, l: size limit, t: time limit)
Output: e: the query generated
1: $timer.start()$
2: $Y \leftarrow \bigcup_{c \in B} var(c)$
3: **if** $|B| > l$ **then**
4:     $e \leftarrow \text{solve}(C_L[Y])$
5:     **if** $\exists c \in B : e \notin sol(c)$ **then**
6:         **return** e
7: $e \leftarrow \text{solve}(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i)$
8: **if** $timer.end() < t$ **then**
9:     $e' \leftarrow \text{solve}(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i, \text{maximize: } obj, \text{time limit:}t - timer.end())$
10:     **if** $e' \neq nil$ **then**
11:         **return** e'
12: **return** e

In order to limit the large runtimes in a more advanced way than by simply imposing a time bound t, Addi et al. proposed a method using conventional solvers named TQ-GEN [1]. It iteratively tries to solve the query generation problem, by gradually reducing the number of variables taken into account by a proportion $\alpha \in]0, 1[$, until a query can be generated within a small time limit τ. This is repeated until either an irredundant query is generated, or a global time bound t is reached, leading to premature convergence. However, choosing the right hyperparameters for t and τ is problem-specific [1] and requires tuning, and thus more interaction with the user.

3.2 Customized solvers

To avoid premature convergence, a CP solver can be customized to store partial assignments that satisfy every $c \in C_L[Y]$ and violate at least one $c \in B[Y]$ during the search. Given an objective function, such as maximizing the number of assigned variables, in every non-failing node of the search tree it will check the above property and, if fulfilled, store the best-scoring partial assignment.

As these customized solvers are guaranteed to find valid partial solutions, their use will never lead to premature convergence. In addition, finding a partial query to return is not time-consuming (especially when combined with specialized search heuristics [25]), even when the bias is large. However, such custom solvers are not publicly available and are typically not based on the latest version of state-of-the-art solvers. This also means that the corresponding active CA methods are heavily tied to those particular customized solvers.

3.3 Projection-based Query Generation

We now introduce a method named *Projection-based Query Generation* (PQ-GEN) that makes it possible to use state-of-the-art conventional solvers for query generation, without premature convergence. Our proposed method is shown in Algorithm 2.

Avoiding indirectly implied constraints. A key observation we make is that when generating a query on line 3, it might be that $\bigcup_{c \in B} var(c) \subset X$, that is, some variables have no more candidate constraints in B. These have become irrelevant, as both lines 6 and 8 are only

concerned with $\kappa_B(e)$, which will not include these variables. So, to generate an irredundant query, it is sufficient to consider only the variables in B. This is not only faster, but also avoids indirectly implied constraints, as these are indirect through variables not used in B.

Thus, our proposed query generator projects the variables down to $Y \subseteq X$, with $Y = \bigcup_{c \in B} var(c)$, thereby simplifying the problem to finding an assignment over $Y \subseteq X$. This will inherently result in a partial assignment when Y is a strict subset of X, without requiring a custom solver. Thus, we first compute the set of variables Y relevant to the query (line 2), and project C_L down to those variables (on lines 4 and 7). The solver then has to prove that there exists a query that satisfies $C_L[Y]$ and violates at least one constraint from B.

Dealing with large biases. As mentioned above, having a large bias B can severely slow down the solver during query generation because propagating the $\bigvee_{c_i \in B[Y]} \neg c_i$ constraint takes a long time. However, we observe that when B contains many constraints, the property that a query e violates at least one of these is usually satisfied without needing to enforce this. Hence, we propose not using this constraint when the bias is larger than some threshold (lines 3 to 6 in Algorithm 2). If in a post-hoc check, it turns out that the generated query violates at least one $c \in B$, it is directly returned (line 6). Otherwise, we again generated a query, this time *with* the constraint enforcing that there must exist a constraint in B that is violated (line 7).

Optimizing the query. The above ensures that we will always find *a* valid query. However, much better queries – according to some objective function – can often be found. This would take additional time, but is safe because, since a valid query has already been found, the optimization can always safely be interrupted. Given a time limit, we can hence call an optimization solver for the remaining time after a first valid query has been found (lines 8-11).

As expressed in Proposition 1, Algorithm 2 is correct.

▶ **Proposition 1.** *Given a bias B, with an unknown target network C_T being representable by B, and a learned constraint set C_L, if nil is returned by Algorithm 2, then the system has converged on $C_T[X]$.*

Proof. When *nil* is returned by Algorithm 2, it means that $\nexists e \in sol(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i)$, with $Y = \bigcup_{c \in B} var(c)$, i.e., $\nexists e \in sol(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i)$. In order to prove convergence over all of X, we must have $\nexists e \in sol(C_L[X] \wedge \bigvee_{c_i \in B[X]} \neg c_i)$. We will now show that when $Y = \bigcup_{c \in B} var(c)$, it means that

$$\nexists e \in sol(C_L[Y] \wedge \bigvee_{c_i \in B[Y]} \neg c_i) \implies \nexists e \in sol(C_L[X] \wedge \bigvee_{c_i \in B[X]} \neg c_i)$$

Assume that Algorithm 2 returns *nil*, i.e., that no assignment exists in a $Y \subset X$ that is accepted by $C_L[Y]$ and rejected by $B[Y]$. This means that all the constraints in $B[Y]$ are proved to be implied by the constraints in $C_L[Y]$. Thus, the remaining constraints in B, that are not proved to be redundant, are the constraints $c \in B \setminus B[Y]$. When we know that $Y = \bigcup_{c \in B} var(c)$ it means that $B[Y] = B$, so $B \setminus B[Y] = \emptyset$. As a result, in this case, all the constraints in B are proved to be implied. Hence, no assignment that is accepted by C_L and rejected by B exists in X. ◀

4 Bottom-up Constraint Acquisition

We start by observing that all current active CA algorithms always consider either the full set of variables X, or a large subset $Y \subseteq X$, in their top-level loop (lines 2-9 in Algorithm 1). This generally leads to complete or almost-complete queries getting generated (line 3 of

■ **Algorithm 3** Growing Acquisition.

Input: Γ, X, D, C_{in} (Γ: the language, X: the set of variables, D: the set of domains, C_{in}: an optional set of known constraints)

Output: C_L : a constraint network

1: $C_L \leftarrow \emptyset$
2: $Y \leftarrow \emptyset$
3: **while** $|Y| \leq |X|$ **do**
4:      $x \leftarrow x \in (X \setminus Y)$
5:      $Y \leftarrow Y \cup \{x\}$
6:      $B \leftarrow \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y \wedge x \in var(c)\}$
7:      $C_L \leftarrow \text{Acq}(Y, D^Y, B, C_L \cup C_{in}[Y])$
8: **return** C_L

Algorithm 1). However, larger queries are generally harder to answer than smaller queries [25]. Also, a large initial query leads to many additional queries getting posed in the scope-finding method on line 8. That is because the worst-case complexity of the best scope-finding methods, in terms of the number of queries required, is $\Theta(log(|Y|))$, where $Y \subseteq X$ is the set of variables considered [25].

Additionally, by directly considering the whole set of variables, the CA algorithm has to represent and operate on the entire set of candidate constraints (i.e., the bias B) at once. The bias is used in many parts of the acquisition process. Hence, the memory requirements and the run time of the acquisition process increase significantly as the bias grows, either because the problems contain more variables or because the language Γ given to the system includes a larger number of relations. This means that, in practice, state-of-the-art active CA methods are only applicable to problems with not too many variables or problems for which the user already has relatively precise knowledge about what constraints the system should consider (which corresponds to the bias being small).

To improve on this, we propose a novel *meta*-algorithm named GrowAcq (Algorithm 3). The key idea is to call a CA algorithm on an increasingly large subset of the variables $Y \subseteq X$, each time using only a relevant unexplored subset of the bias. GrowAcq begins with $Y = \emptyset$ (line 2) and gradually incorporates more variables (lines 3-5). Once a new variable $x_i \in X$ has been added to Y, the new problem becomes to find the new $C_T[Y]$. However, as $C_T[Y \setminus \{x_i\}]$ was already found in the previous iterations, the set of constraints to seek is actually $C_T[Y] \setminus C_T[Y \setminus \{x_i\}]$. To find $C_T[Y] \setminus C_T[Y \setminus \{x_i\}]$, any existing active CA algorithm can be used. We represent this with the function *Acq* (line 7). In every iteration, only a part of the bias B is needed, namely $B[Y] \setminus B[Y \setminus \{x_i\}]$, and as shown in Lemma 1, the bias constructed at line 6 is equivalent to $B[Y] \setminus B[Y \setminus \{x_i\}]$.

▶ **Lemma 1.** *Let Y_i be the set of variables Y in iteration i after line 5 of Algorithm 3 and $B_i = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \wedge x_i \in var(c)\}$ be the bias B constructed at line 6 in iteration i. It holds that $B_i = B[Y_i] \setminus B[Y_{i-1}]$.*

Proof. At line 6 of Algorithm 3, the bias B is constructed. For each iteration i, it is constructed as $B_i = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \wedge x_i \in var(c)\}$. For a set of variables Y_i, the full bias, which includes all candidate constraints, is $B[Y_i] = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i\}$. For the previous iteration, as $Y_{i-1} = Y_i \setminus \{x_i\}$, we know that $B[Y_{i-1}] = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \setminus \{x_i\}\}$. Thus, the additional constraints that are in $B[Y_i]$ and not in $B[Y_{i-1}]$

are the ones with a scope $var(c) \subseteq Y_i$ for which $x_i \in var(c)$:

$$B[Y_i] \setminus B[Y_{i-1}] = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i\} \setminus \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \setminus \{x_i\}\}$$
$$= \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \wedge x_i \in var(c)\} = B_i$$

Hence, it holds that $B_i = B[Y_i] \setminus B[Y_{i-1}]$ for $B_i = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq Y_i \wedge x_i \in var(c)\}$. ◀

This bottom-up approach alleviates the problems described above, i.e., starting from large initial queries and having to represent the whole bias from the beginning, in two ways. First, it naturally leads to partial queries of increasing size in the first step of the "inner" CA system (Algorithm 1 line 5). This is valuable since smaller queries are generally easier for the user to answer [25], and also a smaller initial query leads to a lower worst-case number of additional queries to locate scopes. Second, since the algorithm only stores and uses a small part of the bias at a time (line 6 of Algorithm 3), it is able to handle significantly larger biases than the state-of-the-art. Not representing the whole bias in every iteration does not affect the algorithm's correctness, as we state in Proposition 2.

▶ **Proposition 2.** *Given a bias B built from a language Γ, with bounded arity constraints, and a target network C_T representable by B, GROWACQ is correct (i.e., will learn a constraint set C_L that is equivalent to C_T), as long as a correct (i.e., sound and complete) CA algorithm is used in line 7.*

Proof. (Sketch)

Let us now prove that if any correct algorithm is used in line 7 of Algorithm 3 – like QUACQ, MQUACQ or MQUACQ-2 – GROWACQ remains correct. We will subscript sets with the number of the iteration that they occur in to distinguish between the iterations. Even though the full bias B is never constructed and never kept in memory all at once in GROWACQ, we will still refer to it in this proof and denote it with B, i.e., $B = \{c \mid rel(c) \in \Gamma \wedge var(c) \subseteq X\}$. When we instead write B_i, we refer to the part of the bias that is constructed and used in iteration i (line 6 of Algorithm 3), which is $B[Y_i] \setminus B[Y_{i-1}]$ (Lemma 1).

Soundness. GROWACQ adds constraints to C_L only at line 7 of Algorithm 3. At that line, only constraints returned from the inner interactive CA algorithm are added to C_L. Since the assumption is that a sound algorithm is used in the *Acq* function, GROWACQ is sound.

Completeness. We prove that GROWACQ is complete by proving by induction that, after each iteration i, C_L is equivalent to $C_T[Y_i]$, meaning that after the last iteration, C_L is equivalent to $C_T[X]$. GROWACQ starts with $Y_1 = \emptyset$, so both $C_T[Y_1]$ and B_1 are empty. The first iteration where the algorithm has to actually learn any constraints will be the one where Y grows large enough so that $C_T[Y] \neq \emptyset$. Assume that this happens at iteration k. In this case, $C_T[Y_k]$ will be representable by B_k, because $B_k = B[Y_k] \setminus B[Y_{k-1}]$ and we know that $C_T[Y_{k-1}] = \emptyset$. Since $C_T[Y_k]$ is representable by B_k, it will be successfully learned in line 7, as long as a complete interactive CA algorithm is used.

Assuming now that $C_L = C_T[Y_n]$ holds at the end of the n-th iteration, let us now prove that $C_L = C_T[Y_{n+1}]$ will hold at the end of the $n+1$-th iteration. From the assumption that $C_L = C_T[Y_n]$, it follows that $(B[Y_n] \setminus C_L) \cap C_T = \emptyset$. As a result, B_{n+1}, being equal to $B[Y_{n+1}] \setminus B[Y_n]$ does not exclude any constraint from $C_T[Y_{n+1}]$ that has not already been learned. From this, it follows that $(C_T[Y_{n+1}] \setminus C_L) \subseteq B_{n+1}$, and thus this set of constraints will be learned in line 7 as long as a complete interactive CA algorithm is used. Hence, GROWACQ is complete. ◀

5   Guided query generation

We now turn our attention to the objective function used at line 9 of Algorithm 2. Since when GrowAcq is used, the size of B used in every iteration is reduced, query generation is now often fast, leaving sufficient room for using optimization to find a *good* query.

The objective function used in existing query generation systems [6, 25] tries to maximize the number of constraints from B that are violated by the generated query e. The motivation is that this can potentially help shrink the bias faster. The objective function is

$$e = \arg\max_e \sum_{c \in B} [\![e \notin sol(\{c\})]\!]$$

where $[\![\cdot]\!]$ is the Iverson bracket which converts *True/False* into 1/0.

However, looking only at the number of violated constraints in B does not fully capture what a good query is:

- We want queries that lead to a positive answer to violate many constraints from the bias B, as these can then all be removed from B, shrinking it faster.
- On the other hand, we want queries that lead to a negative answer to violate a small number of constraints from B, as it allows the CA system to find the conflicting constraint faster.

Based on this, in order to generate good queries regardless of the user's answer, we want query generation to minimize the violation of constraints that are in the unknown target set C_T, seeking a query to which the user's answer will be "yes". At the same time, we want to maximize the violation of constraints in B that are not in C_T, so that positive answers can shrink the bias faster (the first bullet point above). Note that we also have the constraint ensuring that at least one constraint from B has to be violated. This means that when $B \setminus C_T = \emptyset$, we want a minimum number of constraints in C_T that we have not already learned to be violated. This leads to negative queries that violate a small number of constraints in B (the second bullet point above).

Assume we have access to an oracle $\mathcal{O}$ that tells us whether a constraint c belongs to the unknown target set or not: $\mathcal{O}(c) = (c \in C_T)$. Using this oracle we can formulate an objective function for query generation, using the reasoning above, as follows:

$$\sum_{c \in B} [\![e \notin sol(\{c\})]\!] \cdot (1 - |\Gamma| \cdot [\![\mathcal{O}(c)]\!]),$$

On the one hand, every time that the oracle returns *False* for a constraint from the bias that is violated by e, the objective function is increased by 1, thereby maximizing the violation of these constraints. Conversely, for constraints where $\mathcal{O}$ returns *True*, we aim to minimize the violations, which requires a reduction in the objective value for each such violated constraint. However, it is possible that violating a set of constraints C (where $\forall c_i \in C \,|\, \mathcal{O}(c_i) = False$) may imply the violation of a constraint c_j with $\mathcal{O}(c_j) = True$. In such cases, if the reduction in the objective value for violating c_j is not large enough, the system will violate both C and c_j, maximizing the objective. To address this issue, we introduce a "penalty" of $|\Gamma|$, which is equal to the upper bound of the number of constraints in each scope. This ensures that the system prioritizes satisfying a constraint with $\mathcal{O}(c_j) = True$, over violating other constraints from B.

Modeling the oracle.   Observe how the current objective of maximizing violations corresponds to using a model of the oracle M that always answers *False*, i.e., that assumes that none of the candidate constraints belong to C_T. On the other hand, if we used an oracle M

that always answers *True*, then the query generation would try to violate as few constraints as possible. However, the $\bigvee_{c_i \in B[Y]} \neg c_i$ constraint would still need to be satisfied, in the extreme case leading every query to violate exactly one constraint from B. Based on this observation, we propose to model the oracle using the following model M, which tries to determine for every constraint c whether violating or satisfying c would lead to the least amount of queries later on in the algorithm.

$$M(c) = \left(\frac{1}{P[c \in C_T]} \leq log(|Y|) \right)$$

On the one hand, in the extreme case, the constraints for which $M(c)$ answers *True* will be violated one by one in the later queries (once most of the constraints for which $M(c)$ answers *False* have been dealt with). Let $P[c \in C_T]$ be a probabilistic estimate of whether c is part of C_T. Then, if the generated queries would violate the constraints with that probability one by one, we would in expectation need $1/P[c \in C_T]$ queries to find a constraint from C_T. For example, for a set of constraints that each has a probability of 25%, 1 in every 4 queries is expected to lead to a $c \in C_T$ being learned.

On the other hand, for each constraint $c \in C_T$ for which $M(c)$ answers *False*, a scope-finding procedure is needed to locate the violated constraint. The most efficient functions commonly used to do it (i.e., FindScope [6] or FindScope-2 [25]) have been shown to require $\Theta(log(|Y|))$ queries to find a violated constraint $c \in C_T$ in the worst case, where Y is the number of variables considered in query generation. As a result, we estimate the number of queries needed in this case as $k \cdot log(|Y|)$, with k a constant. We found $k = 1$ to work well in practice.

Probability estimation. To compute the probability $P(c \in C_T)$ of a constraint $c \in B$, we use a simple approach, considering only information from the relations $rel(c)$ of the constraints. More specifically, to compute $P(c \in C_T)$, we count the number of times a constraint with relation $rel(c)$ has been added to C_L, and divide it by the total number of times that such a constraint has been removed from B. Much more advanced estimation techniques, including machine learning methods, can be used for more accurate estimation. We leave this for future work.

6 Experimental evaluation

In this section, we empirically answer the following research questions:

(Q1) Does using PQ-GEN with conventional solvers avoid premature convergence, and how do CA systems perform when they use it?

(Q2) Does GROWACQ (using MQUACQ-2) perform better than using MQUACQ-2 directly?

(Q3) How does our probability-guided query generation objective function perform compared to the one used in current CA systems?

(Q4) How does the combination of our methods perform?

(Q5) How do our methods perform on problems with a huge bias B?

6.1 Benchmarks

We used the following benchmarks:

Jigsaw Sudoku. The Jigsaw Sudoku is a variant of Sudoku in which the 3×3 boxes are replaced by irregular shapes. It consists of 81 variables with domains of size 9. The target network consists of 811 binary $\neq$ constraints, on rows, columns, and shapes. The bias B was constructed using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$ and contains 19 440 binary constraints.

Murder. The Murder puzzle problem consists of 20 variables with domains of size 5. The target network contains 4 cliques of 10 $\neq$ constraints and 12 additional binary constraints. The bias was initialized with 760 constraints based on the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

Random. We used a problem with 100 variables and domains of size 5. We generated a random target network with 495 $\neq$ constraints. The bias was initialized with 19 800 constraints, using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

Golomb rulers. The problem is to find a ruler where the distance between any two marks is different from that between any other two marks. We built a simplified version of a Golomb ruler with 8 marks, with the target network consisting only of quaternary constraints.[1] The bias, consisting of 238 binary and quaternary constraints, was created with the language $\Gamma = \{\geq, \leq, <, >, \neq, =, |x_i - x_j| \neq |x_k - x_l|\}$.

Job-shop scheduling. The job-shop scheduling problem involves scheduling a number of jobs, consisting of several tasks, across a number of machines, over a certain time horizon. The decision variables are the start and end times of each task. There is a total order over each job's tasks, expressed by binary precedence constraints. There are also constraints capturing the duration of the tasks and that tasks should not overlap on the same machine. The language $\Gamma = \{\geq, \leq, <, >, \neq, =, x_i + c = x_k\}$ was used, with c being a constant from 0 up to the maximal duration of the jobs. We used a problem instance containing 10 jobs, 3 machines (i.e., $|X| = 60$) and a time horizon of 15 steps, leading to a bias containing 14 160 constraints.

6.2 Experimental setup

Let us now give some details about the experimental settings:

- All the experiments were conducted on a system carrying an Intel(R) Core(TM) i9-11900H, 2.50 GHz clock speed, with 16 GB of RAM.
- We measure the total number of queries $\#q$, the average time of the query generation process $\bar{T}_{gen}$ (line 3 of Algorithm 1), the average waiting time $\bar{T}$ per query for the user, and the total time needed (to converge) T_{total}. All times are presented in seconds. The difference between $\bar{T}_{gen}$ and $\bar{T}$ is that the latter takes into account also the queries posed on lines 8-9 of Algorithm 1, which are very fast to compute.
- We evaluate our methods in comparison with the state-of-the-art method MQuAcq-2 [27].
- All methods and benchmarks were implemented in Python [2] using the CPMpy constraint programming and modeling library [16], except for the experiments using custom solvers.[3]
- The results presented in each benchmark, for each algorithm, are the means of 10 runs.

We now discuss the results of our experimental evaluation, based on the questions we posed at the beginning of the section.

6.3 [Q1] Performance of PQ-Gen

Both PQ-Gen, our projection-based query generation approach, and TQ-Gen [1] (discussed in Section 3.1) involve hyperparameters that affect their performance. Thus, we first performed a hyperparameter sensitivity analysis to assess their performance under different

---

[1] The ternary constraints derived when $i = k$ or $j = l$ in $|x_i - x_j| \neq |x_k - x_l|$ were excluded, as also done in the literature [25, 27]

[2] Our code is available online in: `https://github.com/Dimosts/ActiveConLearn`

[3] For the custom solver based query generators from [6, 25], we obtained the implementations (in C++) through personal communication with the authors.

configurations. In tandem with TQ-GEN, we also used the adjust function described in [1]. We used the JSudoku benchmark for this comparison. For TQ-GEN, we fixed the hyperparameter α to 0.8 as recommended in [1], and used $\tau = \{0.05, 0.1, 0.2, 0.3\}$ and $t = \{0.5, 1, 1.5, 2\}$. For PQ-GEN hyperparameters, we used $l = \{3000, 5000, 7500, 10000\}$ and $t = \{0.5, 1, 1.5, 2\}$. Thus, we examined 16 different configurations for each. A summary of the results are shown in Table 1.[4]

■ **Table 1** A summary of the performance of TQ-GEN and PQ-GEN with different configurations.

Problem	$Conv$	$\#q$	T_{max}	T_{total}
MQuAcq-2 with TQ-GEN [1]	32%	7 555	20.66	2 371.40
MQuAcq-2 with PQ-GEN (ours)	100%	6 551	4.42	728.25

Confirming our analysis, with our PQ-GEN there is never a case of premature convergence, no matter what hyperparameters are used. On the other hand, when TQ-GEN is used, the system fails to converge in the majority of cases, and specific hyperparameter values have to be chosen to ensure convergence. In addition, our PQ-GEN shows much better performance both in terms of the number of queries needed and, especially, in terms of runtime.

In more detail, we compared our projection-based query generation (PQ-GEN) with a baseline where we run a conventional CP solver to directly solve the query generation problem, using a one-hour time limit, as well as with query generation methods from the literature, i.e., TQ-GEN and the custom solver based query generators from [6, 27]. For PQ-GEN and TQ-GEN we used the best configuration found in the previous experiment. That is, we run PQ-GEN with $l = 5000$ and $t = 1$ and TQ-GEN with $\tau = 0.2$ and $t = 2$. We used benchmarks that are similar to the ones used in [6, 27]. For consistency, we used the same state-of-the-art query generation objective function across all methods that accept one, i.e., our PQ-GEN and the custom solvers, which tries to maximize the number of violated constraints from B. The results are shown in Table 2.

We can observe that convergence was reached in all cases, except for the baseline in which a conventional solver was used directly using a time limit. Our method, PQ-GEN, and the baseline show similar performance in terms of the number of queries needed. while being much better than TQ-GEN in JSudoku and Random, where B is larger, especially when considering time performance.

On the other hand, when custom solvers are used, we can see that the time performance has improved and the number of queries has decreased. This happens because the custom solver can return a partial assignment of any size, trying only to maximize the value of the objective function used, and utilizing heuristics from the literature, while when our PQ-GEN is used, the query generated has to be a solution in a specific (sub)set of variables, which takes more time to compute. As a result, we observed that custom solvers often return queries that violate more constraints from B, which helps MQuAcq-2 shrink the bias faster in terms of the number of queries needed.

6.4 [Q2 - Q4] Evaluating GrowAcq and guided query generation

Hereafter, we continue our experiments using PQ-GEN, as the motivation was to investigate techniques that work with any solver. As using PQ-Gen allows us to use conventional solvers, able to run on any given benchmark, in contrast to the custom solvers from [6, 27], where specific constraint relations are implemented, from now on we will use all of the benchmarks mentioned in Section 6.1.

---

[4] More details regarding this experiment can be found in Appendix A.

Table 2 Comparing PQ-GEN with state-of-the-art query generators.

Method	Problem	#q	$\bar{T}_{gen}$	$\bar{T}$	T_{max}	T_{total}	$Convergence$
	Using conventional solvers						
MQUACQ-2 BASELINE	JSudoku	6 337	2.05	0.21	11.67	-	0%
	Murder	347	0.09	0.01	0.31	4.69	100%
	Random	5 694	2.90	0.05	20.78	294.94	100%
MQUACQ-2 TQ-GEN [1]	JSudoku	7 153	0.26	0.13	8.46	919.34	100%
	Murder	394	0.04	0.01	0.32	5.22	100%
	Random	5787	5.26	1.23	17.61	7100.44	100%
MQUACQ-2 PQ-GEN (OURS)	JSudoku	6 458	0.77	0.10	3.19	666.91	100%
	Murder	370	0.66	0.03	1.10	12.28	100%
	Random	5 708	0.60	0.04	2.22	233.62	100%
	Using custom solvers						
MQUACQ-2 GenerateQuery.cutoff [6]	JSudoku	5 321	0.99	0.06	2.04	336.42	100%
	Murder	421	0.76	0.13	1.02	53.21	100%
	Random	5 349	0.94	0.04	3.24	198.65	100%
MQUACQ-2 $\max_B$ [25]	JSudoku	5 042	1.00	0.06	2.34	277.36	100%
	Murder	325	0.86	0.04	1.01	12.90	100%
	Random	5 012	0.94	0.02	1.52	95.21	100%

[Q2] Using GrowAcq within MQuAcq-2. We now evaluate the performance of GROWACQ, our proposed bottom-up CA approach. To evaluate it, we used MQUACQ-2, as the inner CA algorithm within GROWACQ (line 7 of Algorithm 3) and compared this to using MQUACQ-2, directly on the full-sized problem. Table 3, top two blocks, presents the results.

We can observe that the usage of GROWACQ results in a reduction of the number of queries in JSudoku, Murder, and Random, while a slight increase can be seen in Golomb. In Job-shop, the increase in the number of queries is somewhat larger (25%). This is the case because the target constraint network in this benchmark is sparse, with most of the iterations of GROWACQ in a $Y \subset X$ not learning any constraint from C_T and only shrinking the bias. So, when the full-sized problem is looked at directly when MQUACQ-2 is used, the bias B can shrink with fewer queries. On the other hand, when the target network is not sparse, there is a decrease in the number of queries of up to 19%, due to the fact that the system can locate the scopes of the constraints faster, starting from a $Y \subset X$ every time. Based on the above observations, we can see that using GROWACQ leads to learning constraints in a lower amount of queries, but on the other hand, needs more queries to shrink the bias.

Finally, although the total time is almost the same in most problems, and slightly increased in JSudoku and Golomb, the average time per query has not noticeably increased, while the maximum time the user has to wait between two queries has decreased significantly (up to 88% in the Job-shop benchmark), due to the overall reduction in the time needed in query generation in almost all problems (as indicated by the $\bar{T}_{qgen}$ column). As the (maximum) waiting time for the user is of paramount importance for interactive settings, we can see that GROWACQ improves this aspect of time performance of interactive CA systems.

[Q3] Guided query generation. In order to evaluate the performance of our proposed objective function for guiding query generation, we compare it with the use of the most popular objective function used in state-of-the-art CA systems, i.e., maximizing violations of constraints from B. The objective functions are utilized in line 9 of Algorithm 2. For this comparison, GROWACQ is used, again with MQUACQ-2 as the inner acquisition algorithm at

■ **Table 3** Evaluation of GrowAcq and the proposed approach for guiding query generation.

Problem	#q	$\bar{T}_{gen}$	$\bar{T}$	T_{max}	T_{total}
MQuAcq-2					
JSudoku	6 458	0.77	0.10	3.19	666.91
Murder	370	0.66	0.03	1.10	12.28
Random	5 708	0.60	0.04	2.22	233.62
Golomb	233	0.96	0.22	1.22	50.89
Job-shop	590	1.03	0.10	5.36	56.23
GrowAcq + MQuAcq-2					
Sudoku	5 863	0.15	0.12	1.98	721.15
Murder	357	0.04	0.02	0.11	7.97
Random	4 804	0.14	0.05	1.30	230.36
Golomb	270	0.80	0.29	1.30	78.47
Job-shop	786	0.13	0.06	0.66	48.18
GrowAcq + MQuAcq-2 $_{guided}$					
JSudoku	3 963	0.15	0.24	1.96	963.42
Murder	250	0.04	0.04	0.27	9.07
Random	4 820	0.14	0.05	1.19	229.47
Golomb	100	0.16	0.27	0.95	27.44
Job-shop	776	0.13	0.06	0.64	47.53

line 7 of Algorithm 3. The results using the guided query generation can be seen in Table 3, bottom-two blocks, comparing GrowAcq + MQuAcq-2 against GrowAcq + MQuAcq-2 $_{guided}$.

We can see that, when using our probability-based guidance for query generation, the number of queries has significantly decreased in JSudoku, Murder, and Golomb, while it has remained nearly the same in Random and Job-Shop. In the latter cases, the number of queries has not decreased because these are under-constrained problems, and thus the probability derived from the constraints' relations was small. This led to maximizing the violations of all constraints in B (i.e., the same behavior as with the existing objective). On the other hand, in the problems that do not have a sparse constraint network, where using the simple counting method to compute the probabilities of the constraints could effectively guide the acquisition system, the decrease observed in the number of queries is substantial (32% in JSudoku, 30% in Murder, and 64% in Golomb). However, as violating constraints one-by-one leads to more queries *generated* at line 3 of Algorithm 1, yet fewer queries at lines 8-9, which are very fast to compute, there is a small increase in the total time on JSudoku.

[Q4] Combination of our methods. Comparing the combination of our methods (i.e., GrowAcq + MQuAcq-2 $_{guided}$) with MQuAcq-2 (Table 3), we can see that combining our bottom-up approach with guiding the query generation greatly outperforms MQuAcq-2 in terms of the number of queries needed to achieve convergence on most of the benchmarks. The number of queries has decreased on all benchmarks except Job-shop, where, because of its sparse target network, we need 23% more queries, as GrowAcq increases the number of queries to converge in underconstrained problems, due to the reasons described in section 6.4, while guiding the query generation does not improve it, as the probabilities estimated are always low. In the rest of the problems, we observe a total decrease of 16% in Random, 39% in JSudoku, 32% in Murder, and up to 60% in Golomb.

These results demonstrate the effectiveness of the proposed methods in reducing the number of queries needed for CA algorithms, which is crucial in interactive scenarios.

6.5   [Q5] Dealing with larger biases

To answer this question, we evaluated GrowAcq and the combination of our methods on larger instances of the Job-shop benchmark, using the same language as before. We used two instances: one with 15 jobs, 11 machines, and 40 steps (denoted as JS-15-11), which resulted in a bias consisting of 542 850 constraints, and one with 19 jobs, 12 machines, and again 40 steps (denoted as JS-19-12), resulting in a bias of 1 037 400 constraints. The results are presented in Table 4.

On the one hand, GrowAcq needs more queries to converge (like on the smaller Job-Shop instance) because the constraint network of this problem is sparse. Yet the total time needed to converge is one order of magnitude lower than in MQuAcq-2, being 24.4 times faster in the instance with a bias size of 0.5 million constraints and 25.6 times faster in the instance with $|B| > 1M$. In addition, the maximum waiting time has drastically decreased by using GrowAcq (and the combination GrowAcq and guiding query generation), from 5 499 seconds to only 3 (resp. 8) seconds in JS-15-11 and from more than 20 371 seconds to only 7 (resp. 6) seconds in JS-19-12. Importantly, the average waiting time is more than 30 times lower when using GrowAcq. Note that, as in the smaller job-shop instance, guiding does not lead to improvement in terms of the number of queries. However, it does not noticeably worsen the time performance of the system.

Hence, the experiments confirm that the proposed methodology can efficiently handle significantly larger sets of candidate constraints than the state of the art, up to 50 times larger than the ones commonly used in the literature [6, 8, 27, 29].

Table 4 Experimental results on instances with a large bias.

Problem	$|B|$	#q	$\bar{T}_{qgen}$	$\bar{T}$	T_{max}	T_{total}
MQuAcq-2						
JS-15-11	$\approx 0.5M$	5 456	66.12	6.25	5 499.76	34 085.73
JS-19-12	$\approx 1M$	8 012	80.99	9.75	20 371.74	78 124.41
GrowAcq + MQuAcq-2						
JS-15-11	$\approx 0.5M$	7 015	0.44	0.20	2.84	1 422.93
JS-19-12	$\approx 1M$	10 309	0.62	0.29	6.92	2 984.77
GrowAcq + MQuAcq-2$_{guided}$						
JS-15-11	$\approx 0.5M$	7 062	0.44	0.20	7.88	1 399.63
JS-19-12	$\approx 1M$	10 219	0.64	0.30	6.19	3 054.68

7   Conclusions

Some of the most important limitations of interactive CA methods are the large number of queries needed to converge, as well as the size of the candidate constraint set that they can handle efficiently. In this work, we presented novel methods to alleviate these issues, improving the efficiency of CA systems. We proposed a bottom-up approach, which allows the system to handle significantly larger biases, reducing the maximum waiting time for the user, and also reducing the total number of queries needed when the target constraint network is not sparse. We also introduced a probabilistic method to guide query generation,

further reducing the number of posted queries when our simple counting method could guide the acquisition system to learn constraints more efficiently. In addition, we presented a new query generation technique, named PQ-Gen, that allows the use of conventional CP solvers, removing the dependency of existing methods on customized solvers to converge. Our experimental evaluation showed that our proposed methods outperform state-of-the-art systems in terms of the number of queries in problems with non-sparse constraint networks, reducing this number up to 60%. In addition, the experiments show that GrowAcq can handle up to 50 times larger biases than the ones commonly used in the literature, allowing CA to tackle increasingly large and complex problems. The biggest avenue for future work is to further investigate additional ways to reduce the number of queries needed, e.g., by using guidance in all parts of the acquisition process (not just the query generation), and with more advanced probabilistic models. Another important avenue is to consider the setting in which user answers can be noisy as has been investigated for passive systems.

References

1 Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. Time-bounded query generator for constraint acquisition. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–17. Springer, 2018.

2 Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

3 Robin Arcangioli, Christian Bessiere, and Nadjib Lazaar. Multiple constraint aquisition. In *IJCAI: International Joint Conference on Artificial Intelligence*, pages 698–704, 2016.

4 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Principles and practice of constraint programming*, pages 141–157. Springer, 2012.

5 Senne Berden, Mohit Kumar, Samuel Kolb, and Tias Guns. Learning max-sat models from examples using genetic algorithms and knowledge compilation. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022.

6 Christian Bessiere, Clement Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C Tsouros, and Toby Walsh. Learning constraints through partial queries. *Artificial Intelligence*, 319:103896, 2023.

7 Christian Bessiere, Remi Coletta, Eugene C Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 123–137. Springer, 2004.

8 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Toby Walsh, et al. Constraint acquisition via partial queries. In *IJCAI*, volume 13, pages 475–481, 2013.

9 Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *European Conference on Machine Learning*, pages 23–34. Springer, 2005.

10 Christian Bessiere, Remi Coletta, Barry O'Sullivan, Mathias Paulin, et al. Query-driven constraint acquisition. In *IJCAI*, volume 7, pages 50–55, 2007.

11 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

12 Eugene C Freuder. Modeling: the final frontier. In *The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London*, pages 15–21, 1999.

13 Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

14 Eugene C Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014.

15   Eugene C Freuder and Richard J Wallace. Suggestion strategies for constraint-based match-maker agents. In *International Conference on Principles and Practice of Constraint Programming*, pages 192–204. Springer, 1998.

16   Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

17   Mohit Kumar et al. Acquiring integer programs from data. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. IJCAI, 2019.

18   Mohit Kumar, Samuel Kolb, and Tias Guns. Learning constraint programming models from data using generate-and-aggregate. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

19   Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 45–52. IEEE, 2010.

20   Nadjib Lazaar. Parallel constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3860–3867, 2021.

21   Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017.

22   Steven D Prestwich. Robust constraint acquisition by sequential analysis. *Frontiers in Artificial Intelligence and Applications*, 325:355–362, 2020.

23   Steven D Prestwich, Eugene C Freuder, Barry O'Sullivan, and David Browne. Classifier-based constraint acquisition. *Annals of Mathematics and Artificial Intelligence*, pages 1–20, 2021.

24   Kostyantyn Shchekotykhin and Gerhard Friedrich. Argumentation based constraint acquisition. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 476–482. IEEE, 2009.

25   Dimosthenis C Tsouros and Kostas Stergiou. Efficient multiple constraint acquisition. *Constraints*, 25(3):180–225, 2020.

26   Dimosthenis C Tsouros and Kostas Stergiou. Learning max-csps via active constraint acquisition. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

27   Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 709–725. Springer, 2019.

28   Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Omissions in constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 935–951. Springer, 2020.

29   Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *24th International Conference on Principles and Practice of Constraint Programming*, 2018.

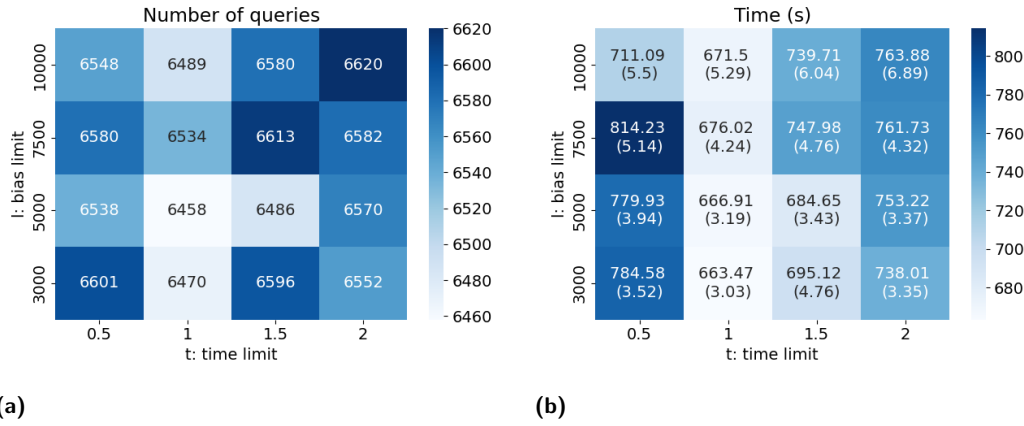## A   Hyperparameter evaluation for PQ-Gen and TQ-Gen

Figure 1 Performance of PQ-Gen with different hyperparameters values, in terms of: a) the number of queries posted and b) the time (s) needed (in brackets we show the maximum waiting time for the user).
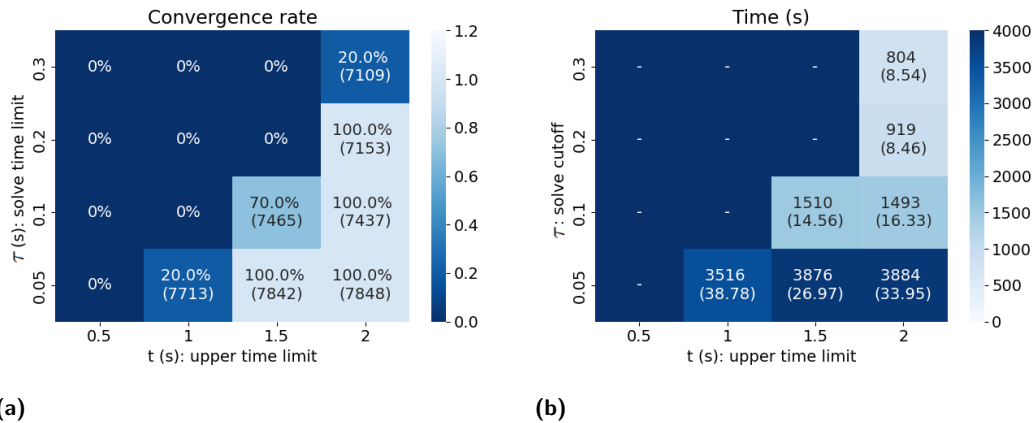

Figure 2 Performance of TQ-Gen with different parameters, in terms of: a) the convergence rate (in brackets we show the number of queries posted when it converged) and b) the time (s) needed (in brackets we show the maximum waiting time for the user).

Both PQ-Gen, our projection-based query generation approach, and TQ-Gen [1] (discussed in Section 3.1) involve hyperparameters that affect their performance. As mentioned in Section 6.3, we performed a sensitivity analysis of the performance with respect to the hyperparameter configuration used of PQ-Gen and TQ-Gen [1]. In this comparison, both query generation methods were used within the state-of-the-art active CA method MQuAcq-2. We used the JSudoku benchmark for this comparison, as from the benchmarks considered in this paper, this is shown to be the hardest one to reach convergence on (see Table 2).

In more detail, we varied the hyperparameters of both PQ-Gen and TQ-Gen to assess their performance under different configurations. While we fixed the hyperparameter α of [1] to 0.8 as recommended in [1], we had to try different values for the time-related hyperparameters, τ and t. We did not use the values proposed by the authors in [1] because we use a different solver and system, and this can affect significantly the time performance.

In our evaluation we used $\tau = [0.05s, 0.1s, 0.2s, 0.3s]$ and $t = [0.5s, 1s, 1.5s, 2s]$ for TQ-Gen. We also used the adjust function described in [1], as it has been shown to improve its performance. For PQ-Gen hyperparameters, we used $l = \{3000, 5000, 7500, 10000\}$ and $t = \{0.5, 1, 1.5, 2\}$. Thus, we examined 16 different configurations for each. The results of our experiments are presented in Figures 1 and 2, respectively, for PQ-Gen and TQ-Gen.

Focusing on Figure 1, we can see that the performance of PQ-Gen is stable across all configurations, both in terms of the number of queries and time performance, having also converged in all cases. Let us now shift our focus to Figure 2 and the performance of TQ-Gen. The first observation is that in the majority of the cases, MQuAcq-2 failed to converge when using TQ-Gen as the query generator. Only when the time limit was set to 2s, we see at least one run achieving convergence for all values of τ. In addition, the performance of MQuAcq-2 using TQ-Gen is highly sensitive to changes in hyperparameter values, particularly with respect to time.

Overall, comparing the results of PQ-Gen and TQ-Gen, we observe that PQ-Gen exhibits superior performance in terms of convergence rate, fully overcoming the issue of premature convergence. PQ-Gen also requires a lower number of queries to reach convergence and offers improved time performance, resulting in reduced waiting times for the user.