Machine Learning-Guided Interactive Constraint Acquisition*

DIMOS TSOUROS†, KU Leuven, Belgium and University of Western Macedonia, Greece
SENNE BERDEN, KU Leuven, Belgium
TIAS GUNS, KU Leuven, Belgium

Background: Constraint Programming (CP) has been successfully used to model and solve complex combinatorial problems. However, modeling is often not trivial and requires expertise, which is a bottleneck to wider adoption of CP. As a result, the field of Constraint Acquisition (CA) has emerged with the aim to (semi-)automate the modeling process. In CA, the goal is to assist the user by automatically *learning* the model. In (inter)active CA, this is done by interactively posting queries to the user, e.g., asking whether a partial solution satisfies their (unspecified) constraints or not.
Objectives: However, a large number of queries is required to learn the model, which is a major limitation of state-of-the-art CA, especially for human users. We believe this is due to the search-based learning of interactive CA, which is based on symbolic concept learning, being mostly *uninformed*. During learning, the system is not aware of any patterns that may appear in the constraints acquired so far, which could be used to guide the rest of the process.
Methods: In this paper, we aim to alleviate this limitation by, for the first time, utilizing statistical Machine Learning (ML) in the context of interactive CA. We propose to use probabilistic classification models to guide interactive CA to generate more informative queries. Specifically, we propose a probability-based objective function to use in the query generation process across all components of interactive CA: the top-level query generation, the scope finding, and the lowest-level constraint finding. To ease the guiding of the scope-finding process, we also propose a novel FindScope function, which includes an explicit query generation step. Effective guidance, however, relies on (probabilistic) estimates of whether candidate expressions belong to the problem or not. To this end, we propose a framework for the training of ML classifiers within the interactive CA process, as well as an expressive feature representation of constraints.
Results: We experimentally evaluate our proposed methods on 7 different problem classes, using different classifiers and feature representations, and show that our methods greatly outperform the state of the art, decreasing the number of queries needed to converge by up to 75%.
Conclusions: Our findings confirm that statistical ML methods can detect patterns in constraint models, while they are being learned, and can be successfully used within the search-based interactive CA for reducing the number of queries needed.

JAIR Track: Constraint Programming and Machine Learning

JAIR Associate Editor: Roni Stern

JAIR Reference Format:
Dimos Tsouros, Senne Berden, and Tias Guns. 2026. Machine Learning-Guided Interactive Constraint Acquisition. *Journal of Artificial Intelligence Research* 86, Article 3 (May 2026), 40 pages. DOI: 10.1613/jair.1.19524

*This paper is an extended version of parts of (Tsouros et al., 2023a) and (Tsouros et al., 2024)
†Corresponding Author.

Authors' Contact Information: Dimos Tsouros, ORCID: 0000-0002-3040-0959, dtsouros@uowm.gr, KU Leuven, Leuven, Belgium and University of Western Macedonia, Kozani, Greece; Senne Berden, ORCID: 0000-0002-6473-5757, senne.berden@kuleuven.be, KU Leuven, Leuven, Belgium; Tias Guns, ORCID: 0000-0002-2156-2155, tias.guns@kuleuven.be, KU Leuven, Leuven, Belgium.

DOI: 10.1613/jair.1.19524

1 Introduction

Constraint programming (CP) is considered one of the main paradigms for solving combinatorial problems in artificial intelligence. In CP, the user declaratively states the constraints over a set of decision variables, after which a solver is used to compute a feasible solution. Although CP has demonstrated success in solving combinatorial problems across various domains, there are still challenges that must be addressed for CP technology to achieve wider adoption. One of the most important challenges is to ease the process of formulating a precise constraint model: expressing a combinatorial problem as a set of constraints over decision variables is not straightforward and requires substantial expertise (Freuder, 1999). Because of this, having to manually model a problem is considered a major bottleneck for the widespread adoption of CP (Freuder and O'Sullivan, 2014; Freuder, 2018).

As a result, a considerable body of research focuses on making CP modeling more accessible, including the development of solver-agnostic modeling languages (Nethercote et al., 2007; Frisch et al., 2008; Guns, 2019) and LLM-based assistants that model a problem given its natural language description (Tsouros et al., 2023b; Michailidis et al., 2024; Kadıoğlu et al., 2024). In addition to these, in the direction of human-in-the-loop constraint modeling, the topic of *Constraint Acquisition (CA)* has received a lot of attention. CA aims to acquire the model of a problem from data. These data are most commonly examples of solutions and non-solutions to the problem. A recent application of CA to a large-scale real-world transmission maintenance scheduling problem, where active CA is used to approximate complex non-linear power-flow constraints, showed its significant potential (Barral et al., 2024).

State-of-the-art CA systems typically use the candidate elimination and version space paradigm (Mitchell, 1978, 1997): given a set of candidate constraints, either provided explicitly as a list of constraints or defined implicitly through a language of constraints, the CA system aims to recognize which candidate constraints are part of the problem. CA can be either *passive* or *interactive*. In *passive acquisition*, the data are provided by the user upfront; typically, a dataset of examples of solutions and non-solutions is given, and based on these examples, the system learns a set of constraints modeling the problem (Beldiceanu and Simonis, 2012; Berden et al., 2022; Bessiere et al., 2004, 2005, 2017; Kumar et al., 2019, 2022; Lallouet et al., 2010; Lombardi et al., 2017).

In contrast to passive learning, *active* or *interactive acquisition* systems learn the constraints through query-based interaction with the user. In the early days, most methods only made use of membership queries, asking the user whether a full assignment of values to the variables constitutes a solution or not (Angluin, 1988; Bessiere et al., 2007, 2017). However, the number of membership queries needed to identify the underlying model can be exponentially large for these methods (Bessiere et al., 2017). This led to a new family of interactive algorithms, using *partial queries* instead (Arcangioli et al., 2016; Bessiere et al., 2013; Lazaar, 2021; Tsouros and Stergiou, 2020, 2021; Tsouros et al., 2019, 2020, 2018; Bessiere et al., 2023b). Such (partial) queries ask the user to classify (partial) assignments to the variables as (partial) solutions or non-solutions, which significantly reduces the number of queries needed (Bessiere et al., 2023b).

Despite recent advancements in interactive CA, the number of queries that current systems need to learn the constraints is still prohibitively large for human users. Even for a small number of constraints, dozens of queries may be needed to learn them. This remains one of the most important limitations of the current state of the art. We hypothesize that this large number of queries stems from the fact that the system uses mostly *uninformed* search-based learning to acquire the constraints. That is, during the acquisition process, the system is not aware, when generating new queries, of any patterns that may exist in the constraints it has acquired and ruled out so far. If it were able to detect these patterns, it could use this information to guide the remainder of the acquisition process. That is, it could pose subsequent queries in an informed way, taking into account estimates of how likely each candidate constraint is to be part of the model. By using these estimates, it could ultimately generate queries in a way that minimizes the number of queries needed in total.

This is the aim of this work. We propose a way to guide the query generation of interactive CA based on probabilistic predictions estimating how likely candidate constraints are to belong to the model. We also show how a statistical ML component can be trained during the interactive CA process to produce these probabilities, and we provide an expressive feature representation of constraints for this ML component to learn over.

More concretely, our contributions are the following:

- We introduce a way to *guide* the top-level query generation used in interactive CA, with the goal of generating queries that learn the set of constraints faster (i.e., requiring fewer queries). Specifically, we propose an objective function for query generation that uses *probabilistic estimates* of whether constraints are likely to hold or not.
- We extend this to guide the other query-posing components of interactive CA, namely its FindScope and FindC methods. These methods are used to identify first the scope (FindScope) and then the relation (FindC) of a specific violated constraint, after the user has identified a top-level query as an infeasible solution. To ease the guiding of FindScope queries, we also introduce a novel variant of this function, called FindScope-3. In contrast to its predecessors, FindScope-3 has an explicitly stated query generation step, which can be guided through the introduction of an objective function. Remarkably, we experimentally find that the novel FindScope-3 even significantly outperforms the state-of-the-art scope-finding function FindScope-2 without any guiding.
- To obtain the probabilistic estimates of how likely constraints are to hold, needed for guiding, we propose to use probabilistic ML classifiers. We show how these classifiers can be trained within the interactive CA process, based on the constraints learned so far, as well as the ones removed from the candidate set of constraints at any point during the acquisition process. We also propose an expressive feature representation of constraints for these classifiers to learn over.
- Through an extensive experimental evaluation, we address several research questions and assess the performance of our proposed approach across 7 benchmark datasets. The results demonstrate that our approach greatly outperforms state-of-the-art constraint acquisition (CA) systems, reducing the number of queries required by up to 75%.

Publication history. This paper is an extension of parts of two published conference papers (Tsouros et al., 2023a, 2024). This paper presents a unified view on the prediction-guided approaches of those two papers and has the following novel components: we simplify the objective function used for guiding, we propose a new FindScope-3 function, we formalize the ML-based interactive CA framework, we propose a richer feature representation, and we perform an extensive evaluation on a larger number of benchmarks.

Structure. The remainder of the paper is organized as follows. Section 2 introduces the necessary background on CA. In Section 3, we present our proposed methods for guiding the query generation process, as well as the new FindScope-3 function. Section 4 describes how to train and apply probabilistic classifiers to estimate which constraints are likely part of the target problem. This section also introduces our proposed feature representation for constraints. Section 5 reports on an extensive experimental evaluation of our approach. Related work is discussed in Section 6. Finally, Section 7 concludes the paper and provides perspectives on future work.

2 Background

We now give the necessary background information for this paper

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) $C = \{c_1, c_2, ..., c_t\}$.

The goal of a CSP is to find an assignment in the set of variables X satisfying the constraints C. 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 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 constraint set C is considered *normalized* when no constraint in C has a scope that is a subset of or equal to the scope of another constraint in 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, it is not allowed by the constraint on $var(c)$. The constraint set $C[Y]$, where $Y \subseteq X$, denotes the set of constraints from C whose scope is a subset of Y. $\kappa_C(e_Y)$ represents the subset of constraints from $C[Y]$ that reject e_Y, i.e., $\kappa_C(e_Y) = \{c \mid c \in C[Y] \wedge e_{var(c)} \notin rel(c)\}$.

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 $C[Y]$.

A *constraint optimization problem* extends a CSP with an *objective function* $f : D^Y \rightarrow \mathbb{Z}$, with $Y \subseteq X$ that assigns a cost to each assignment. The goal is to find a solution $e \in sol(C)$ that minimizes (or maximizes) $f(e)$.

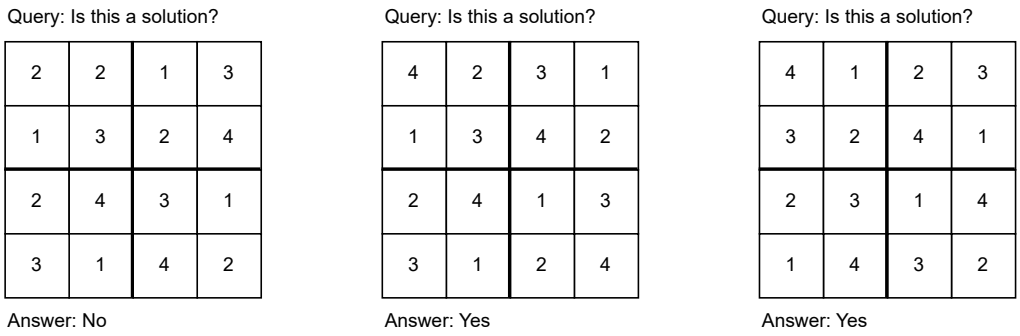
2.2 Interactive Constraint Acquisition

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, such as $=/2$ and $</2$ where the number after / represents the arity. From now on, when referring to binary constraints, we will skip the indication of the arity in the language. 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. For example, for $X = \{x_1, x_2, x_3\}$ and $\Gamma = \{=, <\}$, the bias becomes $B = \{x_1 = x_2,\ x_1 = x_3,\ x_2 = x_3,\ x_1 < x_2,\ \ldots\}$.

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 the unknown C_T. Like other works, we assume that C_T is representable by B, meaning that there exists a $C'_T \subseteq B$ that is equivalent to C_T, i.e., for which $sol(C'_T) = sol(C_T)$. The general interactive CA framework operates analogously to classical version space learning (Mitchell, 1978). That is, it maintains both a subset (C_L) and a superset (B) of C'_T, i.e., $C_L \subseteq C'_T \subseteq B$. The subset C_L is expanded as constraints are confirmed, while the superset B is shrunk as candidate constraints are ruled. At any point in time, C'_T is guaranteed to lie between these two sets. This continues until B contains only C_L and constraints that are implied by C_L, at which point C'_T is identified.

Example 2.1 (Running Example). Assume that we use interactive CA on a 4x4 Sudoku (a description of the Sudoku problem can be found in Section 5). The variables of this problem represent the value assignments to each cell of the Sudoku, i.e., $X = \{x_{11}, x_{12}, x_{13}, \ldots, x_{44}\}$, with x_{ij} representing cell on row i and column j. The target set of constraints C_T defines that all columns, rows and 2x2 blocks must contain only distinct values. A language that can be used in this problem, in order to create a bias B that can represent C_T, is the language $\Gamma = \{\neq\}$, or any language $\Gamma' \supset \Gamma$. For example, using $\Gamma = \{=, \neq, <, >\}$, the bias B will contain one constraint for each of these relations on each pair of variables (as they are binary relations).

In interactive CA, the system interacts with the user while learning the constraints. A *membership query* (Angluin, 1988) 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. The possible answers are *"yes"* and *"no"*. A *partial query* $ASK(e_Y)$, with



(a) Redundant query (b) Irredundant query (c) Redundant query

Fig. 1. Examples of redundant and irredundant membership queries

$Y \subset X$, asks the user to determine if e_Y, which is an assignment in D^Y, is a partial solution, i.e., if $e_Y \in sol(C_T[Y])$. While in passive acquisition there are methods that can handle noisy answers (Prestwich, 2020; Prestwich et al., 2021), this is not the case for interactive acquisition. For this reason, in this work, we follow the assumption that the user answers all queries correctly. Thus, we use the notation $c \in C_T$ iff $\forall\, e \in D^Y$ with $var(c) \subseteq Y \subseteq X$, $ASK(e_Y) =$ "yes" $\implies e_{var(c)} \in sol(c)$.

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 on the subproblem $P_Y = (Y, D[Y], C[Y])$ iff e_Y is rejected by at least one constraint from the bias $B[Y]$ and is a partial solution to the set C_L learned thus far. The first condition captures that $\kappa_B(e_Y)$ cannot be empty, since if $\kappa_B(e_Y)$ were 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 be a solution to $C_L[Y]$, since otherwise the user would certainly answer "no" to the query. Note that a partial query on an example e_Y may be irredundant on Y, but redundant on X. This can happen when the constraints in $\kappa_B(e_Y)$ are implied by C_L and thus cannot be violated by a complete assignment. However, explicitly checking this type of indirect implications can be highly computationally expensive (Tsouros et al., 2023a), and thus we define redundancy of a query based on the variables $Y \subseteq X$ present in the example posted to the user.

Example 2.2 (Redundant Queries). Consider our running example (from Example 2.1). Assume that the CA system has learned all $\neq$ constraints of the 4x4 Sudoku, i.e., $C_L = C_T$, and the remaining candidates are $B = \{x_{11} > x_{44}, x_{12} < x_{22}\}$. Figure 1 shows examples of redundant and irredundant queries at this stage of the interactive CA process. The query shown in Figure 1a is redundant, because it violates a constraint from C_L, and thus the system can already derive that the answer will be negative. The query of Figure 1b is irredundant, as it satisfies C_L, and at the same time it violates the constraint $x_{11} > x_{44}$ from B. Finally, the query shown in Figure 1c is redundant because, although it satisfies all constraints from C_L, it also satisfies both constraints from B.

Algorithm 1 presents the generic process followed in interactive CA through 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 first the system generates an irredundant example (line 3) and posts it as a query to the user (line 5). If the query is classified as positive, then the candidate expressions from B that violate it are removed (line 6). If the example is classified as negative, then the system tries to find one (or more) constraint(s) from C_T that violate it. This is done in two steps. First,

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 : the learned set of constraints

1: $C_L \leftarrow C_{in}$
2: **while** True **do**
3: $e, Y \leftarrow$ QGen(C_L, B)
4: **if** e = nil **then return** C_L ▷ converged
5: **if** $ASK(e_Y)$ = "yes" **then**
6: $B \leftarrow B \setminus \kappa_B(e_Y)$
7: **else**
8: $(B, S) \leftarrow$ FindScope (e, $\emptyset$, Y, B)
9: $(B, C_L) \leftarrow$ FindC (e, S, C_L, B)

the scope of one or more violated constraints is found (the variables that are involved in the constraint) by asking queries and possibly shrinking the bias along the way (line 8). Then, the relations of the constraints with this scope are found, again by asking queries and possibly shrinking the bias (line 9). This process continues until the system converges (line 4).

Convergence. The acquisition process has *converged* on the learned network $C_L \subseteq B$ iff C_L agrees with the set of all partial queries 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 example (i.e., query) could be generated at line 3, as in this case, all constraints in B are redundant. If the first condition (i.e., C_L agrees with E) is true but the second condition ($\exists\, C \subseteq B \mid e \in sol(C)\ \forall\, e \in E \wedge sol(C) \neq sol(C_L)$) 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 (Bessiere et al., 2023b, 2013), MQuAcq (Tsouros and Stergiou, 2020; Tsouros et al., 2018) and MQuAcq-2 (Tsouros et al., 2019) 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. Notice that interactive CA systems consist of three components where (increasingly simpler) queries are asked to the user: (1) Top-level query generation (line 3), (2) Finding the scope(s) of violated constraints (line 8), (3) Finding the relations of constraints in the scope found (line 9). We now describe each of them.

2.2.1 Top-level Query Generation (QGen). Query generation (line 3 of Algorithm 1) aims to find an *irredudant* membership query, i.e., a (possibly partial) assignment that does not violate any constraint from C_L but violates at least one $c \in B$, that will be asked to the user. Thus, it can be formalized as follows:

$$\text{find } e_Y \text{ s.t. } Y \subseteq X \wedge e_Y \in sol(C_L[Y]) \wedge \kappa_B(e_Y) \neq \emptyset, \tag{1}$$

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

Often, this problem is converted to a constraint optimization problem, adding an objective function to find *better* queries. The objective function used in existing query generation systems (Bessiere et al., 2023b; Tsouros and Stergiou, 2020) tries to maximize the number of constraints from B that are violated by the generated query e_Y. The motivation is that this can potentially help shrink the bias faster. The objective function to be maximized is

Algorithm 2 FindScope

Input: e, R, Y, B (e: the example, R, Y: sets of variables, B: the bias)
Output: B, $Scope$ (B: the updated bias, $Scopes$: a set of variables, the scope of a constraint in C_T

1: **function** FindScope(e, R, Y, B)
2: **if** $\kappa_B(e_R) \neq \emptyset$ **then**
3: **if** $ASK(e_R)$ = "yes" **then** $B \leftarrow B \setminus \kappa_B(e_R)$;
4: **else return** $(B, \emptyset)$;
5: **if** $|Y| = 1$ **then return** (B, Y);
6: split Y into $< Y_1, Y_2 >$ such that $|Y_1| = \lceil |Y|/2 \rceil$;
7: **if** $\kappa_B(e_{R \cup Y_1}) = \kappa_B(e_{R \cup Y})$ **then** $S1 \leftarrow \emptyset$
8: **else** $(B, S_1) \leftarrow FindScope(e, R \cup Y_1, Y_2, B)$;
9: **if** $\kappa_B(e_{R \cup S_1}) = \kappa_B(e_{R \cup Y})$ **then** $S2 \leftarrow \emptyset$
10: **else** $(B, S_2) \leftarrow FindScope(e, R \cup S_1, Y_1, B)$;
11: **return** $(B, S_1 \cup S_2)$;

$$f(e_Y) = |\kappa_B(e_Y)| \tag{2}$$

2.2.2 Finding the Scope of Violated Constraints (FindScope). The functions used in the literature (Bessiere et al., 2013; Tsouros and Stergiou, 2020; Bessiere et al., 2023b) to find the scope of violated constraints after a negative answer from the user (line 8 of Algorithm 1) work in a similar way. We outline the general FindScope approach in Algorithm 2, which is the FindScope method from (Bessiere et al., 2023b), and works similar to FindScope-2 (Tsouros and Stergiou, 2020).

FindScope methods recursively map the problem of finding a constraint to a simpler problem by removing blocks of variable assignments from the original query (the one asked in line 3 of Algorithm 1, to which the user answered "no") and asking partial queries to the user. The removed block must contain at least one variable while not including all the present variables, in order to lead to an irredundant query. If after the removal of some variables, the answer of the user changes to "yes", then the removed block contains at least one variable from the scope of a violated constraint. When this happens, FindScope focuses on refining this block, adding some variable assignments back to the query. When, after repeating this procedure, the size of the considered block becomes 1 (i.e., the block contains a single variable), this variable is found to be in the scope of a violated constraint we seek (line 5 of Algorithm 2).

Example 2.3 (FindScope). Figure 2 illustrates an example execution of FindScope on our running 4×4 Sudoku example. The first query is the top-level query, to which the user answers "no" because the assignment violates some same-block constraints. The execution of FindScope then proceeds on this negative example by recursively removing variable assignments in each query, thereby reducing the problem of identifying a violated constraint to simpler subproblems. Once the user provides a positive answer, the system needs to determine which of the previously removed variables (the highlighted part of the fourth query, i.e., $x_{2,1}$ or $x_{2,2}$) belongs to the scope of the violated constraint. That is, it should determine which removed variable assignment led to the user's response changing from negative to positive. To do this, the assignment to variable $x_{2,1}$ is added back. Since the user's answer remains positive in the next query, and the remaining block of variables only contains one variable now (variable $x_{2,2}$), FindScope has identified that this variable must belong to the scope of the constraint. It then continues in the same manner to find the rest of the scope.

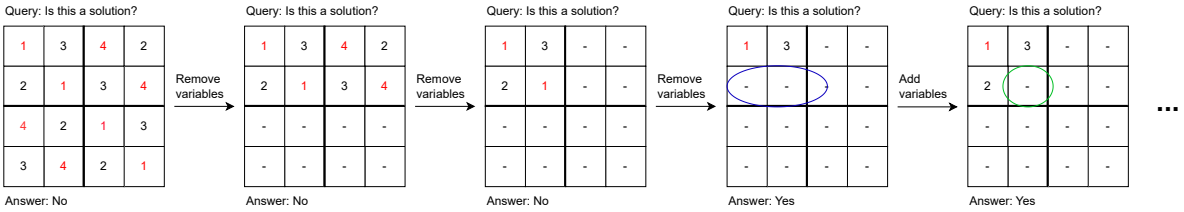


Fig. 2. FindScope execution from Example 2.3

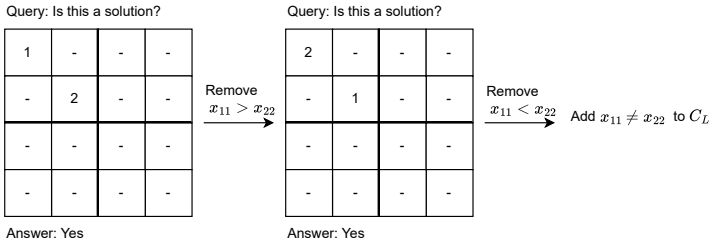


Fig. 3. FindC execution from Example 2.4

The main differences among existing FindScope functions lie in how they prevent redundant queries from being presented to the user. The first FindScope function (Bessiere et al., 2013) did not include the if statements of lines 2, 7, and 9, i.e., it did not check whether a query or a recursive call is redundant. FindScope-2 (Tsouros and Stergiou, 2020) and the FindScope function from (Bessiere et al., 2023b) avoid such redundant queries and recursive calls by adding different kinds of checks that lead to the same result.

2.2.3 Finding the Relation of Violated Constraints (FindC). After the system has located the scope of a violated constraint, it calls function *FindC* to find the violated constraint (line 9 of Algorithm 1). *FindC* asks partial queries focused on the scope returned by *FindScope*. It aims to find which constraint(s) from $B[S]$ are part of C_T, with S being the scope returned by FindScope. We show a general version of FindC in Algorithm 3. It uses the set Δ that consists of the candidate constraints on that scope (initialized at line 2). Then, the main loop starts, where examples are generated and posted to the user, violating some constraints from Δ, but not all (line 4). If no example is generated, then Δ consists of equivalent constraints, which are added to C_L (line 6), after which the updated B and C_L are returned (line 7). If an example is generated, Δ and B are updated based on the user's answer (lines 8-11).

Example 2.4 (FindC). Figure 3 illustrates an example execution of FindC on our running 4×4 Sudoku example. After the scope $\{x_{11}, x_{22}\}$ has been found through the execution of FindScope (Example 2.3), the CA system needs to identify the constraint from C_T in this scope. Assume that the remaining candidate constraints in B for this scope are: $\{x_{11} \neq x_{22}, x_{11} > x_{22}, x_{11} < x_{22}\}$. FindC will try different assignments on the variables of the scope found to identify which of these constraints should be removed and which should be added to C_L. After a positive answer to the first query, the constraint $x_{11} > x_{22}$ can be removed from B. After a positive answer to the second query, the constraint $x_{11} > x_{22}$ can also be removed. As a result, the only remaining candidate is $x_{11} \neq x_{22}$, which is then added to C_L.

Algorithm 3 FindC

Input: e, Y, C_L, B (e: the example, Y: The scope to search, C_L: the learned set of constraints, B: the bias)
Output: c : a constraint in C_T

```
1:  function FindC(e, Y)
2:      initialize Δ based on B;
3:      while true do
4:          Generate e' in D^Y accepted by C_L and rejected by Δ, with κ_Δ(e_Y) ≠ |Δ|;
5:          if e' = nil then
6:              C_L ← C_L ∪ Δ
7:              return (B, C_L) ;
8:          if ASK(e') = "yes" then
9:              B ← B \ κ_B(e');
10:             Δ ← Δ \ κ_Δ(e');
11:         else  Update Δ;
```

Different versions of FindC mainly differ in the assumptions they make about the structure of the target set of constraints. The original FindC function (Bessiere et al., 2013) assumes that the target constraint set is *normalized*: (i) there is at most one constraint per scope, and (ii) if a constraint exists on a scope S, then no constraint $c \in C_T$ has a scope $var(c) \supset S$. The FindC variant of (Bessiere et al., 2016) relaxes the normalization assumption by dropping the second condition: while still assuming at most one constraint per scope, it allows constraints on subscopes to exist. The most recent version (Bessiere et al., 2023b) removes this assumption entirely, allowing conjunctions of constraints within the same scope.

For simplicity, we present the version from (Bessiere et al., 2013). However, the proposed methodology is not tied to this choice and can be applied to all existing FindC variants.

2.2.4 GrowAcq. Particularly relevant to this work is the recently proposed *meta*-algorithm GrowAcq, which can handle large biases and reduces the number of queries needed to reach convergence (Tsouros et al., 2023a). GrowAcq is of particular importance here because it allows the acquisition process to only consider a (small) part of the bias at once. This makes it possible to efficiently optimize an objective function in query generation (as discussed in Section 2.2.1), which is what enables the ML-based guiding we propose in this work.

Concretely, GrowAcq iteratively calls an active CA algorithm (which follows the generic process of Algorithm 1) on an increasingly larger subset of the variables $Y \subseteq X$. This is shown in Algorithm 4. GrowAcq begins with $Y = \emptyset$ (line 2) and gradually incorporates more variables (lines 3-5). Once a new variable $x_i \in X$ is added to Y, the updated problem is to find $C_T[Y]$ e.g. all constraints over the subset of variables Y. However, since $C_T[Y \setminus \{x_i\}]$ was already learned in the previous iterations, the set of constraints to identify is actually

$$C_T[Y] \setminus C_T[Y \setminus \{x_i\}]$$

Thus, only the relevant part of the bias B is needed, namely $B[Y] \setminus B[Y \setminus \{x_i\}]$, and as shown in (Tsouros et al., 2023a), the bias constructed at line 6 is equivalent to $B[Y] \setminus B[Y \setminus \{x_i\}]$. 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), which can be replaced by any state-of-the-art algorithm, like QuAcq, MQuAcq or MQuAcq-2.

2.3 Machine Learning Classification

ML classification is a supervised learning task that involves learning a function over a given dataset. The dataset, denoted as $\mathbf{E}$, is a collection of N training examples, $\mathbf{E} = \{(\mathbf{f}_1, y_1), (\mathbf{f}_2, y_2), ..., (\mathbf{f}_N, y_N)\}$. Each example is a pair

Algorithm 4 Growing Acquisition (GrowAcq)

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 ← ∅
2: Y ← ∅
3: while |Y| ≤ |X| do
4:     x ← pick x ∈ (X \ Y)
5:     Y ← Y ∪ {x}
6:     B ← {c | rel(c) ∈ Γ ∧ var(c) ⊆ Y ∧ x ∈ var(c)}
7:     C_L ← Acq(Y, D^Y, B, C_L ∪ C_in[Y])
8: return C_L
```

$(\mathbf{f}_i, y_i)$, where $\mathbf{f}_i$ is a feature vector from the input space $\mathcal{F}$ and y_i is the corresponding class label from the output space Y. The feature vector $\mathbf{f}_i$ is composed of m features, $\mathbf{f}_i = (\phi_{i1}, \phi_{i2}, \ldots, \phi_{im})$, with each feature ϕ_{ij} being a (quantifiable) property of example i. In classification, Y is a set of possible class labels. An ML classifier aims to learn a function $f_\theta : \mathcal{F} \rightarrow Y$, using a set of learnable parameters θ. These parameters are adjusted during training to minimize a loss function $L(f_\theta(f), y)$ measuring the error between the predicted and actual class labels.

3 Guiding Interactive CA

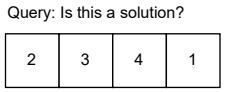
In this section, we show how we can significantly improve the performance of CA systems by guiding the query-based learning mechanism using information from the constraints that have already been learned and the constraints that have been ruled out thus far. First, in Section 3.1, we discuss what constitutes a good query and how this insight can be used to guide top-level query generation (line 3 of Algorithm 1). Since CA systems ASK queries not only in the top-level algorithm on line 5, but also within the FindScope (line 8) and FindC (line 9) components, we also show how the same logic can be used to guide query generation in these components in Sections 3.2 and 3.3, respectively.

3.1 Guiding Query Generation

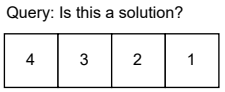
Consider top-level query generation (in line 3 of Algorithm 1). Previous query generation systems use an objective function that tries to maximize the number of constraints from B that are violated by the generated query e (Equation (2)) (Bessiere et al., 2023b; Tsouros and Stergiou, 2020). The motivation is that this can potentially help shrink the bias faster. However, we claim that this measure does not fully capture what constitutes a good query, because the information we get from the query depends on the answer of the user, as follows:

- On the one hand, queries that are answered **positively** lead to the removal of candidate constraints from the bias B. In this case, it is indeed beneficial to violate as many constraints from B as possible, as this will shrink B faster.
- On the other hand, queries that are answered **negatively** lead to the learning of a violated constraint from B. To do this, the system will ask subsequent partial queries through FindScope and FindC, to identify which of the violated constraints in B it should learn. Hence, in this case, it is beneficial to violate as few constraints from B as possible, so that the true conflicting constraint can be found in fewer subsequent queries.

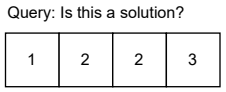
Example 3.1 (Different goals for positive and negative queries). Consider a simple problem, with four variables in one row, where the target set of constraints contains all $\neq$ constraints between all pairs of variables, i.e., all



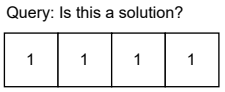
(a) Positive query removing few candidates



(b) Positive query removing many candidates



(c) Negative query directly learning a constraint



(d) Negative query needing more queries to learn a constraint

Fig. 4. "Good" and "bad" queries from Example 3.1

variables need to take a different value. Assume that we use interactive CA to learn the constraints of this problem, with a bias $B = \{\, x_i \neq x_j,\ x_i \leq x_j \mid 1 \leq i < j \leq 4 \,\}$. Figure 4 illustrates example queries that the CA system may post to the user.

We first consider the two *positive* queries (i.e., queries the user answered "yes" to). The first positive example violates only three candidate constraints, namely $\kappa_B(e) = \{x_1 \leq x_4, x_2 \leq x_4, x_3 \leq x_4\}$, which will therefore be removed from B. In contrast, the second positive example violates a substantially larger set of candidates, as $\kappa_B(e) = \{x_1 \leq x_2, x_1 \leq x_3, x_1 \leq x_4, x_2 \leq x_3, x_2 \leq x_4, x_3 \leq x_4\}$). This second query is preferable, as it eliminates more candidates from the bias in a single interaction. Asking the first query would require additional subsequent queries to remove the remaining candidates.

We now turn to the two *negative* queries (i.e., queries the user answered "no" to). The first negative example violates exactly one candidate constraint, $\kappa_B(e) = \{x_2 \neq x_3\}$, whereas the second violates many candidates, namely $\kappa_B(e) = \{x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4\}$). Here, the situation is reversed. A negative query that violates many candidates only indicates that *at least one* of them must belong to the target network, and thus additional queries are required to identify its scope. In contrast, the negative query violating a single candidate constraint allows us to directly add this constraint to the learned network; in this case, neither FindScope nor FindC needs to ask further queries before terminating.

Based on this intuition, we can draw the conclusion that good query generation must balance two opposing goals; maximizing violations of constraints not in C_T, to enable fast bias reduction after positive answers, while minimizing violations of *true* constraints to reduce the cost of finding the exact conflict after negative answers. Our proposed strategy, which we explain below, makes this trade-off explicit by prioritizing queries that are likely to be answered positively first, and only later focusing on queries that isolate individual true constraints.

To generate queries that are good based on this refined perspective, we propose the following strategy. We want query generation to *first* elicit positive answers from the user, by avoiding the violation of constraints that are in the (unknown) target set C_T. At the same time, we want to maximize the violation of constraints in B that are *not* in C_T, so that the positive answer leads to a large removal of constraints from the bias (in line with the first bullet point above). After some iterations of asking such queries, $B \setminus C_T$ will become empty (i.e., only true

constraints, that need to be confirmed, will remain in the bias). Note that, from this point onwards, the user will answer any remaining queries negatively, since any query generated has to violate at least one constraint from B (Equation (1)), and $B \setminus C_T = \emptyset$. At this point, we want the generated query to violate as few constraints from B as possible (in line with the second bullet point above).

We adopt a positive-first, negative-later approach because it is easier to violate candidates outside the target set while satisfying the true constraints, than to satisfy all but one candidates; especially since the latter would require satisfying every constraint not in C_T.

To implement this strategy, we first propose an idealized objective function that assumes access to a hypothetical oracle O that is able to tell whether a constraint c belongs to the unknown target set or not: $O(c) = [c \in C_T]$ (where $[\cdot]$ is the Iverson bracket which converts *True*/*False* into 1/0). Of course, we do not have access to such an oracle in practice, as this would make interactive CA unnecessary, since one would be able to directly extract C_T without requiring any queries. Therefore, after presenting the objective function, we will propose ways to approximate the oracle through machine learning. The oracle will predict which constraints are in C_T, assisting interactive CA in query generation. Interactive CA will then either confirm the predictions of the oracle, or refine them through the new information obtained in each iteration.

Our proposed objective function (to be maximized), using the oracle, is the following:

$$f(e) = \sum_{c \in B} [c \in \kappa_B(e))] \cdot \big(1 - k \cdot O(c)\big), \tag{3}$$

This objective function works as follows. On the one hand, it rewards (increases the objective value by 1) an example e for every violated constraint that the oracle says is not part of C_T (in anticipation of a positive answer). On the other hand, it penalizes (decreases the objective value by $k - 1$) the example for every violated constraints that the oracle indicates *belongs* to C_T, as this would cause a negative answer, which we aim to avoid.

The penalty size k should be chosen large enough for the objective function to prioritize satisfying constraints for which $O(c) = 1$ over violating constraints for which $O(c) = 0$. That is, adding constraints that are likely to lead to a negative answer (and hence will lead to follow-up queries to find the violated constraint) should come at a higher cost. However, we empirically find that setting k too large does not perform well either. We find that $k = |\Gamma|$ (i.e., the number of unique relations considered) works well in practice, and thus use this value in our experiments. By setting the penalty size to $|\Gamma|$, which is equal to the upper bound of the number of constraints in each scope, we ensure that the objective function prioritizes a query satisfying a constraint with $O(c_j) = \mathit{True}$, over a query that violates all constraints from B that share c_j's scope.

Example 3.2 (Guiding query generation). Consider the same problem as in Example 3.1. The objective function that is commonly used in the interactive CA literature is to maximize the number of violated candidates from B. However, this objective rewards equally the queries shown in Figures 4b and 4d, as they both violate the maximum number of constraints that can be violated under this bias. As discussed in the previous example, this is undesirable. The positive query in Figure 4b is indeed more informative, as it shrinks B faster, whereas the negative query in Figure 4d is not considered good, since alternative queries can lead to learning a constraint more efficiently. Conversely, the query shown in Figure 4c, which is a better negative query, receives a worse objective value under this criterion.

Assume now that we use our proposed objective function to guide query generation, exploiting a perfect oracle O. In this case, the query from Figure 4b is correctly preferred to be posed first, as it achieves an objective value of 6, corresponding to the violation of six candidates predicted not to belong to C_T. After this query, all $\leq$ constraints are removed from B, leaving only the $\neq$ constraints from C_T in it. During subsequent query generation steps, our objective function penalizes violations of these remaining constraints. As a result, the generated queries violate only a single candidate, leading to an objective value of -2, such as the query shown in Figure 4c. Queries that

violate additional constraints from C_T are penalized more heavily and therefore receive a lower objective value; for example, the query in Figure 4d has an objective value of −12.

3.1.1 Approximating the Oracle. The objective function given in Equation (3) assumes access to an oracle $O(c)$, which, of course, is not actually available during acquisition because we do not know C_T yet. We propose to approximate this oracle using a machine learning classifier, trained based on previously asked queries and their answers. A probabilistic classifier will predict for each $c \in B$ how likely it is that c is a true constraint (i.e., $P(c \in C_T)$). This classifier is trained on the constraints learned and the constraints ruled out thus far, which we will detail in Section 4. For now, we simply assume that the classifier is available.

We can then approximate the oracle $O(c)$ in two ways:

(1) Using the predicted class: $O_{class}(c) = [P(c \in C_T) \geq 0.5]$. Note that also non-probabilistic classifiers can be used here.
(2) Using the predicted probability itself $O_{prob}(c) = P(c \in C_T)$

We find that option (2) works best in practice, which we will show in the experimental evaluation. We also note that in the conference paper (Tsouros et al., 2023a), we proposed a more intricate binary oracle (specifically, $O_{orig}(c) = [1/P(c \in C_T) \leq \log(|Y|)]$). We later found that option (2) above works equally well, and thus prefer it here for its simplicity. The comparison to the original approximate oracle can be found in the supplementary material.

3.2 Guiding FindScope

Although guiding top-level query generation can result in a significant reduction in the number of queries, guiding this level of the CA system may not always be possible. This can be the case when the problem is hard to solve, and thus finding even an informative query can take a lot of time. However, guiding the queries asked in FindScope can offer important benefits too, while it is generally computationally cheaper because the goal is not to find a solution to the problem but to find subsets of variable assignments given an existing example. We now discuss how to guide this component of interactive CA systems.

As discussed in Section 2, the existing FindScope (Algorithm 2) methods recursively map the problem of finding a constraint to a simpler problem by removing blocks of variable assignments from the given negative example, and asking partial subqueries to the user. In practice, the main problem that must be dealt with in each step is to find a set of variables $Y' \subset Y$ for the next subquery. Depending on the answer of the user we can then update B and decide which part of the problem to focus on next. Existing FindScope functions naively choose $Y' \subset Y$, by splitting the set Y in half. The advantage of this approach is that a logarithmic number of steps can be achieved. However, no information about the violated constraints from B is used, and no guidance is utilized. The set of variables to remove, or keep, in the assignment is usually chosen randomly (Bessiere et al., 2023b) or lexicographically (Tsouros and Stergiou, 2020).

In this paper, we propose a simpler and non-recursive function to find the scope of violated constraints, which uses an explicit query generation step that can then be guided. Our novel FindScope-3 function aims to find the full scope of a violated constraint, instead of trying to find each variable at a time, while also exploiting information about the violated constraints from B when generating a query. We first discuss our new FindScope function, we then focus on its query generation problem, and we finally show how to integrate the ML-based guiding of Equation (3).

3.2.1 FindScope-3: A Constraint-Based FindScope. We now propose FindScope-3, a FindScope function with an explicit query generation step. The query in this case is a projection of the original query e (over the variables Y) to $e_{Y'}$ for any $Y' \subset Y$ subset of variables. Recall that $\kappa_B(e_Y)$ represents the subset of constraints from $B[Y]$ that are violated by e_Y.

Algorithm 5 FindScope-3: our proposed constraint-based FindScope

Input: e, Y, B (e: the example, Y: the set of variables, B: the bias)
Output: *Scope*
1: **while** *True* **do**
2: find Y' s.t. $Y' \subset Y \wedge (\emptyset \subset \kappa_B(e_{Y'}) \subset \kappa_B(e_Y))$
3: **if** $Y' = \emptyset$ **then return** $(B, var(\kappa_B(e_Y)))$
4: **if** $ASK(e_{Y'}) = True$ **then**
5: $B \leftarrow B \setminus \kappa_B(e_{Y'})$
6: **else**
7: $Y \leftarrow Y'$

Generating a query hence corresponds to finding a subset of variables $Y' \subset Y$. This query should be informative, i.e., it should violate at least one constraint of the bias ($\emptyset \subset \kappa_B(e_{Y'})$), and it should violate strictly less constraints than the original query ($\kappa_B(e_{Y'}) \subset \kappa_B(e_Y)$). The query generation hence corresponds to the following satisfaction problem:

$$\text{find } Y' \text{ s.t. } Y' \subset Y \wedge (\emptyset \subset \kappa_B(e_{Y'}) \subset \kappa_B(e_Y)). \tag{4}$$

Making this query generation procedure explicit leads to a much simpler process, which we call FindScope-3. It is shown in Algorithm 5 and takes as input the example e in which we want to find the scope of a violated constraint, given the negatively answered query e, the set of assigned variables Y, and the bias B. It returns the updated B and the scope of a violated constraint. In each iteration, a subset of variables Y' is found based on Equation (4), and then the example $e_{Y'}$ is posted as a query to the user (line 4). If the answer is positive, then the candidate constraints violated by $e_{Y'}$ are removed from B (lines 4-5) and the query generation is repeated. Note that we will never select the same Y' as all its constraints are removed from B, making it uninformative. In case the answer is negative, then we know that there is still at least one violated constraint in $B[Y']$ and hence we can continue focusing just on this $B[Y']$, replacing Y by Y' (lines 6-7). In each iteration, the set of candidates $B[Y]$ considered is reduced: either constraints are removed from B due to a positive answer, or the set of variables Y is reduced. FindScope-3 terminates when all the violated candidates in $B[Y]$ have the same scope, and hence no Y' can be found. This scope is then returned, along with the updated bias.

3.2.2 Query Generation in FindScope-3. To be able to automatically solve the query generation problem of FindScope-3 (Equation (4)), we model it as a CSP, as follows.

First, we introduce a set of boolean variables $BV = \{bv_i \mid i = 1, \ldots, |Y|\}$, where $|BV| = |Y|$. Each bv_i indicates whether variable $x_i \in Y$ is included in the candidate subset $Y' \subset Y \subseteq X$. Thus, $bv_i = 1$ if $x_i \in Y'$, and $bv_i = 0$ otherwise. As a result, the set Y' will be

$$Y' = \{x_i \in Y \mid bv_i = 1\} \tag{5}$$

Then, we need to enforce the constraint $\emptyset \subset \kappa_B(e_{Y'}) \subset \kappa_B(e_Y)$. As we know that $Y' \subset Y$, we can simplify this to

$$0 < |\kappa_B(e_{Y'})| < |\kappa_B(e_Y)|. \tag{6}$$

With $|\kappa_B(e_Y)|$ being a constant value, we only need to represent $\kappa_B(e_{Y'})$. To do this, we introduce a second set of boolean variables $BVC = \{bvc_j \mid j = 1, \ldots, |\kappa_B(e_Y)|\}$, one for each constraint $c_j \in \kappa_B(e_Y)$. Each bvc_j indicates whether c_j is present in $\kappa_B(e_{Y'})$. This occurs when all variables in the scope of c_j are included in Y' (since $c_j \in \kappa_B(e_Y)$). We capture this with the constraint:

$$bvc_j \Leftrightarrow \bigwedge_{v \in \text{var}(c_j)} bv_v \tag{7}$$

where bv_v is the boolean variable in BV corresponding to variable $v \in Y$. Using the boolean variables in BVC, we enforce the cardinality condition $0 < \sum_j bvc_j < |\kappa_B(e_Y)|$, which corresponds to the constraint from Equation (6).

The above CSP formulation captures the conditions of the query generation problem in FindScope-3 (Equation (4), in line 2 of Algorithm 5). However, just choosing any (arbitrarily sized) subset of Y can result in a large number of queries. But now that we have modelled this problem, we can add additional constraints and/or add an objective function in order to improve its performance. For example, if one wants to simulate the "splitting in half" strategy of FindScope, we propose to add the following objective function (to be minimized) to the CSP:

$$f(e_{Y'}) = \left| |Y'| - \left\lfloor \frac{|Y|}{2} \right\rfloor \right| \tag{8}$$

where

$$|Y'| = \sum_{bv_i \in BV} [bv_i] \tag{9}$$

However, the largest performance gains are obtained by incorporating an ML-guided objective function, which we discuss next.

3.2.3 Guiding FindScope-3 Queries. To guide FindScope-3 queries, we propose to use the above CSP formulation of the query generation problem and integrate the objective function from Equation (3). The main change to be considered in the guiding objective from Equation (3) comes from the fact that we now are not generating an assignment e, but deciding which subset of variables Y' from the existing example e to include in the next query $ASK(e_{Y'})$. Thus, we propose to maximize the following objective function:

$$f(e_{Y'}) = \sum_{c \in \kappa_B(e)} [var(c) \subseteq Y'] \cdot \big(1 - k \cdot O(c)\big) \tag{10}$$

where the factor $[e \notin sol(\{c\})]$ from Equation (3) has been replaced by $[var(c) \subseteq Y']$ (recall that here $e_{Y'} \notin sol(\{c\})$ iff $var(c) \subseteq Y'$). The property $var(c) \subseteq Y'$ can be expressed through the boolean variables in BVC.

This objective function works well when the ML classifier is accurate (i.e., can approximate the oracle $O(c)$ accurately). However, there exists a caveat when the ML classifier is not able to learn any meaningful structure, in which case it tends to predict all candidate constraints as *not* belonging to C_T. When this occurs, the use of the guiding objective function (10) reduces to the maximization of the number of constraint violations, which can be highly suboptimal when the user answers the query negatively.

To prevent this, we add an *additional* objective function that provides a reward whenever at most half of the variables are selected in Y'. On the one hand, this ensures that, under the scenario described above, the guiding objective function (10) reduces to the halving strategy, which is preferable to the maximization of constraint violations, as discussed before. On the other hand, it still allows the selection of significantly less than half of the variables when this is deemed advantageous according to the guiding objective (10). This additional objective function (to be maximized) is the following:

$$f(e_{Y'}) = \begin{cases} 1 & \text{if } |Y'| \leq \left\lfloor \frac{|Y|}{2} \right\rfloor, \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

We combine the two objectives in a lexicographic ordering, where (11) is the primary objective, and (10) is the secondary objective. As a result, any query that selects more than half of the variables is strictly dominated by any query that selects less than half. Among all queries satisfying the first condition, the guiding objective (10) is used to select the most informative one, according to the learned constraint likelihoods.

Finally, note that, in (Tsouros et al., 2024), we have also proposed a similar guided query generation step for the original FindScope function. However, we do not discuss this here, as guiding the original FindScope function requires taking into account its specific functionality (e.g., the use of the variable set R), which limits the sets of variables that can be selected, such that the optimal set of variables cannot always be chosen.

3.2.4 Analysis. We now prove some theoretical properties of FindScope-3. We first prove its correctness by proving some of its invariants, and we then focus on its complexity in terms of the numbers of queries.

Proposition 3.3. *Given a bias B, with the assumption that $C_T \subseteq B$, a set of variables Y and an example e with $\emptyset \subset \kappa_{C_T}(e_Y)$, FindScope-3 is correct, i.e., it will return the scope of a constraint $c \in \kappa_{C_T}(e_Y)$.*

Proof. First, let us focus on FindScope-3's invariants.

(Inv. 1) *$\kappa_B(e_Y)$ is reduced in each iteration.* If the example $e_{Y'}$ is classified as positive by the user, then $\kappa_B(e_{Y'})$, with $\kappa_B(e_{Y'}) \supset \emptyset$ (Equation (4)), is removed from B, reducing $|\kappa_B(e_Y)|$. If the answer to the query is negative, then Y is replaced by Y', and thus $\kappa_B(e_Y)$, but not B itself, is reduced due to Equation (4).

(Inv. 2) *No constraint from C_T is ever removed from B.* At every step, the set B is only reduced by removing constraints $c \in \kappa_B(e_{Y'})$ in the case where $e_{Y'}$ is labeled positively by the user (lines 4-5), and thus $\kappa_{C_T}(e_Y) \cap \kappa_B(e_{Y'}) = \emptyset$. Hence, removing $\kappa_B(e_{Y'})$ is sound.

(Inv. 3) *We always have $\emptyset \subset \kappa_{C_T}(e_Y) \subseteq \kappa_B(e_Y)$.* As no constraint from C_T is ever removed from B (Inv 2.), we know that always $\kappa_{C_T}(e_Y) \subset \kappa_B(e_Y)$. Also, the set of variables is reduced only when the answer of the user is negative, and thus $\emptyset \subset \kappa_{C_T}(e_Y)$.

Termination: Since $\kappa_B(e_Y)$ is finite and we reduce it in each iteration (Inv. 1), the loop must terminate after a finite number of iterations.

Correctness: Upon termination, all constraints in $\kappa_B(e_Y)$ must share the same scope, otherwise a Y' would be found due to Equation (4). Moreover, we know that $\emptyset \subset \kappa_{C_T}(e_Y) \subseteq \kappa_B(e_Y)$ (Inv. 3). Therefore, this final set of candidates must contain at least one violated constraint from C_T. Hence, the shared scope of these constraints is the scope of a true constraint $c \in \kappa_{C_T}(e_Y)$. □

We now analyze the complexity of FindScope-3 in terms of the number of queries. We first consider the original version of FindScope-3 and then the complexity when the objective function from Equation (8) is used, i.e., using the "splitting in half" strategy. Throughout the analysis, we denote as α_{min} the minimum arity of constraints in C_T.

Proposition 3.4. *Given a set of variables Y and an example e with $\emptyset \subset \kappa_{C_T}(e_Y) \subseteq \kappa_B(e_Y)$, FindScope-3 needs at most $O(|\kappa_B(e_Y)|)$ queries in the worst case.*

Proof. Using (Inv. 1) of Proposition 3.3, we know that $\kappa_B(e_Y)$ strictly decreases at every iteration. Since $\kappa_B(e_Y)$ is finite, and FindScope-3 asks one query per iteration, the algorithm can ask up to $|\kappa_B(e_Y)|$ queries before termination. □

Proposition 3.5. *Given a set of variables Y and an example e with $\emptyset \subset \kappa_{C_T}(e_Y) \subseteq \kappa_B(e_Y)$, FindScope-3 needs at most $O(|Y| - \alpha_{\text{min}})$ queries in the best case, where all queries at line 4 lead to a negative answer.*

Proof. In the best case, FindScope-3 only generates examples that lead to a negative answer by the user, and thus does not need to eliminate constraints from $B \setminus C_T$ before locating the scope of a constraint in $\kappa_{C_T}(e_Y)$. As a result, at each iteration FindScope-3 replaces Y by a strict subset $Y' \subset Y$ (line 7), and therefore removes at least one variable from Y. In addition, a negative answer implies that the example e_Y still violates at least one constraint $c \in C_T$, and thus $var(c) \subseteq Y$. Since all constraints in C_T have arity at least α_{min}, Y can be strictly reduced at most $|Y| - \alpha_{\text{min}}$ times. As a result, FindScope-3 can ask at most $O(|Y| - \alpha_{\text{min}})$ queries before termination in the best case. □

We now analyze the complexity of FindScope-3 when the objective function from Equation (8) is used. Note that the halving objective only affects how the set of variables Y is reduced following a negative answer. It does not affect the fact that $\kappa_B(e_Y)$ strictly decreases at each iteration (Inv. 1 of Proposition 3.3). As a result, the worst-case upper bound stays the same as in Proposition 3.4. We now focus on the best-case upper bound.

Proposition 3.6. *Given a set of variables Y and an example e with $\emptyset \subset \kappa_{C_T}(e_Y) \subseteq \kappa_B(e_Y)$, FindScope-3 with the objective from Equation (8) needs at most $O\left(\log_2\left(\frac{|Y|}{\alpha_{\min}}\right)\right)$ queries in the best case, where all queries at line 4 lead to a negative answer.*

Proof. As in the best case all queries lead to a negative answer, FindScope repeatedly replaces Y by a strict subset $Y' \subset Y$. We denote as Y_t the set Y after t iterations. Due to the halving objective, the set Y' is chosen to be as close to $\lfloor |Y|/2 \rfloor$ as possible, while satisfying the requirement $\emptyset \subset \kappa_B(e_{Y'}) \subset \kappa_B(e_Y)$ (Equation (4)). Here there are 2 cases for the size of the set Y', based on the available subsets of variables that satisfy the requirements:

Case 1 $Y' \leq \lfloor |Y|/2 \rfloor$: In this case, at least half of the variables in Y are removed at each iteration, as Y is replaced by Y' (line 7) after a negative answer. Hence, $|Y_t| \leq \lfloor |Y_{t-1}|/2 \rfloor$.

Case 2 $Y' > \lfloor |Y|/2 \rfloor$: In this case, the set Y' was the optimal set satisfying the requirements. We will now show that, in this case, this iteration will be the last one to ask a query to the user, and FindScope-3 will then terminate. As Y' was the optimal subset of variables, we know that $\nexists Y'' \subset Y' \subset Y$ s.t. $c < |Y''|$, with $c = |Y| - |Y'|$, satisfying Equation (4), as otherwise Y'' would have a better objective value in Equation (8). If this is not the last iteration of FindScope-3, it would mean that $\exists Y'' \subset Y' \subset Y$ with $|Y''| < c$ and $\emptyset \subset \kappa_B(e_{Y''}) \subset \kappa_B(e_{Y'}) \subset \kappa_B(e_Y)$. However, this would mean that any set T with $Y'' \subset T \subset Y'$ would have $\kappa_B(e_T) \subseteq \kappa_B(e_{Y'}) \subset \kappa_B(e_Y)$. Therefore, T would satisfy the requirements of Equation (4) and would have a better objective value in Equation (8) than Y', which contradicts the optimality of Y'. Hence, $\nexists Y'' \subset Y' \subset Y s.t.$. As a result, when $Y' > \lfloor |Y|/2 \rfloor$ FindScope-3 asks one more query and then terminates.

Based on the above, we either have $|Y_t| \leq \lfloor |Y_{t-1}|/2 \rfloor$ or we immediately terminate after Y_t. Therefore, after t consecutive negative answers, we must have $|Y_t| \leq \lfloor |Y_0|/2^t \rfloor$. As shown in Proposition 3.5, a negative answer implies that e_{Y_t} still violates a constraint $c \in C_T$, and thus $var(c) \subseteq Y_t$. Since all constraints in C_T have arity at least $\alpha_{\min}$, we must have $|Y_t| \geq \alpha_{\min}$ throughout the execution, while zooming in the scope of constraint a $c \in \kappa_{C_T}(e_Y)$. This results to

$$\alpha_{\min} \leq |Y_t| \leq \lfloor \frac{|Y_0|}{2^t} \rfloor \implies \alpha_{\min} \leq \frac{|Y_0|}{2^t} \implies 2^t \leq \frac{|Y_0|}{\alpha_{min}} \implies t \leq \log_2(\frac{|Y_0|}{\alpha_{min}}) \tag{12}$$

Hence, we can have at most $\log_2(\frac{|Y|}{\alpha_{min}})$ iterations, which bounds the number of queries in the best case. □

Note that the best-case upper bound of FindScope-3 is lower than the one of existing FindScope functions, which is in $\Theta(|S| \cdot log_2(Y))$ (Tsouros and Stergiou, 2020; Bessiere et al., 2023b). Also, as the objective function typically used during interactive CA, when unguided, aims to violate a maximum number of constraints from B, this often leads to a large number of violated constraints from C_T. Hence, during the execution of FindScope functions we often have many negative answers, being close to the best-case scenario described above.

3.3 Guiding FindC

After the system has located the scope of a violated constraint, it calls function FindC (Bessiere et al., 2013, 2023b) to find the relation of the violated constraint. To locate this constraint, FindC asks partial queries to the user in

the scope returned by the FindScope module. Different assignments are generated for the variables in the scope given, so as to query for a different subset of candidate constraints, whether they belong to the target problem. In order to do so, FindC functions currently use the following query generation step:

$$\text{find } e'_S \in sol(C_L[S] \wedge (\emptyset \subset \kappa_B(e'_S) \subset \Delta)), \tag{13}$$

with S being the scope found in the previous step and Δ the set of candidates for this scope. Δ is initially equal to the set of violated constraints in the previous example $\kappa_B(e_S)$ (Bessiere et al., 2013) or equal to the entire set of candidate constraints in this scope (Bessiere et al., 2023b), based on which FindC function is used and whether it aims to learn normalized constraint networks or not.

The objective function typically used in this step, in order to again achieve a logarithmic complexity in terms of the number of queries posted, is to try to half the number of violated candidates, minimizing a slack variable b with

$$b = \left\lfloor \frac{|\Delta|}{2} \right\rfloor - \kappa_B(e'_S) \tag{14}$$

We propose to replace this objective function with one that guides the query generation in the same way as in Equations (3) and (10), maximizing the following objective:

$$f(e_S) = \sum_{c \in \Delta} [c \in \kappa_\Delta(e_S))] \cdot \big(1 - k \cdot O(c)\big) \tag{15}$$

4 Using Probabilistic Classification to Guide Interactive CA

As we discussed in Section 3.1.1, the model $\mathcal{M}$ of the oracle O leverages a probabilistic estimate of the likelihood of a given candidate constraint belonging to the problem at hand, $P(c \in C_T)$. We propose to use statistical machine learning techniques, exploiting probabilistic classification in order to calculate $P(c \in C_T)$.

We first present an ML-based framework for interactive CA and then propose a feature representation of constraints to use.

4.1 Framework for ML-guided Interactive CA

In order to use probabilistic classification in this context, we need to build a dataset to learn from. We formally define a dataset E as a collection of N instances, each of which corresponds to a constraint. Each instance is a tuple $(\mathbf{x}_i, y_i)$, $i \in 1, 2, ..., N$, with $\mathbf{x}_i$ being a vector of features representing constraint c_i, and y_i being a (Boolean) label that denotes whether $c_i \in C_T$.

The dataset E is grown incrementally throughout the CA process, as gradually more information is obtained about constraints in the initial bias. This is shown in Algorithm 6. In line 2, the dataset is initialized using the (optionally) provided initial sets of learned constraints C_{in} and excluded constraints C_{excl}. Concretely, each constraint from $C_{in} \cup C_{excl}$ is featurized as $x_i = \phi(c_i)$ and added to E with a positive label if $c \in C_{in}$ and with a negative label otherwise. Later, in the main loop of the algorithm, whenever a constraint is removed from the bias B (because it was determined to not be part of C_T), it is also featurized and added to E with a negative label (line 10). Similarly, whenever a constraint is added to C_L, it added to E with a positive label (line 15). Note that this also happens for constraints learned or excluded within a call to FindScope or FindC, which is why E is given as an argument to these functions (lines 12 and 13).

In every top-level loop of the CA algorithm (line 3), the current dataset E is used to train a probabilistic classifier $\mathcal{M}$ (line 4), as illustrated in Figure 7. This classifier is then used to guide top-level query generation (line 5), FindScope (line 12) and FindC (line 13), the details of which are respectively detailed in Section 3.1, Section 3.2 and Section 3.3.

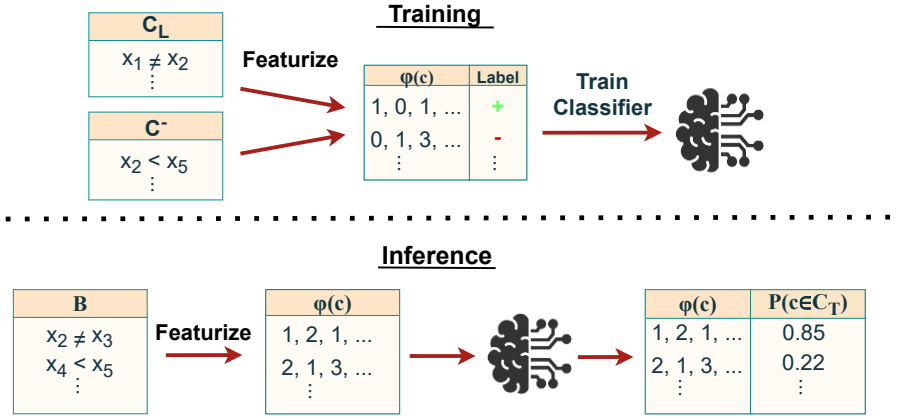


Fig. 5. Probabilistic classifier of constraints: training (top) and inference (bottom).

Algorithm 6 The framework for ML-guided interactive CA (ML-related parts in blue).

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

1: $C_L \leftarrow C_{in}, \quad C^- \leftarrow C_{excl}$
2: $E \leftarrow \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i = \phi(c_i) \wedge \ y_i = [c_i \in C_L] \ \ \forall c_i \in C_L \cup C^-\}$ ▷ Initialize the dataset
3: **while** True **do**
4: $\quad \mathcal{M} \leftarrow$ TrainClassifier(E)
5: $\quad e, Y \leftarrow$ QGen$(C_L, B, \mathcal{M})$
6: $\quad$ **if** e = nil **then return** C_L ▷ Converged
7: $\quad$ **if** $ASK(e_Y)$ = True **then**
8: $\quad\quad C^- \leftarrow C^- \cup \kappa_B(e_Y)$
9: $\quad\quad B \leftarrow B \setminus \kappa_B(e_Y)$
10: $\quad\quad E \leftarrow E \cup \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i = \phi(c_i) \wedge \ y_i = False \ \ \forall c_i \in \kappa_B(e_Y)\}$ ▷ Update dataset
11: $\quad$ **else**
12: $\quad\quad (B, C^-, S) \leftarrow$ FindScope$(e, Y, B, C^-, E, \mathcal{M})$
13: $\quad\quad (B, C^-, C) \leftarrow$ FindC$(S, C_L, B, C^-, E, \mathcal{M})$
14: $\quad\quad C_L \leftarrow C_L \cup C$
15: $\quad\quad E \leftarrow E \cup \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i = \phi(c_i) \wedge \ y_i = True \ \ \forall c_i \in C\}$ ▷ Update dataset

The frequency at which classifier $\mathcal{M}$ is retrained forms an important design choice, as this may affect the waiting time for the user when interacting with the CA system. In this paper, we retrain the classifier right before every top-level query generation, so that the model can incorporate all the information collected in the previous iteration of the loop. Preliminary experiments showed that this yields the best results, compared to more frequent retraining (e.g., every time a constraint is learned or excluded) or less frequent retraining (e.g., after every k iterations, with $k > 1$).

ID	Name	Type	Description
1	Relation	String	Constraint relation
2	Arity	Int	Constraint arity
3	Has_constant	Bool	If a constant value is present
4	Constant	Int	The constant value
5	Var_name_same	Bool	If all variables share the same name
6	Var_dim$_i$_has	Bool	If dimension i is present for all variables
7	Var_dim$_i$_same	Bool	If the index in dimension i is the same for all variables
8	Var_dim$_i$_max	Int	Maximum variable index in dimension i
9	Var_dim$_i$_min	Int	Minimum variable index in dimension i
10	Var_dim$_i$_avg	Float	Average variable index in dimension i
11	Var_dim$_i$_dist	Float	(Maximum) distance between variables in dimension i
12	Var_dim$_i$_ldim$_j$_same	Bool	If the index in dimension i's latent dimension j is the same for variables
13	Var_dim$_i$_ldim$_j$_max	Int	Maximum variable index in dimension's i latent dimension j
14	Var_dim$_i$_ldim$_j$_min	Int	Minimum variable index in dimension's i latent dimension j
15	Var_dim$_i$_ldim$_j$_avg	Float	Average variable index in dimension i's latent dimension j
16	Var_dim$_i$_ldim$_j$_dist	Float	(Maximum) distance between variables in dimension i's latent dimension j

Table 1. Feature representation of constraints

4.2 Feature Representation of Constraints

For classifier $\mathcal{M}$ to effectively detect relevant patterns in dataset E, the constraints must be encoded using an expressive feature representation ϕ. Our proposed feature representation is shown in Table 1, along with a description of each feature. We now provide some additional context to complement these descriptions.

Features 1-5 are self-explanatory. Feature 6–11 refer to the dimensions of variables in the constraint's scope. These pertain to the dimensions in the data structure in which the variables are given. Note that variables are often given in the form of a matrix or tensor (Flener et al., 2001). For example, a natural way to structure the variables representing the cell assignments in Sudoku is as a 9×9 2-dimensional matrix. When variables have such indices, we want to represent this information in the features of the constraint. We encode information per dimension over all variables in the constraint (Feature 6–11). This allows the system to detect common patterns of CP problems, such as constraints over variables occurring in the same row or column, or not being spread out across a dimension.

Features 12–16 refer to *latent dimensions* that are not directly given, but that can be indirectly derived from the variable tensor's dimensions. For example, the vertical and horizontal block indices in a Sudoku grid form latent dimensions. A block index is not explicitly present in the variable matrix, but can easily be obtained by dividing the row/column index by the square root of the grid size. To construct the latent dimensions within a dimension i, we perform integer division with all positive non-trivial divisors of the number of indices in dimension i. For index i and divisor j, the latent dimension index is given by $\left\lfloor \frac{i-1}{j} \right\rfloor + 1$. For example, consider a CA problem in which 8 variables are given in the form of a vector. 8 has two positive non-trivial divisors: 2 and 4. The variables thus have a single dimension, and two latent dimensions: one for divisor 2 and one for divisor 4. Consider now the 6-th variable. This variable has index 6 in the first (and only) dimension, index $\left\lfloor \frac{6-1}{2} \right\rfloor + 1 = 3$ in the first latent dimension, and index $\left\lfloor \frac{6-1}{4} \right\rfloor + 1 = 2$ in the second latent dimension.

Note that a CA problem has as input the set of variables X. Hence, for any CA problem, we can compute the number of dimensions and latent dimensions upfront, and use a fixed-size feature representation throughout the CA run.

Example 4.1 (Feature Representation). Consider again the problem from Example 3.1, containing four variables in one row, where the target set of constraints contains all $\neq$ constraints between all pairs of variables. We illustrate the feature representation for candidate constraint $x_1 \neq x_2$.

(1) Relation: "$\neq$".
(2) Arity: 2, since the constraint involves two variables.
(3) Has_constant: `false`, as no constant appears in the constraint.
(4) Constant: 0 (dummy value, since no constant is present).
(5) Var_name_same: `true`, because both variables share the same base name x.
(6) Var_dim$_1$_has: `true`, since both variables have a first dimension.
(7) Var_dim$_1$_same: `false`, because the variables have different indices in the first (and only) dimension.
(8) Var_dim$_1$_max: 2.
(9) Var_dim$_1$_min: 1.
(10) Var_dim$_1$_avg: 1.5.
(11) Var_dim$_1$_dist: 1.
(12) Var_dim$_1$_ldim$_1$_same: `true`, as $\left\lfloor \frac{1-1}{2} \right\rfloor + 1 = \left\lfloor \frac{2-1}{2} \right\rfloor + 1 = 1$.
(13) Var_dim$_1$_ldim$_1$_max: 1.
(14) Var_dim$_1$_ldim$_1$_min: 1.
(15) Var_dim$_1$_ldim$_1$_avg: 1.
(16) Var_dim$_1$_ldim$_1$_dist: 0.

5 Experimental Evaluation

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

(Q1) How accurately can ML classifiers identify whether a candidate constraint belongs to the target constraint network?
(Q2) How does using ML classifiers to guide top-level query generation in interactive CA affect performance?
(Q3) How does replacing predicted probabilities with binary class labels affect performance?
(Q4) How does guiding the other layers of interactive CA affect performance?

To answer our experimental questions, we perform an extensive evaluation across 7 benchmark problems, using a variety of classifiers. We also assess the performance using feature representations of different levels of expressivity, to evaluate how the quality of the predictions influences the effectiveness of our guiding strategies.

5.1 Experimental Setup

We now discuss the details of our experimental setup.

Feature representations. To evaluate our guiding methods using classification models of different prediction qualities, while also evaluating the impact of the expressivity of the feature representation, we used feature representations of various levels of expressivity in the experiments. This allows us to assess performance with highly expressive feature representations (leading to accurate classifiers), as well as with representations that are not sufficiently expressive to fully capture the problem structure (leading to inaccurate classifiers). Concretely, we used the following representations:

- *Rel*: Using only the features pertaining to the relation of the constraint (features 1-4, Table 1).

- *RelDim*: Using only the features pertaining to the relation of the constraint and the dimensions of its variable indices (features 1-11, Table 1).
- *Full*: Using all features, i.e., features pertaining to the relation of the constraint, the dimensions of its variable indices, and the latent dimensions.

Classifiers. We focus on the following classifiers: Logistic Regression (LR), Random Forests (RF), Decision Trees (DT), Gaussian Naive Bayes (NB), Multi-layer Perceptron (MLP), and Support Vector Machines (SVM). We used RF, DT and NB in their default settings, while we tuned the most important hyperparameters for LR, MLP and SVM. We tuned the classifiers separately for each feature representation, based on their average performance on all benchmarks[1]. For tuning, we used the complete dataset for all benchmarks, having labeled all candidate constraints. A grid search, coupled with 10-fold cross-validation (splitting B into 10 folds), was conducted, using balanced accuracy as the metric to address class imbalance. Hyperparameter combinations surpassing a 10-second training time were omitted to ensure relevance in interactive scenarios, unless there was no hyperparameter combination under that limit, in which case we chose the fastest one.

Comparison We compare performance based on *the number of queries*, which is highly important for the applicability of interactive CA systems in real-world scenarios. We also report on *the maximum user waiting time*, which is of great importance when human users are involved. The results presented are the means of 10 runs to average out randomness in query generation. As the baseline non-guided objective function for top-level query generation, we used *max* (Bessiere et al., 2013), which maximizes the number of violated constraints from B (Equation (2)). This is the most commonly used objective that can be integrated with conventional solvers for use in query generation, and is shown in (Tsouros et al., 2023a) to offer similar results to the objective function for custom solvers from (Tsouros and Stergiou, 2020). For FindScope we used the standard halving strategy from Algorithm 2. For FindC we also used the standard halving objective (Equation (14)).

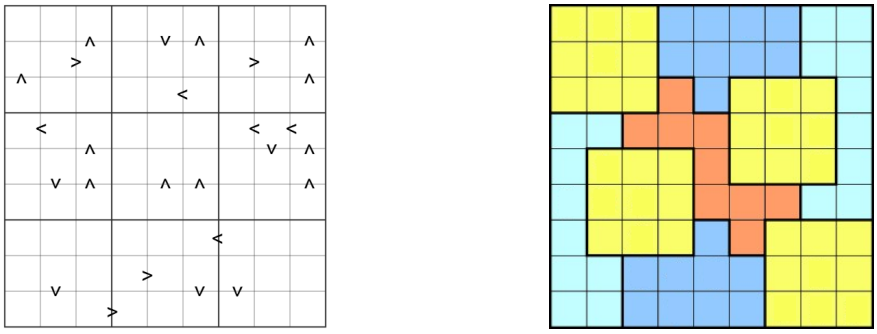
Other aspects of the setup. We now provide additional implementation details:

- All methods and benchmarks were implemented[2] in Python. The PyConA CA library (Tsouros and Guns, 2025) was used to implement the benchmarks and for the interactive CA algorithms. PyConA uses the CPMpy constraint programming and modeling library (Guns, 2019) to model CP problems. OR-Tools' CP-SAT (Perron et al., 2023) was used as the solver, interfaced through CPMpy.
- The guiding techniques are integrated within QuAcq, which is used as the underlying algorithm in GrowAcq. GrowAcq is used because it only has to consider a (small) part of the bias at once. This makes it possible to efficiently optimize an objective function in query generation (as discussed in Section 2.2.1), which is used in the ML-based guiding. FindScope-2 (Tsouros and Stergiou, 2020), which has the same behaviour as the FindScope function from (Bessiere et al., 2023b), was used as the baseline FindScope function. We used the FindC function from (Bessiere et al., 2013).
- We used a time limit of 1 second for all query generation steps, including our newly introduced one in FindScope-3.
- For the classification component we used the Scikit-Learn library (Pedregosa et al., 2011).
- All the experiments were conducted on a system carrying a 12th Gen Intel(R) Core(TM) i7- with 4.90GHz clock speed and 64 GB of RAM.

5.2 Benchmarks

We selected the benchmarks for our experiments to cover a broad range of cases. These include typical benchmarks in the evaluation of CA systems, containing standard structural patterns that frequently appear in CSPs

[1] More details can be found in the appendix
[2] Our code is publicly available at: https://github.com/Dimosts/ML-guided-CA



(a) GTSudoku instance (b) JSudoku instance

Fig. 6. The instances of Jigsaw Sudoku and Greater Than Sudoku used.

(e.g., all-different constraints), as well as unstructured or randomly generated constraints. This allows us to assess performance both in settings where constraint structure can be exploited and in settings where little or no exploitable structure is present. In addition, we included problems that more closely resemble real-world applications to evaluate practical relevance. In more detail, we used the following benchmarks:

9x9 Sudoku (Sudoku) The Sudoku puzzle is a benchmark often used to evaluate CA systems, and is included to evaluate our approach on a problem with regular structure. It is a $n^2 \times n^2$ grid ($n = 3$ for the case we used), which must be completed in such a way that all the rows, columns, and n^2 non-overlapping $n \times n$ squares contain the distinct numbers. This gives a *vocabulary* having 81 variables and domains of size 9. The target constraint network consists of 810 $\neq$ constraints. The bias was initialized with 19 440 binary constraints, using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

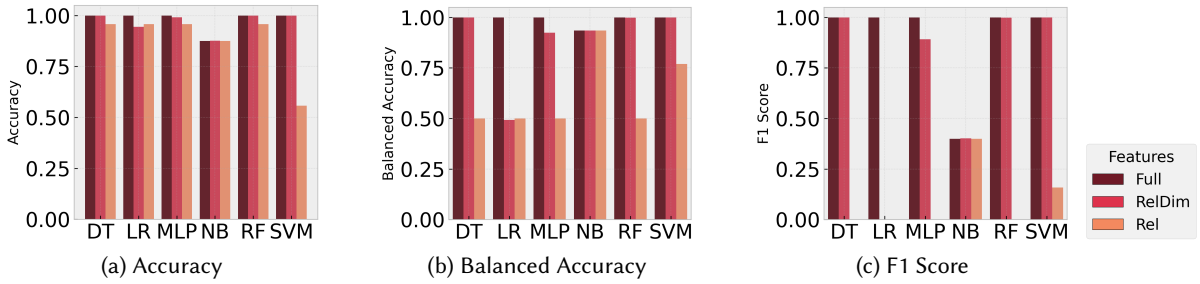
Jigsaw Sudoku. (JSudoku) The Jigsaw Sudoku is a variant of Sudoku in which the 3×3 boxes are replaced by irregular shapes. This is helpful in order to evaluate our approaches in a problem with a less regular structure. We used the instance that is displayed in Figure 6b, It consists of 81 variables with 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.

Greater than Sudoku. (GTSudoku) The Greater Than Sudoku is a variant of Sudoku in which the shapes are the same, but there are also some $>$ constraints between neighboring cells. This is helpful in order to evaluate our approaches to a problem with a structure but with additional non-structured constraints. We used the instance that is displayed in Figure 6b, It consists of 81 variables with domains of size 9. The target network consists of 810 binary $\neq$ constraints on rows, columns, and shapes, and 26 $>$ or $<$ additional constraints. The bias B was constructed using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$ and contains 19 440 binary constraints.

Random$_{\neq}$ (Random) We evaluated our methods in a random problem. 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 29 700 constraints, using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

We also evaluated our methods in the following benchmarks that are closer to real-world problem.

Job-shop scheduling. (JS) 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



(a) Accuracy (b) Balanced Accuracy (c) F1 Score

Fig. 7. Classification results with different classifiers, on different levels of feature representations

machine. The language $\Gamma = \{\geq, \leq, <, >, \neq, =, x_i + c = x_k\}$ was used, for every constant c ranging from 0 up to the maximum duration of the jobs (5 in the intsance we used). We used a problem instance containing 20 jobs, 3 machines, with a start and end variables for each task (i.e., $|X| = 120$). We used a time horizon of 15 steps,. The final bias contained 57 120 constraints.

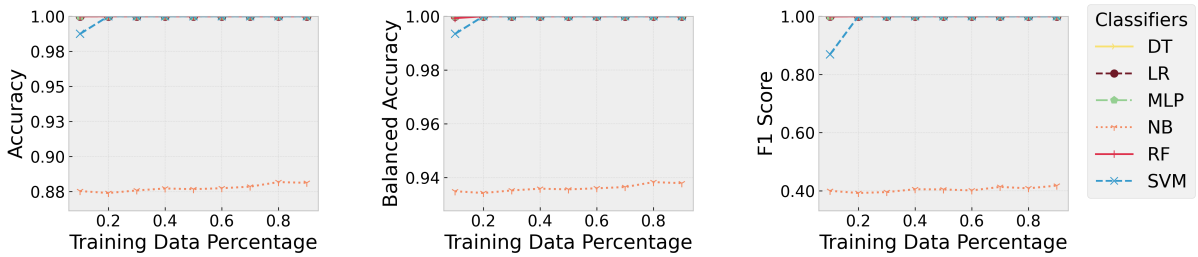
Exam Timetabling (ET) There are ns semesters, each containing cps courses, and we want to schedule the exams of the courses in a period of d days, On each day we have t timeslots. The variables are the courses ($|X| = ns \cdot cps$), having as domains the timeslots they can be assigned on ($D_i = 1, ..., t \cdot d$). There are $\neq$ constraints between each pair of exams. Also, two courses in the same semester cannot be examined on the same day, which is expressed by the constraints $\lfloor x_i/spd \rfloor \neq \lfloor x_j/spd \rfloor$, $\forall i, j$ in the same program. We used an instance with $ns = 9$, $cps = 5$, $d = 10$, and $t = 6$. This resulted in a model with 48 variables and domains of size 90. C_T consists of 1, 128 constraints. The language given is $\Gamma = \{\geq, \leq, <, >, \neq, =, \lfloor x_i/spd \rfloor \neq \lfloor x_j/spd \rfloor\}$, creating a bias of size 7,896.

Nurse rostering (NR) There are n nurses, s shifts per day, ns nurses per shift, and d days. The goal is to create a schedule, assigning nurses to all existing shifts. The variables are the shifts, and there are a total of $d \cdot s \cdot ns$ shifts. The variables are modeled in a 3D matrix. The domains of the variables are the nurses ($D^{x_i} = \{1, ..., n\}$). Each shift in a day must be assigned to a different nurse and the last shift of a day must be assigned to a different nurse than the first shift of the next day. In the instance used in the experiments, we have $d = 7$, $s = 3$ and $ns = 5$. The available nurses are $n = 18$. This results in $|X| = 105$ with domains $\{1, ..., 18\}$. C_T consists of 885 $\neq$ constraints. The bias was built using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$, resulting in $|B| = 32,760$.

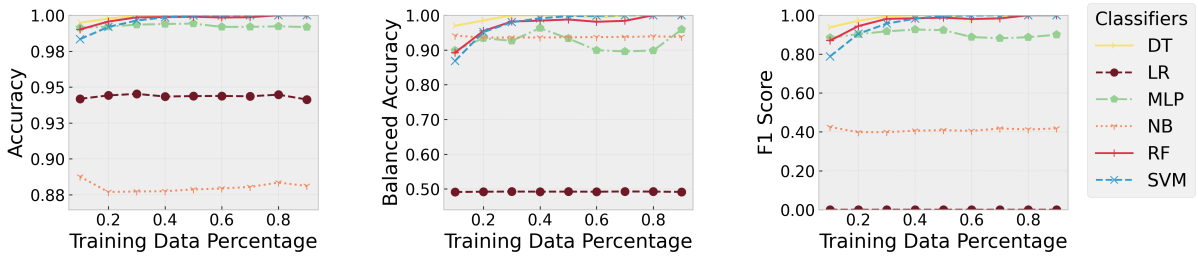
5.3 Results and Discussion

5.3.1 Q1: How Accurate are Classifiers in Identifying True Constraints. For each benchmark, we generated all candidate constraints, assigned the corresponding labels, and constructed a dataset from that. To answer the question, we performed 10-fold cross-validation with each classifier. We run this experiment on all benchmarks with a regular structure (Sudoku, ET, NR), as we want to evaluate the ability of classifiers to identify that structure, and report the average results. We report results for all levels of feature representation: Rel, RelDim, and Full. As evaluation metrics, we use *Accuracy*, *Balanced Accuracy* (to account for the strong class imbalance, as typically fewer than 10% of B carry a positive label), and *F1-score*. The results are presented in Figure 7.

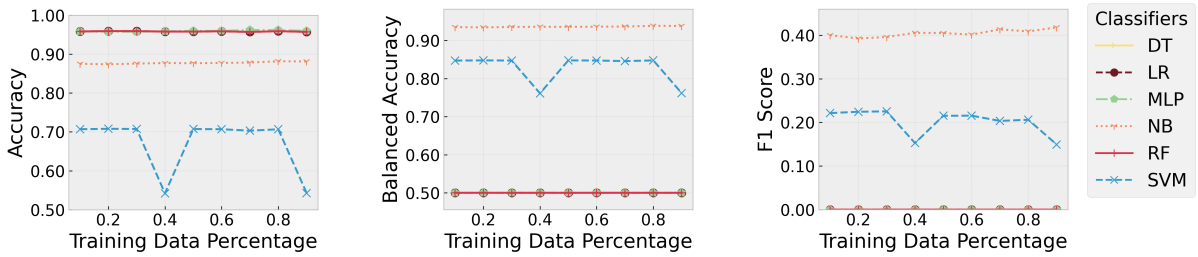
We can notice that, when the full set of features is used, all classifiers considered achieve a perfect accuracy and balanced accuracy, with NB performing slightly worse than the rest, and MLPs performing best. Focusing on F1-score, NB presents quite bad results, but the rest of the classifiers still achieve a score close to 100%. We believe that the worse results of NB are due to its feature independence assumption. Although such near-perfect performance for most classifiers may indicate overfitting within each benchmark, this is not problematic in our context, as the objective is not cross-instance generalization but accurate, instance-specific guidance during



(a) Results using the full feature representation (Full)



(b) Results using features only for the relation and the dimensions (RelDim)



(c) Results using features only for the relation (Rel)

Fig. 8. Classification results when only part of the dataset is available for training, using different feature representations

the acquisition process. The classifier is trained iteratively using constraints learned and ruled out throughout the acquisition process. Its purpose is therefore to guide query generation within that instance rather than to generalize to unseen distributions.

Focusing on the results when fewer features are used, we can see that most classifiers present high scores on all metrics even with the RelDim feature set, except LR and NB, which present very low balanced accuracy and F1-score. On the other hand, when only relation features are used, in feature set Rel, the classifier performances drop, as they cannot distinguish in which scopes they should predict the true constraints. Most classifiers prefer to predict all candidates as false for this feature set (classifying them as not belonging to the target set of constraints), and thus present a balanced accuracy of 50%, while NB predicts all constraints that have relations appearing in

a constraint of C_T as true, which reduces its accuracy. Overall, the results indicate that based on the way the dataset of constraints is created, it is indeed possible to successfully train and use ML models to predict whether a constraint is part of the target network, provided that the features can capture sufficient information about the structure of the target constraints.

However, in order to use the classifiers to assist during the acquisition process – guiding it to generate promising queries, based on the predictions – it is important to evaluate how they perform not only when the labels for 90% of candidate constraints are available, but when only selective parts of the dataset are available (as this is the case during the CA process). Thus, we conducted an experiment to evaluate how the classifiers perform when only a percentage of the dataset is available. We used an increasing portion of the dataset as the training set, to evaluate their performance in different stages of the acquisition process, with the rest of the candidates being the test set. The averages are presented in Figure 8.

We can observe that when the full set of features is used, all classifiers except NB achieve high scores very quickly: even when only 10% of the dataset is available, most classifiers achieve a 100% score in all metrics. When using the intermediate feature representation (i.e., RelDim), most classifiers still achieve high accuracy, balanced accuracy, and F1-score early, and increase their scores when more data becomes available. However, LR and NB are notable exceptions once again, showing very low balanced accuracy and F1-score for all percentages. As expected, when only relation-level features are used – without any scope information – classifier performance drops significantly. We can observe that DT achieves the best results in all metrics from the beginning, when only a small portion of the dataset is labeled, even when the RelDim feature representation is used.

5.3.2 Q2: Using ML Classifiers to Guide Top-level Query Generation. Let us now focus on the effect of using our method for guiding top-level query generation with the probabilistic classifiers. Figure 9 presents the average results on all benchmarks when using the different classifiers for providing the probabilities for the candidate constraints, compared to a baseline that does not guide query generation (dashed line). We present results on each benchmark separately in Figure 10, for all levels of feature representations. We show both the number of queries (left column) and the maximum waiting time for the user (right column).

Focusing on the results when the full feature representation is used, we can see that average decrease in the number of queries is significant with our guiding method, being more than 50% for all classifiers except NB. Focusing more on the results per benchmark (Figure 10), we observe that in all except Random and JS the decrease in the number of queries is significant compared to the non-guided baseline, for most classifiers. This is expected: Random does not have any pattern in its constraints, and JS includes random precedences between jobs that are hard to predict. Moreover, JS is a very sparse constraint network, so many queries are used to eliminate candidates from B rather than to learn new constraints. In the rest of the benchmarks, guiding query generation offers a reduction from 67% (in NR) up to 75% (in Sudoku).

Interestingly, we see improvements in the performance with most classifiers even when only features for the relation are used (Rel), presenting a more than 25% reduction in the number of queries on average when RF, MLP, or DT are used. That is because in all benchmarks, the language includes several additional relations that do not actually appear in the target set of constraints, and thus even using only relation-based features (Rel) allows the classifier to identify constraints with irrelevant relations. Also, we see that when the RelDim features are used, the number of queries is comparable to the performance when the full features are used, in most benchmarks. Using the full features still offers benefits though (mainly in the Sudoku-based problems), additionally reducing the number of queries.

RF and DT are the most classifiers promising across all benchmarks and feature representations, giving the best results on average. We attribute their superior performance to the fact that they already achieve good prediction performance when only a small portion of the constraints is labeled, i.e., at the beginning of the acquisition

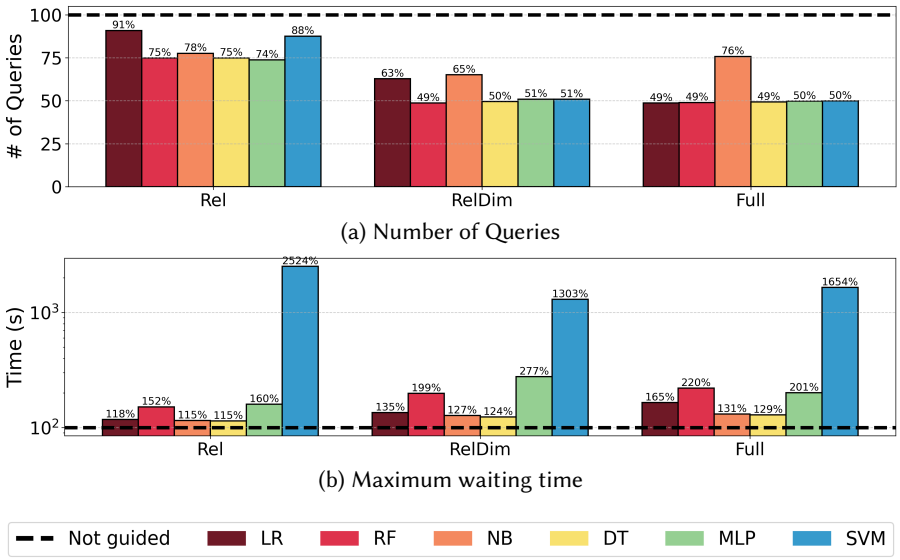


Fig. 9. Results when guiding query generation using probabilistic classification, across different benchmarks and using a variety of classifiers and feature representations. Average on all benchmarks.

process, as shown in Q1. NB presents decent results in most benchmarks, but still performs worse than the other classifiers.

The figure also show the maximum waiting time for the user, which includes 1 second for query generation (based on the imposed cutoff), with the remaining waiting time mostly going to the ML training and prediction. We can see higher waiting times mainly for the SVM classifier, which needs a larger training time, making it impractical for interactive scenarios in many problems (e.g., >100s in Random and JobShop). We also notice slightly higher waiting times when MLPs are used. Other than SVM and MLPs, we can see that the training times do not significantly affect the waiting time of the user, not exceeding 5s in any benchmark, which is very similar to the non-guided baseline.

5.3.3 Q3: Guiding Using the Probabilities or the Predicted Class. To answer this experiment question, we did the same experiment as in Q2, using the predicted class of the classifiers in the objective function (i.e., $O_{class}(c) = [P(c \in C_T) \geq 0.5]$), instead of the probabilities. As this does not affect the training time of the classifiers, we present results only for the number of queries. Figure 11 presents the average results on all benchmarks when using the different classifiers, comparing the use of the probabilities versus the use of the predicted class. We present results on each benchmark separately in Figure 12. We do not present results for the Rel feature representation, as it is already shown that it offers limited benefits with respect to not guiding, compared to the more expressive feature representations. We focus on RelDim (left column) and Full (right column) features, to evaluate how well guiding performs when the predictions of the classifiers are of different quality.

Looking at the average results on all benchmarks (Figure 11), we observe that using the probabilities for candidate constraints presents better results than using the class label. That is because using class labels does not differentiate between constraints that are very likely to belong to a class with constraints, or those that have more uncertainty. When the full set of features is used (Figure 11b), using the probabilities leads to an up to 2% larger reduction in the number of queries. On the other hand, when the RelDim set of features is used (Figure 11a)

Fig. 10. Results when guiding query generation using probabilistic classification, across different benchmarks and using a variety of classifiers and feature representations

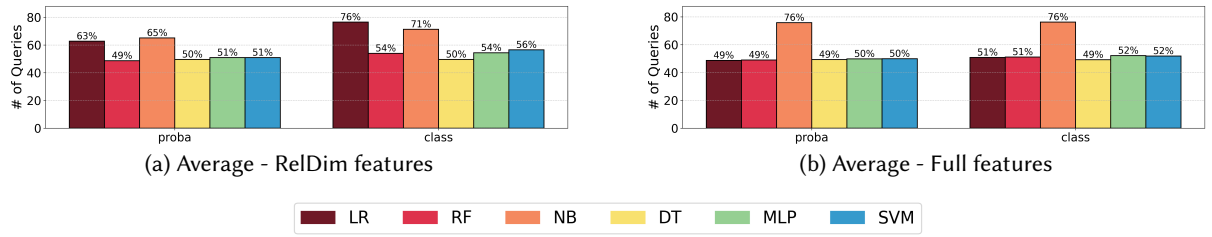


Fig. 11. Average results on all benchmarks comparing the use of probabilities in the objective against the use of the predicted class

and thus the predictions of the classifiers can be slightly worse, we see a larger boost in performance when the probabilities are used, up to 13% for LR. We see that the decrease in the number of queries is larger for classifiers that have worse performance, as the predictions of these classifiers are likely worse, and thus using directly the class label provides subpar guidance.

Looking at the results per benchmark (Figure 12), we can see a similar pattern. In all problems except Random and JS, the reduction in the number of queries is larger for most classifiers when the probabilities are used instead of the binary class label. In the Full features setting, we can see large improvements for the MLP and SVM classifiers, mostly in Sudoku variants with less regular structure than standard Sudoku, and in ET. When the RelDim features are used, we can see improvements in almost all the classifiers in all benchmarks (except Random and JS), with larger improvements for LR (up to 25% in ET), and smaller improvements for DT. Importantly, we can see that the number of queries does not increase (by more than a value attributable to randomness) for any classifier on any benchmark, by using the probabilistic estimate instead of the predicted class.

5.3.4 Q4: Guiding All Layers of Interactive CA.. We now evaluate the effect of guiding the other query-posing layers of interactive CA, besides top-level query generation. We only use RF, as it presented the best performance in the previous experiment, using the Full feature representation. We compare the effect of guiding each layer of interactive CA against the baseline (without guiding) and against guiding all layers, which is the combination of our proposed methods. We also present results for using FindScope-3 in the base non-guided version, and the performance of guiding FindScope-2 using a similar approach, which we presented in the conference paper (Tsouros et al., 2024). Figure 13 illustrates the results for all benchmarks.

Using FindScope-3. First, we can observe that using our new FindScope-3 leads to improvements even in its default, unguided version. The number of queries is reduced in all benchmarks when FindScope-3 is used, from 12% in NR up to 31% in ET. This reduction is higher in benchmarks that have a larger target constraint network. The reduction in the number of queries is possibly due to the fact that our new FindScope-3 does not aim to find one variable of the scope of a violated constraint at a time, but the entire scope directly.

Guiding FindScope. We now turn our attention to the improvements offered by guiding FindScope. We can see that using our objective function to guide both FindScope variants leads to reductions in the number of queries in all benchmarks. We can also observe that guiding our proposed FindScope-3 can lead to even more significant improvements in all benchmarks, reducing the queries by up to 29-33% more in the Sudoku-variants compared to guiding FindScope-2. Overall, the reduction in the number of queries reaches up to 61% in ET and NR compared to the baseline. This improvement is slightly worse than guiding top-level query generation (reduction up to 75% in Sudoku), as expected. That is because top-level query generation has a larger effect on the subsequent queries. However, it shows that additional benefits can be offered by guiding the scope-finding process of interactive CA.

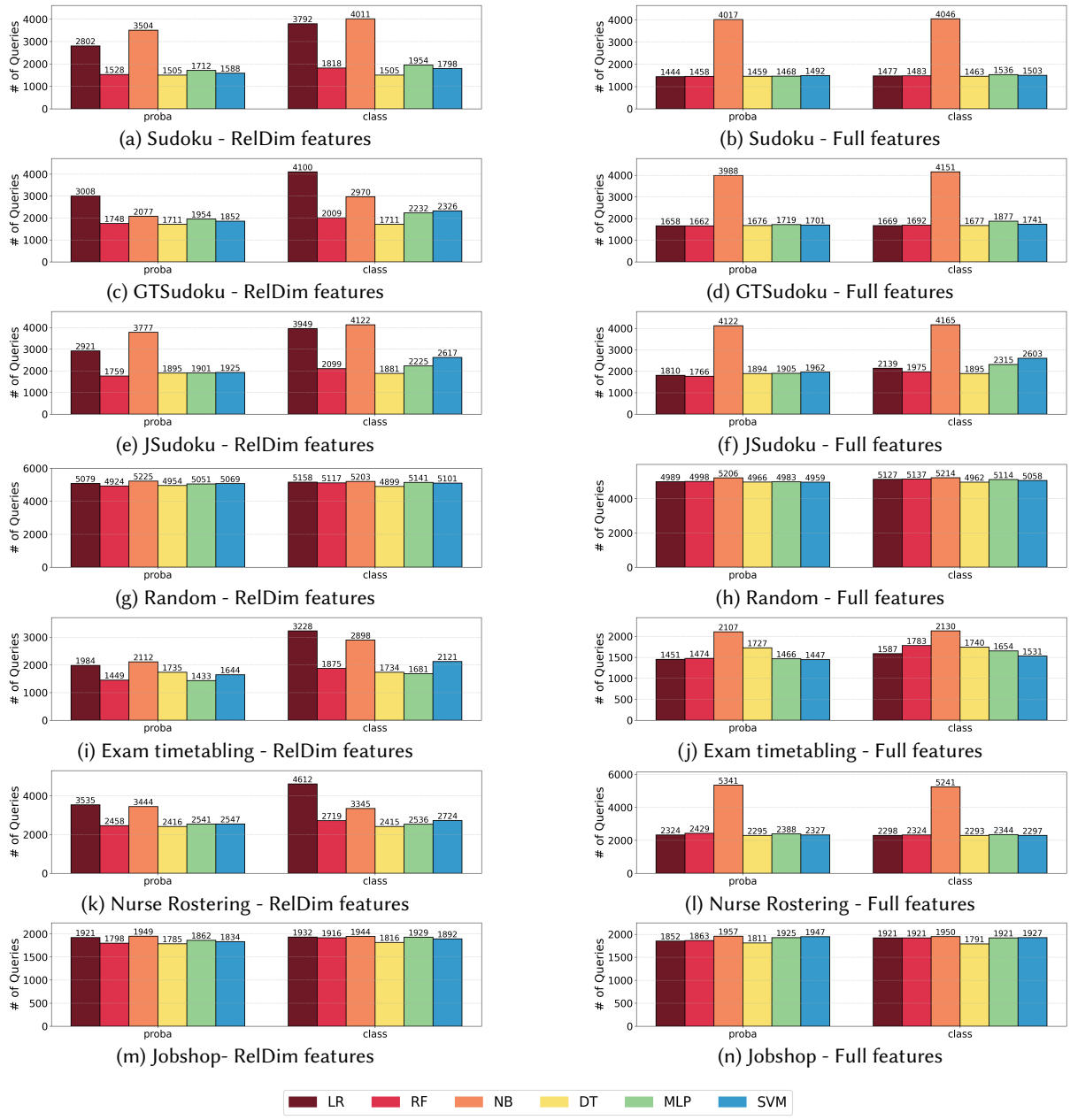


Fig. 12. Results when guiding using the predicted probabilities or the predicted class

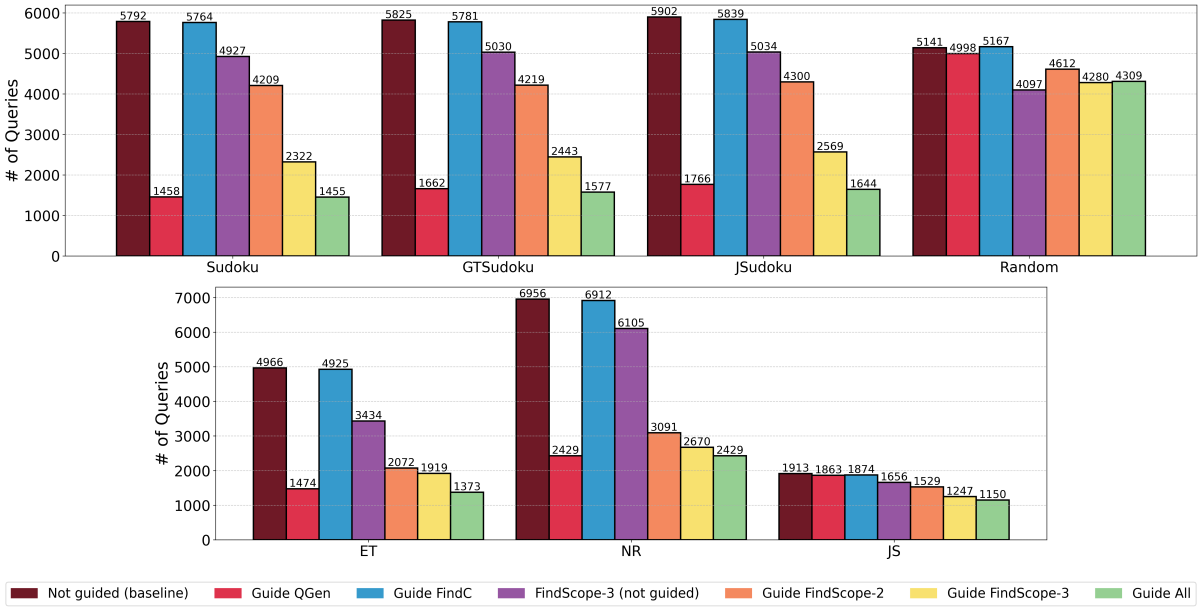


Fig. 13. Evaluating the effect of guiding each layer of interactive CA

Guiding FindC. On the other hand, guiding FindC does not offer similar advantages. Although the number of queries is reduced in all benchmarks (except Random), this reduction is only by 1-2%. This is due to the fact that the queries posted in FindC are typically a very small fraction of the total set of queries posted to the user, as most queries are asked in top-level and in FindScope. As a result, the reduction offered by FindC cannot have such a big impact.

Guiding all layers. Turning our attention to the results when guiding all layers of interactive CA, we can see a comparatively small but additional improvement compared to guiding only top-level query generation or only the FindScope component. A reduction in the number of queries is present in all benchmarks except Random, NR and Sudoku, where no performance improvement is observed. In the rest of the benchmarks, guiding all layers leads to an additional reduction of 6% in GTSudoku, 9% in JSudoku, 5% in ET and 7% in JS, compared to the best performing alternative. The improvement of the combination is relatively small because guiding top-level query generation or FindScope-3 leads to significant reductions already.

Overall, the combination of our methods significantly reduces the number of queries. Compared to the baseline, we can see an overall reduction of 75% in Sudoku, 73% in GTSudoku, 72% in Jsudoku, 16% in Random, 72% in ET, 65% in NR, and 40% in JobShop.

6 Related Work

CA methods can be broadly categorized into two categories: passive and active acquisition.

In passive CA, the system is given a pre-existing set of examples of solutions and possibly non-solutions. Based on these examples, the system learns a set of constraints modeling the problem (Beldiceanu and Simonis, 2012; Berden et al., 2022; Bessiere et al., 2004, 2005, 2017; Kumar et al., 2019, 2022; Lallouet et al., 2010; Lombardi et al., 2017; Bessiere et al., 2023a). These approaches differ mainly in the types of constraints they can learn and the methodologies they employ. Conacq.1 is a version space algorithm for learning fixed-arity constraints (Bessiere

et al., 2004, 2005, 2017), ModelSeeker learns global constraints that are taken from a predefined constraint catalog (Beldiceanu and Simonis, 2012), COUNT-CP is a generate-and-aggregate approach that can learn expressive first-order constraints (Kumar et al., 2022), while SeqAcq (Prestwich, 2020; Prestwich and Wilson, 2024) and BayesAcq (Prestwich et al., 2021) are statistical approaches robust to noise in the pre-existing set of examples.

In this paper we focus on interactive CA methods. In contrast to passive learning, *active* or *interactive* CA systems learn the constraints through interaction with the user, by asking queries that the user answers. 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 (Freuder and Wallace, 1998), where users, when they answer a membership query negatively, also have to provide a violated constraint as the reason for their answer. In order to lower the expertise level required from the user, Bessiere et al. later proposed Conacq.2 (Bessiere et al., 2007, 2017) – an active version of Conacq.1 that uses membership queries and does not require the user to provide any violated constraints. Conacq.2 was in turn extended to also accept arguments (Shchekotykhin and Friedrich, 2009) regarding *why* examples should be rejected or accepted.

A downside of Conacq.2 is that the number of membership queries needed can grow exponentially, as shown by Bessiere et al. (2017). This led to the development of a new family of interactive algorithms, which use *partial queries* instead (Arcangioli et al., 2016; Bessiere et al., 2013; Lazaar, 2021; Tsouros and Stergiou, 2020, 2021; Tsouros et al., 2019, 2020, 2018; Bessiere et al., 2023b). A partial query asks the user to classify a *partial* assignment to the variables. Using partial queries, CA systems are able to converge faster. The first system to use partial queries was called QuAcq (Bessiere et al., 2013, 2016). It starts by asking a top-level query, which, when answered negatively, is followed by a series of derived 'subqueries' asked in order to find an individual constraint that is violated by the original top-level query. This requires a number of subqueries logarithmic in the amount of variables considered. QuAcq was later extended into MultiAcq, which, instead of learning a single constraint per top-level query, learns *all* of the problem's constraints that are violated by the top-level query (Arcangioli et al., 2016). However, MultiAcq has the downside that, for each constraint learned, it requires a number of subqueries linear in the amount of variables. This led to the introduction of MQuAcq, which combines the best of both worlds: it learns all constraints violated by the top-level query in the original logarithmic complexity of QuAcq (Tsouros and Stergiou, 2020; Tsouros et al., 2018). MQuAcq was subsequently extended into MQuAcq2, which improves performance further by exploiting some of the structure present in the constraints already learned (Tsouros et al., 2019). Concretely, it tries to detect incomplete cliques in the constraint model learned thus far, with the hope of then completing these cliques efficiently through targeted partial queries. Furthermore, (Bessiere et al., 2023b) proposed QuAcq2, further improving the algorithmic procedure of finding individual constraints, avoiding redundant queries, and allowing the acquisition of non-normalized constraint networks. Finally, Tsouros et al. (2023a) proposed a meta-algorithm called GrowAcq, which iteratively calls any of the active algorithms discussed above on an increasingly large subset of variables. This approach significantly improves efficiency by allowing the use of larger sets of candidate constraints.

Besides (partial) membership queries, alternative types of queries have also been introduced in the literature. Aiming to weaken the assumptions on the user's ability to answer all queries, Tsouros et al. (2020) introduced the *limited membership query*, allowing omissions in the user's answers. Other types of queries aim to allow more expressiveness in the feedback provided by the user. For example, Bessiere et al. (2014) proposed the use of *generalization queries*, asking the user whether a given pattern of constraints can be generalized on some groups of variables. Daoudi et al. (2015) showed how to extract such groups or types of variables during the acquisition process, in order to assist in generalization, using information derived from the learned constraints. Finally, Daoudi et al. (2016) proposed to ask the user directly whether a specific constraint belongs to the target constraint model, using *recommendation queries*. To find which constraints to recommend, techniques from link prediction (Daoudi et al., 2016) or ML classification (Islah and Mechqrane, 2024) have been used. Following a

similar intuition, instead of only using ML classification for recommendation queries, based on properties of the current constraint network, in this work we show how to guide (partial) membership queries based on properties of learned and excluded constraints. Finally, recently a novel approach using LLMs in CA was introduced with the LLMACQ system (Mechqrane et al., 2024). This approach allows the user to give natural language feedback to the system on why they rejected an example, after which an LLM is used to extract potential constraints from this feedback.

More related to our paper are works that have focused on efficient query generation. These works aim to minimize query generation times while maintaining convergence guarantees (Addi et al., 2018; Addi and Ezzahir, 2019; Tsouros and Stergiou, 2020; Bessiere et al., 2023b; Tsouros et al., 2023a). TQ-GEN (Addi et al., 2018; Addi and Ezzahir, 2019) was introduced to improve query generation times, posing time-limits and trying to generate queries in different parts of the problem if it fails, while PQ-GEN (Tsouros et al., 2023a) projects the query generation problem to only the relevant variables, in order to avoid indirect implications of candidate constraints and speed-up the process. Query generation based on custom solvers, that allow returning incomplete assignments (i.e., non-leaf nodes of the search tree) was also proposed (Tsouros and Stergiou, 2020; Bessiere et al., 2023b). Besides trying to speed-up the generation of informative examples, recent works (Tsouros and Stergiou, 2020; Bessiere et al., 2023b) have also focused on how to generate better queries. Bessiere et al. (2013) and Tsouros and Stergiou (2020) proposed to maximize the violation of candidate constraints, while Bessiere et al. (2023b) showed that maximizing the variables included in the posted queries can be beneficial when a custom solver is used. However, these works do not use information acquired during the learning process to guide the query generation. In contrast to these works, we leverage patterns detected in the learned and excluded constraints to guide query generation, using ML classification for pattern detection. Finally, we show how to guide the queries in all layers of interactive CA systems, not only focusing on top-level queries.

Since our method focuses on guiding query generation at all layers of the acquisition process, it is agnostic to the overarching CA algorithm used, or specific query generation strategy – as long as the system allows sufficient time for optimization in query generation. All interactive CA algorithms that use (partial) membership queries, possibly in addition to other types of queries, and that are based on a set of candidate constraints, can benefit from our proposed methods. Thus, our methods are orthogonal to the advances in the algorithmic procedures described above.

7 Conclusions

One bottleneck of major importance in interactive CA is the number of queries needed to converge. The search-based learning used in interactive CA is *uninformed*: it does not use information about which constraints have already been learned or ruled out. In this work, we tighten the connection between ML and CA, by using, for the first time, statistical ML methods that can learn during the acquisition process, and that can predict whether a candidate constraint is likely to be part of the problem or not. We propose to use probabilistic classification, and to use the predictions from the ML models to guide the search process of CA. To do so, we propose an ML-guided objective function for the query generation problem in all layers of interactive CA, as well as a new FINDSCOPE function that allows for easier guiding. We also propose an expressive feature representation of constraints for the ML classifiers to learn over. Our experimental evaluation shows that the combination of our methods greatly outperforms the state of the art, with the number of queries being decreased by up to 75%. These findings confirm that using statistical ML methods can indeed detect patterns in constraint models, while they are being learned, and that this can enable informed search within interactive CA.

Future work could investigate the use of online learning in this setting, as data becomes available gradually, whereas we currently retrain the classifier from scratch each time. Other opportunities include learning a prior distribution over constraints and transfer learning across different problems, as well as using automated

representation learning rather than our generic but hand-crafted features. We also think that the closer integration with statistical ML techniques can be a stepping stone towards handling inconsistent and incorrect answers from the user. Furthermore, the ML-based implicit pattern detection can also be exploited for generalization during interactive CA, aiming to generalize the patterns detected in a small part of the problem to the full problem early on. Finally, extending interactive CA systems to also be able to learn global constraints and n-ary constraints with constants, such as linear inequalities, is important for expanding the reach of problems learnable.

Acknowledgments

This research received funding from the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt); from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme from the Research Foundation-Flanders (FWO) (Grant No. 11PQ024N); from the framework of H.F.R.I call “4th Call for H.F.R.I.’s Research Projects to Support Postdoctoral Researchers” (H.F.R.I. Project Number: 28553).

References

Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. 2018. Time-Bounded Query Generator for Constraint Acquisition. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10848)*, Willem Jan van Hoeve (Ed.). Springer, 1–17. https://doi.org/10.1007/978-3-319-93031-2_1

Hajar Ait Addi and Redouane Ezzahir. 2019. $$P_a$$-quacq: Algorithm for Constraint Acquisition System. In *Smart Data and Computational Intelligence*. Springer International Publishing, 249–256.

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

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

Hugo Barral, Mohamed Gaha, Amira Dems, Alain Côté, Franklin Nguewouo, and Quentin Cappart. 2024. Acquiring Constraints for a Non-linear Transmission Maintenance Scheduling Problem. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14742)*, Bistra Dilkina (Ed.). Springer, 34–50. https://doi.org/10.1007/978-3-031-60597-0_3

Nicolas Beldiceanu and Helmut Simonis. 2012. A Model Seeker: Extracting Global Constraint Models from Positive Examples. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7514)*, Michela Milano (Ed.). Springer, 141–157. https://doi.org/10.1007/978-3-642-33558-7_13

Senne Berden, Mohit Kumar, Samuel Kolb, and Tias Guns. 2022. 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, Haifa, Israel, July 31 - August 8, 2022 (LIPIcs, Vol. 235)*, Christine Solnon (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:17. https://doi.org/10.4230/LIPICS.CP.2022.8

Christian Bessiere, Clément Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C. Tsouros, and Toby Walsh. 2023b. Learning constraints through partial queries. *Artif. Intell.* 319 (2023), 103896. https://doi.org/10.1016/J.ARTINT.2023.103896

Christian Bessiere, Clément Carbonnel, and Areski Himeur. 2023a. Learning Constraint Networks over Unknown Constraint Languages. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 1876–1883. https://doi.org/10.24963/IJCAI.

2023/208

Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. 2014. Boosting Constraint Acquisition via Generalization Queries. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014) (Frontiers in Artificial Intelligence and Applications, Vol. 263)*, Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan (Eds.). IOS Press, 99–104. https://doi.org/10.3233/978-1-61499-419-0-99

Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. 2004. Leveraging the Learning Power of Examples in Automated Constraint Acquisition. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3258)*, Mark Wallace (Ed.). Springer, 123–137. https://doi.org/10.1007/978-3-540-30201-8_12

Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. 2013. Constraint Acquisition via Partial Queries. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, Francesca Rossi (Ed.). IJCAI/AAAI, 475–481. http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6659

Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. 2005. A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3720)*, João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo (Eds.). Springer, 23–34. https://doi.org/10.1007/11564096_8

Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. 2007. Query-Driven Constraint Acquisition. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, Manuela M. Veloso (Ed.). 50–55. http://ijcai.org/Proceedings/07/Papers/006.pdf

Christian Bessiere, Abderrazak Daoudi, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Younes Mechqrane, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. 2016. New Approaches to Constraint Acquisition. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi (Eds.). Lecture Notes in Computer Science, Vol. 10101. Springer, 51–76. https://doi.org/10.1007/978-3-319-50137-6_3

Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. 2017. Constraint acquisition. *Artif. Intell.* 244 (2017), 315–342. https://doi.org/10.1016/J.ARTINT.2015.08.001

Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, Christian Bessiere, and El-Houssine Bouyakhf. 2015. 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, November 9-11, 2015*. IEEE Computer Society, 413–420. https://doi.org/10.1109/ICTAI.2015.69

Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El Houssine Bouyakhf. 2016. Constraint Acquisition Using Recommendation Queries. In *IJCAI: International Joint Conference on Artificial Intelligence*. 720–726.

Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. 2001. Matrix modelling. In *Proc. of the CP-01 Workshop on Modelling and Problem Formulation*, Vol. 223.

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

Eugene C. Freuder. 2018. Progress towards the Holy Grail. *Constraints An Int. J.* 23, 2 (2018), 158–171. https://doi.org/10.1007/S10601-017-9275-0

Eugene C. Freuder and Barry O'Sullivan. 2014. Grand challenges for constraint programming. *Constraints An Int. J.* 19, 2 (2014), 150–162. https://doi.org/10.1007/S10601-013-9155-1

Eugene C. Freuder and Richard J. Wallace. 1998. Suggestion Strategies for Constraint-Based Matchmaker Agents. In *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1520)*, Michael J. Maher and Jean-Francois Puget (Eds.). Springer, 192–204. https://doi.org/10.1007/3-540-49481-2_15

Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. 2008. Essence: A constraint language for specifying combinatorial problems. *Constraints* 13 (2008), 268–306.

Tias Guns. 2019. 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)*, Vol. 19.

Hamza Islah and Younes Mechqrane. 2024. Machine Learning Based Recommendation Queries for Constraint Acquisition. In *36th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2024, Herndon, VA, USA, October 28-30, 2024*. IEEE, 159–165. https://doi.org/10.1109/ICTAI62512.2024.00031

Serdar Kadıoğlu, Parag Pravin Dakle, Karthik Uppuluri, Regina Politi, Preethi Raghavan, SaiKrishna Rallabandi, and Ravisutha Srinivasamurthy. 2024. Ner4Opt: named entity recognition for optimization modelling from natural language. *Constraints* 29, 3 (2024), 261–299.

Mohit Kumar, Samuel Kolb, and Tias Guns. 2022. Learning Constraint Programming Models from Data Using Generate-And-Aggregate. In *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, Haifa, Israel, July 31 - August 8, 2022 (LIPIcs, Vol. 235)*, Christine Solnon (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:16. https://doi.org/10.4230/LIPICS.CP.2022.29

Mohit Kumar, Stefano Teso, and Luc De Raedt. 2019. Acquiring Integer Programs from Data. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 1130–1136. https://doi.org/10.24963/IJCAI.2019/158

Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. 2010. On Learning Constraint Problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*. IEEE Computer Society, 45–52. https://doi.org/10.1109/ICTAI.2010.16

Nadjib Lazaar. 2021. Parallel Constraint Acquisition. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 3860–3867. https://doi.org/10.1609/AAAI.V35I5.16504

Michele Lombardi, Michela Milano, and Andrea Bartolini. 2017. Empirical decision model learning. *Artif. Intell.* 244 (2017), 343–367. https://doi.org/10.1016/J.ARTINT.2016.01.005

Younes Mechqrane, Christian Bessiere, and Ismail Elabbassi. 2024. Using Large Language Models to Improve Query-based Constraint Acquisition. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 1916–1925. https://www.ijcai.org/proceedings/2024/212

Kostis Michailidis, Dimos Tsouros, and Tias Guns. 2024. Constraint Modelling with LLMs Using In-Context Learning. In *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024 (LIPIcs, Vol. 307)*, Paul Shaw (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:27. https://doi.org/10.4230/LIPICS.CP.2024.20

Tom Mitchell. 1997. Concept learning and the general-to-specific ordering. *Machine Learning* (1997), 20–51.

Tom Michael Mitchell. 1978. *Version spaces: an approach to concept learning*. Technical Report. STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE.

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4741)*, Christian Bessiere (Ed.). Springer, 529–543. https://doi.org/10.1007/978-3-

540-74970-7_38

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. https://doi.org/10.5555/1953048.2078195

Laurent Perron, Frédéric Didier, and Steven Gay. 2023. The CP-SAT-LP Solver (Invited Talk). In *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, Toronto, Canada, August 27-31, 2023 (LIPIcs, Vol. 280)*, Roland H. C. Yap (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:2. https://doi.org/10.4230/LIPICS.CP.2023.3

Steve Prestwich. 2020. Robust Constraint Acquisition by Sequential Analysis. In *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020) (Frontiers in Artificial Intelligence and Applications, Vol. 325)*, Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang (Eds.). IOS Press, 355–362. https://doi.org/10.3233/FAIA200113

Steven D. Prestwich, Eugene C. Freuder, Barry O'Sullivan, and David Browne. 2021. Classifier-based constraint acquisition. *Ann. Math. Artif. Intell.* 89, 7 (2021), 655–674. https://doi.org/10.1007/S10472-021-09736-4

Steven D. Prestwich and Nic Wilson. 2024. A statistical approach to learning constraints. *Int. J. Approx. Reason.* 171 (2024), 109184. https://doi.org/10.1016/J.IJAR.2024.109184

Kostyantyn M. Shchekotykhin and Gerhard Friedrich. 2009. Argumentation Based Constraint Acquisition. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, Wei Wang, Hillol Kargupta, Sanjay Ranka, Philip S. Yu, and Xindong Wu (Eds.). IEEE Computer Society, 476–482. https://doi.org/10.1109/ICDM.2009.62

Dimos Tsouros and Tias Guns. 2025. A CPMpy-based Python library for Constraint Acquisition - PyConA. In *Proc. AAAI 2025 Bridge on Constraint Programming and Machine Learning (CPML)*.

Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. 2023a. Guided Bottom-Up Interactive Constraint Acquisition. In *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, Toronto, Canada, August 27-31, 2023 (LIPIcs, Vol. 280)*, Roland H. C. Yap (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:20. https://doi.org/10.4230/LIPICS.CP.2023.36

Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. 2024. Learning to Learn in Interactive Constraint Acquisition. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 8154–8162. https://doi.org/10.1609/AAAI.V38I8.28655

Dimosthenis C. Tsouros and Kostas Stergiou. 2020. Efficient multiple constraint acquisition. *Constraints An Int. J.* 25, 3-4 (2020), 180–225. https://doi.org/10.1007/S10601-020-09311-4

Dimosthenis C. Tsouros and Kostas Stergiou. 2021. Learning Max-CSPs via Active Constraint Acquisition. In *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021 (LIPIcs, Vol. 210)*, Laurent D. Michel (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 54:1–54:18. https://doi.org/10.4230/LIPICS.CP.2021.54

Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. 2019. Structure-Driven Multiple Constraint Acquisition. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11802)*, Thomas Schiex and Simon de Givry (Eds.). Springer, 709–725. https://doi.org/10.1007/978-3-030-30048-7_41

Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. 2020. Omissions in Constraint Acquisition. In *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve,*

Belgium, September 7-11, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12333), Helmut Simonis (Ed.). Springer, 935–951. https://doi.org/10.1007/978-3-030-58475-7_54

Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. 2018. Efficient Methods for Constraint Acquisition. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11008)*, John N. Hooker (Ed.). Springer, 373–388. https://doi.org/10.1007/978-3-319-98334-9_25

Dimosthenis C. Tsouros, Hélène Verhaeghe, Serdar Kadioglu, and Tias Guns. 2023b. Holy Grail 2.0: From Natural Language to Constraint Models. *CoRR* abs/2308.01589 (2023). https://doi.org/10.48550/ARXIV.2308.01589 arXiv:2308.01589

A Tuning Details

We used RF and NB in their default settings, while we tuned the most important hyperparameters for LR, MLP and SVM. We tuned the classifiers for each separate feature representation, based on their average performance on all benchmarks. For tuning, we used the final dataset for all benchmarks, having labeled all candidate constraints. A grid search, coupled with 10-fold cross-validation, was conducted, using balanced accuracy as the metric to address class imbalance. Hyperparameter combinations surpassing a 10-second training time were omitted to ensure relevance in interactive scenarios, unless there was no hyperparameter combination under that limit, in which case we chose the fastest one.

For **logistic regression**, we explored:

- regularization strength $C \in \{0.1, 1.0, 10.0, 100.0\}$,
- penalties penalty $\in \{\ell_1, \ell_2\}$,
- solvers $\in \{$`liblinear`, `saga`$\}$, and
- maximum iteration counts up to 5000.

The best-performing configuration was $C = 100$, penalty $= \ell_2$, solver=`liblinear`, and max_iter = 2000.

For **support vector machines**, we considered:

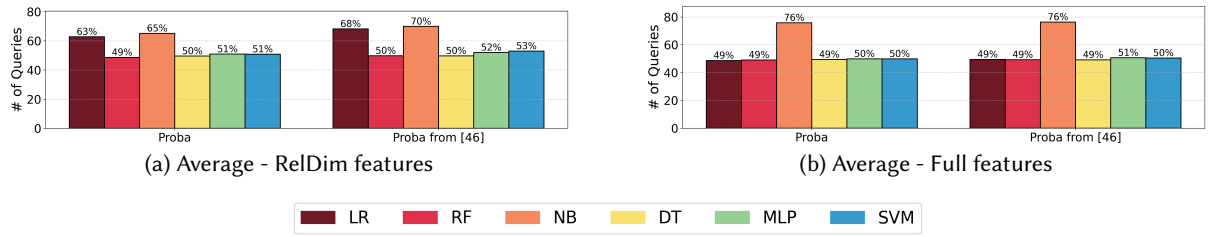
- kernels $\in$ {linear, polynomial, RBF, sigmoid},
- regularization parameter $C \in \{0.1, 1.0, 10.0, 100.0\}$,
- kernel coefficient $\gamma \in \{$`scale`, `auto`$, 0.1, 0.01, 0.001, 0.0001\}$,
- and the use of class balancing and probabilistic outputs.

The optimal configurations varied by feature representation. For the Full feature representation and Rel, the best model used a linear kernel with $C = 0.1$, $\gamma = 0.1$, and balanced class weights. For RelDim, the best model used an RBF kernel with $C = 100$ and $\gamma = 0.1$.

For **MLPs**, we tuned:

- the number and size of hidden layers (single and double-layer networks with 8 to 64 neurons per layer),
- activation functions $\in \{$`relu`, `tanh`$\}$,
- L2 regularization $\alpha \in \{0.0001, 0.001, 0.01, 0.1\}$, and
- learning rate $\in \{0.001, 0.01, 0.1\}$.

The best MLP configuration also depended on the feature representation. For Full and Rel, the optimal model used one hidden layer with 8 neurons, `ReLU` activation, $\alpha = 0.01$, and a learning rate of 0.1. For Rel, the best configuration used two hidden layers with 8 neurons each, `tanh` activation, $\alpha = 0.0001$, and a learning rate of 0.01.



(a) Average - RelDim features

(b) Average - Full features

Fig. 14. Average results on all benchmarks comparing the use of probabilities in the objective against the use of the predicted class

B Objective Function Ablation

We now compare our new probability-based approximation of the oracle with the one we presented in (Tsouros et al., 2023a).

To evaluate this, we did the same experiment as in Q3, using the probabilities as in the oracle from (Tsouros et al., 2023a), as follows:

$$O_{orig}(c) = [1/P(c \in C_T) \leq \log(|Y|)]$$

We compare this with our proposed approach, which uses the predicted probability directly as the oracle:

$$O = P(c \in C_T)$$

Figure 14 presents the average results on all benchmarks when using the different classifiers, comparing the use of the probabilities versus the use of the predicted class. We present results on each benchmark separately in Figure 15. As in Q3, we do not present results for the Rel feature representation, as it is already shown that it offers additional benefits to not guiding, but it can offer limited benefits. We focus on RelDim (left column) and Full (right column) features, to evaluate how well guiding performs when the predictions of the classifiers are of different quality.

Received 13 June 2025; accepted 11 March 2026

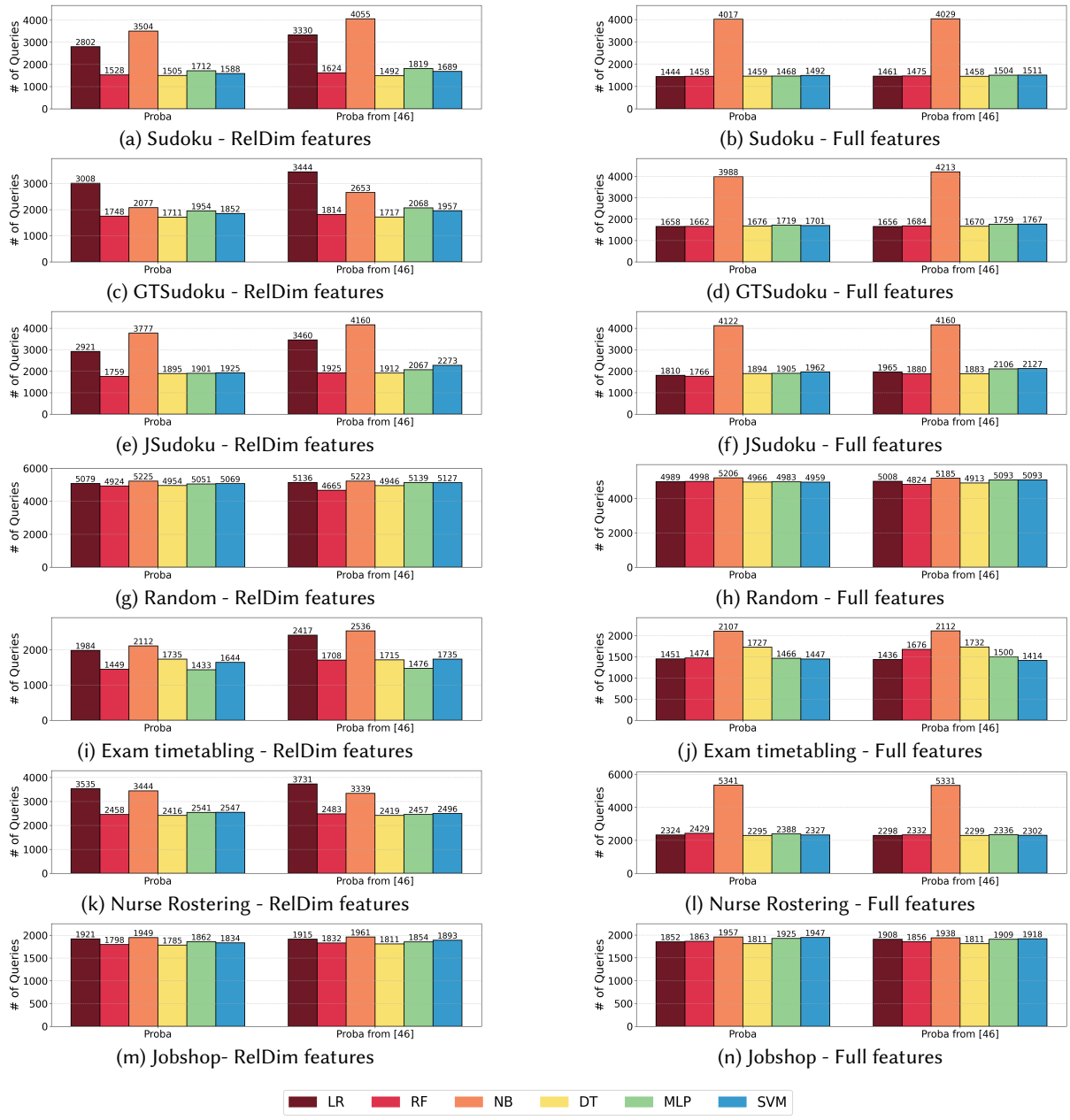


Fig. 15. Comparing the use of our probabilistic oracle in the objective against using the oracle from (Tsouros et al., 2023a)