International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

Enhancing Constraint Acquisition through Hybrid Learning: An Integration of Passive and Active Learning Strategies

Vasileios Balafas
Department of Electrical and Computer Engineering, University of Western Macedonia
Campus ZEP, Kozani, 50100, Greece
v.balafas@uowm.gr

Dimosthenis C. Tsouros
Department of Computer Science, KU Leuven
Celestijnenlaan 200a, Leuven, 3001, Belgium
dimos.tsouros@kuleuven.be

Nikolaos Ploskas
Department of Electrical and Computer Engineering, University of Western Macedonia
Campus ZEP, Kozani, 50100, Greece
nploskas@uowm.gr

Kostas Stergiou
Department of Electrical and Computer Engineering, University of Western Macedonia
Campus ZEP, Kozani, 50100, Greece
kstergiou@uowm.gr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Constraint Programming (CP) is a successful methodology for solving combinatorial problems from various domains. Efficiently modeling the problem at hand as a Constraint Satisfaction Problem is a crucial, but difficult task in CP. Towards this, a recent approach that is attracting increasing interest is Constraint Acquisition, i.e. the (semi)automatic learning of constraints through examples of solutions and non-solutions. This paper introduces a hybrid methodology that combines passive and active learning strategies to acquire both global and fixed arity constraints. This hybrid approach leverages the strengths of both techniques to address their individual limitations. Passive learning rapidly learns constraints from example solutions, while active learning refines and contextualizes constraints through user interaction. The core of the methodology consists of a passive learning module where subsets of variables are compared against global constraints and are validated using a CP solver. Constraints consistently present across multiple solution sets are identified as global constraints that belong to the model. Then, fixed arity constraints are refined through an active learning module with user input. Experiments across various problem types, from simple to complex, demonstrate the efficiency of the proposed hybrid methodology.

Keywords: Constraint Programming; Constraint Acquisition; Modeling

1. Introduction

Constraint Programming (CP) is a successful methodology for solving combinatorial problems from various domains in AI, OR, and computer science, such as scheduling, resource allocation, and planning. Efficiently modeling the problem at hand as a Constraint Satisfaction Problem (CSP) is a crucial, but difficult task in CP. A CSP is modeled by defining variables, domains, and constraints. Variables represent the elements of the problem, domains define possible values for the variables, and constraints describe relationships between variables.

One of the main challenges in CP resides in the modeling phase, where precisely defining the constraints is imperative yet laborious and error-prone because it demands a high level of expertise from the modeler's part. Hence, (semi-)automated modeling is a research direction that is attracting a lot of interest, as it promises to drastically reduce the required expert intervention. Constraint Acquisition (CA) is one of the most promising approaches to (semi-)automated modeling.[1, 2]

There are two types of learning approaches in CA. The first is passive learning (PL), a technique that captures constraints by observing and analyzing example solutions without requiring explicit user input. These solutions can come from previously solved instances of a problem, from historical datasets that contain records of how similar problems were solved in the past, or they can be generated by simulating the problem scenario. PL mainly focuses on acquiring global constraints by analyzing the available solutions and extracting common patterns among them.

On the other hand, there is active learning (AL), a technique that requires explicit user input to acquire constraints. This input can be obtained by asking the user, which may be a human or a software tool, to provide feedback on whether an example (i.e. a set of variable assignments) is a solution or not. AL is useful for acquiring fixed arity constraints, such as binary constraints, and thus can be used to complete a basic model that has been acquired through PL or other means.

However, passive and active acquisition techniques have their respective limitations. PL can use existing data to extract constraints, but in practice it cannot converge to the target model but only approximate it, because it requires an exponential number of examples in order to converge.[3] Thus, the extracted model highly depends on the quality and quantity of the existing data. On the other hand, AL systems can converge to the target model that the user has in mind (under some assumptions), but the amount of interaction, i.e. queries that have to be answered, is quite high for human users. In addition, AL systems are, so far, unable to handle global constraints because to do so, the set of constraints given as input to the AL system, known as the *bias*, would need to have an exponential size.

The existing literature in CA either proposes only PL methods[3–7] or only AL methods,[8–11] but no study combines these approaches to leverage their respective strengths. In this paper, we present a novel hybrid CA method that integrates both PL and AL techniques, resulting in a more efficient and effective constraint-capturing method.

Specifically, PL is first used to identify global constraints in the model. This is done by extracting subsets of variables from the available solutions and validating these against a list of global constraints that may potentially be present in the model, following the way in which the PL system ModelSeeker operates.[4] A novelty in our approach to PL, is that apart from the known set of solutions and the list of global constraints, the PL module is also given as input the full bias of fixed arity constraints that are typically only used by AL systems. As the available solutions are processed, this bias is filtered, removing constraints that violate solutions.

Once all the solutions have been processed and global constraints have been learned, the AL module takes over in order to prove that specific fixed arity constraints belong to the problem. Crucially, the bias that is given as input to the AL module, may now be of (much) smaller size, resulting in a significant reduction in the number of queries that need to be posted to the user. Hence, we obtain a tool that can learn global constraints and can acquire fixed arity constraints that provably belong to the target model, requiring less user involvement than standard AL methods.

To evaluate the effectiveness of the hybrid CA tool, we ran experiments on four different types of problems. Specifically, Sudoku 9×9, GreaterThanSudoku 9×9, the Warehouse Location Problem, and the Virtual Machines to Physical Machines allocation problem in Data Centers. The experimental results demonstrate that our method can learn all the constraints of these problems, significantly reducing the number of queries and the runtime required, compared to running a state-of-the-art AL method on its own. In addition, the hybrid method is able to acquire all the binary constraints that are present in the models, whereas the PL method fails to achieve this when run on its own.

The rest of the paper is structured as follows. Some background on CP and CA is given in Section 2. In Section 3, we review the related work. In Section 4, we present the proposed approach and the hybrid CA tool. An experimental evaluation is given in Section 5. Finally, Section 6 concludes the paper.

2. Background

2.1. *Fundamentals of CP*

Constraint Programming (CP) is a declarative programming paradigm for modeling and solving combinatorial problems with constraints. In CP, problems are modeled as *Constraint Satisfaction Problems (CSPs)*. A CSP[12] is a triple $P = (X, D, C)$, which includes:

- a set of n variables $X = \{x_1, x_2, \cdots, x_n\}$, which denotes the problem's components,
- a set of n domains $D = \{D_1, D_2, \cdots, D_n\}$, where $D_i \subset \mathbb{Z}$ represents the finite set of possible values for variable x_i,
- a set of constraints $C = \{c_1, c_2, \cdots, c_m\}$.

A *constraint* c can be defined as a pair $(var(c), rel(c))$, where $var(c)$ is a subset of X called the *scope* of the constraint, while $rel(c)$ is a relation over the domains of the variables contained in $var(c)$, specifying the permissible assignments. The *arity* of the constraint is given by $|var(c)|$. A *solution* $S = \{s_{x_1}, s_{x_2}, \cdots, s_{x_n}\}$ to a CSP is an assignment of values to all variables, where $s_{x_i} \in D_i$ for $i = 1, 2, \ldots, n$ and for all $c_j \in C$, the assignment S satisfies c_j. The constraint set $C[Y]$, where $Y \subseteq X$, refers to the set of constraints in C with a scope that is a subset of Y. The solution set of a constraint set C includes all variable assignments that satisfy the constraints in C and is denoted by $sol(C)$. A constraint $c \in C$ is considered *redundant* or *implied* if it satisfies the condition $sol(C) = sol(C \setminus c)$. That is, removing c from the set of constraints does not change the solution set.

2.2. *Fixed arity and global constraints*

Fixed arity constraints are mathematical or logical expressions, or other relations, that involve a fixed, usually small, number of variables. Fixed arity constraints are called *binary* if they involve two variables. Typically, fixed arity constraints encompass a range of mathematical operators, such as inequality ($\neq$), equality ($=$), greater than ($>$), less than ($<$), less than or equal to ($\leq$), and greater than or equal to ($\geq$). For example, $x_i \neq x_j$ is a common binary constraint.

Global constraints are constraints over sequences of variables with no fixed arity[13] and typically have more complex semantics than fixed arity constraints. They are called "global" because they involve a global view of the problem, taking into account the relationships over sequences of variables. Global constraints are an important tool in CP for modeling and solving combinatorial optimization problems. Global constraints are defined in terms of a set of parameters that specify their semantics. These parameters include the set of variables involved in the constraint, their domains, and any additional constraints or relationships between them.

Numerous global constraints have been defined, and most of them are described in the ever-expanding global constraints catalog.[14] So far, our hybrid CA system is designed to handle, and potentially learn, the following global constraints (details about them can be found in the on-line catalog): (1) `allDifferent`, (2) `arithm`, (3) `element`, (4) `allEqual`, (5) `notAllEqual`, (6) `circuit`, (7) `atMostNValues`, (8) `atLeastNValues`, (9) `count`, (10) `sum`, (11) `max`, (12) `min`, (13) `lexChainLess`, (14) `lexChainLessEq`, (15) `symmetric`, (16) `allDisjoint`, (17) `nValues`, (18) `tree`. For this study, we define the constraints `allDifferent`, `count`, and `sum`, as they are the ones present in the problems used in our experimental study:

alldifferent Let $X = \{x_1, x_2, \cdots, x_n\}$ be a set of variables. An *alldifferent* constraint on X requires that for any two variables x_i and x_j in X where $i \neq j$, $x_i \neq x_j$. Such a constraint is semantically equivalent to the set of $n \times (n-1)/2$ binary $\neq$ constraints on all pairs of the n variables. However, the non-binary representation as a single *alldifferent* is preferable to its *decomposition* into binary constraints.[15,16]

Count The *count* constraint,[17] denoted as $\text{Count}(X, v, c)$, is a global constraint that involves a set of variables $X = \{x_1, x_2, \cdots, x_n\}$ and two given values v and c. The constraint specifies that exactly c variables from the set X should take the value v.

Sum The *sum* constraint is a global constraint that involves a set of variables and a target sum, which can be either a constant or a variable. The *sum* constraint enforces that the sum of the variables' values is different, less than, equal to, or greater than the target sum. This can be expressed as:

$$\text{sum}(\{x_1, ..., x_n\}, \textit{operator}, z)$$

where *operator* can be any of $\neq$, $=$, $>$, $<$, $\leq$, $\geq$.

2.3. *Constraint Acquisition*

Constraint Acquisition (*CA*) can be defined as the process of learning a set of constraints C from a given set of examples E, where each example $e \in E$ is a tuple of variable assignments and is labeled as either *positive* (satisfying the constraints) or *negative* (violating the constraints).[3,18] In CA, the set of variables and domains (X, D) is referred to as the *vocabulary* of the current problem and is the common knowledge shared by the user and the system.

A (partial) example e_Y is an assignment on a set of variables $Y \subseteq X$. e_Y is rejected by a constraint c if $\text{scope}(c) \subseteq Y$ and the projection $e_{scope(c)}$ of e_Y on the variables in $\text{scope}(c)$ is not in $\text{rel}(c)$. An assignment e_Y is a partial solution if it is accepted by all the constraints $c \in C$ where $scope(c) \subseteq Y$. A complete assignment that is accepted by all the constraints in C is a solution to the problem. A partial assignment e_Y, which is accepted by C, is not necessarily part of a complete solution.

We say that the CA process has *converged* on the learned network $C_L \subseteq B$ if C_L agrees with E, i.e. satisfies all examples in E, and for every other network $C \subseteq B$ that agrees with E, we have $sol(C) = sol(C_L)$. It is possible that the learned network C_L may not correspond to the network that the user has in mind. The target network C_{T} represents all the constraints that are expected to accurately capture the relationships between the variables in the problem.

In *active acquisition*, the learner interacts with the user dynamically while acquiring the constraint network. In such systems, the basic *query* is to ask the user to classify an example as a solution or not a solution. This "yes/no" type of question is called a (complete) *membership query*[19] $ASK(e_X)$, asking the user whether an assignment e_X is a solution to C_{T}. A *partial* query asks the user to classify a partial example as a partial solution or not.

In active CA, apart from the vocabulary, the learner also receives a *language* Γ comprising *fixed arity* constraint relations such as $\neq$, $=$, $>$, $<$, $\geq$, and $\leq$. Utilizing the vocabulary (X, D) and the constraint language Γ, AL systems formulate the *constraint bias* B by generating a constraint for each relation in Γ and each fixed arity combination of variables. This bias includes all potential fixed arity (usually binary) constraints relevant to the problem. We denote this constraint bias as B, which represents the set of all potential fixed arity constraints for the problem. In some cases, *background knowledge* may be available to the AL system. That is, the learned network C_L could be initialized with constraints that are already known to the user.

Example 2.1.

Consider a problem consisting of 8 variables with domains $\{1, ..., 8\}$. The vocabulary (X, D) given to the system would be $X = \{x_1, ..., x_8\}$ and $D = \{D(x_1), ..., D(x_8)\}$ with $D(x_i) = \{1, ..., 8\}$, as all the variables have the same domain. Assume that the problem the user has in mind has to satisfy the constraints $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_3 \neq x_4$. So, the target network C_T of this problem would be the set $\{x_1 \neq x_2, x_1 \neq x_3, x_3 \neq x_4\}$. Also, for simplicity assume that the language (Γ) given to the system by the user contains only the binary relation $\{\neq\}$. In this case, the bias B would contain the given relation for all the possible scopes. As it is a binary relation, $B = \{x_i \neq x_j \mid 1 \leq i < 8 \wedge i < j \leq 8, i \neq j\}$.

If an example $e = \{1, 1, 1, 2, 3, 4, 5, 6\}$ is posted to the user to be classified as positive or negative then ASK(e) will be a complete membership query. If only a partial assignment, for instance $e_Y = \{1, 1, 1, 2, -, -, -, -\}$ with $Y = \{x_1, x_2, x_3, x_4\}$, is posted to the user then e_Y will be a partial query. The answer of the user in both cases will be negative, as there are constraints of C_T that are violated.

In *passive acquisition*, the learner receives a set of examples and a list of, typically, global constraints. The input examples may be unstructured, but the common assumption in the literature is that they are arranged in a (partially) structured way,[4] such as a 2-dimensional array or a list. A *pattern* refers to a recurrent or characteristic arrangement, relationship, or sequence observed within a specific set of variables. For example, in a Sudoku puzzle, a pattern might be observed in the variables composing a row, a column, or a block. *Candidate constraints* are potential relationships or conditions between variables that might hold true within a given problem domain. In the context of CA, they are tentative constraints postulated based on available information, but their validity has not been conclusively determined. Candidate constraints are generated by matching the available global constraints to patterns of variables.

For example, given a set of 9 variables $\mathcal{X} = \{x_1, x_2, \ldots, x_9\}$ arranged in a 3 $\times$ 3 matrix, the arrangements of variables in rows and columns are patterns that can be investigated to identify constraints. Assuming that the only available global constraint is *alldifferent* then the set of candidate constraints would include the six *alldifferent* constraints on the rows and columns of the matrix (i.e. *alldifferent*(x_1, x_2, x_3), *alldifferent*(x_4, x_5, x_6), etc.).

3. Related Work

We now provide an overview of existing works in both passive and active CA.

3.1. *Passive learning algorithms*

In machine learning, PL algorithms are characterized by using a static, predefined dataset to train models. This approach does not involve iterative querying or interaction with the environment for additional data during the training process. PL is especially advantageous in situations where collecting new data is costly or logistically challenging, allowing for the extraction of meaningful patterns and insights without further data input. Numerous works have been proposed in the field of constraint acquisition, employing PL approaches to effectively derive constraint models from predefined datasets. ModelSeeker, proposed by Beldiceanu et al.,[4] is an innovative approach to learning constraint models from positive sets of examples, utilizing the global constraint catalog.[14] ModelSeeker searches for the most suitable constraints from the catalog that are consistent with the provided solutions. It uses various techniques to find the best combination of constraints, such as constraint filtering, constraint composition, and constraint ranking. ModelSeeker, being a PL approach, has some limitations. It may struggle to generate accurate models if the input solutions are not representative or diverse, and it cannot handle fixed arity constraints in an efficient way.

Bessiere et al.[3] proposed a method for acquiring constraint networks called CONACQ (CONstraint ACQuisition), using a clausal representation of the version space CONACQ is only provided with a pool of examples, and it uses frequent pattern-mining techniques to extract constraints from the solutions.

Prestwich et al.[5] proposed a novel approach called Classifier-Based Constraint Acquisition (CLASSACQ), which integrates machine learning with CA. This is done by training a classifier to distinguish between solution and non-solution states, which are then used to derive a constraint model. As an initial demonstration of the potential of CLASSACQ, the authors derived a new CA method named BAYESACQ from a Bernoulli Naive Bayes classifier. The authors also utilized the Bayesian hypothesis testing connection to obtain an even faster CA method using the Sequential Probability Ratio Test, which is not Bayesian but is related.

Prestwich et al.[6] proposed an unsupervised learning method for CA called MINEACQ, which is inspired by data mining techniques. The paper addresses the data collection bottleneck that is incurred when preparing a dataset of known solutions and non-solutions, which requires human effort. MINEACQ can learn constraints from positive-only, negative-only, and positive-negative data, and does not require labels. A

COUNT-CP, introduced by Kumar et al.,[7] is a constraint learner designed to apply the concept of bounded expressions to learn CP constraints. This approach is based on two key observations. In constraint models over finite domain integers, constraints usually consist of Boolean expressions and numeric expressions with a comparison (e.g., $x = y$ and $x \leq y$). Boolean expressions in this context can be seen as a special case of numeric expressions that are equal to 1. COUNT-CP can efficiently learn CP constraints from given examples by searching for suitable bounded numeric expressions that represent the constraints.

Prestwich et al.[20] proposed a method for robust CA using sequential analysis. They introduced SEQACQ (SEQuential analysis-based constraint ACQuisition) which performs fast hypothesis testing by adaptive sampling of training instances. It can learn redundant constraints that cause problems for version space methods. This approach is also robust, and capable of accurately learning constraints even when the data contains numerous errors.

3.2. *Active learning algorithms*

On the other side, AL is a methodology wherein machine learning models enhance their accuracy by selectively querying unlabeled data instances for labeling, which has been extensively documented and analyzed in the literature. Several key contributions stand out in the literature. For instance, the works by Lewis and Gale pioneered the use of uncertainty sampling methods, which select instances for which the model has the least confidence in its predictions.[21] Tong and Koller further developed these ideas by introducing support vector machines to AL, demonstrating the effectiveness of using margin sampling techniques to choose the most informative instances.[22]

Additionally, the Query-by-Committee approach, described by Seung et al., minimizes the version space of the model by querying instances about which a committee of models disagrees the most.[23] This method has been shown to efficiently reduce the number of queries necessary to train a model effectively.

More recent advancements include the work of Freund et al. and Dasgupta, who have contributed to the theoretical foundations of AL, providing bounds on the label complexity and insights into the conditions under which active learning is most beneficial.[24, 25] These studies help clarify when and why active learning can significantly outperform passive learning.

The integration of AL into practical applications has also been highlighted by Settles in his comprehensive survey, which not only reviews algorithms but also discusses real-world applications, indicating a growing trend in the use of active learning strategies in industry and academia.[26] These works collectively showcase the depth and breadth of AL research and underscore its importance in the ongoing evolution of machine learning methodologies.

AL is also successfully used in constraint acquisition to acquire constraints. Bessiere et al. proposed QuAcq (QuickAcquisition), an AL algorithm that interacts with the user by asking them to classify partial queries.[8] The algorithm begins with an initial hypothesis about the constraint network, often starting with no constraints (empty C_L) or minimal information. In each iteration, QuAcq generates a query, which is typically a partial assignment. This query is presented to the user or oracle, who is asked to evaluate whether the partial assignment is consistent or inconsistent with the target constraint network. Based on the user feedback, the algorithm updates its hypothesis about the constraint network by either adding a new constraint to C_L or by removing constraints from B. This process is repeated iteratively until QuAcq has learned a sufficient number of constraints to represent the target constraint network or when a predefined stopping criterion is met.

Arcangioli et al. proposed a method, called MultiAcq, that can learn the maximum number of constraints that are violated by a single negative example.[9] They also introduced several heuristics to balance the trade-off between the time required and the number of queries generated. Both QuAcq and MultiAcq incur large numbers of queries and high CPU times for query generation. To address these issues, Tsouros et al.[10] proposed MQuAcq that combines the primary concept of MultiAcq with QuAcq, resulting in a technique that acquires as many constraints as MultiAcq does following a negative example, but with reduced complexity.

Tsouros et al.[11] introduced MQuAcq-2, an enhanced CA algorithm that builds upon MQuAcq by efficiently learning multiple constraints from negatively generated queries. MQuAcq-2 leverages the structure of the learned network by concentrating on specific violated constraints instead of conducting exhaustive searches in the generated examples. Also, when a constraint is obtained from a negative example, the entire scope of that constraint is eliminated from the example. Although this does not guarantee the discovery of all violated constraints, it can still identify several of them. The main advantage of MQuAcq-2 is its ability to learn multiple constraints from negatively generated queries more efficiently than its predecessors.

Recently, Tsouros et al.[27] addressed two key challenges in CA systems: The excessive number of user queries required and the system's inability to manage large sets of constraints in B. They introduced GROWACQ, a bottom-up method that reduces user wait times and query counts using a probability-driven approach to optimize query creation. Their method outperformed existing CA techniques, reducing query counts by up to 60% and handling much larger constraint sets.

4. Hybrid constraint acquisition

In our proposed methodology, CA is achieved through a synergistic integration of PL and AL techniques. The following subsections provide an overview of the hybrid CA framework, followed by a more detailed description of the PL and AL components. But first, it is important to understand the advantages and disadvantages of PL and AL, which are the fundamental building blocks of our method.

Passive Learning PL has some important advantages. It is highly scalable, making it suitable for situations where a large amount of data is available. As PL does not require interaction, it is more efficient, in terms of human resources, than AL. Also, it can efficiently learn global constraints. However, PL has its drawbacks, with the main one being that convergence to the target network requires an exponential number of examples, but in practice only a relatively small number of examples will be available. This means PL does not guarantee that the learned constraints, be they global or fixed arity, are indeed part of the target model. If the available data is not representative of the problem space, the learned constraints may not accurately capture the target network. It is important to note that, following the literature, we operate under the assumption that any global constraints identified by PL are considered learned. Also, the solution set provided as input to the system is treated as complete and final. In practice, these assumptions are rather restrictive as newly obtained solutions that were not part of the initial training set may invalidate constraints that were considered learned. Lifting these assumptions is an interesting direction for future work.

Active Learning AL on the other hand, usually leads to more robust constraints because it actively queries the user to label the most informative examples as positive or negative. Also, the learned constraints provably belong to the target model, under the assumption that the user answers all queries correctly (which is a standard assumption in the CA literature). However, AL is not without problems. The process requires a lot of input from the user, which can be time-consuming and overwhelming. Also, AL cannot learn global constraints, as this would require a bias of exponential size. AL also operates under certain assumptions, with a basic one being that once a constraint is considered learned, no generated example should contradict it. This means that all the fixed arity constraints that have been inserted to C_L, at any step of the learning process, are considered valid and no query will be generated to invalidate them.

Given these considerations, we propose a hybrid approach that combines PL and AL, in a process depicted in Figure 1, to achieve more effective CA by reducing the learning time and improving the accuracy of constraint learning. Roughly, the PL component extracts initial constraints from the given solution data, while the AL component interacts with the user to refine and improve the learned model.

Our method starts by receiving positive examples of a problem as input, as shown in Figure 1. In the first step, we preprocess this set of examples. This involves examining the format in which the examples are stored (e.g. lists, matrices, structured format) to select suitable patterns for the PL process, as detailed in Subsection 4.1, and mapping the problem's variables to the assignments in the examples. This is a standard step in PL systems such as ModelSeeker and COUNT-CP.

The second step generates the *Bias* B, which encompasses all potential fixed arity constraints relevant to the problem. For instance, for the case of binary constraints, for each pair of variables we generate and add to the Bias the following constraints: $\neq$, $=$, $>$, $<$, $\geq$, and $\leq$. This step is standard for AL methods but not PL ones, as the latter focus on learning global constraints. Adding this step to the PL component of our hybrid method is a novelty of our approach.

In Step 3 of Figure 1, the candidate global constraints are generated. The process starts by applying patterns to the various subsets of assignments in the examples (e.g. per row, per column, all pairs), as the sequences of variables included in the examples could be involved in global constraints. Whenever we associate a sequence of variables (resulting from a recognized pattern) with a global constraint, we term this association as a *candidate constraint.* Furthermore, we use some extra rules to generate any additional parameters that may be needed for certain global constraints (e.g. the target sum in a *sum* constraint).

Then the method is split into two threads that are run in parallel. In the first thread (fourth step in Figure 1) we have a big pool of candidate constraints that will be formulated as small CP models and we will verify if they are satisfied or not using a CP solver, as explained in Subsection 4.1. Constraints failing to satisfy all the assignments of the examples will be discarded. After this step, we will have a list of *Learned Global Constraints*, that form the initial acquired model. The second thread of our system filters the *Bias* of fixed arity constraints that will guide the AL process, removing those constraints that are not satisfied in all the input solutions. These two operations can be performed sequentially, but we have chosen to parallelize them for efficiency reasons.

If the *Learned Global Constraints* include any decomposable constraint, i.e. a constraint that can be equivalently represented by a set of fixed arity constraints, like the *alldifferent*, our method handles it in a specialized way. Specifically, we generate the decomposition of the constraint and insert the resulting set of fixed arity constraints to C_L, which will be given as input to the AL module. As we explain below, this is done so that the AL module can proceed efficiently to the refinement of the model without having to learn these binary constraints from scratch.

The filtered *Bias* is also given as input to the AL module as shown in step 8 of Figure 1, where the AL module runs and iteratively generates and posts partial queries to the user, asking him/her to classify them as solutions to the problem or not. The AL algorithm is an exact active learning method that strategically utilizes each query to substantially contribute to the learning process without the need for a predefined query budget. This allows it to operate efficiently by generating queries that are likely to identify multiple constraints per negative example with the efficiency of logarithmic complexity for locating each constraint's scope. The heuristic used ranks instances based on their potential to uncover new constraints and their likelihood of negative classification, effectively prioritizing queries. The FindScope-2 function, an optimization of FindScope, aims to locate the scope of violated constraints efficiently by avoiding redundancy—skipping queries that do not alter the number of violated constraints detected previously and classifying queries automatically as positive when no violations are found. Meanwhile, the FindC function identifies the specific constraint within the determined scope that the negative example violates, employing a partial querying strategy for precision.

By the end of this process, we will have a list of learned fixed arity constraints, which will augment the global constraints acquired by the PL module.

The following step of our methodology is the post-processing of the list of the learned global and fixed arity constraints, where we remove duplicate and some implied constraints. For example, given two constraints: c1: `AllDifferent`(x1, x2, x3, x4) and c2: `AllDifferent`(x1, x2, x3) then c2 is implied by c1 and thus can be removed without affecting the solution set.

Ultimately, in the last step, again following ModelSeeker, our system presents a list of Learned Global Constraints to the user. Each constraint in this list is accompanied by a probabilistic ranking, calculated based on its relational dependencies. This ranking is systematically derived from the Global Constraint Catalog,[28] with each constraint's position in the ranking determined by its 'implied by' and 'implies' relationships. These relationships are indicative of the constraints' interdependencies and relative strengths, thereby providing an understanding of the constraint's significance within the context of the problem. This last step is necessary in cases where there are several candidate global constraints over the same set of variables that satisfy all of the solutions provided to the PL component.

We now explain in more detail the two main stages of the learning process.

4.1. *Passive learning*

The aim of PL in our methodology is to identify recurring schemes or complex associations among the variables and also to set the stage for the subsequent AL process. The system receives a set of solutions and a list of global constraints as its primary inputs. In our implementation, the acceptable format of the input solutions is based on the format of the data provided for the PTHG 2021 Constraint Acquisition Challenge.[29]

Inspired by the ModelSeeker,[14] we primarily use patterns to identify sequences or configurations of variables that could be involved in a global constraint. The utilization of patterns is a common practice in CA systems, such as ModelSeeker and Count-CP. Our hybrid CA method is designed to accommodate two different problem structures. Problems' examples and/or solutions may be formulated as one-dimensional arrays (lists) or two-dimensional arrays (matrices). In problems structured as matrices, such as Sudoku, we employ specific patterns to identify potential constraints and relationships among the variables. Some notable patterns that we utilize include the following and are depicted graphically in Figure 2:

- Variables in rows: This pattern identifies constraints based on variable values in the same row.
- Variables in columns: Following the preceding pattern, this pattern discerns constraints based on variable values in the same column.
- Diagonals: This pattern reveals constraints based on variable values on the same diagonal.
- Sliding window: This pattern discerns constraints based on variable values within a specific size sliding window.

Our system initializes B, based on a given language Γ encapsulating all potential

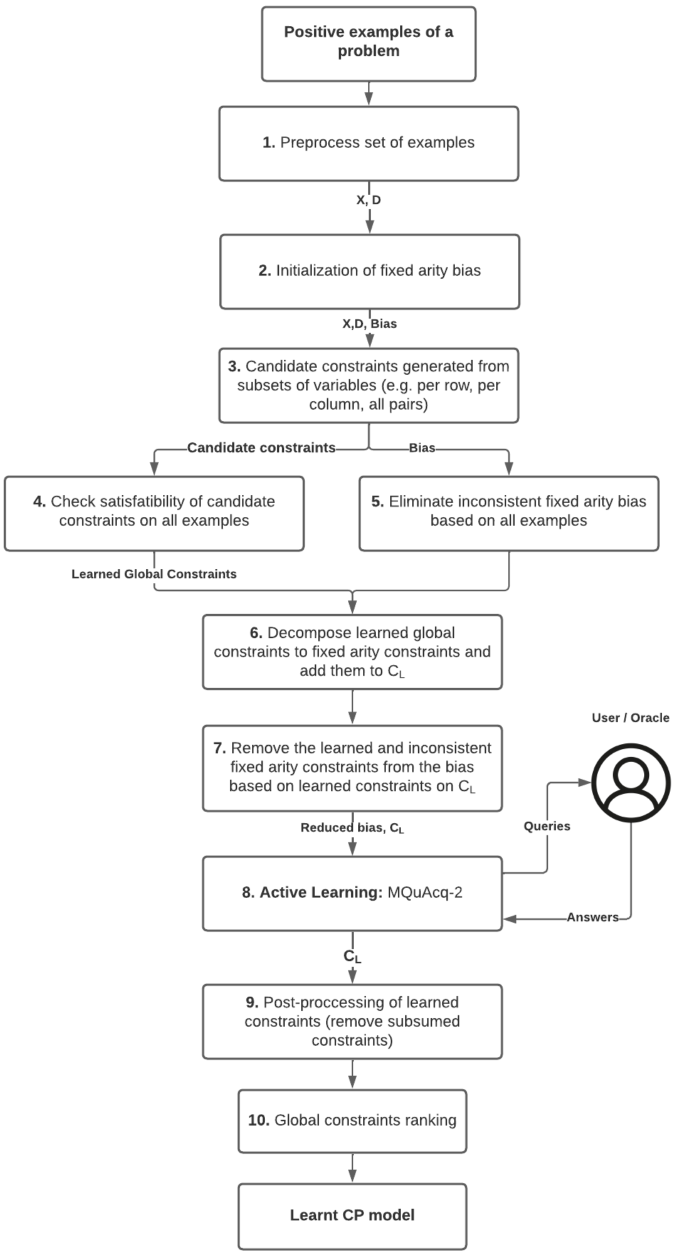

Positive examples of a problem
1. Preprocess set of examples
X, D
2. Initialization of fixed arity bias
X,D, Bias
3. Candidate constraints generated from subsets of variables (e.g. per row, per column, all pairs)
Candidate constraints
Bias
4. Check satisfatibility of candidate constraints on all examples
5. Eliminate inconsistent fixed arity bias based on all examples
Learned Global Constraints
6. Decompose learned global constraints to fixed arity constraints and add them to C_L
7. Remove the learned and inconsistent fixed arity constraints from the bias based on learned constraints on C_L
User / Oracle
Queries
Reduced bias, C_L
8. Active Learning: MQuAcq-2
Answers
C_L
9. Post-proccessing of learned constraints (remove subsumed constraints)
10. Global constraints ranking
Learnt CP model

Fig. 1: Our hybrid constraint acquisition approach

fixed arity relations among the variables. Also, C_L is initialized to the empty set, denoting that no fixed arity constraints have been learned yet.

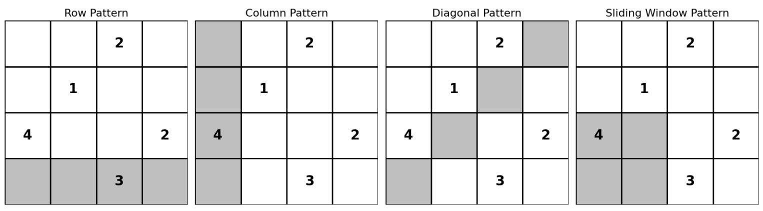


Fig. 2: Sudoku 4×4 with row, column, diagonal, and sliding window patterns.

Identifying and filtering potential global constraints is done systematically. The first step is to identify the candidate constraints based on the patterns mentioned above. Once identified, these candidate constraints are subjected to a filtering process to eliminate constraints that are not satisfied in all solutions. This is done on a constraint-by-constraint basis. Each candidate constraint is individually checked for satisfaction against all available solutions. To achieve this, we create a simple CSP model for each constraint that only includes the variables in the scope of this constraint, and we then iteratively check the satisfiability of the constraint under each solution using a CP solver, such as Choco.[30] This is done by fixing the domains of the variables in the model to the values they have in the currently processed solution and calling the solver to check if the constraint is satisfied under this assignment. This is repeated for all available solutions until the solver fails, signaling that the constraint is not satisfied by some solution, or all solutions have been processed. To handle certain global constraints that need extra parameters, like a *COUNT* or a *SUM* constraint, we also identify the possible values for these parameters.

In addition to candidate global constraints, our method evaluates fixed arity constraints from B, again using the set of provided solutions. Checking if a fixed arity constraint (e.g. $x_i \neq x_j$) is satisfied in a solution is straightforward, and therefore does not require the use of a CP solver. If any such constraint is violated in some solution, it is removed from B, thereby reducing its size. This is a notable novelty of our methodology, as the *Bias* now serves as a link between the PL and AL modules.

Importantly, in our implementation, the Constraint Validation phase for candidate global constraints and the Fixed Arity Constraint Elimination phase of the PL component run in parallel to make the CA process more efficient.

4.1.1. *Decomposable constraints*

As a final step in the PL process, if any learned global constraints are amenable to decomposition, they are processed in the Decomposition Phase, where they are broken down into fixed arity constraints. As *alldifferent* is the most commonly used constraint that is decomposable, we will focus on this constraint when we refer to decomposable constraints hereafter.

Any *alldifferent* constraint is decomposed into binary $\neq$ constraints, which are added to C_L and are removed from B. As all existing AL algorithms can only operate with fixed arity constraints, it is necessary that the AL module "knows" that the $\neq$ constraints have been learned. If they are neither added to C_L nor removed from B then the AL algorithm will try to learn them from scratch, incurring an unnecessarily large number of queries being posted to the user. Otherwise, if these constraints are not added to C_L, but only removed from B, then the user's replies to the examples the system posts may be misled and the system may not be able to converge to the correct constraint network. It has to be noted, that the decomposition of an *alldifferent* does not mean that this constraint is removed from the list of Learned Global Constraints.

Regarding the other binary constraints on pairs of variables participating in an *alldifferent* constraint (i.e. the $=$, $\leq$, $\geq$, $<$, $>$ constraints), we differentiate between constraints that include the equality and the $<$, $>$ ones. As any $\neq$ constraint on two variables x_i, x_j, belonging to the decomposition on an *alldifferent*, is considered learned, we remove the $=$, $\leq$, $\geq$ constraints between these variables from B, as is it has been proved that the two variables must take different values.

However constraints $<, >$ are not removed from B. This is because we cannot conclude if they exist or not in the target network. Removing these constraints from B would make it impossible to learn them in case they are part of C_T in the problem at hand, thus AL would not be able to converge to the correct constraint network. In addition, if they were removed, the AL system might not be able to derive the correct information from the user's answers to queries. There can be cases in which the user's answer is negative due to these constraints, but as they are not in B, the AL system will be directed to look for the constraint(s) that violate the posted example in another part of the problem, resulting in falsely learning constraints that do not belong to the target network.

We now illustrate our PL strategy using the classic Sudoku puzzle, and a variant known as Greater Than Sudoku. A Sudoku problem is composed of a 9x9 grid, divided into nine 3x3 sub-grids or 'blocks'. A correctly resolved Sudoku puzzle guarantees that each row, column, and block encapsulates all digits from 1 to 9 once, without repetition. In the standard CP model for Sudoku we have one variable for every cell, hence 81 variables, all having domain $\{1, \dots 9\}$, and one *alldifferent* constraint for every row, column, and block (hence 27 constraints).

Example 4.1. Consider a small 4×4 Sudoku puzzle. We focus on variables $x1$ to $x4$, which are positioned at the top row of the Sudoku. Assume that the PL module receives only two solutions, which include the following assignments for variables $x1$ to $x4$:

(1) $x1 = 2$, $x2 = 1$, $x3 = 4$, $x4 = 3$
(2) $x1 = 1$, $x2 = 2$, $x3 = 3$, $x4 = 4$

Let us break down how the PL module processes these solutions, and how it modifies B and C_L:

(1) **Initialization of fixed arity Bias:**

- Our method generates all binary constraints on pairs of variables $\{x1, x2, x3, x4\}$ based on the language Γ. These 36 constraints (six per each pair of variables) are added to B.

(2) **Candidate constraints generation and satisfiability test**

- The PL will apply the row pattern to the top row of the variables. This will result in identifying the sequence of variables $\{x1, x2, x3, x4\}$ as a set of variables that could potentially be involved in a constraint. Assuming for simplicity that the only available global constraint is the *alldifferent* then we will get the candidate constraint *alldifferent*$(x1, x2, x3, x4)$.
- This candidate constraint will be checked against the available solutions by building a simple CSP model that will only include variables $\{x1, x2, x3, x4\}$ and using a CP solver to determine if the constraint is satisfied in all solutions. In the first solution, variables $x1$ to $x4$ hold distinct values, meaning that the *alldifferent*$(x1, x2, x3, x4)$ is satisfied. The same holds for the second solution. As there are no other available solutions, the global constraint *alldifferent*$(x1, x2, x3, x4)$ is considered learned and it is added to the *Learned Global Constraints.*

(3) The PL parses each solution and eliminates any fixed arity constraint that is violated. Specifically, the following constraints are removed from B: (a) $x1 < x2$, $x1 \leq x2$, $x1 = x2$ (as in the first solution we have $x1 > x2$) (b) $x1 > x2$, $x1 \geq x2$ (due to the second solution, where $x1 < x2$) (c) $x3 < x4$, $x3 \leq x4$, $x3 = x4$ (as in the first solution $x3 > x4$) (d) $x3 > x4$, $x3 \geq x4$ (due to the second solution, where $x3 < x4$) (e) $x1 = x3$, $x1 > x3$, $x1 \geq x3$, $x1 = x4$, $x1 > x4$, $x1 \geq x4$, $x2 = x3$, $x2 > x3$, $x2 \geq x3$, $x2 = x4$, $x2 > x4$, $x2 \geq x4$ (violated in both solutions).

(4) **Global constraints decomposition**

- At the next step the learned *alldifferent*$(x1, x2, x3, x4)$ is decomposed to the binary constraints: (1) $x1 \neq x2$ (2) $x1 \neq x3$ (3) $x1 \neq x4$ (4) $x2 \neq x3$ (5) $x2 \neq x4$ (6) $x3 \neq x4$. This set of constraints is added to C_L and is removed from B.
- In addition, the following constraints are removed from B: (1) $x1 \leq x3$ (2) $x1 \leq x4$ (3) $x2 \leq x3$ (4) $x2 \leq x4$.
- B includes the following constraints among variables $x1 \ldots x4$: (1) $x1 < x3$ (2) $x1 < x4$ (3) $x2 < x3$ (4) $x2 < x4$. It is now up to the AL module to remove these constraints from B, and thereby to conclude that the only constraint among variables $x1 \ldots x4$ is the *alldifferent.* But importantly, B contains only four constraints among these variables, meaning that fewer queries to the user will be required.

Example 4.2. Consider a Greater Than Sudoku puzzle, where in addition to the standard rules of Sudoku, certain adjacent cells have a greater than ($>$) or less than ($<$) sign between them. Hence, in the CP model of the problem, any two variables corresponding to such cells have a $>$ (resp. $<$) constraint between them, in addition to being involved in at least one *alldifferent* constraint because they are adjacent. Consider two such variables, x_i and x_j, having a constraint $x_i > x_j$. Assume that the solutions provided have the following assignments for x_i and x_j: 1) $x_i = 4$, $x_j = 2$, 2) $x_i = 3$, $x_j = 1$, 3) $x_i = 5$, $x_j = 3$. Let us see what can be deduced about the constraint between x_i and x_j:

- The system will add to B all binary constraints between x_i and x_j from the language Γ: (1) $x_i = x_j$, (2) $x_i \neq x_j$, (3) $x_i > x_j$, (4) $x_i < x_j$, (5) $x_i \geq x_j$, (6) $x_i \leq x_j$.
- Any constraint in B that violates some solution is removed. This holds for constraints $x_i = x_j$, $x_i \leq x_j$, $x_i < x_j$.
- As x_i and x_j are involved in some *alldifferent* constraint, there are two cases;
  (1) The *alldifferent* is not learned by the PL module. In this case, constraints $x_i > x_j$, $x_i \geq x_j$, $x_i \neq x_j$ remain in B, and therefore the PL module is unable to learn the specific constraint that holds between variables x_i and x_j.
  (2) The *alldifferent* is learned by the PL module. In this case, after its decomposition is generated, the constraint $x_i \neq x_j$ will be added to C_L and removed from B, as it is considered learned. Constraint $x_i \geq x_j$ will also be removed from B. However, constraint $x_i > x_j$ will remain in B, meaning that the system is so far unable to determine whether it belongs to the target network or not.

4.2. *Active learning*

The AL component focuses on refining the model learned by the PL component. This is achieved by adding to C_L, and thereby acquiring, specific fixed arity constraints, and by removing other fixed arity constraints from B, and thereby proving that they do not belong to the target network.

Our hybrid CA system can incorporate any AL algorithm, as the only means of communication between the PL and AL components is the initialization of C_L and B that the PL performs for the AL algorithm. We currently use the MQuAcq-2 algorithm. Details about the mechanics of AL algorithms can be found in the AL papers referenced in Section 3, and specifically about MQuAcq-2 in the paper by Tsouros et. al.[31] In this section, we focus on the integration of MQuAcq-2 in our hybrid CA system.

MQuAcq-2 takes as input a set of variables X, a set of domains D, the *Bias* B, and the set of learned constraints C_L, and it outputs the updated learned constraint network C_L. In our method, the algorithm starts by initializing C_L to the known constraints that we get as input from the PL process (if any exist), whereas the AL algorithms such as MQuAcq-2 typically initialize C_L to the empty set. Then, MQuAcq-2 generates (partial) examples and asks the (human or software) user to classify them as solutions or not. If the user classifies an example as a solution, MQuAcq-2 will remove the constraints rejecting the example from B. If the example is classified as a non-solution, MQuAcq-2 will identify one or more constraints from B that conflict with the example and add them to C_L.

MQuAcq-2 and other active CA algorithms generate queries at three different levels. The first level called *Top-Level Query Generation*, builds a (partial) example to focus on. If this is classified as negative then the second level, implemented by function *Findscope*, queries the user on subsets of the top-level query to identify the scope of the constraint to acquire. Once the scope has been located, the third level, implemented by the function *FindC*, seeks the specific relation that holds over this scope.[10] The generation of top-level queries is the process which mostly affects the runtimes. In our system, the query generation strategy of MQuAcq-2 is designed to ensure that at least one constraint from the B aiming to maximize the violated constraints and is formulated as follows:

$$\text{Minimize} \quad \sum_{c \in B} c$$

where c represents an individual constraint from the set B. The constraints in B are treated as soft constraints that MQuAcq-2 aims to violate. To ensure that at least one constraint is violated, the following constraint is added to the model:

$$\sum_{c \in B} c < |B|$$

Where $|B|$ denotes the number of constraints in B. This inequality ensures that the total sum of the satisfied constraints is less than the total number of constraints, thereby guaranteeing that at least one constraint is not satisfied. Since the MQuAcq-2 algorithm is exact, there is no budget for queries like in classical AL and the algorithm terminates when no other example could be generated.

The algorithm terminates when all the constraints in B have been explored and either confirmed as belonging to C_T, and therefore moved to C_L, or rejected. The output of MQuAcq-2 is the set of learned fixed arity constraints C_L that can be combined with the learned global constraints from PL to form a complete CSP model. As an illustration, consider again the Sudoku and Greater Than Sudoku examples 4.1 and 4.2.

Example 4.3. Regarding Example 4.1, recall that after the completion of the PL phase, the constraint *alldifferent*$(x1, x2, x3, x4)$ has been learned, and B includes the following constraints among variables $x1 \ldots x4$: (1) $x1 < x3$ (2) $x1 < x4$ (3) $x2 < x3$ (4) $x2 < x4$. MQuAcq-2 will at some point generate one or more positive (partial) examples that are violated by these four constraints, and will thus remove them from

B. For instance, it may generate a partial example with the following assignments for variables $x1 \ldots x4$: $x1 = 4$, $x2 = 3$, $x3 = 2$, $x4 = 1$. The user will label this example as positive, and therefore the system will remove all constraints that violate it (i.e. the four constraints above) from B. Hence, the AL component will prove that the only constraint among variables $x1 \ldots x4$ is the *alldifferent*, something that the PL module was unable to do.

Now consider Example 4.2 and let us assume that the PL module has learned the *alldifferent* constraint that includes variables x_i and x_j. In this case, constraint $x_i \neq x_j$ will be added to C_L and removed from B by the PL module. Also, all other binary constraints between x_i and x_j will be removed from B, except for constraint $x_i > x_j$. The AL phase, then commences, seeking to refine the constraint model. MQuAcq-2 may post a (partial) query including the following assignment for x_i and x_j: $x_i = 1$, $x_j = 2$. This query will be classified as negative by the user because the target constraint between x_i and x_j is violated. The algorithm will then seek the constraint responsible, and after a series of partial queries have been posted and replied, it will eventually focus on $x_i > x_j$, which will be removed from B and added to C_L. Hence, the AL module will be able to learn that this constraint is part of C_T, something that the PL module was unable to do.

5. Experimental evaluation

In this section, we present the results of experiments that evaluate the efficiency of our hybrid CA system. Our tool, implemented in Java 11 and utilizing the Choco solver[a], was tested on four problems. Namely, Sudoku, Greater Than Sudoku, the Warehouse Location Problem, and the Virtual Machines to Physical Machine allocation problem. Subsection 5.1 describes these problems. The experimental results and an analysis of their implications is given in Subsection 5.2.

5.1. *Benchmarks*

Sudoku and Greater Than Sudoku Sudoku is a puzzle game played on a 9×9 grid, with some cells initially filled in. The objective is to complete the grid using numbers from 1 to 9, ensuring that every row, column, and 3×3 block contains unique numbers.[32] The Sudoku puzzle is widely used as a benchmark for CA algorithms. Its CSP model includes a set of 81 variables, one for each cell in the grid, having domain $\{1, \ldots 9\}$, and a set of *alldifferent* constraints across each row, column, and 3×3 block.

Greater Than Sudoku is a variant of Sudoku where in addition to the *alldifferent* constraints, there are binary $>$ and/or $<$ constraints between some adjacent variables (i.e. variables corresponding to adjacent cells in the puzzle). These additional

[a]Our code is available online at our GitHub repository *https://github.com/mpvasilis/JAIT-HybridConstraintAcquisition*

binary constraints make Greater Than Sudoku more challenging for PL methods compared to the classic Sudoku.

Warehouse Location Problem The Warehouse Location Problem (WLP)[33] involves finding the best locations to open warehouses to minimize the overall expense of serving customers. For the purposes of this study, we approach WLP as a satisfaction problem and do not consider the optimization criterion. The target model for the WLP is as follows:

Variables: Let $W = \{w_1, w_2, \ldots, w_n\}$ be the set of potential warehouse locations, and $C = \{c_1, c_2, \ldots, c_m\}$ be the set of customers. We define the following variables: A_c: A variable for each customer $c \in C$, where $A_c = i$ if customer c is served by warehouse w_i, and $A_c = 0$ if no warehouse is assigned to customer c.

Domains: $A_c \in \{0, 1, 2, \ldots, n\}$ for each customer $c \in C$.

Constraints:

(1) Every customer must be served by exactly one warehouse: $1 \leq A_c \leq n, \quad \forall c \in C$.
(2) The maximum number of customers a warehouse i can supply is limited to K_i: Count($\{A_c$: c ∈ C$\}$, i, K_i), ∀ i ∈ {1, ..., n}.

The first constraint ensures that each customer is assigned to one and only one warehouse. Variable A_c represents the assignment of customer c to a warehouse. The second constraint ensures that the number of customers assigned to each warehouse does not exceed its designated capacity.

In practice, additional side constraints may exist in a WLP. For instance, a constraint stating that two specific customers must not be served by the same warehouse. Hence, for the purposes of this study, we extend the model above to include constraints such as: $A_i \neq A_j$, i.e. a binary constraint indicating that customer $i \in C$ must not be served by the warehouse that servers the customer $j \in C$.

VM to PM Allocation Problem The Virtual Machine to Physical Machine (VM to PM) allocation problem is a resource allocation problem, where the goal is to optimally allocate a set of virtual machines to a set of physical machines in a Data Center. Each Virtual Machine (VM) has specific resource requirements, such as CPU and RAM, while each Physical Machine (PM) has a limited capacity for these resources. As with the WLP, we approach this problem as a satisfaction one. The target model is as follows:

Variables: Let n be the number of VMs and m be the number of PMs.

- $vm\_allocated_i \in \{1, \ldots, m\}$: An integer variable representing the index of the PM to which the ith VM is allocated, where $i \in \{1, \ldots, n\}$.
- CPU_i: The CPU requirement of the ith VM, where $i \in \{1, \ldots, n\}$.
- CPU_j: The CPU capacity of the jth PM, where $j \in \{1, \ldots, m\}$.
- RAM_i: The RAM requirement of the ith VM, where $i \in \{1, \ldots, n\}$.

- RAM_j: The RAM capacity of the jth PM, where $j \in \{1, \ldots, m\}$.

Constraints:

(1) CPU capacity constraints: For each PM $j \in \{1, \ldots, m\}$, the sum of the CPU requirements of the VMs allocated to it must not exceed its CPU capacity.

$$\text{sum}(\{CPU_i * (vm\_allocated_i == j) : i \in \{1, \ldots, n\}\}, \leq, CPU_j) \quad \forall j \in \{1, \ldots, m\} \tag{1}$$

(2) RAM capacity constraints: For each PM $j \in \{1, \ldots, m\}$, the sum of the RAM requirements of the VMs allocated to it must not exceed its RAM capacity.

$$\text{sum}(\{RAM_i * (vm\_allocated_i == j) : i \in \{1, \ldots, n\}\}, \leq, RAM_j) \quad \forall j \in \{1, \ldots, m\} \tag{2}$$

(3) VM allocation constraints: Each VM $i \in \{1, \ldots, n\}$ must be allocated to exactly one PM.

$$vm\_allocated_i \in \{1, \ldots, m\}, \quad \forall i \in \{1, \ldots, n\} \tag{3}$$

As in the WLP, side constraints may exist. For this study, we consider additional constraints between pairs of variables specifying whether two VMs must not be allocated to the same PM: $vm\_allocated_i \neq vm\_allocated_j$, i.e. a binary constraint indicating that VM i must not be allocated to the same PM as the VM j.

5.2. *Experimental Results*

This section describes the methodology we used for our experiments:

(1) We used the Hamming distance[34] to generate a broad spectrum of 1,000 diverse solutions for each problem, following the methodology proposed in.[35] For any given problem, we model it as a CSP and use the Choco solver to generate solutions iteratively. For each new solution obtained, we ensure that its Hamming distance from each of the previously generated solutions is greater than a predefined threshold of 10. The Hamming distance between two solutions is calculated by counting the number of assignments where the two solutions differ. This diversity is crucial for ensuring that our learning process is not biased toward a specific subset of solutions.
(2) In the experiments, we sequentially increased the number of available solutions from zero to 1,000. Through this, we aimed to observe the incremental impact of adding more solutions to the learning process.
(3) Upon completion of the learning process, we created plots to visualize the following: a) The correlation between the amount of learned global constraints and the number of solutions, b) the correlation between the number of AL queries and the number of solutions, c) the correlation between the

generated AL bias and the number of solutions, d) the effect of the number of solutions on the runtime of PL and AL.

The experiments were carried out on a computational system with an AMD Ryzen Threadripper PRO 3975WX 32-Core CPU with a clock speed of 3.50GHz, complemented by 64 GB of DDR4 RAM, running under Ubuntu 22.04. The results are presented in a series of graphs (Figures 3, 4, 5, and 6), showing the number of global constraints acquired, the number of AL queries, the size of the fixed arity bias, and the AL and PL running times.

5.2.1. *Sudoku* 9×9 *experimental results*

The experimental results for the standard Sudoku 9×9 puzzle are presented in Figure 3. Figure 3a illustrates the efficiency of the PL process in acquiring global constraints. The system successfully identified all 27 *alldifferent* constraints, without wrongly learning any other constraint, by examining a set of only 20 solutions. This is consistent with findings reported in the literature with respect to ModelSeeker,[4] which also acquired all the *alldifferent* constraints with a small solution sample.

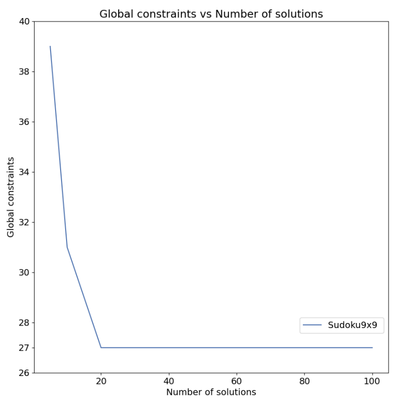
Figure 3b gives the number of queries required by AL for constraint refinement. As the target model for Sudoku only includes the 27 *alldifferent* constraints, the AL module does not add any new constraints to the basic model, but it verifies that no additional binary constraints exist in the model. Figure 3c shows the number of constraints left in B after the completion of the PL process.

Without the preparatory phase of PL (i.e. with 0 solutions available), AL alone requires an excessive number of queries (over 6,000) to learn all the binary $\neq$ constraints (with a total bias of 19,440 binary constraints). However, as we introduce more solutions, the burden on AL diminishes substantially. The query count drops to only 71 and the bias size drops to 182 constraints after processing just 5 solutions. This is due to the narrowing down of B, achieved during the PL phase. With 20 solutions, the number of queries drops to 26 (for a bias size of 40). With 100 solutions only one query is required to remove the remaining constraints from B.

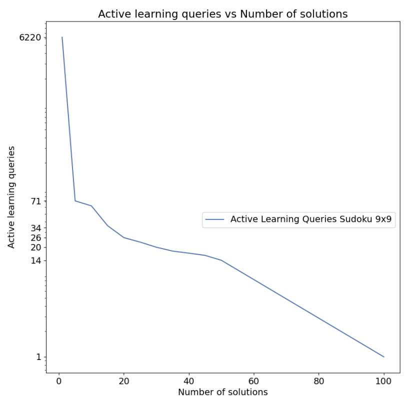
Figures 3d and 3e display the runtimes of the PL and AL phases, respectively. The runtimes are rounded off to the nearest second. Notably, the integration of PL with AL leads to a sharp drop in the total time required for the AL phase, after only a few solutions have been given to the PL component. Regarding the runtime of the PL phase, it is negligible, reaching approximately 1 second as the number of solutions becomes larger than 20.

5.2.2. *Greater Than Sudoku* 9×9 *Experimental Results*

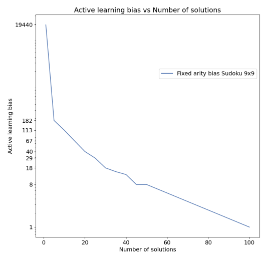
We generated three Greater Than Sudoku puzzles, having 8, 16, and 24 binary inequality constraints, respectively. Figure 4 demonstrates the results of this study. As depicted in Figure 4a, with just 20 solutions, the algorithm learned all 27 *alldifferent* constraints for the puzzle having 8 additional inequalities. However, for puzzles



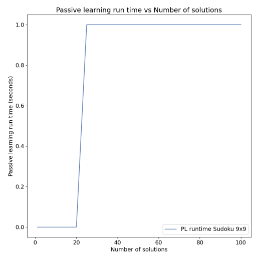
(a) Acquired global constraints



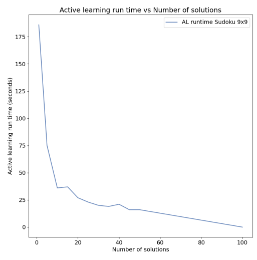
(b) AL queries



(c) Fixed arity bias



(d) PL runtime



(e) AL runtime

Fig. 3: Sudoku 9×9 experimental results

with 16 and 24 inequalities, a minimum of 100 solutions was necessary to learn the 27 *alldifferent* constraints without wrongly acquiring constraints that do not exist in the target network. This is because, in the presence of an increasing number of inequality constraints, candidate *alldifferent* constraints that are not part of the target network may be invalided by fewer variable assignments. Hence, a larger number of solutions needs to be processed so that such variable assignments appear in a solution.

Figure 4b demonstrates the advantages of integrating PL with the AL process. When using AL alone, it took over 6,200 queries to acquire the constraints in the Greater Than Sudoku instances. However, with as few as 5 solutions processed by the PL module, the query count for AL starts to drop substantially.

Specifically, for 20 solutions we get the following results: for the puzzle with 8 binary constraints, the system precisely learned the 27 *alldifferent* constraints, initiated 392 AL queries, and had an AL bias with 621 constraints. For the puzzle with 16 constraints, there were 30 *alldifferent* constraints learned (i.e. it wrongly acquired 3 extra constraints), 504 AL queries, and an AL bias of 596 constraints. In the puzzle with 24 constraints, the system learned 31 *alldifferent* constraints, conducted 468 AL queries and had an AL bias with size 764. When giving 100 solutions to the PL phase, in all three puzzles the 27 *alldifferent* constraints were learned without wrongly acquiring any extra constraints. In the 8-constraint puzzle there were 42 AL queries and an AL bias size of 40. In the 16-puzzle 107 AL queries were posted, and the AL bias included 79 constraints. Lastly, in the 24-constraint puzzle there were 78 AL queries and an AL bias with 59 constraints.

In Figures 4d and 4e, we give the runtimes of the PL and AL phases, respectively. As in Sudoku, integrating PL with AL significantly reduces the total time of the AL phase after giving only a few solutions to the PL component. In Figure 4e we can observe that the AL exhibits peaks at 15-20 solutions and 35 solutions for the three instances. These peaks are the result of variations in the number of top-level queries generated by MQuAcq-2.

PL gives minimal runtimes for a small number of solutions. When the number of solutions is between 20 and 100, the runtime of the PL phase increases moderately and averages about 1 second. As expected, the runtime increases as the number of solutions that need to be processes grows. As with Sudoku, the runtimes are rounded to the nearest second, resulting in some peaks in the curves from 0 to 50 solutions due to this rounding process.

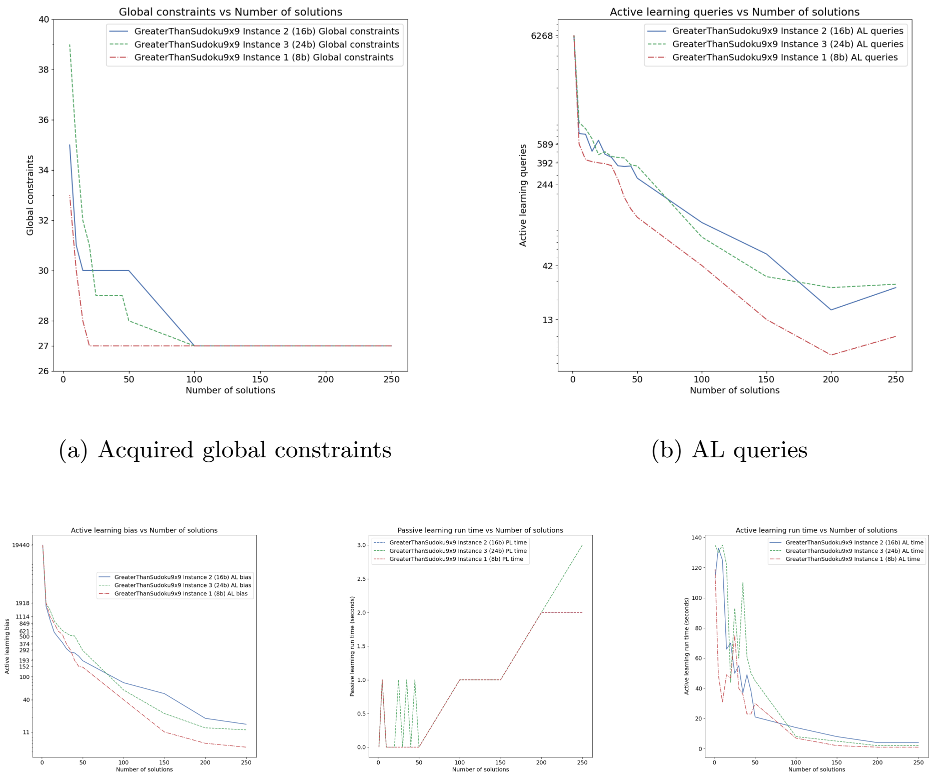
5.2.3. *WLP experimental results*

We generated four instances of the WLP that comprise both global and fixed arity constraints. The first instance consists of 10 warehouses, 20 customers, and 10 binary constraints. For the second instance these numbers are 15, 30 and 15, respectively. In the third instance we have 25 warehouses, 50 customers, and 25 binary constraints. Lastly, for the fourth instance these numbers are 30, 60, and 30.

The results are demonstrated through a series of plots in Figure 5. In the first two instances of the WLP our system successfully acquired the precise number of *Count* constraints (10 and 15 respectively) by utilizing 20 solutions. In the third instance, the system acquired all 25 *Count* constraints after processing 30 solutions. Finally, in the last instance, the system required 35 solutions to accurately learn the 30 *Count* constraints.

Figure 5b shows the number of AL queries posted during the process. When solely utilizing the AL module, 870 queries were needed for the first instance (with 1,140 constraints in B), while the largest instance required 5,610 queries (10,620 constraints in B). Note that the *Count* constraint cannot be decomposed into fixed arity constraints. However, the PL module still removes from B any binary constraint that is not satisfied in all solutions, resulting in a bias of smaller size being passed to the AL module. Hence, as demonstrated in Figure 5b, by running the PL module with only a few solutions, the amount of AL queries sharply drops.

Specifically, for 20 solutions, in the first instance the system learned 12 global



(a) Acquired global constraints

(b) AL queries

(c) Fixed arity bias

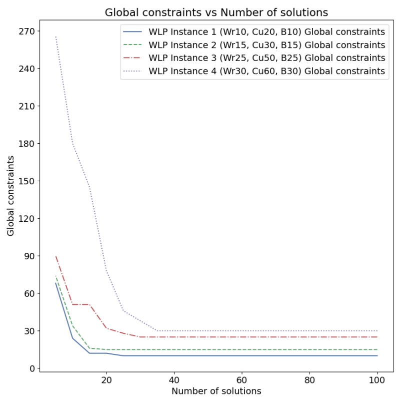
(d) PL runtime

(e) AL runtime

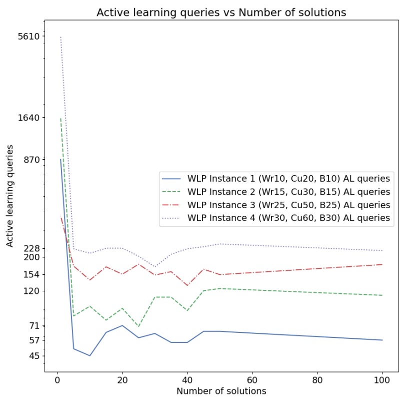
Fig. 4: Greater Than Sudoku 9 × 9 experimental results

constraints, initiated 71 queries, and had a bias size of 180. With the same number of solutions, the second instance recorded 15 learned global constraints, 92 queries, and a bias of size 429. In the third instance we had 32 learned global constraints, 154 queries, and a bias of size 1,220. For the fourth instance, there were 78 learned global constraints, 288 queries, and a bias of 1,766 constraints. At 100 solutions, the precise number of *Count* constraints was acquired in all four instances. In the first instance the system posted 67 queries, in the second instance, 112 queries were posted, in the third instance we had 178 queries. Finally, in the fourth instance we had 220 queries. Importantly, in none of the instances was the PL module able to refine the bias to consist exclusively of 'not equal to' constraints.

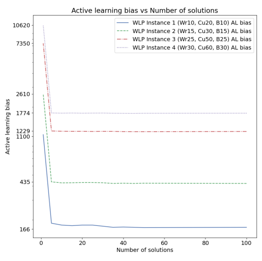
In the AL runtime plot (Figure 5e), we observe a similar pattern as in the Sudoku benchmarks, i.e. a sharp drop as only a few solutions become available, meaning that the size of B starts to shrink. In contrast, the PL runtime graph (Figure 5d) shows a different behavior. The PL runtimes remain relatively constant or increase only slightly for the less complex instances. However, for the fourth instance we observe a clear and steady increase in runtime as the number of solutions increases.



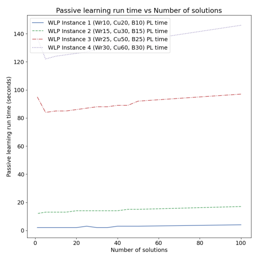
(a) Acquired global constraints



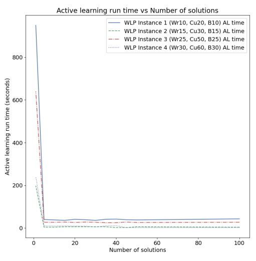
(b) AL queries



(c) Fixed arity bias



(d) PL runtime



(e) AL runtime

Fig. 5: WLP experimental results

5.2.4. *VM to PM allocation problem experimental results*

In the last of our experiments, we generated four instances of the VM to PM allocation problem. The outcomes of these experiments are illustrated in Figure 6. The first instance consisted of 12 VMs, 8 PMs, and 8 binary constraints. The second instance encompassed 18 VMs, 10 PMs, and 10 binary constraints. The third instance involved 24 VMs, 12 PMs, and 12 binary constraints. Lastly, the fourth instance comprised 30 VMs, 14 PMs, and 14 binary constraints.

Figure 6a shows the number of global constraints identified by the PL module. In the first instance, we learned the 16 *Sum* global constraints using only 20 solutions. In the second instance, approximately 35 solutions were required to accurately learn the 20 *Sum* constraints. In the the third the acquisition of all 24 *Sum* constraints was achieved using 40 solutions. Lastly, for the fourth instance, the system required 100 solutions to precisely learn the 28 *Sum* constraints. However, the binary $\neq$ constraints were not acquired during the PL phase.

Figure 6b shows the number of AL queries required. When using AL exclusively, a total of 56 queries were needed to learn the 8 binary constraints from a bias of size 396. For the second instance, 81 queries were necessary to learn the 10 binary constraints from a bias of size 918. In the case of the third instance, 95 queries were required to learn the 12 binary constraints from a bias of size 1,656. Finally, for the last instance, 127 queries were needed to learn the 14 binary constraints from a bias of size 2,610.

As in the case of WLP, the combination of PL with AL achieved a reduction in the number of queries required. For the first instance, the use of only 10 solutions resulted in the number of queries dropping from 56 to 25. For the second instance, the query count decreased to 39 with 10 solutions and further dropped to 20 with 40 solutions. In the third instance, the queries were reduced to 30 using 50 solutions, while in the fourth instance, the queries were reduced to 30 with 100 solutions.

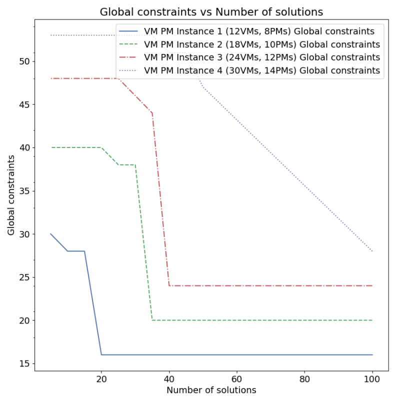
Figure 6d shows the PL runtime, which gradually increases with the number of solutions, as in the case of the WLP. As expected, the runtimes of the PL are higher for the larger instances compared to the smaller ones. Figure 6e shows the AL runtime, which again as in the WLP, decreases sharply as the number of solutions increases, due to the smaller size of the bias received from the PL module.

6. Conclusions

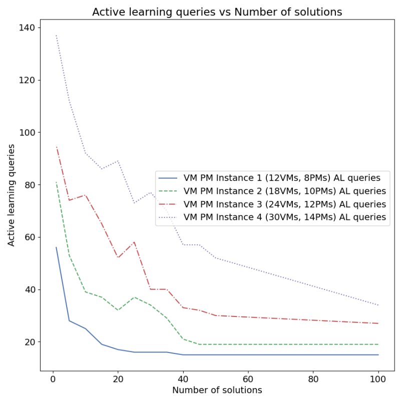
This paper introduced a tool for CA that blends AL and PL strategies, achieving the efficient acquisition of global and fixed arity constraints for CSPs. The PL module is based on generating patterns from potential solutions, which may contain constraints. These patterns are then compared with an array of global constraints and confirmed using a CP solver. Constraints that are present across all solution sets are added to the list of learned constraints. Afterward, unsatisfactory fixed arity constraints are identified and removed, and the remaining ones are channeled to the AL module for further acquisition and enhancement through user interaction.

Testing our methodology on a variety of simple and complex problems demonstrated its effectiveness in learning global and fixed arity constraints. The results indicate that the system is capable of learning the precise global constraints of all problem instances. By combining AL and PL, we were able to learn the remaining fixed arity constraints that the PL alone could not learn. Furthermore, we observed a significant decrease in the number of queries required by the AL module when the PL module was used in combination.

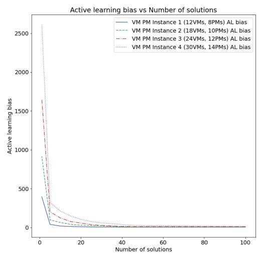
We believe that this innovative hybrid approach contributes towards making CA more efficient and effective. In the future, efforts can be directed towards improving and expanding this method to make it more adaptable to a broader range of complex problems.



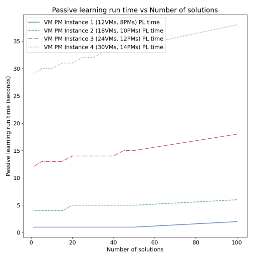
(a) Acquired global constraints



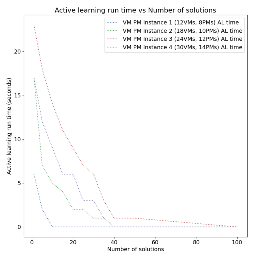
(b) AL queries



(c) Fixed arity bias



(d) PL runtime



(e) AL runtime

Fig. 6: VM to PM allocation problem experimental results

Acknowledgments

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the 4th Call for HFRI PhD Fellowships (Fellowship Number: 9446).

References

1. B. O'Sullivan, Automated modelling and solving in constraint programming, *AI Magazine* **24**(1) (2010) 75–88.
2. C. Bessiere, R. Coletta, E. C. Freuder and B. O'Sullivan, Leveraging the learning power of examples in automated constraint acquisition, in *Principles and Practice of Constraint Programming – CP 2004* Springer Berlin Heidelberg, (Springer, 2004), pp. 123–137.
3. C. Bessiere, F. Koriche, N. Lazaar and B. O'Sullivan, Constraint acquisition, *Artificial Intelligence* **244** (2017) 315–342.
4. N. Beldiceanu and H. Simonis, Modelseeker: Extracting global constraint models

from positive examples, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **10101 LNCS**(284715) (2016) 77–95.

5. S. D. Prestwich, E. C. Freuder, B. O'Sullivan and D. Browne, Classifier-based constraint acquisition, *Annals of Mathematics and Artificial Intelligence* **89** (2021) 655–674.
6. S. Prestwich, Unsupervised constraint acquisition, in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)* (IEEE, 2021), pp. 256–262.
7. M. Kumar, S. Kolb and T. Guns, Learning Constraint Programming Models from Data Using Generate-And-Aggregate, in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, ed. C. Solnon *Leibniz International Proceedings in Informatics (LIPIcs)* **235**, (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022), pp. 29:1–29:16.
8. C. Bessiere, R. Coletta, E. Hébrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper and T. Walsh, Constraint Acquisition via Partial Queries, in *IJCAI: International Joint Conference on Artificial Intelligence* (Beijing, China, 2013), pp. 475–481.
9. R. Arcangioli, C. Bessiere and N. Lazaar, Multiple Constraint Acquisition, in *IJCAI: International Joint Conference on Artificial Intelligence* (New York City, United States, 2016), pp. 698–704.
10. D. C. Tsouros, K. Stergiou and P. G. Sarigiannidis, Efficient methods for constraint acquisition, in *Principles and Practice of Constraint Programming*, ed. J. Hooker (Springer International Publishing, Cham, 2018), pp. 373–388.
11. D. C. Tsouros, K. Stergiou and C. Bessiere, Structure-driven multiple constraint acquisition, in *Principles and Practice of Constraint Programming*, eds. T. Schiex and S. de Givry (Springer International Publishing, Cham, 2019), pp. 709–725.
12. W. J. van Hoeve, The alldifferent Constraint: A Survey, *arXiv e-prints* **cs.PL/0105015** (May 2001) p. cs/0105015.
13. F. Rossi, P. van Beek and T. Walsh, Chapter 4 constraint programming, in *Handbook of Knowledge Representation*, eds. F. van Harmelen, V. Lifschitz and B. Porter, *Foundations of Artificial Intelligence* **3** (Elsevier, 2008) pp. 181–211.
14. N. Beldiceanu, M. Carlsson, S. Demassey and T. Petit, Global constraint catalogue: Past, present and future, *Constraints* **12** (2007) 21–62.
15. J.-C. RÉGIN, A filtering algorithm for constraints of difference in csps, in *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence AAAI'94*, (AAAI Press, 1994), p. 362–367.
16. K. Stergiou and T. Walsh, The difference all-difference makes, in *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 1 IJCAI'99*, (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999), p. 414–419.
17. N. Beldiceanu and E. Contejean, Introducing global constraints in chip, *Mathematical and Computer Modelling* **20**(12) (1994) 97–123.
18. L. De Raedt, A. Passerini and S. Teso, Learning constraints from examples, *Proceedings of the AAAI Conference on Artificial Intelligence* **32** (Apr. 2018).
19. D. Angluin, Queries and concept learning, *Machine Learning* **2** (1988) 319–342.
20. S. D. Prestwich, *Robust constraint acquisition by sequential analysis* (IOS Press, 2020).
21. D. D. Lewis and W. A. Gale, A sequential algorithm for training text classifiers, in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Springer-Verlag New York, Inc., 1994), pp. 3–12.
22. S. Tong and D. Koller, Support vector machine active learning with applications to

text classification, in *Proceedings of the 18th International Conference on Machine Learning* (Morgan Kaufmann Publishers Inc., 2001), pp. 999–1006.
23. H. S. Seung, M. Opper and H. Sompolinsky, Query by committee, in *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory* (ACM, 1992), pp. 287–294.
24. Y. Freund, H. S. Seung, E. Shamir and N. Tishby, Selective sampling using the query by committee algorithm, *Machine Learning* **28**(2-3) (1997) 133–168.
25. S. Dasgupta, Analysis of a greedy active learning strategy, in *Advances in Neural Information Processing Systems* (MIT Press, 2004), pp. 337–344.
26. B. Settles, Active learning literature survey (2009), Computer Sciences Technical Report 1648.
27. D. C. Tsouros, S. Berden and T. Guns, Guided Bottom-Up Interactive Constraint Acquisition, in *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, ed. R. H. C. Yap *Leibniz International Proceedings in Informatics (LIPIcs)* **280**, (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023), pp. 36:1–36:20.
28. N. Beldiceanu, M. Carlsson and J.-X. Rampon, Global Constraint Catalog (2012).
29. H. Simonis and E. Freuder, Pthg21 challenge (August 2021).
30. C. Prud'homme, J.-G. Fages and X. Lorca, Choco solver documentation, *TASC, INRIA Rennes, LINA CNRS UMR* **6241** (2016).
31. D. C. Tsouros, K. Stergiou and C. Bessiere, Structure-driven multiple constraint acquisition, in *Principles and Practice of Constraint Programming*, eds. T. Schiex and S. de Givry (Springer International Publishing, Cham, 2019), pp. 709–725.
32. H. Simonis, Sudoku as a constraint problem **12**, Citeseer, (CP Workshop on modeling and reformulating Constraint Satisfaction Problems, 2005), pp. 13–27.
33. P. Van Hentenryck, *The OPL optimization programming language* (Mit Press, 1999).
34. S. Z. Li and A. Jain (eds.), *Hamming Distance*, in *Encyclopedia of Biometrics*, eds. S. Z. Li and A. Jain. (Springer US, Boston, MA, 2009), Boston, MA, pp. 668–668.
35. T. Petit and A. C. Trapp, Finding Diverse Solutions of High Quality to Constraint Optimization Problems, in *IJCAI. International Joint Conference on Artificial Intelligence* (Buenos Aires, Argentina, 2015).