Constraint Acquisition Based on Solution Counting

Christopher Coulombe
Université Laval, Québec, Canada

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

We propose CABSC, a system that performs Constraint Acquisition Based on Solution Counting. In order to learn a Constraint Satisfaction Problem (CSP), the user provides positive examples and a Meta-CSP, i.e. a model of a combinatorial problem whose solution is a CSP. This Meta-CSP allows listing the potential constraints that can be part of the CSP the user wants to learn. It also allows stating the parameters of the constraints, such as the coefficients of a linear equation, and imposing constraints over these parameters. The CABSC reads the Meta-CSP using an augmented version of the language MiniZinc and returns the CSP that accepts the fewest solutions among the CSPs accepting all positive examples. This is done using a branch and bound where the bounding mechanism makes use of a model counter. Experiments show that CABSC is successful at learning constraints and their parameters from positive examples.

2012 ACM Subject Classification Computing methodologies → Modeling methodologies

Keywords and phrases Constraint acquisition, CSP, Model counting, Solution counting

Digital Object Identifier 10.4230/LIPIcs.CP.2022.27

1 Introduction

Constraint solvers are used to solve complex combinatorial problems. They require an expert to model the problem using the constraints available in the solver. The model creation is a crucial step, but is often time-consuming. One way to save time to the expert is to suggest a model based on sample solutions. For instance, a hospital that wants to automatize the creation of their work schedules for its staff might provide to the experts previous schedules. Assisted with software, the expert wants to discover what constraint generated the examples. While some of these constraints are already known and even written on legal documents, there are as important constraints that are not written but are part of the work culture. These are the constraints for which a constraint acquisition software becomes handy.

When two constraints are candidates for a model, the one that was the most likely used to generate the sample solutions is the most restrictive one [14]. Different approaches exist to decide which constraint is the most restrictive. There are mainly statistical approaches [13, 14] and approaches based on a ranking system [6] (that includes many other criteria). Current methods analyze the constraint in isolation. However, adding to a model a constraint that accepts many solutions can reduce more the solution space than adding a constraint that accepts few solutions. It all depends on the interaction between the constraints in the model. We propose the first approach that takes into account this interaction. It uses a model counter to make sure that the constraints suggested to the expert are those that are the most likely to explain the observed sample solutions given the constraints that were already identified to be part of the model.

In this paper, we propose CABSC, an algorithm for Constraint Acquisition Based on Solution Counting. CABSC uses examples of solutions to evaluate which constraints to keep from a chosen set of candidates. The selection process is based on solution counting using model counters, an approach which differs from the current methods detailed in Section 2.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 27; pp. 27:1–27:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The definitions for our approach are given in the Section 3, followed by a practical explanation in Section 4. Experiments are explained in Section 5 and discussed in Section 6.

2 General Background

Constraint acquisition is an intricate problem that can be solved in a few ways. A first idea called passive learning requires examples of solutions and/or non-solutions. A system chooses which constraint represents best the examples from a preselection of constraints. The preselected pool of constraints from which the model is built is called a bias. Other methods use active learning and generate examples of solution and ask an expert to classify the examples given. From a bias, the system choses the best set of constraints according to the answer provided.

Passive learning systems exploit the idea that the underlying structure of the given examples gives information about the model to learn. Beldiceanu and Simonis [6] created a Model Seeker that learns constraints from a catalog given positive and negative examples. The constraints of the catalog that accepts the positive examples and reject the negative examples are sorted with the more likely constraints having a higher rank. The sorting system is based on a ranking value that is a function of multiple parameters, including the number of solutions satisfying the constraint [5]. A constraint accepting fewer solutions is more likely to be the constraint that generated the examples as there is a lesser chance that the examples are a product of a coincidence. To work, this method needs to make the hypothesis that the constraints learned are independent of each other. That hypothesis is not what transpires in real applications and may result in errors. Two constraints with a small but near identical set of solutions would be picked over two constraints accepting more solutions if picked individually but very few solutions when combined. This is counterproductive as the idea is often to complete an already existing model or to learn multiple constraints at the same time.

Picard-Cantin et al. [13] approached the problem with a statistical approach with the idea that the constraint that best explains the examples is the most improbable one. Equation (1) was therefore used by Picard-Cantin et al. [14] to calculate the probability of the constraints where $G_{\mathrm{C}}(P)$ is the probability that a random assignment satisfies the constraint C with the parameters P. The parameters can be, for instance, the coefficients of a linear equation. $S_{\mathrm{C}}(P)$ is the solution set that satisfies the constraint C with the parameters P. The probability is calculated for a constraint over n variables. $prob(e)$ is the probability to observe an assignment e of n variables and $prob(e_i)$ is the probability to observe an assignment of a single variable.

$$G_{\mathrm{C}}(P) = \sum_{\boldsymbol{e} \in S_{\mathrm{C}}(P)} \mathrm{Prob}(\boldsymbol{e}) = \sum_{\boldsymbol{e} \in S_{\mathrm{C}}(P)} \prod_{i=1}^{n} \mathrm{Prob}(e_i) \tag{1}$$

A hypothesis of independence between the variables of the constraints is applied in the equation. Whenever a variable is in the scopes of multiple constraints, the hypothesis of independence between the variables becomes an approximation. In all cases, the preferred constraints are the ones with a small number of solutions but the independence hypothesis can lead to an erroneous ranking of the constraints. Moreover, this system was not designed for learning multiple constraints and requires solution counting algorithms specialized for each constraint.

Another approach was suggested by Bessiere et al. [8] which consists of creating a model from partial queries, a form of active learning, with an algorithm called QuAcq. The system creates an example and asks an expert whether the presented values are valid. The system adapts the learned constraints depending on the provided answer. Recently, QuAcq was improved with a new version called QuAcq2 [7]. In some cases, QuAcq and QuAcq2 can require a number of queries too high to be efficiently answered by a person. The number of queries can go as high as $n^2 \log(n)$ where n is the number of variables of the problem [7]. Multiple authors tackled this problem such as Daoudi et al. [10], Addi et al. [2], Addi et al. [1], Arcangioli and Lazaar [3], Tsouros et al. [20] and Tsouros et al. [19], but up to thousands of queries can still be needed.

3 The CABSC approach

The CABSC approach (Constraint Acquisition Based on Solutions Counting) we introduce fulfills three goals:

1. To lift the hypothesis of independence between variables;
2. To allow learning multiple constraints;
3. To work with any set of constraints for which filtering algorithms exist, rather than solution counting algorithms.

CABSC models the process of learning constraints as a Meta-CSP. As will be described in Section 3.1, a Meta-CSP is a combinatorial problem whose solution is a CSP. In our case, the solution is the CSP we learn from the examples. When modeling the Meta-CSP, we list the mandatory constraints, i.e. the constraints that we know belong to the model, and also the possible constraints, those that could belong to the model. The variables of the Meta-CSP encode the possible activation of a constraint and also the parameters of the constraints, such as the coefficients of a linear constraint. Solving the Meta-CSP provides the learned model. To do so, we use a branch and bound to decide which constraint to keep and identify the values of the parameters. Our approach uses constraint programming to model a Meta-CSP and to define a family of CSPs from which we can learn. We therefore do not aim to learn any CSP but the optimal CSP among a set programmed through constraint programming. This approach is inspired from regression where one defines a family of functions (e.g. linear functions) and aims at finding the function from this family that best fits the data. Here, we aim at finding the CSP from a family of CSPs defined by the Meta-CSP that best explains the examples.

As there are multiple candidate constraints that could belong to the learned model, we follow Beldiceanu and Simonis [6] and Picard-Cantin et al. [13] by selecting the constraints that minimize the number of solutions. However, instead of analyzing the constraints individually like Beldiceanu and Simonis [6] and Picard-Cantin et al. [13], our system reasons globally on all constraints which allows us to consider multiple different constraints at once.

In order to lift the hypothesis that variables and constraints in a CSP are independent, we directly count the solutions of a model using a model counter. The solution to our Meta-CSP is therefore a CSP whose constraints are satisfied by all observed examples and is as restrictive as possible, i.e. it minimizes the number of solutions.

Our approach has two main differences from existing methods. The first difference is that constraint programming, through the declaration of a Meta-CSP, is used to define a family of CSPs from which we can learn. A second difference from most existing methods is that we use a criterion with a global view on the model to learn by considering the constraints to learn as a whole instead of individually.

3.1 Definition of a Meta-CSP

Following [16], a CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, \text{dom}, C\rangle$ where X is a n-tuple of variables $X = \langle X_1, X_2, \ldots, X_n\rangle$, dom is a function that maps a variable in $X_i \in X$ to a set of values, called domain, that can be assigned to the variable X_i, C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t\rangle$. A constraint C_j is a pair $\langle R_j, S_j\rangle$ where $S_j \subseteq X$ is the scope of the constraint and R_j is a relation on the variables in S_j. In other words, R_j is a subset of the Cartesian product of the domains of the variables in S_j. A solution to the CSP $\mathcal{P}$ is an assignment to the variables $X = v_1, \ldots, X_n = v_n$ such that $v_j \in \text{dom}(X_j)$ $\forall 1 \leqslant j \leqslant n$ and each C_j is satisfied in that the tuple $\langle v_1, \ldots, v_n\rangle$ projected onto S_j is a tuple in R_j.

We extend the definition of a CSP to a Meta-CSP. The solution of a Meta-CSP is a CSP. In our case, it is the CSP we want to learn. A Meta-CSP is a tuple $M = \langle X, P, \alpha, \text{dom}, E, C\rangle$ where $X = \langle X_1, \ldots, X_n\rangle$ are the decision variables, $P = \langle P_1, \ldots, P_q\rangle$ are the parameter variables, $\alpha = \langle \alpha_1, \alpha_2, \ldots\rangle$ are the activation variables, dom is a function that maps a variable in $X \cup P \cup \alpha$ to a set of values that can be assigned to the variable, E is the example matrix of dimensions $m \times n$, and $C = \{C_1, \ldots, C_t\}$ is a set of constraints. A row $e_i = \langle e_{i,1}, \ldots, e_{i,n}\rangle$ of matrix E satisfies $e_{i,j} \in \text{dom}(x_j)$ and is a solution to the CSP we want to learn. The examples of the matrix must satisfy the constraints that we want to learn.

A constraint C_j is a quadruple $\langle R_j, S_j, P_j, \alpha_j\rangle$ where $S_j \subseteq X$ is the scope of the constraint, $P_j \subseteq P \cup \alpha$ its parameters set and $\alpha_j \in \alpha$ its activation variable. For instance, for a linear constraint, the parameters P_j are the coefficients that need to be learned. To each constraint C_j is associated the activation variable α_j with domain $\text{dom}(\alpha_j) \subseteq \{\perp, \top\}$. Deciding whether α_j is true ($\top$) is equivalent to deciding whether the constraint appears in the learned model. One can force a constraint to appear in the learned model by setting $\text{dom}(\alpha_j) = \{\top\}$ in the definition of the Meta-CSP. The relation R_j is a set of the assignments accepted by the constraint along with the parameters given to the constraint: $R_j \subseteq \bigtimes_{x \in S_j} x \times \bigtimes_{p \in P_j}$.

A solution to the Meta-CSP is an assignment to the parameter variables $P_1 = p_1, \ldots, P_q = p_q$ and an assignment to the activation variables $\alpha_1 = r_1, \ldots, \alpha_t = r_t$ such that $r_j \in \text{dom}(\alpha_j)$ for all constraints C_j, $p_k \in \text{dom}(P_k)$ for all $1 \leqslant k \leqslant q$. Finally, the examples must satisfy the activated constraint, i.e. $\forall 1 \leqslant j \leqslant t,\ \alpha_j \implies \forall i\ \langle e_{i,1}, \ldots, e_{i,n}, p_1, \ldots, p_q\rangle \in R_j$.

4 Framework

4.1 The Language

We augmented the MiniZinc language [12] to model a Meta-CSP. The declaration of constraints in a Meta-CSP differs from the one in a CSP in two ways. First, the constraints had to be rewritten in MiniZinc to include the Boolean activation variable. This avoids writing explicitly, for each constraint, the underlying constraints needed for such variables. Second, when declaring the scope of a constraint, the indices of the decision variables in X need to be stored in the constraint. Indeed, the constraint's filtering algorithm needs a map of the decision variables in its scope to the columns of the matrix of examples E. Therefore, constraints used for the Meta-CSP have different specifications from what is possible within MiniZinc, which is why the language had to be augmented. The MiniZinc language was also modified to better communicate with the solver we developed, i.e. imports and heuristics were adapted to give a better control. Even though the modifications to MiniZinc do not change its fundamental structure, the way to write a Meta-CSP is made significantly easier.

Listing 1 provides a code snippet written in the augmented MiniZinc language. A set of two-dimensional points are given as solutions of an unknown CSP problem. We know that the x and y coordinates of these points are nonnegative. We do not know whether these points are subject to a linear inequality or an elliptic inequality. This Meta-CSP will tell us.

```
1  set: domain = 1..10;
2  array: x = [1]; %Points are (x,y)
3  array: y = [2];
4  array: x_y = [1..2];
5  var domain: a;
6  var domain: b;
7  var domain: c;
8  var 0..1: activation1;
9  var 0..1: activation2;
10
11 constraint Linear(x, [1], ">=", 0, true); % x >= 0
12 constraint Linear(y, [1], ">=", 0, true); % y >= 0
13 constraint Linear(x_y, [a,b], "<=", c, activation1);  % a*x + b*y <= c
14 constraint Ellipse(x_y, [a,b], "<=", c, activation2); % a*x^2 + b*y^2 <= c
15 constraint Xor(activation1, activation2, true);
```

Listing 1 Code snippet of the augemented MiniZinc

The decision variables x and y are declared on lines 2 and 3. As their values are known for each example, they are not declared as variables using the keyword *var* but rather as constants corresponding to the column numbers in the example matrix E.

Line 11 declares the first constraint of the problem. It is interpreted as follows: It is a linear constraint whose scope is the decision variable x, whose coefficient vector is [1], whose comparison operator is $\geqslant$, and whose right-hand side is 0. It can be interpreted as $[1]^T x \geqslant 0$. The activation variable is set to *true*, which means that this constraint is known to belong to the CSP. Line 12 imposes $y \geqslant 0$ with a similar constraint. Line 13 encodes the first constraint that we want to learn. It is a linear constraint over the variables x and y whose coefficients and right-hand side are unknown and are represented by the parameter variables a, b, and c. Finally, it is unknown whether this constraint belongs to the CSP. The activation variable `activation1` will be set to 1 if it belongs and 0 otherwise. Line 14 encodes the second constraint that we want to learn. It is an elliptic constraint centered at the origin where parameter variables a, b, and c are reused. The activation variable `activation2` is used for this constraint. Line 15 shows an example of a constraint over two activation variables meaning that exactly one constraint among the linear and the elliptic constraint can be activated. This is an example of how one can define the bias (i.e. the family of CSPs from which the CSP is learned) and exploit the full richness of CP to model the learning process.

A constraint can be satisfied by all examples even if the solver chooses not to learn it by setting its activation variable to $\bot$, unlike a reified constraint which would be set to $\bot$ only if the examples are not satisfied.

Figure 1 is a graphical representation of the problem encoded in Listing 1. The curves represent both candidate constraints: the linear candidate and the elliptic candidate. The dots are the sample solutions that are provided.

We are looking for the CSP that is the most likely the one that generated the points provided in the example matrix E, i.e. the CSP that accepts the fewest solutions among all CSPs that accepts all solutions in E. We see in Figure 1 that the Elliptic constraint accepts 9 solutions while the linear constraint accepts 10 solutions. Therefore, our approach learns that an elliptic inequality fits best the examples with parameters $a = 1, b = 2, c = 10$, `activation1` $= \bot$ and `activation2` $= \top$, which confirms the visual intuition.

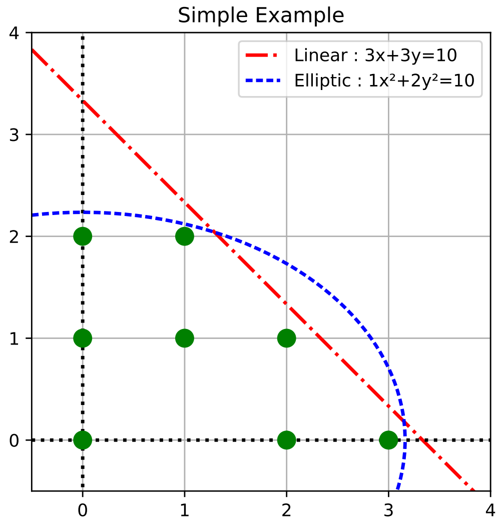

Figure 1 Simple example

4.2 The Solver

We created a custom solver called CabscSolver that reads the Meta-CSP written in the augmented MiniZinc language and the example matrix E. This solver finds the CSP that accepts the fewest solutions among all CSPs that accept all examples. CabscSolver uses a branch and bound to solve the problem. The branching variables are the activation and parameter variables $\alpha \cup P$. After branching, constraint propagation is triggered. Let $C(\vec{x}, \vec{p}, \alpha)$ be a constraint where $\vec{x}$ is the vector of decision variables, $\vec{p}$ is the vector of parameter variables, and α is the activation variable. Only the domains of $\vec{p}$ and α need to be filtered as the values of the decision variables are provided by the examples. To filter the constraint, one needs to filter the expression $\alpha \implies \bigwedge_{i=1}^{m} C(e_i|\vec{x}, \vec{p}, \top)$ where $e_i|\vec{x}$ is the projection of the i^{th} example over the decision variables in the scope of the constraint. The filtering can take place only when the value of the activation variable α is known. Indeed, if α is false ($\bot$), the constraint is satisfied and no filtering is required. If α is true ($\top$), a conjunction of constraints needs to be filtered. Each component of the conjunction can be filtered independently, but a more sophisticated algorithm might process the examples in batch to gain in efficiency. The choice is specific to each constraint. In the example of Listing 1, if variable `activation1` is set to $\top$ during the search process, the linear constraint filters values 1 and 2 from the domain of c as the point $(x, y) = (3, 0)$ prevents the linear constraint to be satisfied when $c \leqslant 2$.

In order to make the branch and bound effective at minimizing the number of solutions accepted by the CSP we want to learn, one needs to compute a lower bound on this number of solutions. This computation is carried in two phases. In the first phase, we detect if a situation occurs where it is possible to deduce which CSP accepts the fewest solutions, regardless whether this CSP accepts the examples or not. If such a CSP can be deduced, the second phase launches a model counter to compute the number of solutions for this CSP.

Some constraints have monotonic parameters with respect to the number of solutions they accept [14]. For instance, consider the linear constraint $c^T x \leqslant b$ where the parameters c

and b are a vector of nonnegative coefficients and a nonnegative right-hand side. The vector x contains the decision variables. It is clear that the number of solutions accepted by this constraint decreases as the values in c increases and b decreases. In order to obtain the most restrictive constraint, one needs to fix the parameters c to their greatest values in their domains and b to its smallest value. If all parameter variables with more than one value in their domain are monotonic and all constraints agree to set these variables to the same values (either largest or smallest) in order to minimize the number of solutions, then we can proceed to the second phase and compute a lower bound on the number of solutions. Otherwise, we use the number of examples as the trivial lower bound as this is the minimum number of solutions the CSP can accept. Since parameter variables can be subject to constraints, it is possible that fixing the value of the parameter variables leads to inconsistencies. In such a case, the CSP used to calculate the lower bound has no solution. Even if that CSP has no solution, multiple CSPs can exist further in the search tree. We therefore still use the number of examples as a lower bound on those nodes.

In the second phase, the parameter variables are set to their most restrictive value and activation variables that are not set to *false* are forced to be *true* in order to have the maximum number of activated constraints. This results in a CSP $\mathcal{A}$ for which the number of solutions needs to be determined. There exists a few model counters in the literature such as the exact probabilistic model counter GANAK [17] or the approximate model counter ApproxMC4 [9, 18]. Both of these counters can only approximate the number of solutions of a model written as a CNF file. CabscSolver encodes the constraints of $\mathcal{A}$ into a pseudo-Boolean language that is translated to a CNF using the MiniSat+ module NaPS [11]. This CNF is given to the model counter which calculates the number of solutions of the model. This number is used as a lower bound on the number of solutions of the learned CSP for the current node of the branch and bound.

Executing the model counter is the most time-consuming operation in the whole search process. Since the parameter variables are often fixed to the same values (due to their monotonicity), it is worth implementing a cache system. Therefore, before calling the model counter, the system checks whether the generated model was previously counted, and if so, returns the number of solutions previously found.

The resulting algorithm is summarized in Figure 2. The next branching is defined by the best-first-search heuristics, i.e. the open node with the smallest lower bound is expanded. When the lower bound of a node is greater than the number of solutions of the incumbent CSP, this node is closed.

5 Experiments

5.1 Implementation

We implemented CabscSolver in Python[1]. While this interpreted language leads to a slow execution, in practice, most of the computation time is spent in the model counters. We use GANAK [17] and ApproxMC4 [9, 18] as model counters that are both efficiently implemented in C/C++.

GANAK is a probabilistic exact model counter [17]. Using the parameter δ, GANAK guarantees with a probability of at least $1-\delta$ that the value provided is an exact count. The approximate model counter ApproxMC4 [9, 18] was also integrated to our solver to count

[1] The code and the benchmarks will be available on the authors' web sites.

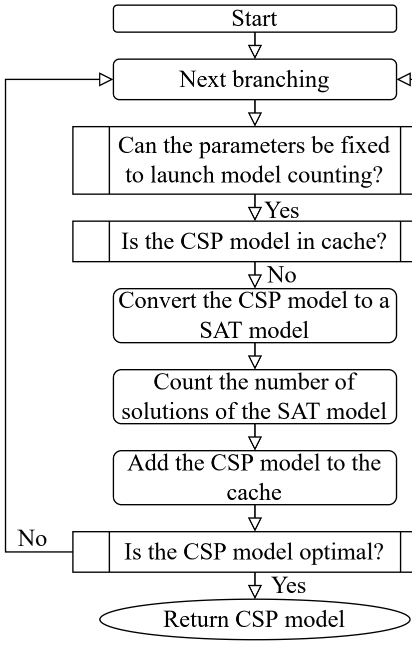

Figure 2 Flow chart for the CABSC approach

the number of solutions since some calculations are much faster with this counter. Let F be the real number of solutions of a model. ApproxMC4 gives an approximation of F with a configurable confidence. More specifically, it returns a count that is guaranteed to be within $[\frac{F}{(1+\epsilon)}, F \cdot (1+\epsilon)]$ with a probability of at least $1-\delta$, where ϵ and δ are the configurable parameters. The chosen values for the parameters ϵ and δ are discussed in Section 5.3.

To read the Meta-CSP models, using the parsing toolkit Lark, we implemented, from scratch, a parser that interprets a subset of the MiniZinc language [12] to which we add the necessary augmentations. MiniZinc was not changed in any way other than the required augmentations. This allows us to efficiently communicate the Meta-CSP models to the solver.

5.2 Instances

We try to learn the constraints inspired from nurse scheduling problems. The problem consists of creating a schedule that respects a set of predetermined rules. In these schedules, the increments used are days, meaning that we are only preoccupied on a daily basis whether the nurses work or not. Let $\eta \in \{2, 3, 4\}$ and $d \in \{7, 14, 21, 28\}$ be the number of nurses and days in a schedule (with $\eta d \leqslant 56$). All instances have a matrix of decision variables $[[X_{(1,1)}, \ldots, X_{(1,d)}], \ldots, [X_{(\eta,1)}, \ldots, X_{(\eta,d)}]]$. Each variable of the matrix represents a day of work for a nurse with its domain being $\{0, 1, 2\}$. $X_{i,j}$ takes the value 0 if the nurse i does not work on day j. If the nurse i does work during day j, $X_{i,j}$ takes the value 1 or 2, depending on whether the nurse works in room 1 or 2.

In the first benchmark, denoted **Sequence**, we want to learn one of these two constraints on the rows of the matrix.

$$\textsc{Sequence}([X_{i,1}, \ldots, X_{i,d}], l, u, k, V) \qquad \forall 1 \leqslant i \leqslant \eta \tag{2}$$

$$\textsc{Among}(t_1, t_2, [X_{i,7w+1}, \ldots, X_{i,7(w+1)}], V) \qquad \forall 1 \leqslant i \leqslant \eta,\ \forall 0 \leqslant w < \frac{d}{7} \tag{3}$$

Constraint (2) is the Sequence constraint [4] that is satisfied when at least l and at most u variables in a window $X_{i,j}, \ldots, X_{i,j+k-1}$ of k consecutive variables are assigned to a value in the set V. This constraint is used to spread out the workload of the nurses over the days without underload nor overload. The parameters l, u, and k are unknown and need to be learned. Their domains are given by $\text{dom}(l) = \text{dom}(u) = \text{dom}(k) = [0, 7]$ and are subject to $l \leqslant u < k$. The set V is known and fixed to $\{1, 2\}$ as these are the values that represent a nurse who is working. Constraint (3) simply constrains the number of work days to be at least t_1 and at most t_2 every week. The parameter variables t_1 and t_2 have for domain $\text{dom}(t_1) = \text{dom}(t_2) = [0, 7]$. One, and only one, constraint among (2) or (3) must be activated. We therefore constrain the activation variables of both constraints with a `Xor`, just like the line 15 of Listing 1. The benchmark **Sequence** is composed of 368 instances generated with distinct constraints, parameters, and examples. These instances satisfy the Sequence constraint and the parameters lie in the intervals $l, u \in [1, 6]$ and $k \in [2, 7]$.

The second benchmark, denoted **Complex**, inherits all the characteristics of the **Sequence** benchmark, including the constraint to learn, to which additional known constraints are added on the decision variables. These constraints have for goal to encode a more realistic situation where constraints that we want to learn are mixed with constraints that are known. For each column $[X_{(1,d)}, \ldots, X_{(\eta,d)}]$ of the matrix that represents the schedule for the day d, we have the constraint $\text{Among}(b, 3, [X_{(1,d)}, \ldots, X_{(\eta,d)}], V)$ where $b = 1$ if d is a Monday, Tuesday, Wednesday, or Thursday and $b = 2$ otherwise. This constraint and its parameters are known and added to the Meta-CSP with an activation variable set to $\top$. This constraint does not need to be learned. For instances with 3 or more nurses, we also have another known constraint $X_{(\eta,j)} = 0 \vee X_{(\eta-2,j)} = 0 \quad \forall j \in \{1, \ldots, d\}$ in order to prevent nurse η from working at the same time as $\eta - 2$. When applicable, this constraint is also included in the Meta-CSP as a known constraint. The **Complex** benchmark has 247 instances that satisfy the Sequence constraint with the parameters lying in the intervals $l, u \in [1, 6]$ and $k \in [2, 7]$.

In the third benchmark denoted **Vacation**, the Meta-CSP is identical to the one of **Complex**. However, the examples E that are provided to the solver are particular: nurses can be non-working for 7 consecutive days. This represents a situation where the staff goes on leave during the vacation period. These leaves violate the Sequence constraint and force the solver to activate the Among constraint and learn its parameters t_1 and t_2. The examples were created such that nurse η never takes a vacation but other nurses do. For a problem spanning w weeks, nurses globally take no more than w weeks of vacation. We generated 272 instances for this benchmark such that the instances satisfy the Among constraint. The parameters lie in the intervals $t_1 \in [2, 3]$ and $t_2 \in [3, 7]$.

The last benchmark **Overtime** uses the same Meta-CSP as **Complex** and **Vacation**, but the examples E provided to learn the CSP differ from **Vacation** on one point: rather than leaving for vacations for 7 consecutive days, the nurses in the **Overtime** benchmark work on a stretch of 7 consecutive days. This represents a situation when the hospital is understaffed and nurses need to work overtime. This benchmark has 304 instances such that the instances satisfy the Among constraint and the parameters lie in the intervals $t_1 \in [2, 7]$ and $t_2 \in [4, 7]$ with the restriction $t_1 \leqslant t_2$.

For all benchmarks, the solver aims to learn exactly one constraint among (2) and (3). The selection depends on the known constraints added to the Meta-CSPs and the examples.

5.3 Experimental Setup

For each instance, the CSP we want to learn was written in the MiniZinc language [12] and used to randomly generate up to a thousand solutions. The Meta-CSP model was written in our augmented-MiniZinc language in order to learn which constraint, between the SEQUENCE and the AMONG constraints, is activated and what are the parameters that were used to generate the examples.

CabscSolver supports two model counters. We first used the solver with the model counter GANAK [17]. By setting the parameter δ to 0.05, we state that the value returned by the model counter is guaranteed to be exact with a probability of at least 0.95. Tighter guarantees can be used, but the time taken to count the number of solutions of the models increases accordingly. Using this model counter and this configuration, we nevertheless assume the given number of solutions to be exact. GANAK was used with a maximum cache size of 2000 Mb. We ran all benchmarks on the solver using this model counter.

As a second series of tests, we used a mix of ApproxMC4 [9, 18] and GANAK. Some CSP models are faster to evaluate with ApproxMC4, so we tried to make CABSC faster using both model counters. Since ApproxMC4 is not an exact model counter, we did not want to run both model counters at the same time and simply use the result returned by the fastest of the two. When using both model counters, GANAK and ApproxMC4 are simultaneously launched. If GANAK finishes first, ApproxMC4 is terminated. If ApproxMC4 finishes first, GANAK is terminated only if the returned result is conclusive. Indeed, ApproxMC4 returns a solution count that is guaranteed to be within an interval with a parametrized confidence. A solution returned by this model counter could be largely underestimated, which could lead to the wrong CSP model being learned. If F is the exact number of solutions of a CSP, the number of solutions returned by ApproxMC4 lies in $[\frac{F}{(1+\epsilon)}, F \cdot (1+\epsilon)]$ with probability $1-\delta$. When ApproxMC4 returns a number of solutions that is $(1+\epsilon)$ times greater than the number of solutions accepted by the incumbent CSP, the computation of GANAK is halted, and the node is closed, i.e. no children of this node will be explored in the search tree. Otherwise, we draw no conclusion and let GANAK terminate its computation. ApproxMC4 is rather used as a means to close nodes faster than substituting GANAK.

The same way we assumed that GANAK would return exact values, we assume that ApproxMC4 does not give a solution count that is lower than the minimum value of the interval. We used $\delta = 0.10$ and $\epsilon = 0.5$ which means that the count calculated is guaranteed to be in the range $[\frac{F}{1.5}, 1.5F]$ with a probability of at least 0.90. A lower probability is accepted from ApproxMC4 than GANAK since the main focus of using ApproxMC4 is to count CSP models faster than GANAK.

We ran the experiments on a computer with the following configuration: CentOS 7.6.1810, 32 GB ram, Processor Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 32 Cores. We simultaneously launch 7 instances of the solver.

From each instance, random subsets of 1, 2, 3, 5, 10, 25, and 100 examples were used. Each time, the top 3 solutions are returned by the solver, and we verify that one of these solutions is the one used to generate the examples. For the **Sequence** and **Complex** benchmarks, the expected constraint to be learned is the SEQUENCE constraint with parameters l, u, and k. For the **Vacation** and **Overtime** benchmarks, the examples violate the SEQUENCE constraint, and the AMONG constraint is expected to be learned with parameters t_1 and t_2.

6 Results and Discussion

Figure 3 presents the results obtained when running CabscSolver using only GANAK for the four benchmarks presented at Section 5.2. On the y-axis is the number of examples that are given to the solver. On the x-axis is the proportion of instances for which the solution is the best one returned by the solver, the second best, the third best, or whether the CSP that

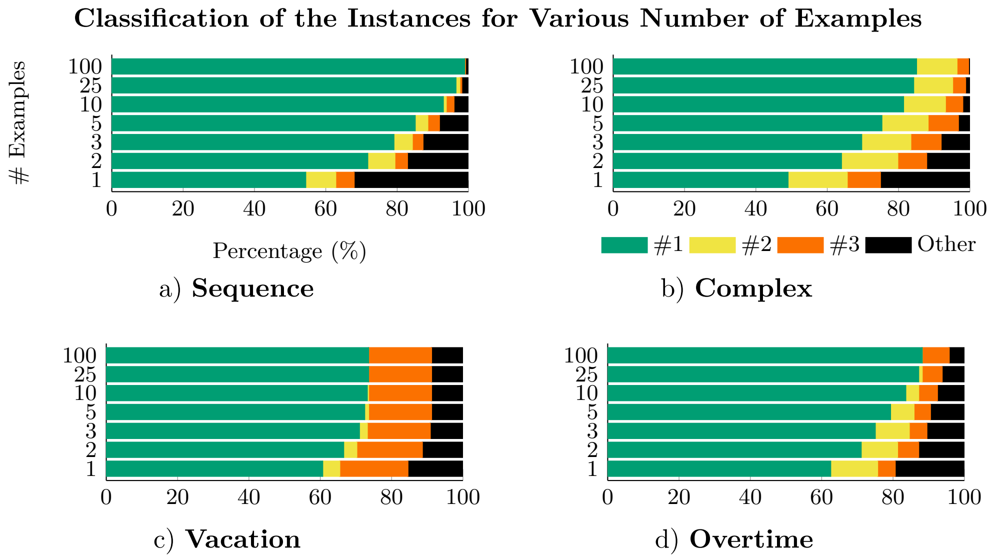

Figure 3 Classification of the instances in percentage for each number of examples. CabscSolver uses GANAK as the only model counter.

was used to produce the examples does not appear at all in the top-3 learned models. We recall that the solver returns the CSP that minimizes the number of solutions.

6.1 Accuracy

CABSC performs generally well as seen in Figure 3. Each benchmark presents a distinct behavior regarding the quality of the results. The first observable behavior is that CABSC succeeds in learning the CSP that was used to generate the data as seen with the benchmark **Sequence**. In this simpler case, the solver has to count the solutions of a conjunction of SEQUENCE constraints, i.e. the constraints to learn. With few examples, our approach stays coherent with the results of Picard-Cantin et al. [13] where they reach above 70% accuracy with a single example of solution, and around 85% accuracy with 5 examples. Extending the number of examples drastically reduces the margin of incorrectly learned instances while the number of examples needed is still relatively low. With only 25 examples, 96.47% of the instances resulted in a correctly learned SEQUENCE constraint at the first try. A few instances could not be resolved even with 100 examples. The unsolved instances occur when the solver finds a more restrictive constraint than the one that was used to generate the examples. This can happen if all the examples given are not enough to filter out parameters that would make the constraints more restrictive. This is why we see that with more examples given, fewer instances remain unsolved. The same phenomenon happens with the **Complex** benchmark where we see an efficient progression as the number of given examples increases.

Finding a more restrictive constraint is not the only way to get an incorrect model. As Figure 3 c) shows, the results for the **Vacation** benchmark converge toward a point where increasing the number of examples does not affect the results while still having a non-negligible proportion (8.82%) of unsolved instances. This is caused by multiple CSPs that are tied. A tie occurs when two distinct CSPs have the same number of solutions. In an instance from **Vacation**, the constraint we want to learn restricts 2 nurses to work a minimum of 3 days and a maximum of 4 days from Monday to Sunday. Since at least one nurse is required to work each day and that a nurse can work a maximum of 4 days within the week, the only way to satisfy the requirements is by having a first nurse working 4 days and the second nurse working 3 or 4 days. It is impossible for one of the nurses to work fewer than 3 days without violating the constraints. The problem comes when setting the value for the minimum number of days a nurse can work during the week. Consider a second selection of parameters where a minimum of 2 working days is required instead of 3. The same solutions are available since this change in parameters does not add solutions. The same goes with a minimum of 1 or 0 working day. This situation leads to four distinct CSPs with the same solution space. Since the objective is to find the CSP accepting the fewest solutions, these four CSPs are equivalent and the solver returns them in an arbitrary order. Most of the unsolved instances in the benchmark **Vacation** have the correct CSP in fourth position, which would have been first if the branching heuristics broke ties differently. We did not observe in our benchmarks situations where the solution spaces differ which let us believe that these models are equivalent. If we pretend for a moment that the **Vacation** benchmark was completed using heuristics that break ties without errors, we obtain the Figure 4.

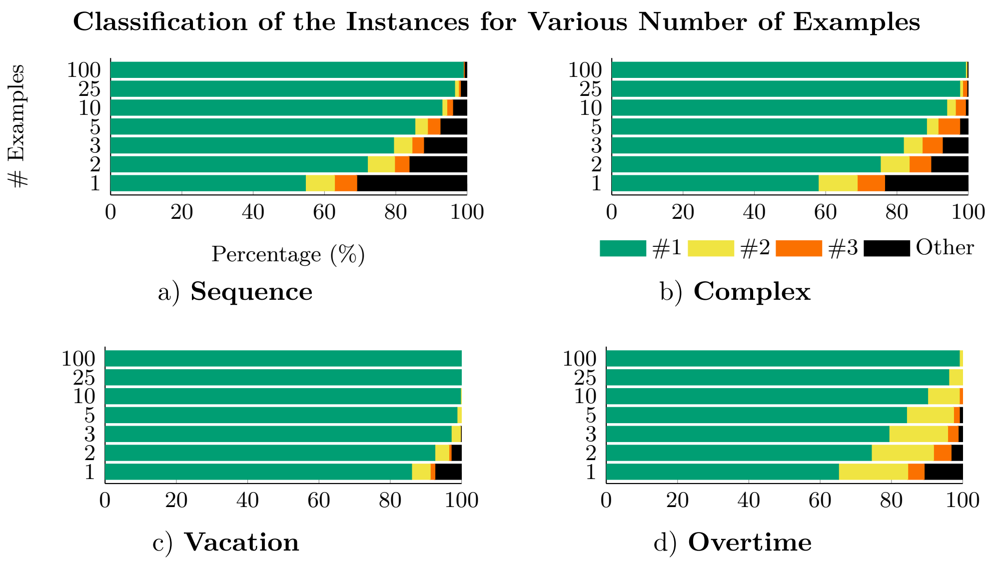

Figure 4 Hypothetical best results for each benchmark

Figure 4 shows that this hypothetical heuristic allows solving perfectly the **Vacation** benchmark using as few as 10 examples. Improvements are also present with the other benchmarks. This confirms that finding equivalent CSPs is the main reason why the solver does not succeed to correctly learn some CSPs.

The unsolved instances from the **Complex** benchmark are mainly caused by constraints found more restrictive than the correct one while the unsolved instances from the **Vacation** benchmark are mostly caused by equivalent CSPs. The unsolved instances of the **Overtime** benchmark are caused by a mix of these two reasons.

The final results show that our model can accurately learn the constraints even when the schedules contain vacations, overtime, or constraints that interfere with the constraints one wants to learn. Few examples are needed to obtain good results. These figures demonstrate that CABSC can learn constraints with the right parameters in diverse situations.

6.2 Execution Time

6.2.1 Using GANAK alone

For the **Complex** benchmark, model counting represents on average 93.6% of the time spent in the solver. Solution counting is a #P-difficult problem with few effective algorithms. Even with state-of-the-art tools, computing a lower bound on the number of solutions can take several minutes. The bound that took the longest time to compute by GANAK took 648 seconds. Figure 5 represents the time taken to solve all instances, i.e. the $(368 + 247 + 272 + 304) \times 7$ instances that come from the four benchmarks that were solved with 1, 2, 3, 5, 10, 25, and 100 examples using only GANAK as a model counter. Most of the instances are solved within a minute, but the solving time quickly and abruptly rises. This time limitation comes from a few main elements.

First, the size of the Meta-CSP greatly impacts the time needed for CABSC to find a solution. This size is measured in the number of parameter variables and activation variables since their number affects the depth of the search tree, thus the number of nodes explored in the branch and bound. For our instances, a few hundreds nodes could be observed on average resulting in around 30 to 60 unique calls to a model counter.

Second, the examples also impact the total runtime in two ways. With a higher number of examples, the solver is able to filter out more values from the domain of the parameter variables which directly decreases the number of potential calls to a model counter. Using a single example, the instances in the **Complex** benchmark takes on average 385.3 seconds to solve. With a hundred examples, the average time drops to 306.9 seconds, an improvement of 20.35%. The second way the solving time is impacted by the examples is with their length, i.e. the number of decision variables. The more decision variables, the more Boolean variables in the SAT model to count. For this reason, we were not able to learn the constraints of schedules with a horizon of 56 days or more.

Lastly, all bounds do not take the same computation time. Indeed, we obtain SAT instances with various numbers of Boolean variables and clauses. The internal structure of these SAT instances can also vary. The bound that is the slowest to calculate uses a SAT instance with 672 Boolean variables and 1172 clauses and takes 648 seconds to count. The

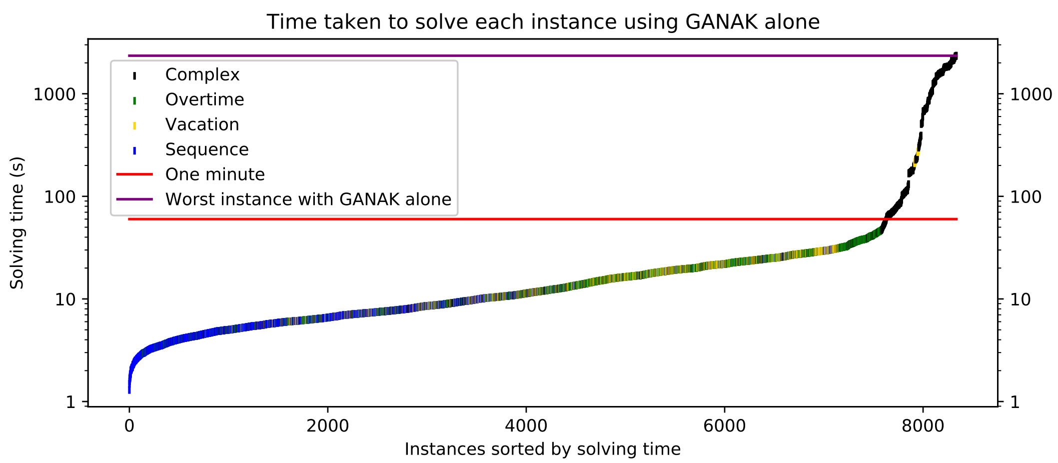

Figure 5 Measures of time for all instances using GANAK alone

Boolean model with the greatest number of variables has 804 variables and 2052 clauses and is counted in 0.11 seconds. This demonstrates that the counting time does not only depend on the number of decision variables, but also the structure of the problem.

6.2.2 Using both GANAK and ApproxMC4

One method used to improve the time needed to solve a Meta-CSP is by combining a probabilistic exact model counter with an approximate model counter. This allows some CSP models to have their solutions counted quicker. The way ApproxMC4 was added to CabscSolver was to use it to prune CSP models from the search tree when the number of solutions was reasonably far from the number of solutions of the best CSP model found so far, as explained in Section 5.3.

This method is a lot faster than using GANAK as the only model counter as demonstrated by the Figure 6. The worst instance with GANAK alone lasted 2350 seconds while the same instance lasted 1099 seconds using ApproxMC4. The arithmetic average solving time of the **Complex** drops from 333.0 seconds to 158.7 seconds. This represents an improvement of 52.3% in average. The geometric average drops from 54.2 seconds to 41.0 seconds, an improvement of 24.4%.

The results obtained using both GANAK and ApproxMC4 have a lower accuracy by a small margin. While the accuracy of the results for the **Sequence**, **Vacation** and **Overtime** benchmarks remain unchanged, **Complex** suffers slight changes when few examples of solutions are given. Since the results have no significant differences to be seen on a graph, the changes are textually reported. With a single example of solution, the percentage of correctly learned CSP models drops from 48.99% to 48.48%. When using two examples of solutions, the percentage of correctly learned CSP models drops from 63.97% to 63.56% and with three examples, it drops from 69.64% to 68.83%. When using five examples of solutions or more, adding ApproxMC4 do not change the results anymore. All the other accuracy results are exactly the same, whether ApproxMC4 was used or not.

The lack of changes in the accuracy of **Sequence**, **Vacation** and **Overtime** benchmarks is mainly caused by the fact that ApproxMC4 returns approximations that are often too

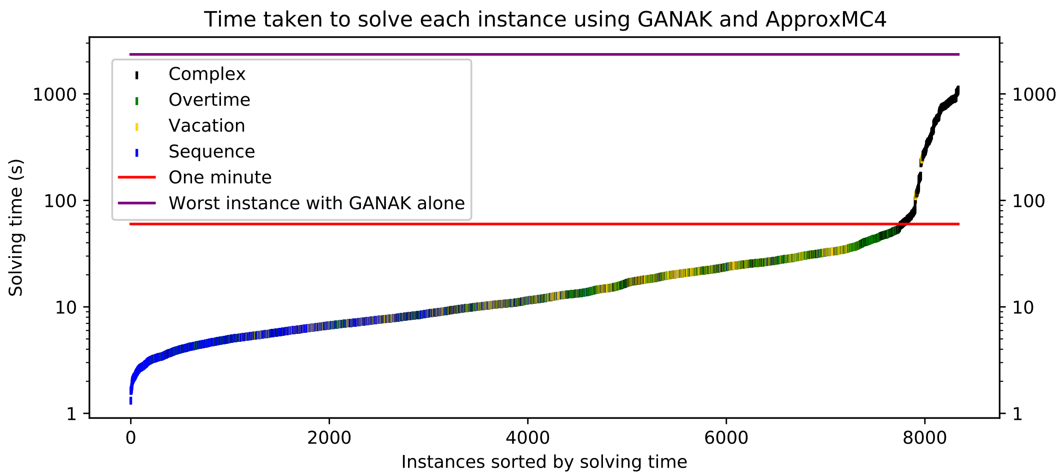

Figure 6 Measures of time for all instances using GANAK with ApproxMC4

close to take into account. The solver then has to ask GANAK to finish calculating the number of solutions of the CSP model regardless of the time needed by ApproxMC4. For the **Complex** benchmark, many CSP models were approximated by ApproxMC4 a lot faster than GANAK could and with values that allow pruning many nodes. ApproxMC4 sometimes overestimates the count of solutions outside the wanted interval of values. Since we used $\delta = 0.10$ for the model counters, ApproxMC4 therefore has a probability of at most 0.10 to return values outside the wanted interval. This can cause many of the evaluations to accidentally prune correct CSP models, which can cause the Meta-CSP not to be properly solved. On the opposite side, it is possible to see improvements in the CSP learned due to overestimations that prune CSP models that would be learned if counted exactly. This happened on few instances from the **Complex** benchmark where the correct CSP went from being the third suggestion to the second. Since the correct CSP was not suggested as a first choice, the accuracy of correctly learned CSP models did not improve from these.

6.3 Potential Improvements

There exist several open source model counters that are efficient at counting SAT models, but fewer available programs to count the solutions of a CSP. Translating Sequence constraints into pseudo-Boolean constraints and then to CNF offers no guarantee in the efficiency of the model. Directly counting the solution of a CSP could be faster and would certainly prevent from translating the model.

Parallelization could also speed up the exploration of the search tree. An approach like Embarassingly Parallel Search [15] could be appropriate, but also parallelization within the model counters would be suited as it is offered by ApproxMC3 [9, 18].

7 Conclusion

We introduced CABSC, a technique for Constraint Acquisition Based on Solution Counting. Our approach learns the CSP that accepts all provided examples but that minimizes the size of its solution space. This criterion has proven to return good solutions. The branch and bound uses model counters to compute a bound on the number of solutions for a given CSP. Experimental results show that CABSC successfully learns models and require few examples for our benchmarks.

References

1. Hajar Ait Addi and Redouane Ezzahir. P_a-quacq: Algorithm for constraint acquisition system. In *Smart Data and Computational Intelligence*, pages 249–256, 2019.
2. Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. Time-bounded query generator for constraint acquisition. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018)*, pages 1–17, 2018.
3. Robin Arcangioli and Nadjib Lazaar. Multiple constraint acquisition. In *Proceedings of the 2015 International Conference on Constraints and Preferences for Configuration and Recommendation and Intelligent Techniques for Web Personalization*, pages 16—-20, 2015.
4. Nicolas Beldiceanu and Évelyne Contejean. Introducing global constraints in chip. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. Nicolas Beldiceanu and Helmut Simonis. A constraint seeker: Finding and ranking global constraints from examples. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, pages 12–26, 2011.

6 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, pages 141–157, 2012.

7 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. Partial queries for constraint acquisition. Technical Report abs/2003.06649, CoRR, 2020.

8 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In Francesca Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 475–481, 2013.

9 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 3569–3576, 2016.

10 Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El-Houssine Bouyakhf. Constraint acquisition with recommendation queries. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 720–726, 2016.

11 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

12 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543, 2007.

13 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning parameters for the sequence constraint from solutions. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pages 405–420, 2016.

14 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning the parameters of global constraints using branch-and-bound. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, pages 512–528, 2017.

15 Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 596–610, 2013.

16 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

17 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-19)*, pages 1169–1176, 2019.

18 Mate Soos and Kuldeep S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, pages 1592–1599, 2019.

19 Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, pages 709–725, 2019.

20 Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, pages 373–388, 2018.