Using Large Language Models to Improve Query-based Constraint Acquisition

Younes Mechqrane[1] , **Christian Bessiere**[2] , **Ismail Elabbassi**[1]

[1]Ai movement - International Artificial Intelligence Center of Morocco - University Mohammed VI Polytechnic, Rabat, Morocco

[2]CNRS, University of Montpellier, France

{younes.mechqrane, ismail.elabbassi}@um6p.ma, bessiere@lirmm.fr

Abstract

Most active constraint acquisition systems suffer from two weaknesses. They require the explicit generation of the set of potential constraints (the bias), whose size can be prohibitive for practical use of these systems, and the answers to queries contain little information. In this paper, we introduce ACQNOGOODS, an active learning schema that does not require the construction of a bias. We then propose LLMACQ, an active learning system that incorporates a Large Language Model component in the ACQNOGOODS schema. LLMACQ interprets the user's answers given in natural language, leading to more informative communication. As our experiments show, the non requirement of a bias in extension combined to the higher-level communication with the user allow LLMACQ to learn constraints of any arity and to dramatically decrease the number of queries.

1 Introduction

In Constraint programming (CP), the user specifies the constraints of the problem and the solver searches for solutions. Specifying the constraints of the problem can be a challenge for non-CP experts. The purpose of constraint acquisition is to offer assistance to the user in specifying the constraints.

There are two main categories of constraint acquisition systems: passive learning algorithms and active learning algorithms. In passive learning systems, the user provides a set of examples of solutions and/or non-solutions, and the system learns a set of constraints that correctly classifies these examples (CONACQ1 [Bessiere *et al.*, 2004; Bessiere *et al.*, 2005], MODELSEEKER [Beldiceanu and Simonis, 2012], SEQACQ [Prestwich, 2020], COUNT-CP [Kumar *et al.*, 2022]). Passive learning systems may fail when the user does not provide a set of examples sufficiently representative of the problem. Active learning can help overcome this limitation by asking the user queries whose answers will help the system in learning missing constraints for converging on a unique constraint network. In CONACQ2, complete assignments of the variables of the problem are proposed to the user for classification as positive or negative [Bessiere *et al.*, 2007; Bessiere *et al.*, 2017]. CONACQ2 may require an exponentially large number of queries to converge. To overcome this issue, QUACQ asks the user to answer partial queries, that is, queries involving a subset of the problem variables [Bessiere *et al.*, 2013]. Numerous QUACQ-based algorithms have been introduced in the literature, incrementally improving the basic version (e.g., MULTIACQ [Arcangioli *et al.*, 2016], MQUACQ [Tsouros *et al.*, 2018], MQUACQ2 [Tsouros *et al.*, 2019], PREDICT&ASK [Daoudi *et al.*, 2016], QUACQ2 [Bessiere *et al.*, 2023a]).

A common weakness of all these systems is that the communication between the user and the learner is limited to "yes/no" answers. The consequence is that we need strong restrictions on the learning bias if we want to have a chance to learn a network in a reasonable time and/or number of examples. Some systems, such as MODELSEEKER or COUNT-CP, put strong restrictions on where a constraint may lie (e.g., on rows, columns of a matrix). Most others put strong restrictions on the kind of relations to be used to define constraints (e.g., CONACQ-based, SEQACQ, QUACQ-based). The problem is even more accurate on active learners because they usually need to explicitly represent the learning bias, that is, the set of all possible constraints that could belong to the target network. For instance, in a problem with 50 variables, the relation stating that the sum of three variables is equal to a given constant k would give rise to $\binom{50}{3} = 19,600$ constraints in the bias. Two recent works address this issue. In [Bessiere *et al.*, 2023b], a passive learner is proposed that does not need any language or bias to be explicitly represented because its goal is to learn the (small set of) relations that will be used to express the whole problem. GROWACQ is an active learner that builds the bias step by step by adding variables incrementally during the learning process. It is shown that it allows the system to deal with much larger biases than other QUACQ-based systems [Tsouros *et al.*, 2023].

In this paper we address the two issues described above: the problem of the size of the learning bias and the low amount of information contained in user's answers. Our first contribution is an active constraint acquisition schema, called ACQNOGOODS, that is able to acquire any constraint network without the need of a bias. (In fact, it does not even need a language). Like QUACQ-based systems, ACQNOGOODS uses partial queries. To avoid the need of a bias, ACQNOGOODS learns forbidden tuples (aka, nogoods) rather than

constraints. The downside is obviously a potentially large number of queries. But ACQNOGOODS is not intended to be used as a standalone acquisition system. It will serve as a basic framework in which our main contribution will be embedded.

Our main contribution is to improve QUACQ-based constraint acquisition by enhancing the amount of information contained in the user's answers to queries. A first attempt in that direction was proposed in [Freuder and Wallace, 1998]. When asked to classify a negative example, the user provides a 'correction', that is, the set of constraints of the problem that make the example negative. Such a scenario requires the user to be able to articulate and express the constraints rejecting the example, which can be too hard for a non-CP expert. In this paper, we consider the scenario in which, when the user classifies an example as negative, she provides a sentence in natural language that describes a reason why the example is not satisfactory. Allowing natural language communication should make the task much easier to the user, and the impressive success of Large Language Models (LLMs) tells us that it is the right time to do this. We then propose LLMACQ, a constraint acquisition system based on ACQNOGOODS that embeds natural language understanding capabilities. The goal is to showcase how natural language processing techniques can be used in active constraint acquisition algorithms. We show that by fine-tuning BERT [Devlin *et al.*, 2019] with a pre-defined language of relations, we provide LLMACQ with capabilities for extracting a constraint from the user's feedback as long as the constraint is defined by a relation in our pre-defined language. Our experiments demonstrate that LLMACQ significantly outperforms QUACQ2, requiring far fewer queries while successfully learning constraints of any arity. This is not the first time natural language processing is used to help expressing problems in CP. In [Kiziltan *et al.*, 2016], a complete textual description of a problem is used to detect constraints. However, to our knowledge, this is the first time that natural language understanding techniques are used to extract constraints from user's feedback in the context of active constraint acquisition.

The rest of the paper is organized as follows. Section 2 gives the necessary background. Section 3 describes the ACQNOGOODS algorithm. In Section 4, we show how we can fine-tune BERT to handle user's feedback. LLMACQ is described in Section 5. Section 6 presents the experimental results. Section 7 concludes the paper.

2 Background

In the following we assume that the user and the learner share a common *vocabulary* to communicate. The vocabulary is a finite set of variables X and a domain D for these variables. A constraint c is defined by a pair $(var(c), rel(c))$, where $rel(c)$ is the relation specifying which combinations of values (or *tuples*) are allowed for the variables $var(c)$. $var(c)$ is called the scope of c and $|var(c)|$ the arity of $rel(c)$. A *constraint network* is a set C of constraints on the vocabulary (X, D). The projection $C[S]$ of a constraint network C on a subset S of variables is the set $\{c \in C \mid var(c) \subseteq S\}$. An example e is a (partial/complete) assignment on a set of variables $var(e) \subseteq X$. e is *rejected* by a constraint c if and only if $var(c) \subseteq var(e)$ and the projection $e_{var(c)}$ of e on $var(c)$ is not in $rel(c)$. The constraint with scope $var(c)$ that rejects only the tuple $e_{var(c)}$ is called a *nogood* and is denoted by $(var(c), \neg e_{var(c)})$. A complete assignment e of X is a solution of C if none of the constraints in C reject e. The set of solutions of C is denoted by $sol(C)$. In addition to the vocabulary, the learner may own a *language* Γ of relations from which it can build constraints on specified sets of variables. A constraint *bias* is a set of constraints built from a language Γ on the vocabulary (X, D). Formally speaking, the bias is the set $\{c \mid (var(c) \subseteq X) \wedge (rel(c) \in \Gamma)\}$.

The *target network* is a constraint network T on (X, D) such that $sol(T)$ is equal to the set of solutions of the problem that the user has in mind. A *query* $\texttt{Ask}(e)$, with $var(e) \subseteq X$ and $e \in D^{var(e)}$, asks the user whether e satisfies all the constraints in $T[var(e)]$ or not. Depending on whether the answer is "yes" or "no", the example is called *positive* or *negative*, respectively.

3 Learning Nogoods

In this section, we present the ACQNOGOODS constraint acquisition schema. Like QUACQ-based algorithms, ACQNOGOODS uses partial queries to learn the target network. But, as opposed to QUACQ-based algorithms, ACQNOGOODS does not need any predefined language or bias. ACQNOGOODS does not learn constraints whose relation belongs to a pre-defined language. ACQNOGOODS learns nogoods.

3.1 Technical Description

ACQNOGOODS is presented in Algorithm 1. ACQNOGOODS takes a vocabulary (X, D) as input and returns a learned network L composed of a set of nogoods such that L has the same solutions as the target network T. ACQNOGOODS initializes L to the empty set (line 1). The set `solutions`, used to store the complete examples classified as positive by the user, is also initialized to the empty set (line 2). At each iteration of the while loop in line 3, ACQNOGOODS tries to generate a complete example e that is not already in `solutions` and that is not rejected by any of the nogoods learned so far (line 4). If it fails to generate such an example, ACQNOGOODS has converged and returns the network L, which has the same solutions as T (line 5). Otherwise, ACQNOGOODS asks the user to classify e. If the user's answer is positive, e is added to `solutions` (line 7). If not, ACQNOGOODS calls function `OneNogood` that returns a minimal scope S such that e_S is forbidden by a constraint of T (line 9). $(S, \neg e_S)$ is added to L (line 10).

Algorithm 2 describes function `OneNogood`. Its behavior is very similar to that of function `FindScope` [Bessiere *et al.*, 2013]. `OneNogood` inherits the logarithmic complexity in number of queries of `FindScope`.

The inputs of `OneNogood` are an example e, two sets of variables R and Y, and a Boolean ask_query. An invariant of `OneNogood` is that e violates at least one constraint of the target network T, whose scope is a subset of $R \cup Y$. Any recursive call of `OneNogood` returns a subset of Y that is a subset of the scope of a constraint in T that is violated

Algorithm 1: ACQNOGOODS

Input: X, D
Output: a learned network L equivalent to T

1 $L \leftarrow \emptyset$;
2 `solutions` $\leftarrow \emptyset$;
3 **while** *true* **do**
4 $\quad e \leftarrow$ generate $e \in sol(L) \setminus$ `solutions` ;
5 $\quad$ **if** $e = \bot$ **then return** *"convergence on L"* ;
6 $\quad$ **else**
7 $\quad\quad$ **if** $Ask(e)$ = yes **then** add e to `solutions` ;
8 $\quad\quad$ **else**
9 $\quad\quad\quad S \leftarrow$ `OneNogood`$(e, \emptyset, X, false)$;
10 $\quad\quad\quad$ add $(S, \neg e_S)$ to L ;

Algorithm 2: Function `OneNogood`

Input: an example e, two scopes R, Y, and a Boolean ask_query
Output: a scope S such that e_S is a nogood

1 **if** ask_query **then if** $\texttt{Ask}(e_R) = no$ **then return** $\emptyset$;
2 **if** $(|Y|) = 1$ **then return** Y ;
3 split Y into $\langle Y_1, Y_2 \rangle$ such that $|Y_1| = \lceil |Y|/2 \rceil$;
4 $S_1 \leftarrow$ `OneNogood`$(e, R \cup Y_1, Y_2, true)$;
5 $S_2 \leftarrow$ `OneNogood`$(e, R \cup S_1, Y_1, (S_1 \neq \emptyset))$;
6 **return** $S_1 \cup S_2$;

by e. The Boolean ask_query, when false, tells us that we already know that the answer to $\texttt{Ask}(e_R)$ will be "yes" (because R cannot contain the scope of a constraint rejecting e). If ask_query is true, we ask the user to classify e_R. If the answer is "no", R contains the scope of a constraint in T that rejects e. Thus, Y is not needed to cover that scope, and an empty set is returned (line 1). Line 2 is only reached if e_R does not violate any constraint. Knowing that $e_{R \cup Y}$ violates a constraint, if Y is a singleton, it must be part of the scope of a violated constraint in $e_{R \cup Y}$ and line 2 returns Y. If no return conditions are met, Y is divided into two balanced segments, Y_1 and Y_2 (line 3), and `OneNogood` is called recursively to elucidate the variables involved in a constraint violated by $e_{R \cup Y}$ (lines 4 and 5). The whole process is logarithmic in the number of variables.

3.2 Correctness of ACQNOGOODS

Proposition 1 (Partial correctness)**.** *Given a target network T, the network L returned by* ACQNOGOODS *is such that $sol(L) = sol(T)$.*

Proof. Completeness. Assume that $sol(L) \not\subseteq sol(T)$. Let e be a solution of L that is not solution of T. e cannot be in the set `solutions` because the only way for an example to enter `solutions` is to have been classified as positive by the user (Algorithm 1, line 7), which is impossible because e is not solution of T. Hence, when e was generated in line 4 of Algorithm 1 (because $e \in sol(L)$), it has been classified as negative by the user. As a result, `OneNogood` was called and a nogood rejecting e has been added to L (Algorithm 1, lines 9 and 10). This contradicts the assumption that $e \in sol(L)$.

Soundness. Assume now that $sol(T) \not\subseteq sol(L)$. This means that there exists a nogood $(S, \neg e_S)$ in L that rejects a solution e of T. For $(S, \neg e_S)$ to be in L, S must have been extracted by `OneNogood` from an example e' that was classified as negative by the user. Following the proof of correctness of `FindScope` in [Tsouros and Stergiou, 2020], we know that `OneNogood` extracts a minimal subset S of X such that e'_S violates a constraint in T. Therefore, every superset of e'_S, including e, will violate a constraint from T. This contradicts the assumption that e belongs to $sol(T)$. □

Proposition 2 (Termination)**.** ACQNOGOODS *terminates.*

Proof. The nogoods learned by ACQNOGOODS are not redundant because every new learned nogood is extracted from an example that satisfies all the nogoods learned so far (Algorithm 1, line 4). Thus, the number of nogoods learned by ACQNOGOODS is finite. In addition, at each iteration of the main loop of ACQNOGOODS (Algorithm 1, line 3), either we learn a new solution e of T, or a new nogood (Algorithm 1, lines 7 and 10). Hence, ACQNOGOODS terminates because both the number of solutions of T and the number of non-redundant nogoods are finite. □

Theorem 1 (Correctness)**.** *Given a target network T, Algorithm 1 converges on a network L such that $sol(L) = sol(T)$.*

Proof. Direct from propositions 1 and 2. □

4 Using an LLM for Constraint Extraction

In ACQNOGOODS and in all QUACQ-based algorithms, the user only provides "yes/no" answers to the queries of the system. That is, the user only confirms whether an example meets her requirements or not. In this section, we consider the scenario in which, when the answer is "no", the user provides a sentence in natural language describing a reason why the example is negative. For instance, the user may answer that "we must get 40 when we add up the variables x_4 and x_5" or "make sure 80 is the exact distance between x_{76} and x_{59}". To capture this scenario, we extend the definition of $\texttt{Ask}(e)$ queries:

> An *extended* query $\texttt{E-Ask}(e)$, with $var(e) \subseteq X$ and $e \in D^{var(e)}$, asks the user whether e satisfies all the constraints in $T[var(e)]$ or not. If the answer is not "yes", the user provides a sentence of feedback describing one constraint rejecting e.

When receiving such a sentence, our constraint acquisition system will use the power of large language models to interpret the text and identify the constraint outlined by the user.

As a state-of-the-art pre-trained language model, BERT (Bidirectional Encoder Representations from Transformers [Devlin *et al.*, 2019]) has achieved significant results in many language understanding tasks, particularly in tasks involving token and sequence classification. To adapt BERT to natural language processing tasks in a specific domain, an appropriate fine-tuning strategy is required.

In our context, the identification of a constraint from a user's sentence can be divided in two subtasks: *Relation identification* and *Parameters identification*. Relation identification is the task of identifying the relation the user has in mind. For instance, if the answer of the user is "make sure 80 reflects the exact distance between x_{76} and x_{59}", it

should understand that the relation is $||_{val}$, which characterizes the fact that the absolute value of the difference between two variables is equal to a numerical constant. Relation identification involves performing sequence classification, each class corresponding to a relation in the given language Γ of possible relations. Parameters identification can be seen as a token classification task where the classes associated with the tokens (i.e., words) are used to identify the parameters of the constraint. For instance, if the answer of the user is again "make sure 80 reflects the exact distance between x_{76} and x_{59}", the result of the token classification should be the sequence $[0, 0, 3, 0, 0, 0, 0, 0, 1, 0, 2]$, which means that the ninth word in the user's answer is the first parameter of the constraint, the eleventh word is the second parameter, and the third word is the third parameter. Words that correspond to zeros do not count as parameters of the constraint. Instead of processing relation identification and parameters identification as two separate subtasks, we chose to apply a multi-task learning strategy, where we simultaneously fine-tune BERT on both relation and parameters identification tasks. The goal is to boost efficiency by learning shared representations between these two related tasks.

Fine-tuning BERT begins with a step of data preprocessing, involving tokenization and dealing with variations in sequence length. In tokenization, input sentences are split into smaller units, or tokens, which can be either words or subwords. This process is essential for adapting the text to the input format of BERT, which is based on a fixed vocabulary. After tokenization, each sequence is prefixed with a [CLS] token. This special token plays a key role in sequence classification tasks. It is designed to capture the context of the entire input sequence. A [SEP] token is used to mark the end of the sentence, followed by padding to ensure a uniform input structure for the neural network. Thus, the format for a sentence is [CLS] *sentence* [SEP], plus as many [PAD] tokens as needed to meet the fixed length requirement.

Once the data has been preprocessed, we use a dual-headed approach to handle both relation and parameters classification (see Figure 1). The relation classification head, which is a linear classification layer shown in blue in Figure 1, uses the [CLS] token representation, which after passing through BERT's layers, contains aggregated information from the entire input sequence. On the other hand, the parameters classification head, the other linear classification layer shown in green in Figure 1, works with the representations of individual tokens. Each token representation, obtained from BERT's final layer, is used for token-level classification.

During training, minibatches are processed separately for each task. 50% of the minibatches focus on relation classification while the other 50% focus on parameters classification. This separate processing ensures that the model is evenly trained on both types of tasks. After each minibatch is processed, the model computes the categorical loss for the task at hand and uses it to perform backpropagation, where the gradients are calculated, and the weights of the neural network are adjusted accordingly. By doing so, the neural network gradually improves its ability to accurately classify both relations and parameters.

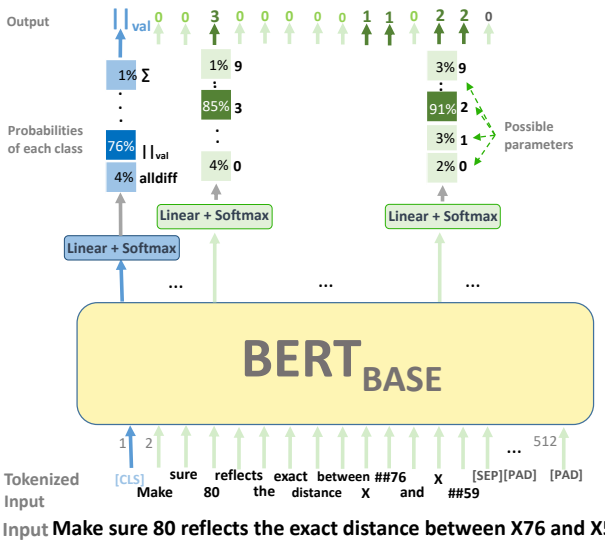


Figure 1: Illustration of the fine-tuned BERT model used on the sentence "Make sure 80 reflects the exact distance between $X76$ and $X59$". After tokenization, and with the addition of the special tokens [CLS], [SEP] and [PAD], the input sentence becomes: "[CLS] Make sure 80 reflects the exact distance between X $\#\#76$ and X $\#\#59$ [SEP][PAD]...[PAD]". This tokenized sentence is then passed through the fine-tuned BERT. The sequence classification head, in blue, uses the [CLS] token representation for relation classification. Simultaneously, the parameters classification head, in green, focuses on the representations of individual tokens. The relation identification head predicts $||_{val}$ as the most likely relation. This relation refers to the absolute value of the difference between two parameters, $param_1$ and $param_2$, being equal to a third parameter, $param_3$: $|param_1 - param_2| = param_3$. The parameter identification head, in green, classifies the token 80 as class 3, the tokens X and $\#\#76$ as class 1, X and $\#\#59$ as class 2, and all other tokens as class 0. This means that $X76$, $X59$, and 80 are identified respectively as the first, the second and third parameters of the relation $||_{val}$. The other tokens do not count as parameters of the constraint. The fine-tuned BERT successfully identifies the relation and its parameters.

5 Constraint Acquisition with an LLM

Now that we have a way to exploit the user's feedback given in natural language, we are ready to present a revised version of ACQNOGOODS that will use this feedback to improve constraint acquisition. In addition to the extended queries $\texttt{E-Ask}(e)$ defined in Section 4, we also need another type of query, already used in [Beldiceanu and Simonis, 2012; Daoudi *et al.*, 2016], to validate the understanding of the BERT component:

> A *validation* query $\texttt{V-Ask}(c)$ asks the user whether the constraint c belongs to the target network T or not. The answer is "yes" or "no".

Our new constraint acquisition algorithm, called LLMACQ, is described in Algorithms 3 and 4. The general structure of LLMACQ is similar to that of ACQNOGOODS. The first difference is that LLMACQ uses $\texttt{E-Ask}(e)$ queries instead of $\texttt{Ask}(e)$ queries. When the answer to a $\texttt{E-Ask}(e)$ query is not "yes", the feedback provided by the user is given to

Algorithm 3: LLMACQ

Input: X, D, maxPos
Output: a learned network L equivalent to T

1 $L \leftarrow \emptyset$;
2 solutions $\leftarrow \emptyset$; $\#pos \leftarrow 0$;
3 **while** $\#pos <$ maxPos **do**
4 $e \leftarrow$ generate $e \in sol(L) \setminus$ solutions ;
5 **if** $e = \bot$ **then return** *"convergence on L"* ;
6 **else**
7 $Answer \leftarrow$ E-Ask(e);
8 **if** $Answer = yes$ **then**
9 add e to solutions ; $\#pos \leftarrow \#pos + 1$;
10 **else**
11 $\#pos \leftarrow 0$;
12 $c \leftarrow$ BERT $(Answer)$;
13 **if** V-Ask$(c) = yes$ **then** add c to L ;
14 **else**
15 $S \leftarrow$ OneConstraint$(e, \emptyset, X, false)$;
16 **if** $S \neq$ "$exit$" **then** add $(S, \neg e_S)$ to L ;
17 **return** *"pseudo-convergence on L"* ;

Algorithm 4: Function OneConstraint

Input: an example e; two scopes R, Y, and a Boolean ask_query
Output: a scope S such that e_S is a nogood or "exit" if a constraint from T has been added to L

1 **if** ask_query **then**
2 $Answer \leftarrow$ E-Ask(e_R);
3 **if** $Answer \neq yes$ **then**
4 $c \leftarrow$ BERT $(Answer)$;
5 **if** V-Ask$(c) = yes$ **then**
6 add c to L ; **return** "$exit$" ;
7 **else return** $\emptyset$;
8 **if** $|Y| = 1$ **then return** Y ;
9 split Y into $< Y_1, Y_2 >$ such that $|Y_1| = \lceil |Y|/2 \rceil$;
10 $S_1 \leftarrow$ OneConstraint$(e, R \cup Y_1, Y_2, true)$;
11 **if** $S_1 =$ "$exit$" **then return** "$exit$" ;
12 $S_2 \leftarrow$ OneConstraint$(e, R \cup S_1, Y_1, (S_1 \neq \emptyset))$;
13 **if** $S_2 =$ "$exit$" **then return** "$exit$" ;
14 **return** $S_1 \cup S_2$;

the fine-tuned BERT component. BERT analyzes the feedback to identify a violated constraint (Algorithm 3, line 12). The constraint c inferred from this feedback is only added to the network L if the user confirms that it belongs to the target network T (Algorithm 3, line 13). If the user does not confirm, LLMACQ defaults to running OneConstraint, which is a modified version of OneNogood (Algorithm 3, line 15). During the execution of OneConstraint, whenever a partial example is classified as negative by the user, OneConstraint calls BERT to infer a constraint from the user's feedback (Algorithm 4, line 4). If BERT succeeds in identifying a new constraint that is validated by the user in line 5, this constraint is added to L (line 6) and OneConstraint returns "exit" (lines 6, 11, and 13). Otherwise, OneConstraint continues running until it returns a nogood.

We applied another change to LLMACQ compared to ACQNOGOODS to avoid being trapped in long series of positive queries when target networks have a large number of solutions. ACQNOGOODS needs to find all the solutions of T to prove convergence (Algorithm 1, lines 3-5). However, the target network may have been found far before all solutions have been generated in a query. We thus force LLMACQ to stop when it reaches a long sequence of complete positive examples. Such a sequence probably means that all the constraints of the target network have already been learned. LLMACQ uses a counter $\#pos$ to track the number of consecutive complete positive examples (Algorithm 3, lines 2 and 9). $\#pos$ is reset to zero at each negative answer (line 11). LLMACQ stops as soon as $\#pos$ reaches a given cutoff maxPos (line 3). This new termination condition for LLMACQ outputs a network L that has not been proved to have the same solutions as the target network T (line 17). This state is called *pseudo-convergence* in [Addi *et al.*, 2018].

Theorem 2 (Correctness)**.** *Given a target network T and a sufficiently large cutoff maxPos, LLMACQ terminates and converges on a network L such that $sol(L) = sol(T)$.*

Proof. Assume maxPos is equal to $|D^X|$, which is obviously greater than $|sol(T)|$. Hence, the condition $\#pos <$ maxPos in line 3 of Algorithm 3 is always satisfied and has no effect on the correctness of LLMACQ. If BERT always fails to extract a constraint from the user's feedback, Algorithms 3 and 4 behave respectively like Algorithms 1 and 2. If we learn a new constraint thanks to the user's feedback, this constraint is added to L only if the user confirms its membership to T (Algorithm 3, line 13 and Algorithm 4, line 5). In addition, this constraint cannot be redundant with regard to the current constraints in L because it was inferred from a negative example that satisfies all the constraints learned so far (Algorithm 3, line 4). As a result, learning a constraint thanks to BERT is equivalent to learning multiple nogoods (all the tuples forbidden by c) in one shot. This can only reduce the number of queries needed to learn the target network. □

6 Experiments

In this section, we first describe how our BERT component is fine-tuned. We then introduce the benchmark problems used to evaluate LLMACQ and to compare it to QUACQ2. Finally, we report the results of acquiring problems with LLMACQ and QUACQ2 and we analyze these results.

Fine-tuning BERT was done on Google Colab GPU-accelerated environment. QUACQ2 was parameterized with exactly the same setting as in [Bessiere *et al.*, 2023a], notably a time out of 1 hour. In LLMACQ, the solver used in line 4 is Google OR-Tools CP-SAT. The value of the cutoff maxPos was set to 10. The source code used to obtained the results reported in this paper is available at https://github.com/mechqrane/LLmAcq.

6.1 Fine-Tuning and Evaluation of BERT

For our experiments we trained BERT on a vocabulary $X = \{x_1, \ldots, x_{100}\}$, $D = \{1, \ldots, 10\}$, and a language $\Gamma = \{$Alldifferent, $\sum_{=}^{val}, \sum_{\neq}^{val}, \neq, =, <, >, \leq, \geq, \|_{val}, \nparallel_{val}, \neq_{val}, =_{val}\}$, where val is a numeric value in $\{1, 2, \ldots, 100\}$, $\sum_{=}^{val}$ is $\sum_{j=1}^{k \leq 10} x_{i_j} = val$, $\sum_{\neq}^{val}$ is $\sum_{j=1}^{k \leq 10} x_{i_j} \neq val$, $\|_{val}$, $\nparallel_{val}$, $\neq_{val}$ and $=_{val}$ are $|x_i - x_j| =$

{"text": "we need distinct values within X92 X10 X28 and X45", "labels": "[0, 0, 0, 0, 0, 1, 2, 3, 0, 4]", "task_name": "token_classification"}
{"text": "we need distinct values within X92 X10 X28 and X45", "labels": "[0]", "task_name": "sequence_classification"}
{"text": "82 is required to match the total of X22 X20 and X39", "labels": "[10, 0, 0, 0, 0, 0, 0, 1, 2, 0, 3]", "task_name": "token_classification"}
{"text": "82 is required to match the total of X22 X20 and X39", "labels": "[1]", "task_name": "sequence_classification"}
{"text": "67 should be the length spanning between X10 and X9", "labels": "[3, 0, 0, 0, 0, 0, 0, 1, 0, 2]", "task_name": "token_classification"}
{"text": "67 should be the length spanning between X10 and X9", "labels": "[9]", "task_name": "sequence_classification"}

Figure 2: A sample of data generated for fine-tuning BERT

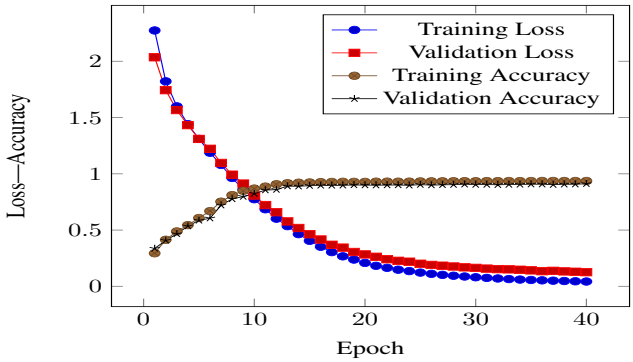


Figure 3: Training and Validation Losses and accuracies

val, $|x_i - x_j| \neq val$, $x_i \neq val$ and $x_i = val$, respectively. It is worth noticing that the bias that corresponds to this Γ and that is required by QUACQ-based algorithms, would have a size growing tremendously fast with $|X|$.

We used CHATGPT-4 to generate templates for the relations in Γ. A template is a sentence without its parameters (i.e., variables and numerical constants). For instance, "$param_1$ *differs from* $param_2$" is a template for the relation $\neq$. The use of templates is crucial for enabling sentence annotation via a script. Manual annotation would be highly time-consuming because the process of annotation entails assigning a class to each word within the sentence. For each relation, we asked CHATGPT-4 to generate 100 distinct templates to ensure a variety of formulations. Based on these templates, we produced 200 sentences for each relation in Γ. To produce a sentence for a relation, we randomly select one of the templates of that relation. We fill that template with randomly generated parameters. These generated parameters are used to fill in the corresponding placeholders in the selected template. The maximum number of variables allowed in a single relation is restricted to 10. We allocated 80% of this generated data for training purposes and the remaining 20% for validation. Figure 2 provides a sample of the annotated generated data. Observe that each sentence is used both for relation identification and for parameters identification tasks. In relation identification, the sentence is labeled with a vector containing a single value to denote the specific relation the sentence implies. In parameters identification, the sentence is labeled with a vector pinpointing the parameters. An element of the vector with value $k \neq 0$ indicates that the word at that position in the sentence represents the k^{th} parameter of the constraint. Elements equal to zero represent words that are not parameters.

We used the PyTorch implementation of BERT BASE and AdamW optimizer [Loshchilov and Hutter, 2019] with a learning rate of 10^{-5} and a batch size of 128. The maximum number of epochs was set to 40. All the sentences were tokenized with the default tokenizer of BERT. The learning curves of the combined losses and accuracy rates are shown in Figure 3. The BERT model achieves an accuracy of 91% on the validation data.

To be able to mimic as much as possible an interaction with a human user, we had to use LLMACQ with data unseen by the fine-tuned BERT during its training phase. We asked CHATGPT-4 to generate 20 additional templates per relation that are thus distinct from those used for training BERT. These are the templates used to generate user's feedback.

6.2 Benchmark Problems

We used the following benchmark problems to evaluate the performance of LLMACQ.

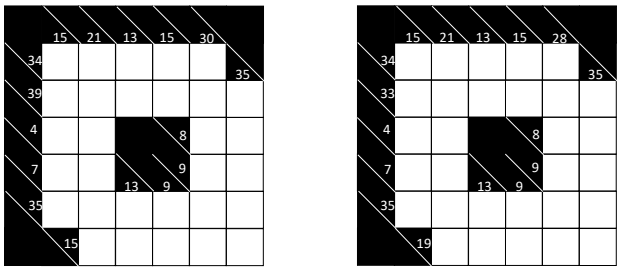
Purdey. Four families stopped by Purdey's general store, each to buy a different item. They all paid with different means. The problem is to match each family with the item they bought and how they paid for it. The target network contains three `Alldifferent` constraints and some additional binary constraints given in the description of the puzzle.

Zebra. The target network of the well-known Lewis Carroll's zebra problem contains five `Alldifferent` constraints and 14 unary and binary constraints given in the description of the problem.

Sudoku. The Sudoku puzzle involves filling cells with numbers from 1 to 9 in a 9×9 grid so that each row, column, and square does not contain twice the same number. The target network contains 27 `Alldifferent` constraints on rows, columns, and squares.

Kakuro. The Kakuro puzzle is played on a grid with white and black cells (see Figure 4). The goal is to fill the blank cells with numbers from 1 to 9 so that the sum of the numbers in each horizontal (resp. vertical) block matches the clue given in the black cell at the left (resp. top) of the block, and no number is repeated within a block. The target network contains sixteen `Alldifferent` constraints and sixteen $\sum_{=}^{val}$ constraints, where val corresponds to the clue of the given block. We used two instances of Kakuro with the same grid and different clues. Kakuro1 has a single solution. Kakuro93 has 93 solutions.

Regarding the evaluation of QUACQ2, the presence of global constraints in the target networks would require the addition of these global constraints to the language Γ. However, global constraints lead to a bias of exponential size, which significantly hurts the performance of QUACQ2. For the experiments with QUACQ2, we replaced all occurrences of `Alldifferent` constraints by cliques of disequalities in all target networks. Hence, Purdey, Zebra, and Sudoku can be learned by QUACQ2 with exactly the same language and bias as in [Bessiere *et al.*, 2023a], the bias containing respec-



(a) Kakuro1 (b) Kakuro93

Figure 4: Two Kakuro puzzles with same shape but different clues

tively 396, 2,700, and 19,440 unary and binary arithmetic constraints. For Kakuro it is more tricky because the target network also contains $\sum_{=}^{val}$ global constraints, which, as opposed to `Alldifferent`, cannot be decomposed. To be able to capture the Kakuro problem, we add all the possible occurrences of $\sum_{=}^{val}$ with the required arity and value of val to Γ. The two instances in Figure 4 contain blocks of size 2 with possible sums $4, 7, 8, 9, 13, 15$, blocks of size 5 with possible sums $15, 19, 34, 35$, and blocks of size 6 with possible sums $21, 28, 30, 33, 35, 39$. It gives us 16 possible $\sum_{=}^{val}$ constraints to add to Γ, leading to a bias containing $6\binom{30}{2} + 4\binom{30}{5} + 6\binom{30}{6} = 4,135,284$ $\sum_{=}^{val}$ constraints.

6.3 Results

All the results reported in Table 1 are the averages of ten runs. We report the total number `#E-Ask` of extended queries asked by LLMACQ or `OneConstraint`, the number `#V-Ask.M` of validation queries asked within the main loop of LLMACQ, the number #no(`V-Ask.M`) of such queries that received a negative answer, and the number `#V-Ask.OC` of validation queries asked by `OneConstraint`. `#Nogoods` counts the number of nogoods learned by LLMACQ. The total `#TotQ` of all kind of queries is thus equal to `#E-Ask` + `#V-Ask.M` + `#V-Ask.OC`. The columns "Learned?" and "Converged?" respectively tell us whether the network L returned by LLMACQ has the same solutions as the target network and whether LLMACQ has proved convergence. For QUACQ2, we report the number $\#QA$ of queries required to learn the target network and the number $\#QC$ required to converge.

Let us first concentrate on the internal behavior of LLMACQ. We first observe that the number of extended queries `#E-Ask` is higher than the number of validation queries `#V-Ask.M` + `#V-Ask.OC` because `#E-Ask` contains positive answers, which do not lead to validation queries. We also observe that the number #no(`V-Ask.M`) of validation queries asked in the main loop that received a negative answer is low but not null. This indicates that BERT occasionally struggles to understand what the user had in mind. However, these occasional failures are generally compensated by the call to `OneConstraint` before being forced to learn a nogood. During the execution of `OneConstraint`, the user's answer may be rephrased, focused on a subset of variables, helping BERT to identify the constraint. The number `#Nogoods` of times `OneConstraint` eventually learns a nogood because it failed to figure out the constraint the user had in mind is less than 1 on all the problems. In most cases these learned nogoods are subsumed by a constraint learned later. But we found a case in which a nogood is not subsumed by any constraint. On the first two cells of the third row of Kakuro, we observed that when the nogood $(2, 2)$ and the constraint $\sum_{=}^{4}$ are learned first, then the `Alldifferent` constraint becomes redundant and is not learned. The nogood $(2, 2)$ is thus non-redundant. If the `Alldifferent` constraint is learned before the nogood, then the nogood is not learned.

Regarding convergence, we observe that LLMACQ fails to prove convergence on Sudoku and Kakuro93. The lack of convergence on these problems is due to the pseudo-convergence mechanism that forces LLMACQ to stop after maxPos =10 consecutive complete examples classified positively (Algorithm 3, line 9). This mechanism prevents LLMACQ from entering an exhaustive enumeration of all the solutions of the problem. For Kakuro93, which has 93 solutions, removing the cutoff (or equivalently setting it to $+\infty$) allows LLMACQ to converge in 166.7 queries. The additional queries compared to the 83.7 in Table 1 are positive extended queries due to the enumeration of solutions. For Sudoku, convergence would simply be impossible as Sudoku has more than $6 \cdot 10^{21}$ solutions.[1] Regardless of the convergence condition, column "Learned?" in Table 1 shows us that for all problems, LLMACQ is able to learn a network that has the same solutions as the target network. For Purdey, Zebra, and Kakuro1, it is guaranteed by convergence. For Kakuro93, checking equivalence between the target network and the learned network was trivial because of the small number of solutions. For Sudoku, LLMACQ consistently learned 21 `Alldifferent` constraints in each of the ten runs. Following the results in [Demoen and de la Banda, 2014], we were able to check that on each run, the 6 missing constraints were redundant. Hence, the learned network is equivalent to the target network.

Let us now compare the performance of LLMACQ and QUACQ2. In Table 1 we see a huge difference between the total number of queries `#TotQ` asked by LLMACQ compared to the number of queries $\#QC$ asked by QUACQ2. On all problems except Purdey, LLMACQ requires orders of magnitude less queries than QUACQ2. Relaxing the convergence condition does not help QUACQ2 to improve its performance. We see that for QUACQ2, learning the target network ($\#QA$) is almost as hard as proving convergence ($\#QC$). QUACQ2 is not able to learn Kakuro1 and Kakuro93 before reaching the 1-hour time limit whereas LLMACQ learns Kakuro1 and Kakuro93 in less than 90 seconds (with less than 90 queries). The failure of QUACQ2 on these two instances is due to the huge size of the bias handled by QUACQ2. LLMACQ does not need the construction of the bias. It only requires a language on which BERT has been trained.

6.4 Discussion

Our experiments confirmed the assumption that when the answers from the user contain more information than the simple "yes/no" of most active constraint acquisition systems,

[1] https://en.wikipedia.org/wiki/Mathematics_of_Sudoku

	LLMACQ								QUACQ2	
	#E-Ask	#V-Ask.M	#no(V-Ask.M)	#V-Ask.OC	#Nogoods	#TotQ	Converged?	Learned?	#QA	#QC
Purdey	41.1	13.7	4.4	8.0	0.7	62.8	Yes	Yes	173.2	179.6
Zebra	32.8	20.0	3.3	4.2	0	57.0	Yes	Yes	565.9	572.1
Sudoku	68.9	21.4	7.5	11.1	0.4	101.4	No	Yes	6945.5	6948.0
Kakuro1	52.2	24.0	5.1	10.1	0.4	86.3	Yes	Yes	time out	time out
Kakuro93	53.1	24.0	3.7	6.6	0.3	83.7	No	Yes	time out	time out

Table 1: LLMACQ vs QUACQ2 in terms of numbers of queries and convergence capabilities. Time out = 1 hour.

the number of queries required to learn the target network decreases significantly. We can object that despite the BERT natural language interface between the user and the learner, the user may not always be able to answer queries as informative as what we have used in our experiments. Interestingly, LLMACQ has the ability to adapt to the user's skills. We have already seen that if the user is not able to articulate any meaningful feedback for a negative example, LLMACQ simply learns a nogood. There can be less extreme cases in which the user can articulate a feedback, but not as informative as we would expect. Take for instance any problem containing `Alldifferent` constraints. `Alldifferent` is one of the few global constraints that has both the contractibility property (i.e., if `Alldifferent` holds on a set S of variables, then `Alldifferent` also holds on every subset of S [Maher, 2009]) and the decomposability property (i.e., any decomposition into a set of smaller `Alldifferent` constraints is semantically equivalent to the original `Alldifferent` as long as every pair of variables is involved in at least one of these smaller `Alldifferent` constraints [Bessiere and Van Hentenryck, 2003]). Because of these two properties, when a negative example violates an `Alldifferent` constraint, the user may be tempted to pinpoint an `Alldifferent` constraint on a subset of the variables of the `Alldifferent` that is in the target network.

Being uniquely composed of `Alldifferent` constraints, Sudoku is the perfect benchmark to illustrate the impact of the quality of the feedback provided by the user on the acquisition process. We built the problem Sudoku-binary by replacing the 27 `Alldifferent` constraints of the target network of Sudoku by the equivalent 810 disequalities. The results of LLMACQ on Sudoku-binary will illustrate the case of a user not familiar at all with constraint programming, who, when asked to classify an example in which two cells cell_i and cell_j in the same row are equal, always responds that the example is negative because cell_i and cell_j have to be different.

LLMACQ asks 1649.5 queries in average of the ten runs to reach pseudo-convergence on Sudoku-binary. This is an order of magnitude more than the 101.4 queries of LLMACQ on Sudoku (see Table 1). The reason is simple: Each learned constraint requires at least one extended query and one validation query, and the target network of Sudoku-binary contains 810 constraints whereas the target network of Sudoku has only 27. Over the ten runs, the network learned by LLMACQ when pseudo-convergence is reached contains between 684 and 690 constraints. We do not know whether these networks are equivalent to the target network of Sudoku-binary –checking it would be extremely time-consuming– but their number of constraints is consistent with the number of non-redundant constraints conjectured in [Demoen and de la Banda, 2014]. A last information on the Sudoku-binary experiment is that LLMACQ has learned zero nogoods. This is probably due to the fact that the scope is so small that the BERT component can more easily understand what constraint the user has in mind.

When compared to the Sudoku experiment, the experiment on Sudoku-binary confirms that the more informative the feedback from the user (e.g., large arity `Alldifferent` constraints), the better the performance of LLMACQ. By being able to handle both kinds of feedback, LLMACQ is able to leverage the user's skills.

7 Conclusion

We have introduced ACQNOGOODS, an active constraint acquisition schema. By learning nogoods instead of constraints, ACQNOGOODS eliminates the need to construct the bias. But the main objective of this paper was to apply natural language processing technology within the realm of constraint acquisition. We proposed LLMACQ, a system based on ACQNOGOODS that incorporates a large language model component. LLMACQ is thus able to interpret natural language feedback from the user. This more informative communication with the user together with the absence of bias in extension allows LLMACQ to learn constraints of any arity and to dramatically decrease the number of queries required to learn the target network. This novel active learning approach shows the potential of LLMs in constraint acquisition and suggests greater improvements by a more advanced use of LLMs.

Acknowledgments

This work was partially supported by the TAILOR project, funded by EU Horizon 2020 research and innovation programme under GA No. 952215, and by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under GA No. ANR-19-PI3A-0004.

References

[Addi *et al.*, 2018] Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. Time-bounded query generator for constraint acquisition. In *Proceedings of*

the 15th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018), volume 10848 of *Lecture Notes in Computer Science*, pages 1–17, Delft, The Netherlands, 2018. Springer.

[Arcangioli *et al.*, 2016] Robin Arcangioli, Christian Bessiere, and Nadjib Lazaar. Multiple constraint acquisition. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 698–704, New York, NY, USA, 2016. IJCAI/AAAI Press.

[Beldiceanu and Simonis, 2012] 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)*, volume 7514 of *Lecture Notes in Computer Science*, pages 141–157, Québec City, QC, Canada, 2012. Springer.

[Bessiere and Van Hentenryck, 2003] Christian Bessiere and Pascal Van Hentenryck. To be or not to be ... a global constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794, Kinsale, Ireland, 2003. Springer.

[Bessiere *et al.*, 2004] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 123–137, Toronto, Canada, 2004. Springer.

[Bessiere *et al.*, 2005] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of the 16th European Conference on Machine Learning (ECML 2005)*, volume 3720 of *Lecture Notes in Computer Science*, pages 23–34, Porto, Portugal, 2005. Springer.

[Bessiere *et al.*, 2007] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pages 50–55, Hyderabad, India, 2007.

[Bessiere *et al.*, 2013] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013*, pages 475–481, Beijing, China, 2013. IJCAI/AAAI.

[Bessiere *et al.*, 2017] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artif. Intell.*, 244:315–342, 2017.

[Bessiere *et al.*, 2023a] Christian Bessiere, Clément Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C. Tsouros, and Toby Walsh. Learning constraints through partial queries. *Artif. Intell.*, 319, 2023.

[Bessiere *et al.*, 2023b] Christian Bessiere, Clément Carbonnel, and Areski Himeur. Learning constraint networks over unknown constraint languages. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023*, pages 1876–1883, Macao, SAR, China, 2023. ijcai.org.

[Daoudi *et al.*, 2016] Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El-Houssine Bouyakhf. Constraint acquisition with recommendation queries. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 720–726, New York, NY, USA, 2016. IJCAI/AAAI Press.

[Demoen and de la Banda, 2014] Bart Demoen and Maria Garcia de la Banda. Redundant sudoku rules. *Theory Pract. Log. Program.*, 14(3):363–377, 2014.

[Devlin *et al.*, 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*, pages 4171–4186, Minneapolis, MN, USA, 2019. Association for Computational Linguistics.

[Freuder and Wallace, 1998] Eugene C. Freuder and Richard J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, volume 1520 of *Lecture Notes in Computer Science*, pages 192–204, Pisa, Italy, 1998. Springer.

[Kiziltan *et al.*, 2016] Zeynep Kiziltan, Marco Lippi, and Paolo Torroni. Constraint detection in natural language problem descriptions. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 744–750, New York, NY, USA, 2016. IJCAI/AAAI Press.

[Kumar *et al.*, 2022] Mohit Kumar, Samuel Kolb, and Tias Guns. Learning constraint programming models from data using generate-and-aggregate. In *28th International Conference on Principles and Practice of Constraint Programming, CP 2022*, volume 235 of *LIPIcs*, pages 29:1–29:16, Haifa, Israel, 2022. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Loshchilov and Hutter, 2019] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[Maher, 2009] Michael J. Maher. Open contractible global constraints. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 578–583, Pasadena, California, 2009.

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

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

[Tsouros *et al.*, 2018] 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*, volume 11008 of *Lecture Notes in Computer Science*, pages 373–388, Lille, France, 2018. Springer.

[Tsouros *et al.*, 2019] Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *Proceedings 25th International Conference on Principles and Practice of Constraint Programming, CP 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 709–725, Stamford, CT, USA, 2019. Springer.

[Tsouros *et al.*, 2023] Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. Guided bottom-up interactive constraint acquisition. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming, CP 2023*, volume 280 of *LIPIcs*, pages 36:1–36:20, Toronto, Canada, 2023. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.