Active Constraint Acquisition Using Large Language Models

YOUNES MECHQRANE*, Ai movement, University Mohammed VI Polytechnic, Morocco
CHRISTIAN BESSIERE, CNRS, University of Montpellier, France

Background: In Constraint Programming, constraint acquisition is the subfield that addresses the issue of modelling problems as sets of constraints, so that they can be solved by a constraint solver. This is a way to dramatically extend the use of Constraint Programming technology. 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.
Objectives: 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 (LLM) component in the AcqNogoods schema. LlmAcq interprets the user's answers given in natural language, leading to more informative communication.
Methods: LlmAcq was instantiated with (i) a fine-tuned LLM encoder and (ii) a prompt-engineered LLM decoder. We evaluated both variants on classical logic-puzzle benchmarks (Purdey, Zebra, Sudoku, Kakuro). For greater realism, we also collected written feedback from 12 human subjects and used these data to design a pseudo-real experiment.
Results: Across all benchmarks, LlmAcq dramatically decreases the number of queries, while learning constraints of arbitrary arity without the need of an explicit bias. The version of LlmAcq using a decoder LLM shows better accuracy and an interesting abstraction capability that allows it to learn several constraints from a single user's answer.
Conclusions: Our results suggest that combining natural-language feedback with bias-free learning is a promising step toward more user-friendly Constraint Programming modeling.

JAIR Track: Constraint Programming and Machine Learning

JAIR Associate Editor: Ferdinando Fioretto

JAIR Reference Format:
Younes Mechqrane and Christian Bessiere. 2026. Active Constraint Acquisition Using Large Language Models . *Journal of Artificial Intelligence Research* 85, Article 11 (February 2026), 28 pages. DOI: 10.1613/jair.1.19277

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, Coletta, Freuder, et al. 2004; Bessiere, Coletta, Koriche, 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

*Corresponding Author.

Authors' Contact Information: Younes Mechqrane, younes.mechqrane@um6p.ma, Ai movement, University Mohammed VI Polytechnic, Rabat, Morocco; Christian Bessiere, bessiere@lirmm.fr, CNRS, University of Montpellier, Montpellier, France.

DOI: 10.1613/jair.1.19277

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, Coletta, O'Sullivan, et al. 2007; Bessiere, Koriche, 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, Coletta, Hebrard, 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, Stergiou, and Sarigiannidis 2018), MQuAcq2 (Tsouros, Stergiou, and Bessiere 2019), Predict&Ask (Daoudi et al. 2016), QuAcq2 (Bessiere, Carbonnel, Dries, et al. 2023)).

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 amount of 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, Carbonnel, and Himeur 2023), 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, Berden, 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 feedback 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 it. We then propose LlmAcq, a constraint acquisition system based on AcqNogoods, which embeds natural language understanding capabilities through the use of an LLM. The goal is to showcase how natural language processing techniques can be used in active constraint acquisition algorithms. We show that by fine-tuning an encoder LLM (Bert (Devlin et al. 2019)) or by adequately prompting a decoder LLM (Mistral (Mistral AI 2023, 2024)) with a predefined language of relations, we provide LlmAcq with capabilities for extracting constraints from the user's feedback. A first experiment demonstrates that LlmAcq using a decoder LLM has a better understanding of user's feedbacks than LlmAcq

using an encoder LLM, thus requiring fewer queries to learn the target network. Both of these variants of LLMACQ significantly outperform QUACQ2, requiring far fewer queries while successfully learning constraints of any arity. We then describe a test performed with students to understand the kind of feedback sentences a human can provide during an acquisition process. The Sudoku logic puzzle was used for this test. The results of this test allowed us to design an experiment as close as possible to what would happen in real conditions. The results we report are very promising. They show the great abstraction capabilities of decoder LLMs when facing human answers, allowing them to learn multiple constraints from a single user's answer. Problems can then be learned with very few queries.

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. More recently, an end-to-end approach was proposed in (Michailidis et al. 2024). The authors propose to use an LLM to map a natural language problem description directly to an executable constraint program. They use intermediate representations and retrieval-augmented in-context learning to produce runnable models in a Python-based CP framework. To our knowledge, our work is the first one that uses natural language understanding techniques 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. Section 4 presents the LLMACQ algorithm, which uses an LLM component to interpret user's answers to queries. In Section 5, we describe two possible implementations of the LLM component, one that uses an encoder and has to be fine-tuned, whereas the other uses a decoder, for which special prompts have to be designed. Section 6 presents the experimental results and 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* $\mathsf{Ask}(e)$, 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. However, as opposed to QUACQ-based algorithms, ACQNOGOODS does not require any predefined relation language or constraint bias. ACQNOGOODS learns nogoods on the fly, deriving each nogood directly from a negative example.

Algorithm 1: AcqNogoods

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

```
1  L ← ∅ ;
2  solutions ← ∅ ;
3  while true do
4      e ← generate e ∈ sol(L) \ solutions ;
5      if e = ⊥ then return "convergence on L" ;
6      else
7          if Ask(e) = yes then add e to solutions ;
8          else
9              S ← OneNogood(e, ∅, X, false) ;
10             add (S, ¬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 Ask(e_R) = no then return ∅ ;
2  if |Y| = 1 then return Y ;
3  split Y into ⟨Y1, Y2⟩ such that |Y1| = ⌈|Y|/2⌉ ;
4  S1 ← OneNogood(e, R ∪ Y1, Y2, true) ;
5  S2 ← OneNogood(e, R ∪ S1, Y1, (S1 ≠ ∅)) ;
6  return S1 ∪ S2 ;
```

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, Coletta, Hebrard, 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 to `OneNogood` returns a subset of Y that is a subset of the scope of a constraint in T that is violated 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 requires a number of queries logarithmic in the number of variables.

3.2 Correctness of AcqNogoods

Proposition 3.1 (Partial correctness). *Given a target network T, the network L returned by AcqNogoods is such that $sol(L) = sol(T)$.*

Proof. *Completeness.* Assume that $sol(L) \nsubseteq sol(T)$. Let e be a solution of L that is not a 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 a 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 $sol(T) \nsubseteq sol(L)$. 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 3.2 (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. □

Corollary 3.3 (Correctness). *Given a target network T, Algorithm 1 converges on a network L such that $sol(L) = sol(T)$.*

Proof. Direct from propositions 3.1 and 3.2. □

4 Using User's Feedback and LLMs for Constraint Acquisition

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 some *feedback* in the form a sentence in natural language specifying 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 "The sum of the values of each row must be equal to 100".

To capture this scenario, we extend the definition of `Ask`(e) queries:

> An *extended* query `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 "no", the user provides a sentence of feedback describing a reason why e is negative. A reason why e is negative can be the description of one

constraint in $T[var(e)]$ that is violated by e. It can also involve several constraints (e.g., "the sum of the values of each row must be equal to 100").

If the user is able to provide feedback sentences with their "no" answers, we need to process these feedback sentences to be able to use them in the learning process. In the following we will use the function `LlmExtractC`. Given a sentence *Sentence* of feedback from the user, `LlmExtractC`(*Sentence*) calls an LLM to interpret the sentence and to try to extract constraints from this sentence. `LlmExtractC` returns the extracted constraints. How to implement `LlmExtractC` will be the subject of Section 5. The interpretation/extraction step performed by the LLM does not have any correctness guarantee. Thus, the set of constraints returned by `LlmExtractC` is considered as a suggestion of constraints from the target network to be validated by the user. To validate these suggested constraints, we need an additional type of query, similar to what was used in (Beldiceanu and Simonis 2012; Daoudi et al. 2016):

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

Answering validation queries requires a kind of skill different from extended queries. Deciding whether a constraint belongs to the target network can be seen as a non-trivial task when complicated global constraints are involved. However, the experiments will show us that good LLMs make very few mistakes, if any. If the user prefers ease of use to guarantee of correctness, they can simply trust the LLM and drop the validation step (that is, systematically answering "yes" to validation queries).

4.1 LLMACQ: An Algorithm Using User's Feedback

We are now ready to present LLMACQ, a new constraint acquisition algorithm that uses feedback from the user. LLMACQ is described in Algorithms 3, 4, and 5. The general structure of LLMACQ is similar to that of ACQNOGOODS (see Algorithm 4). The first difference is that LLMACQ uses `E-Ask`(e) queries instead of `Ask`(e) queries. When the answer to a `E-Ask`(e) query is not "yes", the feedback provided by the user is given to the LLM component through a call to `LlmExtractC` (Algorithm 4, line 12). `LlmExtractC` returns a set C of constraints inferred from this feedback. The set C is given to the function `ValidateC` (Algorithm 4, line 13). Every constraint c from C that is not already in L and for which the user confirms that c belongs to the target network T is added to the network L by `ValidateC` (Algorithm 3, lines 2–4). If at least one constraint has been added to L, the Boolean *modifiedL* is set to true (Algorithm 3, line 5) and `ValidateC` returns true. If no new constraint is learned, `ValidateC` returns false and LLMACQ defaults to running `FindScope&C`, which is a modified version of `OneNogood` (Algorithm 4, line 14). During the execution of `FindScope&C` (Algorithm 5), whenever a partial example is classified as negative by the user (line 3), `FindScope&C` calls `LlmExtractC` to extract constraints from the user's feedback (line 4). If `ValidateC` succeeds in validating and adding one or more new constraints to L among those extracted from the user feedback, `FindScope&C` returns "exit" (line 5). Otherwise, `FindScope&C` returns $\emptyset$ (line 6) and `FindScope&C` goes back to its previous recursive call and continues running until it learns a constraint or returns a nogood.

We applied another change to LLMACQ compared to ACQNOGOODS. ACQNOGOODS can be trapped in long series of positive queries when target networks have a large number of solutions because 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 once it starts repeatedly generating complete positive examples. Such a sequence of consecutive positive examples 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 answers to a query (Algorithm 4, lines 1 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 2). 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 16). This state is called *premature convergence* in (Addi et al. 2018).

Algorithm 3: `ValidateC`

Input: a set C of constraints, and the learned network L
Output: a Boolean true iff the learned network L has been modified

```
1 modifiedL ← false;
2 foreach c ∈ C do
3     if (c ∉ L) and (V-Ask(c) = yes) then
4         add c to L;
5         modifiedL ← true;
6 return modifiedL;
```

Algorithm 4: LlmAcq

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

```
1  L ← ∅; solutions ← ∅; #pos ← 0;
2  while #pos < maxPos do
3      e ← generate e ∈ sol(L) \ solutions ;
4      if e = ⊥ then return "convergence on L" ;
5      else
6          Answer ← E-Ask(e) ;
7          if Answer = yes then
8              add e to solutions;
9              #pos ← #pos + 1;
10         else
11             #pos ← 0;
12             C ← LlmExtractC(Answer);
13             if ¬ValidateC(C, L) then
14                 S ← FindScope&C(e, ∅, X, false);
15                 if S ≠ "exit" then add (S, ¬e_S) to L ;
16 return "premature convergence on L" ;
```

4.2 Correctness and Analysis

We prove that LlmAcq is correct and we observe that LlmAcq is not redundancy-free.

Theorem 4.1 (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 2 of Algorithm 4 is always satisfied and has no effect on the correctness of LlmAcq. If the LLM always fails to extract a constraint from the user's feedback, Algorithms 4 and 5 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 3). 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

Algorithm 5: Function `FindScope&C`

Input: an example e, two scopes R, Y, a Boolean *ask_query*, and the learned network L
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 ← E-Ask(e_R) ;
3      if Answer ≠ yes then
4          C ← LlmExtractC(Answer);
5          if ValidateC(C, L) then return "exit";
6          else return ∅;
7  if |Y| = 1 then return Y ;
8  split Y into ⟨Y1, Y2⟩ such that |Y1| = ⌈|Y|/2⌉;
9  S1 ← FindScope&C(e, R ∪ Y1, Y2, true);
10 if S1 = "exit" then return "exit" ;
11 S2 ← FindScope&C(e, R ∪ S1, Y1, (S1 ≠ ∅));
12 if S2 = "exit" then return "exit" ;
13 return S1 ∪ S2;
```

learned so far (Algorithm 4, line 3). As a result, learning a constraint thanks to the LLM 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. □

Like QuAcq1 (Bessiere, Coletta, Hebrard, et al. 2013), and unlike QuAcq2 (Bessiere, Carbonnel, Dries, et al. 2023), AcqNogoods (resp. LlmAcq) is subject to redundancy in the queries (resp. extended queries) it asks: the same positive query (resp. extended query) may be asked several times inside `OneNogood` (resp. `FindScope&C`). QuAcq2 solved this issue by checking whether there remain constraints in the bias that could be learned from a query. As AcqNogoods and LlmAcq do not build an explicit bias, applying the technique used in QuAcq2 would simply consist in storing the answers to all positive –extended– queries already asked to avoid asking them again in the future. Our experimental results with LlmAcq will show that `FindScope&C` is so seldom called that applying this heavy fix would save a negligible number of queries. We thus decided not to store extended query answers. LlmAcq is also subject to redundancy in the validation queries it asks: the same negative validation query may be asked several times if the LLM repeatedly fails to understand the feedback and repeatedly returns the same wrong constraint. The fix is again very simple: store the answers to all negative validation queries already asked to avoid asking them again in the future. As in the case of extended queries, we decided not to store validation query answers.

5 Extracting Constraints with an LLM

In this Section we describe how to use an LLM to extract constraints from the feedback provided by the user. We explore two families of LLMs: encoder models and decoder models. Encoder models, such as Bert (Devlin et al. 2019), produce contextual embeddings for token or sentence-level classification tasks. However, they typically require an explicit fine-tuning step to achieve good performance on a new application. Decoder models, such as Mistral (Mistral AI 2023, 2024), can sequentially generate text. They often demonstrate few-shot learning capabilities, where they can adapt to new tasks with few examples and with little fine-tuning or no fine-tuning at all (Tom B. Brown and al. 2020). In Subsection 5.1, we describe how an encoder model can be adapted for

constraint extraction. In Subsection 5.2, we describe how carefully crafted prompts allow a decoder model to extract constraints from sentences without any fine-tuning.

5.1 Using an Encoder LLM for Constraint Extraction

As a state-of-the-art pre-trained LLM, BERT 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 extraction of a constraint from a user's sentence can be divided into 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}", the LLM should understand that the relation ensures that the absolute value of the difference between two parameters $param_1$ and $param_2$ are equal to a third parameter, $param_3$: $|param_1 - param_2| = param_3$. 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. On the example in Figure 1, after tokenization, the input sentence (at the bottom of the figure) becomes "[CLS] Make sure 80 reflects the exact distance between x ##76 and x ##59 [SEP][PAD]...[PAD]".

Once the data has been preprocessed, we use a dual-headed approach to handle both relation and parameters classification. The relation classification head is a linear classification layer, shown in blue in Figure 1. The relation classification head uses the [CLS] token representation. After passing through BERT's layers, it contains aggregated information from the entire input sequence. The parameters classification head is also a linear classification layer, shown in green in Figure 1. It works with the representations of individual tokens. Each token representation, obtained from BERT's final layer, is used for token-level classification. On the example in Figure 1, the tokenized sentence is passed through BERT's layers. The relation classification head, in blue, predicts $||_{val}$ as the most likely relation. Simultaneously, the parameters classification head, in green, focuses on the representations of individual tokens. It 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 x_{76}, x_{59}, 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.

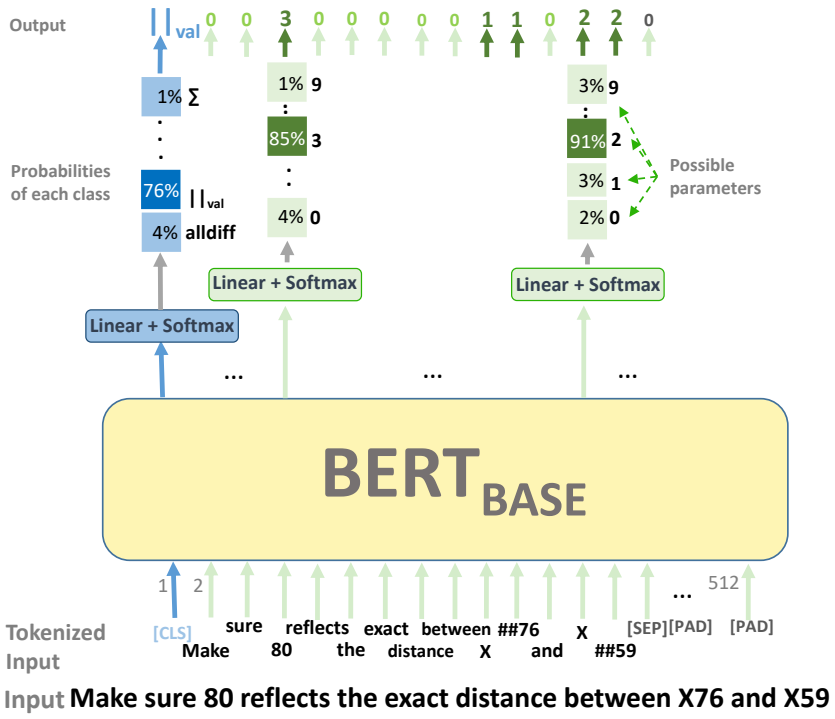


Fig. 1. Illustration of the fine-tuned BERT model used on the sentence "Make sure 80 reflects the exact distance between x_{76} and x_{59}".

In the experiments described in Section 6, we fine-tuned BERT in such a way that it is able to extract the kind of constraints that are used in the logic puzzles that we used. We used a vocabulary $X = \{x_0, \ldots, x_{99}\}$, $D = \{1, \ldots, 10\}$, and a language $\Gamma = \{\texttt{all\_different}, \sum_{=}^{val}, \sum_{\neq}^{val}, \neq, =, \leq, \geq, \|_{val}, \nparallel_{val}, \neq_{val}, =_{val}\}$, 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 $|x_i - x_j| = val$, $|x_i - x_j| \neq val$, $x_i \neq val$ 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 –in English– to ensure a variety of formulations. Based on these templates, we produced 200 sentences for each relation in Γ, where each sentence expresses a single constraint. 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. Figure 2 provides a sample of the annotated generated data. Bear in mind 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

```
 {"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, 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"}
```


Fig. 2. A sample of data generated for fine-tuning Bert

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.

5.2 Using a Decoder LLM for Constraint Extraction

As explained in Section 5.1, encoder LLMs require fine-tuning. In contrast, decoder LLMs excel in generating task-specific outputs just by carefully designing prompts. Our approach to interpreting user's feedback involves two steps. First, a relation identification prompt classifies the user's feedback into one relation. Second, depending on the identified relation, a parameter identification prompt allows the LLM to identify the relevant variables and parameters. As opposed to the encoder setting –where the identification of the relation and its parameters can be done in one step–, decoder LLMs lose accuracy when asked to do both tasks jointly. (In preliminary tests, we observed that single-pass prompts increased cascading errors, a misidentified relation systematically producing wrong parameters.)

In the following, we describe how we designed prompts to guide the decoder LLM in this two-steps constraint extraction (that is, relation identification and parameters identification). The prompts should provide the LLM with role description, task definition, and illustrative examples to help it generate the desired output.

The relation identification prompt (see Figure 3) instructs the LLM in classifying a given sentence into one of the predefined relations in the language Γ. The prompt includes:

- Role description: "You are a highly skilled constraint programming assistant."
- Task definition: "Your task is to classify sentences into types of relations involving variables from the set X."
- The list of possible relations: Enumeration of the relations in Γ.
- Output instructions: "The output must be in JSON format with the attribute: "relation": m", where m is the code corresponding to the relation described in the sentence.

You are a highly skilled constraint programming assistant. Your task is to classify sentences into types of relations involving variables from the set X. The relations are expressed using the following language: {all_different, sum_equals_value, sum_not_equals_value, not_equal, equal, less_than_or_equal_to, greater_than_or_equal_to, absolute_difference_equals_value, absolute_difference_not_equals_value, not_equal_to_value, equal_to_value }.
You have to identify the type of relation in each sentence and output the corresponding code for that relation.
The output must be in JSON format with the attribute: "relation": m.
Mapping of relations to values of m and examples:
- For all_different, m = 0.
Examples:
"The variables x_0, x_1, and x_2 must take different values."
"In x_0, x_1, and x_2, there are two identical numbers."
- For sum_equals_value, m = 1.
Examples:
"The sum of the variables x_0, x_1, and x_2 must equal 10."
"The variables x_3, x_{11}, and x_2 must add up to 10."
- For sum_not_equals_value, m = 2.
Examples:
"The total of the variables x_2, x_1, and x_5 should not be equal 10."
"The variables x_0, x_{12}, and x_3 must not add up to 10."
- For not_equal, m = 3.
Examples:
"The variable x_0 must not equal the variable x_1."
"x_2 and x_3 cannot have the same value."
- For equal, m = 4.
Examples:
"The variable x_0 must equal the variable x_1."
"x_4 is exactly equal to x_5."
- For less_than_or_equal_to, m = 5.
Examples:
"The variable x_0 must be less than or equal to the variable x_1."
"x_6 must be smaller than x_7."
- For greater_than_or_equal_to, m = 6.
Examples:
"The variable x_0 must be greater than or equal to the variable x_1."
"x_8 must be at least as large as x_9."
- For absolute_difference_equals_value, m = 7.
Examples:
"The absolute difference between x_0 and x_1 must be equal to 5."
"The distance between x_0 and x_2 must be 2."
- For absolute_difference_not_equals_value, m = 8.
Examples:
"The absolute difference between x_0 and x_1 must not be equal to 3."
"The gap between x_3 and x_4 cannot be exactly 1."
- For not_equal_to_value, m = 9.
Examples:
"The variable x_0 must not be equal to 7."
"x_1 must not take the value 5."
- For equal_to_value, m = 10.
Examples:
"The variable x_0 must be equal to 4."
"x_2 has to take the value 6."

Fig. 3. Relation Identification Prompt

You are an experienced constraint programming assistant tasked with extracting the parameters of constraints from sentences.
The sentences express constraints of type "sum_equals_value".
The output must be in JSON format with the following structure:
"sum_equals_value constraints": [{"vars" : [$x_1, x2, \ldots, xk$], "param" : p}, ...]
Where each object represents a sum equals constraint, with "vars" being the variables involved in the sum, and "param" being the value the sum should equal.

Here is an example:
Problem description: "The sum of x_{34}, x_{21}, x_{57}, and x_0 must be equal to 100. Also, the sum of x_5 and x_6 must be equal to 50."

Output:

{"sum_equals_value constraints": [{"vars": ["x_{34}", "x_{21}", "x_{57}", "x_0"], "param": 100}, {"vars": ["x_5", "x_6"], "param": 50}]}

Fig. 4. Parameter Extraction Prompt for sum_equals_value constraints

- Mapping of relations to their code and examples: Illustrations of how sentences correspond to specific relations and their associated code.

For parameter extraction, we designed distinct prompts tailored to each type of relation. These prompts include:

- Role description: "You are an experienced constraint programming assistant tasked with extracting parameters of constraints from sentences."
- Task definition: A detailed explanation of the specific type of constraint to be extracted.
- Output instructions: "The output must be in JSON format with the following structure", followed by instructions that depend on the constraint type.
- Examples: An illustrative example of sentence and the corresponding output.

As an example, Figure 4 presents the prompt used to extract variables for the sum_equals_value constraint. Similar prompts have been created for all relations in the language Γ, each tailored to extract the relevant variables and parameters for the corresponding relation.

6 Experiments

This experimental section is composed of several subsections containing several kinds of experiments. Section 6.1 introduces the benchmark problems used to evaluate LLMACQ. In Section 6.2, we present and analyze the results obtained by LLMACQ using either an encoder LLM or decoder LLM. Section 6.3 compares the results obtained by QUACQ2 to those obtained by LLMACQ on the same benchmark problems. In Section 6.4 we show the impact of the kind of user's answers on the efficiency of LLMACQ. Section 6.5 gives the results of a Sudoku test performed on students to better understand how human users formulate their answers. Finally, in Section 6.6 we make a simulation of a human user learning Sudoku with LLMACQ and we analyze the results.

6.1 Experimental Setting and Benchmark Problems

Our implementation of LLMACQ uses Google OR-Tools CP-SAT as the solver in line 3 of Algorithm 4 and maxPos is set to 10. Later in this section, we will see that maxPos = 10 is uselessly too large for the benchmark problems used in this paper. It would be interesting to learn the right value of maxPos that fits the type of problem being learned, but it is beyond the scope of this paper.

In Section 6.2, we used BERT BASE as encoder and Mistral as decoder. BERT BASE (about 7B parameters) was fine-tuned on a Google Colab GPU-accelerated environment, using 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. During training, minibatches are processed separately for each task. 50% of the minibatches focus on relation classification whereas 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. All the sentences were tokenized with the default tokenizer of Bert. We allocated 80% of this generated data for training purposes and the remaining 20% for validation. The Bert model achieves an accuracy of 91% on the validation data. The Mistral decoder was used via an API using the "open-mistral-7b" model (Mistral AI 2023), referred to as small-mistral. This model is one of the smallest variants in the Mistral family but it offers a fair comparison with the Bert Base encoder version due to their similar parameter counts of roughly seven billion. In Section 6.6, we switched to the larger "mistral-large-2407" model (Mistral AI 2024) (referred to as large-mistral) because, in our preliminary tests, we found that small-mistral failed to make effective use of the problem description provided in the prompt.

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

Purdey. (Mason 1997). 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.

Zebra. This is the well-known Lewis Carroll's problem. There are five houses. Each of the five houses is painted a different color, and their inhabitants are of different nationalities, drink different beverages, smoke different brands of cigarettes, and own different pets. The problem is to match each house with its color, nationality, drink, cigarette brand, and pet.

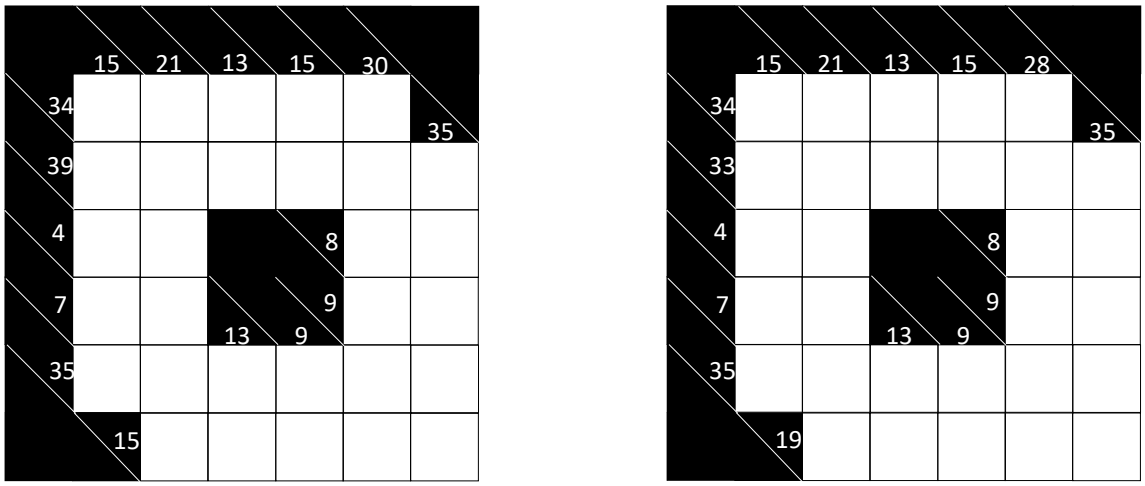
Sudoku. The Sudoku puzzle involves filling cells with numbers from 1 to 9 in a 9×9 grid so that each row, column, and square does not contain twice the same number.

Kakuro. The Kakuro puzzle is played on a grid with white and black cells. 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. We used two instances of Kakuro with the same grid and different clues (see Figure 5). Kakuro1 has a single solution. Kakuro93 has 93 solutions.[1]

For all these benchmark problems, we have to define the corresponding target network that will be used to classify the examples as positive (i.e., no constraint violated), or negative (i.e., at least one constraint violated). We observe that all the problems above contain `all_different` constraints. The `all_different` constraint has the particular property that it is decomposable into binary disequality constraints on the same variables (Bessiere and Van Hentenryck 2003). That is, `all_different`$(x_1, \ldots, x_n)$ is equivalent to the clique of binary constraints $x_i \neq x_j, i, j \in 1..n, i < j$. Any human user is able to recognize this property. Given a –partial– example containing the same value for two variables involved in the same `all_different` constraint, any human user will notice that the `all_different` is violated, even if not all the variables of that `all_different` constraint belong to the example. To avoid the weird case in which an example would be classified as positive because the violated `all_different` constraint is not fully instantiated, for every `all_different` constraint in a target network, we add the –redundant– corresponding clique of binary disequalities to that target network.

Here are the target networks we used to represent our benchmark problems.

[1]Kakuro93 is not a canonical Kakuro as it has several solutions. We included it to have one more instance without a unique solution.



(a) Kakuro1 (b) Kakuro93

Fig. 5. Two Kakuro puzzles with same shape but different clues

Purdey. The target network contains three `all_different` constraints (each with its corresponding clique of disequalities) and some additional binary constraints given in the description of the puzzle.

Zebra. The target network contains five `all_different` constraints (each with its corresponding clique of disequalities) and 14 unary and binary constraints given in the description of the problem.

Sudoku. The target network contains 27 `all_different` constraints on rows, columns, and squares (each with its corresponding clique of disequalities) .

Kakuro. The target network contains sixteen `all_different` constraints (each with its corresponding clique of disequalities) and sixteen $\sum_{=}^{val}$ constraints, where val corresponds to the clue of the given block.

6.2 Encoder Versus Decoder LLMs

In this first experiment, we compare the performance of LLMAcq depending on whether it uses an encoder LLM or a decoder LLM. The encoder and the decoder are used exactly as described in Sections 5.1 and 5.2. When using LLMAcq, we have not only to simulate the "yes/no" answers of the user. We need to generate feedback sentences each time an example (complete or partial) is classified as negative in line 6 of Algorithm 4 or in line 2 of Algorithm 5. Given a negative example, we first compute the set of all the constraints of the target network that are violated by the example. We pick one constraint from this set, prioritizing global constraints over their decompositions, and randomly select one of the templates associated with the relation of that constraint. It is important to notice that for each relation, we asked CHATGPT-4 to generate 20 new templates, distinct from those used during the fine-tuning of BERT, to avoid the risk of biased experiments when using the encoder LLM. (For the decoder LLM, we also use these new templates despite there is no risk of bias as no fine-tuning was performed.) Once a template has been selected, we instantiate it by replacing placeholders (e.g., variable names) with their actual values in the example. For instance, if the example assigns two variables x_3 and x_5 the same value whereas the target network contains the constraint $x_3 \neq x_5$, and if the selected template is "$param_1$ *should*

Table 1. Comparison of LlmAcq using an encoder LLM (Bert) or a decoder LLM (small-mistral) in terms of numbers of queries, convergence, and learning capabilities.

Problem	LLM used	LlmAcq							
		#E-Ask	#V-Ask.M	#no(V-Ask.M)	#V-Ask.FSC	#Nogoods	#TotQ	Converged?	Learned?
Purdey	Bert	41.1	13.7	4.4	8.0	0.7	62.8	Yes	Yes
	small-mistral	18.5	13.2	1.2	1.8	0.2	33.5	Yes	Yes
Zebra	Bert	32.8	20.0	3.3	4.2	0.0	57.0	Yes	Yes
	small-mistral	34.1	20.0	4.0	4.9	0.0	59.0	Yes	Yes
Sudoku	Bert	68.9	21.4	7.5	11.1	0.4	101.4	No	Yes
	small-mistral	32.0	21.0	0.0	0.0	0.0	53.0	No	Yes
Kakuro1	Bert	52.2	24.0	5.1	10.1	0.4	86.3	Yes	Yes
	small-mistral	25.0	24.0	0.0	0.0	0.0	49.0	Yes	Yes
Kakuro93	Bert	53.1	24.0	3.7	6.6	0.3	83.7	No	Yes
	small-mistral	35.8	24.0	0.3	0.4	0.0	60.2	No	Yes

be distinct from $param_2$," we substitute x_3 for $param_1$ and x_5 for $param_2$ to produce the feedback sentence: "x_3 *should be distinct from* x_5." Observe that the sentences built from such templates express a single constraint, exactly as in Section 5.1. Hence, the calls to `LlmExtractC` in line 12 of Algorithm 4 and line 4 of Algorithm 5 return a single constraint.

All the results reported in Table 1 are the averages of ten runs. For the two variants of `LlmExtractC` (encoder with Bert and decoder with small-mistral), we report the total number `#E-Ask` of extended queries asked by LlmAcq or `FindScope&C`, 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.FSC` of validation queries asked by `FindScope&C`. `#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.FSC`. 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.

Let us first concentrate on the internal behavior of LlmAcq. We observe that for both variants of LlmAcq the number of extended queries `#E-Ask` is higher than the number of validation queries `#V-Ask.M` + `#V-Ask.FSC` because `#E-Ask` includes positive answers, which do not lead to validation queries. We also observe that the number #no(`V-Ask.M`) of validation queries in the main loop that received a negative answer is low but not always null for both Bert and small-mistral. This indicates that both LLM variants occasionally struggle to understand the user's feedback. However, these occasional failures are generally compensated by the call to `FindScope&C` before being forced to learn a nogood. During the execution of `FindScope&C`, the user's answer may be rephrased and focused on a subset of variables, helping the LLM in identifying the constraint. The number `#Nogoods` of times `FindScope&C` eventually learns a nogood because it failed to figure out the constraint the user had in mind is less than 1 for both LLM variants across all problems. In most cases, these learned nogoods are subsumed by a constraint learned later. However, we found a case in which a nogood is not subsumed by any constraint. In

the Kakuro1 benchmark, when the nogood $(2, 2)$ and the constraint $\sum_{=}^{4}$ are learned first, the `all_different` constraint becomes redundant and is not learned. Thus, the nogood $(2, 2)$ is non-redundant.

Regarding convergence, we observe that LLMACQ fails to prove convergence on Sudoku and Kakuro93 for both variants. The lack of convergence on these problems is due to the premature convergence mechanism that forces LLMACQ to stop after maxPos =10 consecutive complete examples classified positively (Algorithm 4, 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. For example, BERT variant of LLMACQ converges in 166.7 queries. The additional queries compared to the 83.7 reported in Table 1 are positive extended queries due to the enumeration of solutions. For Sudoku, convergence would be impossible as Sudoku has more than $6 \cdot 10^{21}$ solutions.[2]

Regardless of the convergence condition, the "Learned?" column in Table 1 shows that for all problems, both variants of LLMACQ are able to learn a network that has the same solutions as the target network. For Purdey, Zebra, and Kakuro1, this equivalence is guaranteed by convergence. For Kakuro93, checking equivalence between the target network and the learned network was straightforward due to the small number of solutions. For Sudoku, LLMACQ consistently learned 21 `all_different` constraints in each of the ten runs. Following the results in (Demoen and Banda 2014), we verified that in each run, the 6 missing constraints were redundant. Hence, the learned network is equivalent to the target network.

Let us now compare the differences between LLMACQ with an encoder (BERT) and LLMACQ with a decoder (SMALL-MISTRAL). The results follow a very similar pattern for most of the measures that we display in Table 1. The most noticeable difference in behavior between the BERT variant and the SMALL-MISTRAL variant is visible in the number #no(`V-Ask.M`) of validation queries asked in the main loop of LLMACQ that reject the constraint proposed by the LLM. Except on Zebra, the constraints suggested by BERT are rejected significantly more often than those suggested by SMALL-MISTRAL. As a consequence, `FindScope&C` is called more often in the BERT variant than in the SMALL-MISTRAL variant. These more frequent calls to `FindScope&C` in the BERT variant lead to more extended queries and more validation queries inside `FindScope&C` (see `#V-Ask.FSC` and `#E-Ask`) and more nogoods learned (see `#Nogoods`). This weaker quality of suggestions is the reason for the overall greater number of queries `#TotQ` required by the BERT variant to learn the target network. Given that both LLMs have approximately the same size, this tends to show that a decoder LLM using prompts is more suited than a fine-tuned encoder for the task of interpreting feedback from a user.

6.3 Comparison with QUACQ2

In this section we compare the performance of LLMACQ to QUACQ2, an almost state of the art active constraint acquisition system.[3]

To be able to learn our benchmark problems with QUACQ2, we need to define the language Γ that will be used to build the learning bias. Now, the target network of all our benchmark problems contain the global constraint `all_different`. Adding global constraints to the language Γ would lead to a bias of exponential size. Fortunately, as observed in Section 6.1, every `all_different` constraint is equivalent to its corresponding clique of disequalities, and these cliques belong to the target networks (as described in Section 6.1). Hence, Purdey, Zebra, and Sudoku can be learned by QUACQ2 with exactly the same language and bias as in (Bessiere, Carbonnel, Dries, et al. 2023),[4] the bias containing respectively 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 `all_different`, cannot be decomposed. To make it as easy as possible for QUACQ2 to capture the Kakuro

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

[3] The recent GROWACQ has shown better performance than QUACQ2 but it is of the same order of magnitude (Tsouros, Berden, et al. 2023).

[4] Code available at https://github.com/Dimosts/QuAcq2.

Table 2. Performance of QuAcq2 in terms of number of queries for learning the target network (#QA) or for convergence (#QC). T0 stands for 'Time Out' and was set to 1 hour. In order to help comparison with LlmAcq, we repeat the total number of queries, convergence, and learning information when using the small-mistral decoder LLM (Data from Table 1).

	QuAcq2		LlmAcq		
Problem	#QA	#QC	#TotQ	Converged?	Learned?
Purdey	173.2	179.6	33.5	Yes	Yes
Zebra	565.9	572.1	59.0	Yes	Yes
Sudoku	6945.5	6948.0	53.0	No	Yes
Kakuro1	T0	T0	49.0	Yes	Yes
Kakuro93	T0	T0	60.2	No	Yes

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 5 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.

The results of QuAcq2 are reported in Table 2. As in Table 1, the results are the averages of ten runs. We report the number #QA of queries required to learn the target network and the number #QC required to converge. If we compare to the numbers reported in Table 1, we observe a significant difference between the total number of queries #TotQ asked by LlmAcq (whatever variant) compared to the number of queries #QC asked by QuAcq2. Across all problems except Purdey, QuAcq2 requires orders of magnitude more queries than LlmAcq. Relaxing the convergence condition does not help QuAcq2 to improve its performance. We see that learning the target network (#QA) is almost as hard as proving convergence (#QC). In addition, we observe that QuAcq2 is not able to learn Kakuro1 and Kakuro93 before reaching the 1-hour time limit, whereas LlmAcq learns these problems in a few minutes whatever the LLM variant. The failure of QuAcq2 on these two instances is due to the large size of the bias handled by QuAcq2. In contrast, LlmAcq does not require the construction of the bias.

6.4 The Case of Over-Specific Feedback Sentences

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, the number of queries required to learn the target network decreases significantly. We can object that despite the natural language interface between the user and the learner, the user may not always be able to answer queries as informatively as 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 a problem containing the ubiquitous `all_different` constraint and an example classified negative because that `all_different` constraint is violated. It could happen that the user gives an answer pinpointing a pair of variables assigned the same value in the example instead of pinpointing the whole `all_different` constraint that is violated. As seen in Section 6.1, such an answer would be correct thanks to the decomposability property of the `all_different` constraint.

Sudoku is the perfect benchmark to illustrate the impact of such over-specific feedback sentences on the acquisition process because it is uniquely composed of `all_different` constraints. We did an experiment in which we removed the 27 `all_different` constraints from the target network of Sudoku, keeping only the corresponding cliques of disequalities (i.e., 810 binary constraints). As a consequence, the feedback sentences always pinpoint one pair of variables assigned the same value instead of the 9-ary `all_different` constraint that these two variables violate. LLMACQ using SMALL-MISTRAL asks 1697.7 queries in average of the ten runs to reach premature convergence. This is more than an order of magnitude more than the 53.0 queries required by LLMACQ using SMALL-MISTRAL on Sudoku (see Table 1). The reason is simple: Expressing Sudoku with binary constraints involves 810 disequalities, and each learned constraint requires at least one extended query and one validation query. LLMACQ never actually reaches 810 learned constraints. Over the ten runs, the network returned by LLMACQ at premature convergence contains between 773 and 792 constraints. The fact that LLMACQ did not learn the 810 binary constraints is not a proof that the learned network is not equivalent to the target network because the binary target network of Sudoku contains many redundancies. (In (Demoen and Banda 2014), it was shown that there exist networks with 648 binary disequalities that are equivalent to the target network with 810 constraints.) Nevertheless, having learned more than 648 constraints is not a guarantee that the learned network is equivalent to the target network. For each of the ten networks returned by LLMACQ over the ten runs, we computed the F_1 score. The average F_1 score over the ten runs is 0.96. This means that on some runs, LLMACQ has reached premature convergence on a network that is not equivalent to the target network, though very close to it. This similarity explains why LLMACQ was able to generate 10 (=maxPos) consecutive solutions of the learned network that are also solutions of the target Sudoku network.

A last information on this experiment is that LLMACQ has not learned a single nogood. This is probably due to the fact that the scope of the constraint pointed out in the feedback sentence is so small that the LLM component can more easily understand what constraint the user has in mind.

6.5 Experiment with Human Users

In the previous section, we raised the issue of over-specific users' answers. To avoid speculating on what a user can or cannot respond, we designed an experiment with real human users giving feedback on erroneous Sudoku grids.

We designed an experiment to see what kind of feedback a human user could give in front of negative examples of their problem. Twelve undergraduate students from the master of AI of the University of Paris Cité participated in the test. These students had zero skills in constraint programming. We thought it is important that the students do not have any knowledge in constraint programming to avoid a possible bias in the kind of answers they will give. Still, as master's students, they can be considered experts in their field –here, Sudoku– just like any user applying constraint programming is an expert in their own domain. Before starting the test, the twelve students confirmed that they all know the Sudoku puzzle, so that we did not have to define it before starting the test. (Giving the definition of the problem right before performing the test could have influenced their responses.)

The twelve students were given ten filled Sudoku grids. The ten grids are reported in Figure 6. On each grid, some cells are colored in gray and the number inside the cell is written in bold to emphasize on some type(s) of violation of the rules of Sudoku. (Giving grids without any hint on where to watch would have put the burden on finding the errors more than giving feedback on why it is not a solution of Sudoku.) The ten grids each illustrate a distinct violation scenario:

(1) Two cells on the same row contain the same number;
(2) Two cells on the same column contain the same number;
(3) Two cells in the same 3×3 block contain the same number;
(4) Two cells in the same row and in the same 3×3 block contain the same number;

(5) Two cells in the same column and in the same 3×3 block contain the same number;
(6) A cell contains a number that already appears in the same row and in the same column;
(7) Three cells in the same 3×3 block contain the same value;
(8) Three cells in the same column contain the same value;
(9) Three cells in the same row contain the same value;
(10) Multiple cells across rows, columns, and 3×3 blocks contain the same value.

The only guidelines that were given to the students are the following sentence (in French) and that they have fifteen minutes to answer the ten questions.

> *Chacune des grilles ci-dessous contient des cases grisées qui désignent une ou des raisons pour lesquelles la grille n'est pas un sudoku valide. Pour chaque grille, en se basant sur ces cases grisées, écrivez une phrase expliquant pourquoi cette grille n'est pas un sudoku.*

which ChatGPT translates as:

> *Each of the grids below contains shaded cells that indicate one or more reasons why the grid is not a valid Sudoku. For each grid, based on these shaded cells, write a sentence explaining why this grid is not a valid Sudoku.*

We collected 120 answers (ten from each of the twelve students). Here are a few examples of answers given by the students during the test. (The students answered in French but for the sake of readability, we report the ChatGPT translations of their answers.)

Grid 1: "It's not a sudoku because the digit 9 is present two times on row 4." **[Student 4]**
Grid 1: "This is not a sudoku because a row cannot contain the same number more than once." **[Student 5]**
Grid 2: "Two '9's in the same column." **[Student 6]**
Grid 3: "In the square formed by rows 1, 2, 3 and columns a, b, c, the same digit (1) appears twice." **[Student 3]**
Grid 10: "4 rows have identical numbers (3, 5, 7, 8). 2 columns have identical numbers (d, f). 3 blocks have identical numbers (d, e, f, 1, 2, 3), (d, e, f, 4, 5, 6), and (d, e, f, 7, 8, 9)."**[Student 10]**

The first information that we discovered from the analysis of the answers collected through this test is that human users, even when non-specialist of constraint programming, never give answers as specific as the binary responses described in Section 6.4.

The second important information that we derived from this test is that human users often provide answers that use some implicit background knowledge about the problem (rows, columns, squares). Thanks to this implicit background knowledge, answers may only implicitly refer to variables. For instance, the answer of Student 5 to Grid 1 (see above) indicates an `all_different` relation across variables in row 4 without explicitly naming the variables.

6.6 Adapting LlmAcq to Feedbacks Containing Implicit Information

The experiment with human users presented in Section 6.5 shows us that human users may provide feedbacks that use implicit background knowledge of the problem. Thanks to this implicit background knowledge, a feedback may not explicitly refer to variables. What to do in front of feedbacks that do not explicitly specify the variables in the violated constraint? Regarding encoder models, there is little hope to be able to retrieve the information included in feedbacks that do not explicitly contain the variable names. We saw in Section 5.1 that Bert relies on token classification to extract the variable names inside the user feedback. In the absence of variable names, Bert lacks the capacity to infer which specific variables are involved. Interestingly, in decoder models, it is easy to give any kind of extra information to the LLM through adequate prompting. In this Section we show that even when feedbacks do not explicitly specify the variables involved in a constraint, LlmAcq using a decoder LLM can learn the target network.

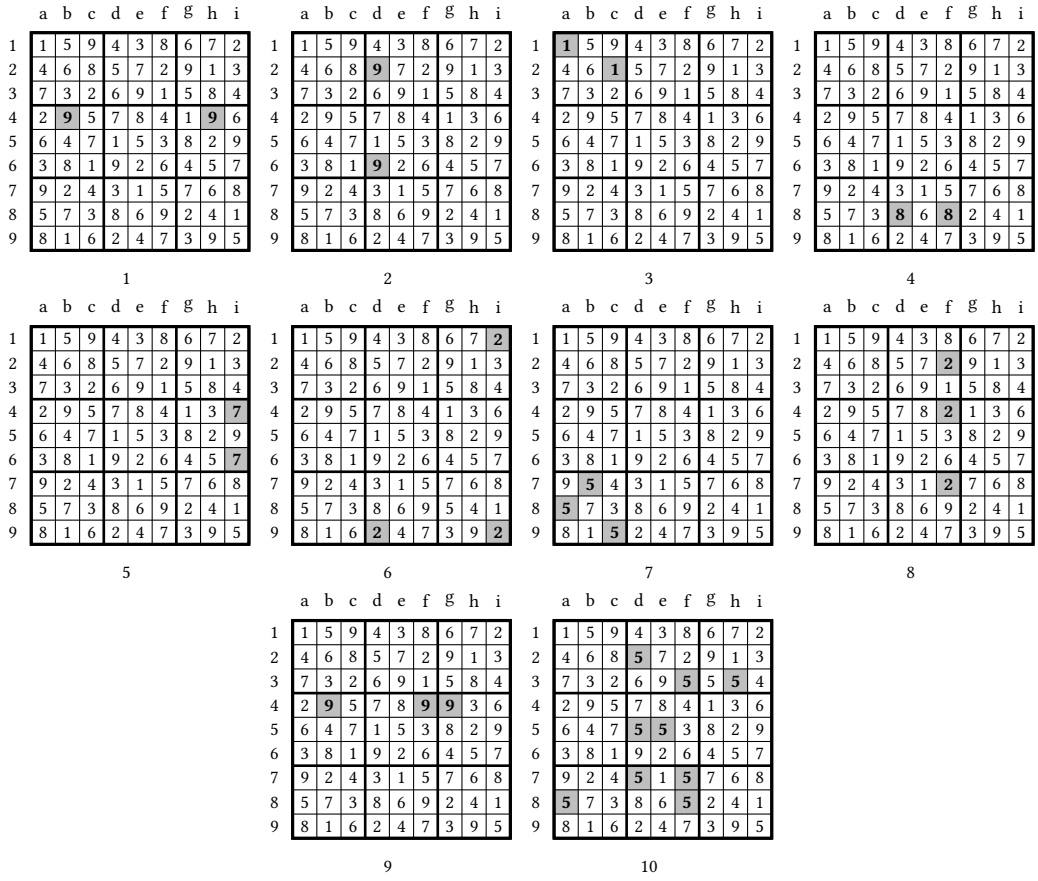


Fig. 6. The ten invalid Sudoku grids used for our test

We designed an experiment in which the data from the student test of Section 6.5 is used to simulate human users learning the Sudoku. For each student, we create a digital user whose answers to LLMACQ will be of the kind used by the corresponding student in the test. The ten feedbacks collected from Student k are converted into reusable parameterized templates by replacing numerical values by parameters. For instance, the answer of Student 4 in front of Grid 1 (i.e., *"It's not a Sudoku because the digit 9 is present two times on row 4."*) is transformed into the template: *"It's not a Sudoku because the digit {value} is present two times on row {row}"*. See Figures 7 and 8 for the complete list of templates of Users 4 and 11.

The ten templates generated from the ten answers of Student k are used to simulate the answers of User k during an execution of LLMACQ to learn the Sudoku. Each time a complete negative example e is presented to the user in line 6 of Algorithm 4 or a partial negative example in line 2 of Algorithm 5, a feedback has to be generated. For each of the ten violation scenarios described in Section 6.5, we collect every occurrence of this scenario in the example. For instance, if the example being processed contains twice value 6 in row 3 and twice value 5 in row 7, we collect two occurrences of scenario 1. And so on for each scenario. Then, among all collected occurrences of scenarios, we randomly pick one. We retrieve the corresponding template from Student k, and we

```
"scenario 1":  "It's not a sudoku because the digit {value} is present two times on row {row}."
"scenario 2":  "It's not a sudoku because the digit {value} is present two times on column
{column}."
"scenario 3":  "It's not a sudoku because the digit {value} is present two times in section {section}."
"scenario 4":  "It's not a sudoku because the digit {value} is present two times on row {row} and
in section {section}."
"scenario 5":  "It's not a sudoku because the digit {value} is present two times on column
{column} and in section {section}."
"scenario 6":  "It's not a sudoku because the digit {value} is present two times on row {row}
and on column {column}."
"scenario 7":  "It's not a sudoku because the digit {value} is present three times in section
{section}."
"scenario 8":  "It's not a sudoku because the digit {value} is present three times on column
{column}."
"scenario 9":  "It's not a sudoku because the digit {value} is present three times on row
{row}."
"scenario 10": "It's not a sudoku because the digit {value} is present more than once
in columns {columns_violations}; in rows {rows_violations}. It's also the case for section {section}."
```

Fig. 7. Templates extracted from User 4 sentences (translated into English by ChatGPT)

```
"scenario 1":  "There are two '{value}' on the same row, which contravenes the constraints
of the game."
"scenario 2":  "There are two '{value}' on the same column, so the grid is not valid."
"scenario 3":  "Two '{value}' in the same square. Replacing the one further down on
the right with another digit seems to solve the grid."
"scenario 4":  "Two '{value}' on the same row. The one on the right should be another digit."
"scenario 5":  "Two '{value}' on the same column, the top one should be another digit."
"scenario 6":  "Two errors: there is a pair of '{value}' on the same column, and one of these
forms another pair with a third {value} on the same row."
"scenario 7":  "Three '{value}' in the same square. This is invalid."
"scenario 8":  "Three '{value}' on the same column, also invalid."
"scenario 9":  "Three '{value}' on the same row. Also invalid."
"scenario 10": "A mass of various errors: {value} lined up in columns, rows, and on the same
pairs in multiple squares."
```

Fig. 8. Templates extracted from User 11 sentences (translated into English by ChatGPT)

replace the placeholders (e.g., value of a digit, row number, etc.) by their value in the example. In the illustrative example above, if the occurrence of scenario that is randomly picked is the one expressing repetition of value 6 in row 3, and if we are simulating Student 4, the feedback returned will be: *"It's not a Sudoku because the digit 6 is present two times on row 3"*.

As most of the sentences given by the students of the test do not contain explicit variable names, we need to provide the LLM with a prompt telling it what these abstract notions, such as row or column, mean. Our conjecture is that if a feedback sentence uses an abstract notion to refer to objects in the problem to acquire, this means that the user who gave this feedback has this notion in their background knowledge. (A similar assumption was made in (Bessiere, Coletta, Daoudi, et al. 2014), where variable types are known by the system,

Table 3. Results of simulation of human users learning Sudoku with LLMACQ: Queries, and learned constraints for each user.

User	#E-Ask	#yes(E-Ask.M)	#V-Ask.M	#no(V-Ask.M)	#V-Ask.FSC	#TotQ	#Nogoods	#C	Learned?
1	20.9	10.0	25.5	1.0	3.0	49.4	0.0	27.0	Yes
2	28.5	10.0	22.9	1.4	1.8	53.2	0.0	23.0	Yes
3	33.4	10.0	23.6	1.6	4.9	61.9	0.3	23.8	Yes
4	24.5	10.1	22.4	0.0	0.5	47.4	0.0	22.9	Yes
5	26.9	10.0	23.1	0.0	0.0	50.0	0.0	23.1	Yes
6	35.7	10.0	14.9	3.0	23.2	73.8	1.0	27.0	Yes
7	26.5	10.0	21.2	1.2	2.6	50.3	0.0	22.5	Yes
8	32.0	10.0	23.2	0.7	1.9	57.1	0.1	24.1	Yes
9	26.3	10.0	15.8	1.9	11.4	53.5	0.1	22.2	Yes
10	25.7	10.0	21.5	1.5	2.4	49.6	0.1	21.8	Yes
11	43.8	10.0	11.8	2.4	19.1	74.7	1.1	26.7	Yes
12	28.4	10.0	16.4	1.7	19.1	63.9	0.8	27.0	Yes
13	*12.0*	*10.0*	*27.0*	*0.0*	*0.0*	*39.0*	*0.0*	*27.0*	Yes

and the user can decide whether a learned constraint should be applied to all scopes of variables of the same type.) In the case of Sudoku and our twelve students, it is clear from their answers that all students knew about rows, columns and squares. As "row"/"column"/"square" were the words the most commonly used in the test to refer to rows/columns/squares, we defined these three words, and only these three words, in the prompt that we give to the LLM. Importantly, we have not designed a specific prompt for each user. We used the same prompt (see Figure 9) for all users. The experiments will show that even for those students who used the words '*section*' or '*block*' instead of '*square*', the LLM is often able to catch the meaning.

Table 3 reports the results for the 12 students as described above, plus a "virtual best" student that we will describe later. For this experiment we used the decoder model LARGE-MISTRAL (Mistral AI 2024). (Preliminary tests with SMALL-MISTRAL were giving bad results.) The LLM was fed with the prompts of Figures 3 and 4 as described in Section 5, to which we added the prompt of Figure 9 that describes rows, columns, and squares. Each number in the table is the average over ten runs.

A first observation is that the lack of explicit specification of the variables involved in the `all_different` constraints does not hurt so much the performance compared to the case where feedbacks were explicitly specifying the variables in the scope of the constraints. LLMACQ with decoder and explicit specification of the variables requires 53.0 queries in average of the ten runs (see #TotQ in the row "Sudoku" in Table 1). LLMACQ with decoder plus a prompt defining rows/columns/squares and *without* explicit specification of variables requires 57.1 queries in average of the twelve users (average of the twelve first entries in column #TotQ in Table 3).

If we look at the details per user, we observe a non-negligible variance of performance in number of queries from one user to the other. The total number of queries #TotQ goes from 47.4 queries in average of the ten runs for User 4 to 74.7 queries for User 11. We also observed a higher variance over the ten runs for those users with high average number of queries than for those with low average number of queries (standard deviation is 1.35 for User 4 whereas it is 35.92 for User 11 –not reported in the Table).

```
We have a grid of 9x9 cells divided into rows, columns, and blocks. Rows are identified by letters
(a, b, c, ..., i), and columns are identified by numbers (1, 2, ..., 9). This creates a coordinate
system where each cell is labeled as `row letter' followed by `column number' (e.g., 1a, 2b).
81 variables are needed for this problem. Variables are labeled to reflect their position
clearly for easier referencing. Each variable is labeled as `x' followed by `row number' and
`column letter' (e.g., x1a).

Here is how these variables are related to the grid:

Variables per Row:
Row 1: [x1a, x1b, x1c, x1d, x1e, x1f, x1g, x1h, x1i]
Row 2: [x2a, x2b, x2c, x2d, x2e, x2f, x2g, x2h, x2i]
Row 3: [x3a, x3b, x3c, x3d, x3e, x3f, x3g, x3h, x3i]
Row 4: [x4a, x4b, x4c, x4d, x4e, x4f, x4g, x4h, x4i]
Row 5: [x5a, x5b, x5c, x5d, x5e, x5f, x5g, x5h, x5i]
Row 6: [x6a, x6b, x6c, x6d, x6e, x6f, x6g, x6h, x6i]
Row 7: [x7a, x7b, x7c, x7d, x7e, x7f, x7g, x7h, x7i]
Row 8: [x8a, x8b, x8c, x8d, x8e, x8f, x8g, x8h, x8i]
Row 9: [x9a, x9b, x9c, x9d, x9e, x9f, x9g, x9h, x9i]

Variables per Column:
Column 1: [x1a, x2a, x3a, x4a, x5a, x6a, x7a, x8a, x9a]
Column 2: [x1b, x2b, x3b, x4b, x5b, x6b, x7b, x8b, x9b]
Column 3: [x1c, x2c, x3c, x4c, x5c, x6c, x7c, x8c, x9c]
Column 4: [x1d, x2d, x3d, x4d, x5d, x6d, x7d, x8d, x9d]
Column 5: [x1e, x2e, x3e, x4e, x5e, x6e, x7e, x8e, x9e]
Column 6: [x1f, x2f, x3f, x4f, x5f, x6f, x7f, x8f, x9f]
Column 7: [x1g, x2g, x3g, x4g, x5g, x6g, x7g, x8g, x9g]
Column 8: [x1h, x2h, x3h, x4h, x5h, x6h, x7h, x8h, x9h]
Column 9: [x1i, x2i, x3i, x4i, x5i, x6i, x7i, x8i, x9i]

Variables per 3x3 Square:
Square 1 (Top-left):     [x1a, x1b, x1c, x2a, x2b, x2c, x3a, x3b, x3c]
Square 2 (Top-middle):   [x1d, x1e, x1f, x2d, x2e, x2f, x3d, x3e, x3f]
Square 3 (Top-right):    [x1g, x1h, x1i, x2g, x2h, x2i, x3g, x3h, x3i]
Square 4 (Middle-left):  [x4a, x4b, x4c, x5a, x5b, x5c, x6a, x6b, x6c]
Square 5 (Center):       [x4d, x4e, x4f, x5d, x5e, x5f, x6d, x6e, x6f]
Square 6 (Middle-right): [x4g, x4h, x4i, x5g, x5h, x5i, x6g, x6h, x6i]
Square 7 (Bottom-left):  [x7a, x7b, x7c, x8a, x8b, x8c, x9a, x9b, x9c]
Square 8 (Bottom-middle):[x7d, x7e, x7f, x8d, x8e, x8f, x9d, x9e, x9f]
Square 9 (Bottom-right): [x7g, x7h, x7i, x8g, x8h, x8i, x9g, x9h, x9i]
```

Fig. 9. Sudoku Grid Description

Why User 4 is the best case? In Table 3 we see that for User 4 (and also for User 5, who is the second best in total number of queries #TotQ), Column #no(V-Ask.M) is equal to zero. All the constraints returned by `LlmExtractC` in line 12 of Algorithm 4 were confirmed as correct. This accuracy can be explained by the clarity of User 4's feedbacks (see Figure 7). The LLM never fails to understand that the relation to learn is `all_different`. In addition, in most cases, the LLM understands the row, column, and/or 3×3 square where the violation occurs. However, we see that the number of calls to `FindScope&C` is not zero (see Column #V-Ask.FSC). The reason is

```
"scenario 1":  "In each row, all values must be distinct."
"scenario 2":  "In each column, all values must be distinct."
"scenario 3":  "In each square, all values must be distinct."
"scenario 4":  "In each row and in each square, all values must be distinct."
"scenario 5":  "In each square and in each column, all values must be distinct."
"scenario 6":  "In each row and in each column, all values must be distinct."
"scenario 7":  "In each square, all values must be distinct."
"scenario 8":  "In each column, all values must be distinct."
"scenario 9":  "In each row, all values must be distinct."
"scenario 10": "In each row, each column, and each square, all values must be distinct."
```

Fig. 10. Templates of the virtual best user (#13)

that, despite the sentences are quite explicit, they use the word "section" to refer to 3×3 squares. The LLM is occasionally fooled by such sentences, and `LlmExtractC` returns constraints that are already in the currently learned network L instead of the one(s) the user was referring to. Function `ValidateC` (Algorithm 3) detects that L has not been modified, triggering a call to `FindScope&C` (line 14 of Algorithm 4). We can make a last important observation on User 4 –but also on most of the other users– by analyzing the number of extended queries necessary to learn the Sudoku network. If we ignore the 10 positive answers (i.e., #yes(`E-Ask.M`)), whose only purpose is to reach maxPos and prove premature convergence, the number of extended queries effectively used for learning (i.e., `#E-Ask` − #*yes*(`E-Ask.M`)) is lower than the number `#C` of constraints learned. The reason is that some of the feedback sentences allow `LlmExtractC` to return several constraints. For instance, when the feedback sentence is built from the template *"It's not a sudoku because the digit {value} is present two times on row {row} and on column {column}."* [scenario 6, User 4], `LlmExtractC` can return an `all_different` on a row and an `all_different` on a column.

At the other extreme of efficiency in number of queries, we find Users 11. A closer look at this user reveals interesting behaviors. On the one hand, we see that together with User 6, it has the highest number of rejection of constraints suggested by the LLM (see Column #no(`V-Ask.M`)). The explanation can be found by observing the feedback sentences. These sentences are frequently ambiguous. See for instance scenario 2 in Figure 8: *"There are two '{value}' on the same column, so the grid is not valid."*. By not specifying which row, column, or 3×3 square contains the violation, the LLM makes guesses based on the grid description provided in the prompt. These guesses can lead to very different behaviors. In some cases, `LlmExtractC` returns an `all_different` constraint that has already been learned. This leads `ValidateC` to return false, then triggering a call to `FindScope&C`. In other cases, `LlmExtractC` returns a correct `all_different` constraint not yet learned. Finally, in some cases, the uncertainty in the feedback sentence allows the LLM to make correct abstractions. For instance, when the sentence is built from the template *"We have the same digit {value} on the same row."* [scenario 1, User 12], `LlmExtractC` returns an `all_different` constraint for each row in one shot.

Regarding the number of nogoods learned, we observe a correlation with the total number `#TotQ` of queries, and also with the number #no(`V-Ask.M`) of rejected validations in the main procedure of LlmAcq. All these three measures are symptoms of low quality of the user's feedback. The clearer and the more precise the feedback, the fewer calls to `FindScope&C` and the fewer nogoods are learned.

As for the number of constraints, unlike the experiments reported in Section 6.2, the number of learned constraints here always exceeds 21. This outcome arises because constraints are not necessarily learned one by one. Several constraints can be acquired in one shot.

Across the 120 runs (10 per user), we observe that the number of positive answers #yes(E-Ask.M) to extended queries in the main procedure of LlmAcq is equal to maxPos = 10 for all users except User 4, for which it is 10.1 (see Column #yes(E-Ask.M) in Table 3). This means that across the 120 runs, there is only one run –from User 4– where one valid solution is generated before learning all non-redundant constraints. This suggests that setting maxPos to 2 would have been sufficient for learning Sudoku for all 12 users, saving 8 queries on each run.

In our analysis of the results in Table 3, we said that when a sentence is not explicit enough, LlmExtractC can either make a wrong suggestion or extract several constraints in one shot. Is there a good level of abstraction to be used in feedback sentences that would outperform all the feedbacks provided by the students in our test? To answer this question, we handcrafted ten templates that take the best from what we observed, that is, abstract enough to allow the LLM to learn several constraints at a time, precise enough to avoid mistakes from the LLM. These ten handcrafted templates are reported in Figure 10. They will be used to generate the feedback sentences from a virtual user that we call User 13. We did the same experiment with this User 13 as we did with the twelve others. The results are reported at the bottom of Table 3. We see that this virtual user outperforms all other users. It requires 39 queries in total to reach premature convergence. A closer look at these 39 queries shows us that among the 12 extended queries (Column #E-Ask), only two are negative. The ten others are the ten consecutive positive answers –#yes(E-Ask.M)– required to reach the cutoff maxPos and thus to return premature convergence. Furthermore, the 27 other queries are all positive validation queries asked in the main LlmAcq procedure (#V-Ask.M) to validate the 27 all_different constraints extracted by the LLM. Not a single one of the constraints suggested by LlmExtractC was rejected, hence no call to FindScope&C. As a consequence, on this last experiment, if we trust enough the LLM not to ask validation queries, the Sudoku model is learned in 2 queries only.

To conclude on this experiment, we wanted to be sure that our results are not biased by the fact that Sudoku is an extremely common logic game that the LLM could have learned somewhere else. We relaunched the same experiment without the prompt in Figure 9. Instead of extracting constraints, the LLM asks what a row, column, or square is, failing to learning any constraint.

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 an LLM component. We described how to build such an LLM component either using an encoder LLM or a decoder LLM. LlmAcq associated with this LLM component is able to interpret natural language feedback from the user. This highly informative communication with the user, together with the release of the need for a bias in extension, allows LlmAcq to dramatically reduce the number of queries required to learn the target network and to learn constraints of any arity. We also made an experiment to collect real feedback from human users, and we used these data to design a pseudo-real experiment. This last experiment shows how much LlmAcq is robust to any kind of user's feedback and has good abstraction capabilities. This work shows the great potential of LLMs in active constraint acquisition.

Acknowledgments

This paper is an extended version of a paper presented at IJCAI 2024 (Mechqrane et al. 2024).
We thank the students of the master of artificial intelligence of University Paris Cité 2024 for having accepted to participate in the Sudoku test used in this paper.

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

H. A. Addi, C. Bessiere, R. Ezzahir, and N. Lazaar. 2018. "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)* (Lecture Notes in Computer Science). Vol. 10848. Springer, Delft, The Netherlands, 1–17.

R. Arcangioli, C. Bessiere, and N. Lazaar. 2016. "Multiple Constraint Acquisition." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*. IJCAI/AAAI Press, New York, NY, USA, 698–704.

N. Beldiceanu and H. Simonis. 2012. "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)* (Lecture Notes in Computer Science). Vol. 7514. Springer, Québec City, QC, Canada, 141–157.

C. Bessiere, C. Carbonnel, A. Dries, et al.. 2023. "Learning constraints through partial queries." *Artif. Intell.*, 319.

C. Bessiere, C. Carbonnel, and A. Himeur. 2023. "Learning Constraint Networks over Unknown Constraint Languages." In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023*. ijcai.org, Macao, SAR, China, 1876–1883.

C. Bessiere, R. Coletta, A. Daoudi, N. Lazaar, Y. Mechqrane, and E. Bouyakhf. 2014. "Boosting Constraint Acquisition via Generalization Queries." In: *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014* (Frontiers in Artificial Intelligence and Applications). Vol. 263. IOS Press, Prague, Czech Republic, 99–104.

C. Bessiere, R. Coletta, E. C. Freuder, and B. O'Sullivan. 2004. "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* (Lecture Notes in Computer Science). Vol. 3258. Springer, Toronto, Canada, 123–137.

C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C. Quimper, and T. Walsh. 2013. "Constraint Acquisition via Partial Queries." In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013*. IJCAI/AAAI, Beijing, China, 475–481.

C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. 2005. "A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems." In: *Proceedings of the 16th European Conference on Machine Learning (ECML 2005)* (Lecture Notes in Computer Science). Vol. 3720. Springer, Porto, Portugal, 23–34.

C. Bessiere, R. Coletta, B. O'Sullivan, and M. Paulin. 2007. "Query-Driven Constraint Acquisition." In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*. Hyderabad, India, 50–55.

C. Bessiere, F. Koriche, N. Lazaar, and B. O'Sullivan. 2017. "Constraint acquisition." *Artif. Intell.*, 244, 315–342.

C. Bessiere and P. Van Hentenryck. 2003. "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* (Lecture Notes in Computer Science). Vol. 2833. Springer, Kinsale, Ireland, 789–794.

A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar, and E. Bouyakhf. 2016. "Constraint Acquisition with Recommendation Queries." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*. IJCAI/AAAI Press, New York, NY, USA, 720–726.

B. Demoen and M. G. de la Banda. 2014. "Redundant Sudoku rules." *Theory Pract. Log. Program.*, 14, 3, 363–377. doi:10.1017/S1471068412000361.

J. Devlin, M. Chang, K. Lee, and K. Toutanova. 2019. "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*. Association for Computational Linguistics, Minneapolis, MN, USA, 4171–4186.

E. C. Freuder and R. J. Wallace. 1998. "Suggestion Strategies for Constraint-Based Matchmaker Agents." In: *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP98)* (Lecture Notes in Computer Science). Vol. 1520. Springer, Pisa, Italy, 192–204.

Z. Kiziltan, M. Lippi, and P. Torroni. 2016. "Constraint Detection in Natural Language Problem Descriptions." In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*. IJCAI/AAAI Press, New York, NY, USA, 744–750.

M. Kumar, S. Kolb, and T. Guns. 2022. "Learning Constraint Programming Models from Data Using Generate-And-Aggregate." In: *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming, CP 2022* (LIPIcs). Vol. 235. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Haifa, Israel, 29:1–29:16.

I. Loshchilov and F. Hutter. 2019. "Decoupled Weight Decay Regularization." In: *Proceedings of the 7th International Conference on Learning Representations, ICLR 2019*. OpenReview.net, New Orleans, LA, USA.

J. Mason. 1997. "Purdey's General Store." *Dell Logic Puzzles*, April, 10.

Y. Mechqrane, C. Bessiere, and I. Elabbassi. 2024. "Using Large Language Models to Improve Query-based Constraint Acquisition." In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI 2024)*. ijcai.org, Jeju, South Korea, 1916–1925.

K. Michailidis, D. Tsouros, and T. Guns. 2024. "Constraint Modelling with LLMs Using In-Context Learning." In: *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming, CP 2024* (LIPIcs). Vol. 307. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Girona, Spain, 20:1–20:27.

Mistral AI. 2023. *Introducing Mistral 7B*. https://mistral.ai/fr/news/announcing-mistral-7b/. Accessed: 2024-11-05. (2023).

Mistral AI. 2024. *Mistral Large 2407 Model*. https://mistral.ai/news/mistral-large-2407/. Accessed: 2024-11-05. (2024).

S. Prestwich. 2020. "Robust Constraint Acquisition by Sequential Analysis." In: *Proceedings of the 24th European Conference on Artificial Intelligence, ECAI 2020* (Frontiers in Artificial Intelligence and Applications). Vol. 325. IOS Press, Santiago de Compostela, Spain, 355–362.

Tom B. Brown and al.. 2020. "Language Models are Few-Shot Learners." *CoRR*, abs/2005.14165. https://arxiv.org/abs/2005.14165 arXiv: 2005.14165.

D. C. Tsouros, S. Berden, and T. Guns. 2023. "Guided Bottom-Up Interactive Constraint Acquisition." In: *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming, CP 2023* (LIPIcs). Vol. 280. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Toronto, Canada, 36:1–36:20.

D. C. Tsouros and K. Stergiou. 2020. "Efficient multiple constraint acquisition." *Constraints An Int. J.*, 25, 3-4, 180–225.

D. C. Tsouros, K. Stergiou, and C. Bessiere. 2019. "Structure-Driven Multiple Constraint Acquisition." In: *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming, CP 2019* (Lecture Notes in Computer Science). Vol. 11802. Springer, Stamford, CT, USA, 709–725.

D. C. Tsouros, K. Stergiou, and P. G. Sarigiannidis. 2018. "Efficient Methods for Constraint Acquisition." In: *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming, CP 2018* (Lecture Notes in Computer Science). Vol. 11008. Springer, Lille, France, 373–388.

A Online Resources

The source code used to obtained the results reported in this paper is available at https://github.com/mechqrane/JAIR_CODE_SOURCE

Received 27 May 2025; accepted 2 December 2025