Omissions in Constraint Acquisition

Dimosthenis C. Tsouros, Kostas Stergiou, Christian Bessiere

Omissions in Constraint Acquisition

Dimosthenis C. Tsouros[1], Kostas Stergiou[1], and Christian Bessiere[2]

[1] Dept. of Electrical & Computer Engineering,
University of Western Macedonia,
Kozani, Greece
dtsouros@uowm.gr, kstergiou@uowm.gr
[2] CNRS, University of Montpellier,
Montpellier, France
bessiere@lirmm.fr

Abstract. Interactive constraint acquisition is a special case of query-directed learning, also known as "exact" learning. It is used to assist non-expert users in modeling a constraint problem automatically by posting examples to the user that have to be classified as solutions or non-solutions. One significant issue that has not been addressed in the literature of constraint acquisition is the possible presence of uncertainty in the answers of the users. We address this by introducing Limited Membership Queries, where the user has the option of replying "I don't know", corresponding to "omissions" in exact learning. We present two algorithms for handling omissions. The first one deals with omissions that are independent events, while the second assumes that omissions are related to gaps in the user's knowledge. We present theoretical results about both methods and we evaluate them on benchmark problems. Importantly, our second algorithm can not only learn (a part of) the target network, but also the constraints that cause the user's uncertainty.

1 Introduction

A major bottleneck in the use of Constraint Programming (CP) is modeling. Expressing a combinatorial problem as a constraint network requires considerable expertise in the field. Hence, one of the major challenges in CP research is that of efficiently obtaining a good model of a real problem without relying on expert human modellers [14, 16, 15]. Constraint acquisition can assist non-expert users in modeling a constraint problem automatically. It has started to attract a lot of attention as constraint acquisition systems can learn the model of a constraint problem using a set of examples that are posted as queries to a human user or to a software system [6, 8, 7, 20, 19].

Active or *interactive* constraint acquisition systems interact with the user while learning the constraint network. This is a special case of query-directed learning, also known as "exact learning" [11, 10]. State-of-the-art constraint acquisition algorithms like QuAcq [5], MQuAcq [20] and MQuAcq-2 [19] use the version space learning method [18], extended for learning constraint networks. In such systems, the basic query is to ask the user to classify an example as a solution or not solution. This "yes/no" type of question is called membership query [1], and this is the type of query that has received the most attention in active constraint acquisition.

One significant issue that has not been addressed in the literature is the presence of uncertainty, or even errors, in the answers of the user. The constraint acquisition algorithms that have been proposed are guaranteed to perform well and learn the target constraint network under the assumption that queries are always answered with certainty and correctly. However, this is not a realistic assumption as questions posted by the algorithm can be too difficult for humans to always answer reliably. Thus, it may happen that the user is uncertain about her/his answers or even gives erroneous answers. In this paper we deal with the case where the user is uncertain about her/his answers, leaving the case of erroneous answers for future work.

In the context of exact learning, uncertainty is typically captured through "omissions" in the replies of the users. The omissions can be persistent or not. In this work, we focus on persistent omissions. Such omissions have been studied for several classes of concepts [17, 13, 3, 2]. Two main models of omissions in answers to membership queries have been introduced:

1. Learning from Randomly Fallible Teachers (RFT) [3, 2]: The omissions are assumed to be independent, as "independent coin flips the first time each query is made" [3].
2. Learning from a Consistently Ignorant Teacher (CIT) [13]: In this model it is assumed that the omissions are related to a gap in the user's knowledge. Thus, the omissions are not only persistent but also they are consistent with the rest of the answers of the user. This means that if the answers to some queries imply a particular answer to another query, the latter cannot be answered with an omission.

Concerning RFT, Angluin et. al [3] presented an algorithm that can learn the target concept using equivalence and incomplete membership queries. In the exact learning model defined by [2] the learning system can learn exactly a target concept using equivalence and membership queries with at most some number l of errors or omissions in the answers of the user to the membership queries. This model introduced the *limited* membership queries (LMQ) and the *malicious* membership queries. A LMQ may be answered either by precisely classifying the example, or with the special answer "I don't know" that corresponds to an omission. In a malicious membership query, the classification by the user may be wrong. The examples answered with "I don't know" are allowed to be classified arbitrarily by the final hypothesis of a learning algorithm. The above models have been extended to other classes of concepts [9]. Focusing on learning from a CIT, Frazier et al. [13] introduced learning algorithms for several concept classes like k-term DNF formulas, decision trees etc.

In this work we address the problem of uncertainty in user answers in the context of constraint acquisition, for the first time. We focus on persistent omissions inspired by both the RFT and CIT models. We are specifically interested in the important case where there exist relations about the entities (variables) of the problem that the user is uncertain about. As a result, the user may find it difficult to classify some examples with absolute certainty. To address this, we introduce LMQs in constraint acquisition and propose two methods to handle "I don't know" replies by extending the state-of-the-art algorithm MQuAcq-2.

The first method is a baseline one that simply ignores omissions, assuming that nothing can be learned from them. The reasoning behind this is inspired by the RFT model, where the omissions are assumed to be independent.

The second method, which is our main contribution, is related to the CIT learning model and is based on the assumption that through the interaction between the learner and the user it may be possible to identify and exploit the gap in the user's knowledge, i.e. the "uncertain" constraints. For instance, the user may not be certain if a large example is a solution or not because of uncertainty about a relation between some variables. However, if a part of the example that does not include these "problematic" variables is posted to the user, then she/he may be able to classify it with certainty. Our method exploits the idea of posting partial examples that are built by dividing an example that was classified as an omission, to seek the parts of the example that cause the confusion to the user, and hence, to learn the "uncertain" constraints. As we demonstrate, this method does not only learn such constraints, but using knowledge inferred while seeking them, it also significantly cuts down the number of queries and the cpu time required for convergence.

We prove the correctness of our main method and give complexity results for both methods. We also present experimental results that evaluate our methods in the context of both RFT and CIT learning.

The rest of the paper is organized as follows. Section 2 gives background on interactive constraint acquisition. Section 3 focuses on the proposed methods. Experiments are presented in Section 4. Section 5 concludes the paper.

2 Background

The *vocabulary* (X, D) is a finite set of n variables $X = \{x_1, ..., x_n\}$ and a domain $D = \{D(x_1), ..., D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values for x_i. The vocabulary is the common knowledge shared by the user and the constraint acquisition system. A *constraint* c is a pair (rel(c), var(c)), where var(c) $\subseteq X$ is the *scope* of the constraint and rel(c) is the relation between the variables in var(c). rel(c) specifies which of their assignments are allowed. $|var(c)|$ is called the *arity* of the constraint. A *constraint network* is a set C of constraints on the vocabulary (X, D). A constraint network that contains at most one constraint for each subset of variables (i.e., for each scope) is called a *normalized constraint network*. Following the literature, we will assume that the constraint network is normalized. Besides the vocabulary, the learner has a *language* Γ consisting of *bounded arity* constraints.

An example e_Y is an assignment on a set of variables $Y \subseteq X$. e_Y violates a constraint c iff var(c) $\subseteq Y$ and the projection $e_{var(c)}$ of e_Y on the variables in the scope var(c) of the constraint is not in rel(c). A complete assignment e_X that is accepted by all the constraints in C is a solution to the problem. $sol(C)$ is the set of solutions of C. An assignment e_Y is called a partial solution iff it is accepted by all the constraints in C with a scope $S \subseteq Y$. Observe that a partial solution is not necessarily part of a complete solution.

Using terminology from machine learning, concept learning can be defined as learning a Boolean function from examples. A *concept* is a Boolean function over D^X

that assigns to each example $e \in D^X$ a value in $\{0, 1\}$, or in other words, classifies it as negative or positive. The target concept f_T is a concept that assigns 1 to e if e is a solution to the problem and 0 otherwise. In constraint acquisition, the target concept, also called target constraint network, is any constraint network C_T such that $sol(C_T) = \{e \in D^X \mid f_T(e) = 1\}$. The *constraint bias* B is a set of constraints on the vocabulary (X, D), built using the constraint language Γ. The bias is the set of all possible constraints from which the system can learn the target constraint network. $\kappa_B(e_Y)$ represents the set of constraints in B that reject e_Y.

In exact learning, the question asking the user to determine if an example e_X is a solution to the problem that the user has in mind is called a *membership query* $ASK(e)$. In the following we will use the terms *example* and *query* interchangeably. The answer to a membership query is positive if $f_T(e) = 1$ and negative if $f_T(e) = 0$. A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, asks the user to determine if e_Y, which is an assignment in D^Y, is a partial solution or not. We assume that all queries are answered correctly or with an omission by the user.

In partial queries the assumption is that the user considers only the assigned variables in the query posted. So, if the relation between the variables is insufficiently clear because some variable assignments are missing, then this does not affect the user's answer. Classifying a partial example as positive does not mean that it is necessarily part of a complete solution.

For instance, assume that we have a constraint $c \in B \wedge c \in C_T$ with $var(C) = \{x_1, x_2, x_3, x_4\}$. An example violating this constraint that includes assignments to all four variables will be classified by the user as negative. However, if the system asks the user to classify an example e_{x_1,x_2,x_3}, which does not include an assignment to x_4, the above constraint is not taken into account. So, if no other constraint from the target network is violated in this example, it will be classified as positive.

The acquisition process has *converged* on the learned network $C_L \subseteq B$ iff C_L agrees with E and for every other network $C \subseteq B$ that agrees with E, we have $sol(C) = sol(C_L)$.

3 Omissions in Constraint acquisition

State-of-the-art constraint acquisition algorithms are based on version space learning. Initially, the given language Γ is used to construct the bias B. Then the system iteratively posts membership queries to the user in order to learn the constraints of the target network. Each example posted as a query must satisfy C_L, i.e. the network that has already been learned so far, and violate at least one constraint from B. A query that satisfies these criteria is *informative*, as whatever the user's answer is, the version space's size will shrink. In case of a positive answer, each constraint $c \in B$ that violates the posted example can be removed from B (i.e. all the constraint networks containing c are removed from the version space). In case of a negative answer, one or more of the violated constraints are certainly in C_T. So, the system will search to find the scope of one, some, or all of them, depending on the algorithm used.

This is done through a function called *FindScope* in QuAcq, and its enhanced variant *FindScope-2* [20] used by MQuAcq and MQuAcq-2. Once a scope has been located, the

function *FindC* [5] is used to learn the specific constraint (i.e. its relation). *FindScope-2*, upon which we build, finds the scope of a violated constraint of the target network by successively removing entire blocks of variables from the query, and posting the resulting partial query to the user.

The above reasoning assumes that all the answers of the user are correct and also does not allow for uncertainty (omissions) in the answers. However, both of these assumptions are not realistic as humans make mistakes and are not always certain of their answers. In this paper we deal with the problem of uncertainty. We introduce the use of Limited Membership Queries (LMQ) to constraint acquisition, and propose methods to handle them. A LMQ may be answered by the user either by classifying the example as a solution ("yes") or not ("no") of the problem, or with a third option, namely an "I don't know" answer. In this paper we are mainly interested in cases where the user is uncertain about the existence or non-existence of a relation between some entities (i.e. variables) of the problem, and the type of the relation if one exists.

In general, there are some questions that arise when an "I don't know" answer is encountered: First of all, is it possible to learn something from this query? And if we believe that it is possible, what can we learn and how can we learn it?

We argue that in case we have consistent answers, it is possible to learn from an omission. We define as the *omission network* C_{OM} the set of "uncertain" constraints that the user does not know if they should be included in the target network or not, i.e. the gap in the knowledge of the user. This set may contain constraints both from C_T and from $B \setminus C_T$. Thus, if we iteratively split the initial query into partial ones then by posting these partial queries to the user we may be able to isolate one or more scopes that cause the uncertainty.

Considering these, we propose and compare two different methods for handling omissions in constraint acquisition:

1. Queries answered as "I don't know" are simply ignored, under the assumption that we cannot discover anything through such queries. So, after an omission, we can save the query to avoid posting it again, and move on to generate a new one. This method is inspired by the RFT learning model, where it is assumed that the reason of the omission cannot be learned.
2. The second method assumes that each omission is caused by relations between the variables that the user is uncertain about, and that the relevant sets of variables can be identified. When the user answers with an omission, the system commences a search for the "confusing" constraints using a reasoning similar to the search for violating constraint that algorithms like QuAcq and its variants apply at negative examples. This method corresponds to the CIT of concept learning, where it is assumed that the reason of the omission is a gap in the knowledge of the user about the problem.

We now describe and analyze the proposed methods for dealing with omissions. Both extend MQuAcq-2 [19], but they can be used in conjunction with any constraint acquisition algorithm.

3.1 Ignoring Omissions

This is a simple method, called MQuAcq-2-OM1, where any query e_Y answered as "I don't know" is ignored by the system, under the assumption that we cannot discover anything through such a query. Thus, after an omission, the example posted as a query is stored, to avoid posting it again, and then a new example is generated. Note that the following scenario is possible: The user may answer negatively to a query, and as a result the algorithm will search for one or more violated constraints following a similar process to MQuAcq-2 (and also QuAcq/MQuAcq). However, as partial queries are posted to the user during this process, some of these partial queries may be answered by an omission because the user may be certain that the initial query is not a solution, but may not be certain that some part of the query violates any constraint or not. Such partial queries are also stored and then bypassed.

Algorithm 1 depicts MQuAcq-2-OM1. The system repeatedly generates an example e satisfying C_L and rejecting at least one constraint from B (line 4). If it has not converged, it tries to acquire multiple constraints of C_T violating e. At first it posts the example to the user (line 8). If the answer of the user is positive, the set $\kappa_B(e_{Y'})$ of all the constraints from B that reject $e_{Y'}$ is removed from the bias (line 9). In case the answer is negative, it tries to learn a constraint by using the functions *FindScope-2* and *FindC* (lines 14-17).

In case of an omission, the example, which may be a complete or a partial one, is added to the set E' (line 11) and then the algorithm stops trying to learn any more constraints of C_T in this example (line 12). The set E' stores all the examples that lead to an omission, in order to avoid generating the same assignments in future queries, as shown at line 4. *FindScope-2* has been modified to apply the same reasoning when searching in partial queries. We omit the pseudocode of this modified version of *FindScope-2* for space reasons. In case *FindScope-2* has added any partial example $e_{Y''}$ to E', then the algorithm breaks the loop again (lines 18-19), stopping the search for more violated constraints of the target network in this specific partial query. If no omissions have occured, the algorithm removes the entire scope of the acquired constraint at line 20, trying to learn multiple non-overlapping constraints. This iterative process ends when the example $e_{Y'}$ does not contain any violated constraint from the bias (line 21), or if an omission has occured at some point. In these cases, the system cannot learn more violated constraints from this example, so it generates a new example at line 4 and starts over.

We now analyze the complexity of MQuAcq2-OM1 in terms of the number of queries required. We assume that l is the maximum number of omissions.

Proposition 1. *Given a bias B built from a language Γ, with bounded arity constraints, a target network C_T and a number of omissions l, MQuAcq-2-OM1 uses $O(|C_T| \cdot (\log |X| + \Gamma) + |B| + l)$ number of queries to converge.*

Proof. MQuAcq-2 needs $O(|C_T| \cdot (\log |X| + \Gamma) + |B|)$ queries in order to find the C_T and converge when we do not have omissions [19]. As our handling of omission guaranties that no query will be posted more than once, the maximum number of omissions is l. Also, the omissions do not affect the maximum number of queries needed

Algorithm 1 MQuAcq-2-OM1

Input: B, X, D (B: the bias, X: the set of variables, D: the set of domains)
Output: C_L : a constraint network

1: $C_L \leftarrow \emptyset$;
2: $E' \leftarrow \emptyset$;
3: **while** true **do**
4: Generate e_Y in D^Y accepted by C_L and rejected by B, with $e_Y \neq e'_Y \mid e'_{Y_2} \in E' \wedge Y \subseteq Y_2$;
5: **if** e = nil **then return** "C_L converged";
6: $Y' \leftarrow Y$;
7: **do**
8: answer $\leftarrow$ASK($e_{Y'}$);
9: **if** answer = "yes" **then** $B \leftarrow B \setminus \kappa_B(e_{Y'})$;
10: **else if** answer = "I don't know" **then**
11: $E' \leftarrow E' \cup e_{Y'}$;
12: break;
13: **else**
14: $Scope \leftarrow FindScope\text{-}2(e_{Y'}, \emptyset, Y', false)$;
15: $c \leftarrow FindC(e_{Y'}, Scope)$;
16: $C_L \leftarrow C_L \cup \{c\}$
17: $B \leftarrow B \setminus \{c \in B \mid var(c) = Scope\}$;
18: **if** $\exists\, Y'' \subset Y' \mid e_{Y''} \in E'$ **then**
19: break;
20: $Y' \leftarrow Y' \setminus Scope$;
21: **while** $\kappa_B(e_{Y'}) \neq \emptyset$

to learn the constraints from C_T and to converge. As a result, MQuAcq-2-OM1 uses $O(|C_T| \cdot (\log |X| + \Gamma) + |B| + l)$ number of queries to converge. □

As the number l of omissions can be equal to the maximum number of examples that can be generated in the worst case, i.e. $l \leq |D|^{|X|}$, the above complexity, as well as the space complexity of the algorithm, is exponential, which of course is a major drawback. However, under the assumption that omissions are random independent events, meaning that there are no "uncertain constraints", then the algorithm will converge once B is empty, as does MQuAcq-2.

On the other hand, in case the omissions are related to a gap in the user's knowledge (i.e. to "uncertain" constraints), their number can be exponential. This is because the "uncertain" constraints are not learned, and therefore the system cannot distinguish them from "normal" constraints. Consider the case where C_T has been learned (hence C_L is equivalent to C_T) and the constraints from $B \setminus (C_T \cup C_{OM})$ have already been removed from B. Now the system will repeatedly try to build examples that satisfy C_L and violate at least one constraint from C_{OM} in line 4. Each such example will be answered with an omission because the user cannot tell if the violation of a constraint from C_{OM} makes the example a non-solution or not. As the number of examples that satisfy C_L and violate at least one constraint from C_{OM} is exponential, and all of these

examples have to be generated in the worst case in order for the algorithm to terminate, the number of omissions is exponential.

3.2 Exploiting Omissions

We now present our main method for handling omissions, which is called MQuAcq-2-OM2, and is based on the assumption that each omission is due to uncertainty from the user's part about one or more relations between the variables, i.e. due to a gap in knowledge. In contrast to the first method, instead of discarding omissions, we now try to derive useful information from them.

The main idea is to iteratively divide an example that was answered by an omission into partial ones, in a way similar to how negative answers are handled, until a set of variables (a scope) that causes uncertainty to the user is discovered. Then the constraints corresponding to this scope can be learned and removed from B to avoid generating subsequent queries that contain the same "source of uncertainty". Another potential gain is that during this process we may come across partial queries that are answered positively, meaning that the constraints that violate them can also be removed from B.

We introduce a new set, $C_{L_{OM}}$, which stores all the "uncertain" constraints that are removed from B via an omission. We also modify the query handling process to locate scopes causing omissions and avoid violating the constraints that confuse the user in future queries, through the use of $C_{L_{OM}}$. As we now explain, each of the three possible answers by the user to a query over an example e requires different handling.

- After a *positive answer*: The constraints from B that reject the example posted are removed, as in all constraint acquisition algorithms.
- After an *omission*: In this case a partial example of e can lead either to an omission, if the variables that confuse the user are still in the partial example, or to a positive answer, in case one or more of the variables in the scope of the omission are removed. So, we can find the scope of the omission with a procedure similar to the one used in *FindScope-2*, exploiting the positive answers in order to locate the variables of the scope.
- After a *negative answer*: In such a case a partial example of e can be answered in any possible way. If one or more of the variables in the scope of the violated constraint(s) are removed, we can have a positive answer or an omission if a constraint that confuses the user is violated. In addition, the answer can still be "no", if all the variables of the constraint of C_T that is violated are still in the partial example. Hence, after a negative answer we must search for a violated constraint but we may also find the scope of an omission.

Based on these, we introduce two functions for finding the scope of a violated constraint (*FindScope-NO*) and the scope of an omission (*FindScope-OM*) so as to handle all the possible query answers.

MQuAcq-2-OM2 is depicted by Algorithm 2. The system generates an example accepted by the learned network and $C_{L_{OM}}$, while violating at least one constraint from B (line 4). We want the example to satisfy $C_{L_{OM}}$ to avoid violating any constraint that will lead to an omission. Queries answered as "I don't know" are handled at lines 11-14

Algorithm 2 MQuAcq-2-OM2

Input: B, X, D (B: the bias, X: the set of variables, D: the set of domains)
Output: $C_L, C_{L_{OM}}$: constraint networks

```
 1: C_L ← ∅;
 2: C_{L_OM} ← ∅;
 3: while true do
 4:     Generate e_Y in D^Y accepted by C_L and C_{L_OM} while rejected by B;
 5:     if e = nil then return "C_L converged";
 6:     Y' ← Y;
 7:     do
 8:         answer ← ASK(e_{Y'});
 9:         if answer = "yes" then B ← B \ κ_B(e_{Y'});
10:         else if answer = "I don't know" then
11:             OMS ← FindScope-OM(e_{Y'}, ∅, Y', false);
12:             C_{L_OM} ← C_{L_OM} ∪ FindC(e_{Y'}, OMS));
13:             B ← B \ {c ∈ B | var(c) = OMS};
14:             Y' ← Y' \ OMS;
15:         else
16:             Scope ← FindScope-NO(e_{Y'}, ∅, Y', false);
17:             c ← FindC(e_{Y'}, Scope);
18:             C_L ← C_L ∪ {c}
19:             B ← B \ {c ∈ B | var(c) = Scope};
20:             Y' ← Y' \ Scope;
21:             if κ_{C_{L_OM}}(e_{Y'}) ≠ ∅ then
22:                 Y' ← Y' \ {var(c) | c ∈ κ_{C_{L_OM}}(e_{Y'})};
23:     while κ_B(e_{Y'}) ≠ ∅
```

by calling *FindScope-OM*. This function finds the scope responsible for the omission, as we explain below, and stores it in OMS (line 11). Then, it finds the specific "uncertain" constraint (i.e. the relation) and removes it from B, adding it to $C_{L_{OM}}$ (lines 12-13). Finally, the scope found is removed from Y' (line 14), so that MQuAcq-2-OM2 can continue searching.

Queries answered negatively are handled by calling *FindScope-NO* at line 16. As explained below, this function not only finds the scope of a violated constraint but sometimes it can also find the scope of an omission. Thus, if such a scope is found (line 21), the algorithm removes the scope from Y' at line 22.

We now focus on the new functions, *FindScope-OM* and *FindScope-NO*. Both use the reasoning of *FindScope-2*, i.e. successively removing approximately half of the variables and posting a partial query. If after such a removal, the answer of the user changed then we know that the removed block contains at least one variable from the scope of a constraint we seek (a constraint from C_T or C_{OM}).

FindScope-OM (Algorithm 3) is similar to *FindScope-2*. It takes as parameters a (partial) example e_Y that has led to an omission, two sets of variables R and Y, initialized to the empty set and to Y respectively, and a Boolean variable ask_query. An invariant in any recursive call is that the example e violates at least one constraint from

Algorithm 3 FindScope-OM

Input: e, R, Y, ask_query (e: the example, R,Y: sets of variables, ask_query: boolean)
Output: $Scope$: a set of variables, the scope of an omission

```
1:  function FindScope-OM(e, R, Y, ask_query)
2:      if ask_query ∧ |κ_B(e_R)| > 0 then
3:          if rej ≠ |κ_B(e_R)| then
4:              if ASK(e_R) = "I don't know" then
5:                  rej ← |κ_B(e_R)|;
6:                  return ∅;
7:              else  B ← B \ κ_B(e_R);
8:          else  return ∅;
9:      if |Y| = 1 then return Y;
10:     split Y into < Y_1, Y_2 > such that |Y_1| = ⌈|Y|/2⌉;
11:     S_1 ← FindScope-OM(e, R ∪ Y_1, Y_2, true);
12:     S_2 ← FindScope-OM(e, R ∪ S_1, Y_1, (S_1 ≠ ∅));
13:     return S_1 ∪ S_2;
```

C_{OM}, whose scope is a subset of $R \cup Y$. The number of violated constraints from B is stored in rej, to avoid posting redundant queries to the user in any recursive call.

If *FindScope-OM* is called with ask_query = true and e_R violates at least one constraint from B (line 2) but not the same number of constraints as the previous query posted (line 3), it posts e_R as a query to the user (line 4). In case of an omission it returns the empty set (line 6), in order to remove some variables from R in the previous call. If the answer is "yes", it removes all the constraints from the bias that reject e_R and continues. Thus, it reaches line 9 only in the case where e_R does not violate any constraint from C_{OM}. Because we know that e violates at least one constraint whose scope is a subset of $R \cup Y$, in case Y is a singleton it is returned (line 9). The set Y is split in two balanced parts (line 10) and the algorithm searches recursively, in sets of variables built using R and these parts, for the scope of a violated constraint of C_{OM}, in a logarithmic number of steps (lines 11-13).

FindScope-NO (Algorithm 4) handles the case of a negative answer by the user. It operates in a slightly different way than *FindScope-OM* because after the removal of some variables, the answer of the user may change from "no" either to "yes" or to "I don't know" (see Lemma 3 below). This is because after the removal of one or more variables of the violated constraint, the user may now be confused by another constraint in the partial example formed, not being sure if the partial example is positive or not. In such a case, we continue the search in two directions. First, *FindScope-OM* is called in order to locate the scope of the omission and store it in OMS (line 8) and then find the "uncertain" constraint, which is then removed from B and added to $C_{L_{OM}}$ (line 9), as in lines 11-13 of MQuAcq-2-OM2. Also the function *FindScope-NO* continues the search for the scope of the violated constraint of C_T.

Now, let us illustrate the behaviour of our proposed approach.

Example 1. Assume that the vocabulary (X, D) given to the system is $X = \{x_1, ..., x_8\}$ and $D = \{D(x_1), ..., D(x_8)\}$ with $D(x_i) = \{1, ..., 8\}$, the target network C_T is the set

Algorithm 4 FindScope-NO

Input: e, R, Y, ask_query (e: the example, R,Y: sets of variables, ask_query: boolean)
Output: $Scope$: a set of variables, the scope of a constraint in C_T

```
1: function FindScope-NO(e, R, Y, ask_query)
2:     if ask_query ∧ |κ_B(e_R)| > 0 then
3:         if rej ≠ |κ_B(e_R)| then
4:             answer ← ASK(e_R);
5:             if answer = "yes" then B ← B \ κ_B(e_R);
6:             else if answer = "I don't know" then
7:                 if κ_{C_{L_OM}}(e_R) = ∅ then
8:                     OMS ← FindScope-OM(e_R, ∅, R, false);
9:                     C_{L_OM} ← C_{L_OM} ∪ FindC(e_R, OMS));
10:                    B ← B \ {c ∈ B | var(c) = OMS};
11:            else
12:                rej ← |κ_B(e_R)|;
13:                return ∅;
14:        else return ∅;
15:    if |Y| = 1 then return Y;
16:    split Y into < Y_1, Y_2 > such that |Y_1| = ⌈|Y|/2⌉;
17:    S_1 ← FindScope-NO(e, R ∪ Y_1, Y_2, true);
18:    S_2 ← FindScope-NO(e, R ∪ S_1, Y_1, (S_1 ≠ ∅));
19:    return S_1 ∪ S_2;
```

$\{\neq_{34}, \neq_{56}\}$, $C_{OM} = \{\neq_{34}\}$ and $B = \{\neq_{ij} \mid 1 <= i < 8 \wedge i < j <= 8\}$. Also, assume that the example generated at line 4 of MQuAcq-2-OM2 is $e = \{1, 4, 2, 2, 3, 3, 5, 6\}$.

The system will post e as a query at line 8 of MQuAcq-2-OM2. The answer will be "no" as it violates constraint $\neq_{56}$. Thus, *FindScope-NO* is called to find the scope of a violated constraint. Table 1 shows the trace of its recursive calls. A dash (-) in columns e_R and *ASK* means that no query is posted to the user, due to one of the conditions at lines 2 and 3 (e.g., at call 0 of *FindScope-NO*, as ask_query = false and $R = \emptyset$, the condition at line 2 does not hold). Recall that queries are only on the variables in R.

When half of the variables are removed from the query at recursive call 1 of *FindScope-NO*, the answer of the user changes to "I don't know". So, *FindScope-OM* is called to find the cause of uncertainty (line 8 of *FindScope-NO*). Its trace of recursive calls is also shown in Table 1. After 4 queries it finds the scope of the omission constraint and returns it (back at the 0 call), so FindC is called at line 9 of *FindScope-NO*. *FindScope-NO* then continues searching to find the violated constraint from $C_T \setminus C_{OM}$ that is responsible for the negative answer in the first place. It will be found after 3 queries and returned (back at the 0 call of *FindScope-NO*).

Analysis of MQuAcq2-OM2: We now prove the correctness of MQuAcq-2-OM2. That is, we prove that the constraints it adds to C_L and $C_{L_{OM}}$ belong indeed there, and it converges having learned all the constraints of C_T and of C_{OM} that it possibly can. We first give three lemmas showing that for each possible answer to a query ASK(e_Y), the possible answers we can have in partial queries of the form ASK($e_{Y'}$), with $Y' \subset Y$, are the ones informally described previously for the CIT model. Then we give a proposition

Table 1. Behavior of MQuAcq-2-OM2 in Example 1

Recursive calls of *FindScope-NO*					
call	R	Y	e_R	*ASK*	*return*
0	$\emptyset$	$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$	-	-	$\{x_5, x_6\}$
1	x_1, x_2, x_3, x_4	x_5, x_6, x_7, x_8	$\{1, 4, 2, 2, -, -, -, -\}$	"I don't know"	$\{x_5, x_6\}$
Go to *FindScope-OM*					
0	$\emptyset$	x_1, x_2, x_3, x_4	-	-	$\{x_3, x_4\}$
1	x_1, x_2	x_3, x_4	$\{1, 4, -, -, -, -, -\}$	"yes"	$\{x_3, x_4\}$
1.1	x_1, x_2, x_3	x_4	$\{1, 4, -, -, -, -, -\}$	"yes"	$\{x_4\}$
1.2	x_1, x_2, x_4	x_3	$\{1, 4, -, -, -, -, -\}$	"yes"	$\{x_3\}$
back to *FindScope-NO*					
1.1	$x_1, x_2, x_3, x_4, x_5, x_6$	x_7, x_8	$\{1, 4, 2, 2, 3, 3, -, -\}$	"no"	$\emptyset$
1.2	x_1, x_2, x_3, x_4	x_5, x_6	-	-	$\{x_5, x_6\}$
1.2.1	x_1, x_2, x_3, x_4, x_5	x_6	$\{1, 4, 2, 2, 3, -, -\}$	"yes"	$\{x_6\}$
1.2.2	x_1, x_2, x_3, x_4, x_6	x_5	$\{1, 4, 2, 2, -, 3, -\}$	"yes"	$\{x_5\}$

regarding the soundness of *FindScope-OM* and *FindScope-NO*. Proofs of Lemmas are ommitted for space reasons.

Lemma 1. *If ASK(e_Y) = "yes" then for any $Y' \subset Y$ it holds that ASK($e_{Y'}$) = "yes".*

Lemma 2. *If ASK(e_Y) = "I don't know" then for any $Y' \subset Y$ we can have ASK($e_{Y'}$) = "I don't know" or ASK($e_{Y'}$) = "yes".*

Lemma 3. *If ASK(e_Y) = "no" then a partial query in $Y' \subset Y$, ASK($e_{Y'}$) can return any of the possible answers.*

Proposition 2. *If* FindScope-OM *(resp.* FindScope-NO*) is given an example e_Y and returns a scope S then there exists a violated constraint $c \in C_{OM}$ (resp. $c \in C_T \setminus C_{OM}$) with $scope(c) = S$.*

Proof. An invariant of *FindScope-OM* is that the example e violates at least one constraint from C_{OM}, whose scope is a subset of $R \cup Y$ (i.e. ASK($R \cup Y$) = "I don't know"). Also, it reaches line 9 only in the case that ASK(e_R) = "yes". Thus, by Lemma 1, for $Y' \subset Y$ it holds that ASK($e_{Y'}$) = "yes", i.e. e_R does not violate any constraint from C_{OM}. Also, *FindScope-OM* returns variables only at line 9, in case Y is a singleton. As a result, for any $x_i \in S$ we know that ASK(S) = "I don't know" and ASK($S \setminus x_i$) = "yes". Hence, S is definitely a scope of a constraint from C_{OM}. □

Theorem 1. *Given a bias B built from a language Γ with bounded arity constraints, a target network C_T representable by B, and an omission network C_{OM}, MQuAcq-2-OM2 is correct.*

Proof. Soundness. MQuAcq-2-OM2 learns constraints and adds them to $C_{L_{OM}}$ or C_L only by using the function *FindC* in a scope found by *FindScope-OM* or *FindScope-NO* respectively. By Proposition 2, when *FindScope-OM* returns a scope S, then there exists a violated constraint $c \in C_{OM}$ with $scope(c) = S$ and when *FindScope-NO* returns a scope S, then there exists a violated constraint $c \in C_T \setminus C_{OM}$ with $scope(c) = S$. Also, *FindC* has been proved to be correct for normalized target networks [7]. Hence,

MQuAcq-2OM2 is sound, as for every constraint c added to C_L it holds that $c \in C_T \setminus C_{OM}$ and for every constraint c added to $C_{L_{OM}}$ it holds that $c \in C_{OM}$.

Completeness. An example generated at line 4 of MQuAcq-2-OM2 must violate at least one constraint from B. Given such an example e_Y, MQuAcq-2-OM2 will find at least one constraint from C_T (lines 16-20) or C_{OM} (lines 11-14), if one exists, and then remove it from B (lines 13,19). It finds the scope of a constraint of $C_T \setminus C_{OM}$ using *FindScope-NO* and of C_{OM} using *FindScope-OM*. After a scope has been located, it had been proved that *FindC* will find a constraint in the given scope if one exists [7]. The same applies to constraints of C_{OM}, as the procedure is exactly the same, because when ASK(e_Y) = "I don't know" then for any $Y' \subset Y$ we can have ASK($e_{Y'}$) = "I don't know" or ASK($e_{Y'}$) = "yes" (Lemma 2). If no constraint can be learned by an example (i.e. $\kappa_{C_T}(e_Y) = \kappa_{C_{OM}}(e_Y) = \emptyset$), it will remove all the violated constraints from B (line 9). Thus, the size of B will decrease after each query. The algorithm terminates only when no example can be generated at line 4. In this case, the system has converged as C_L agrees with E and for every other network $C \subseteq B$ that agrees with E and, we have $sol(C) = sol(C_L)$. Hence, the system learned any constraint in $C_T \setminus C_{OM}$ that could be learned. As no constraint from B can be violated, the same applies for C_{OM}. Hence, MQuAcq-2-OM2 is complete. □

Proposition 3. *Given a bias B built from a language Γ, with bounded arity constraints, a target network C_T and an omission network C_{OM}, MQuAcq-2-OM2 uses $O(|C_T| \cdot (\log|X| + |\Gamma|) + |B| + l)$ number of queries to converge, with $l <= |C_{OM}| \cdot (\log|X| + |\Gamma|)$.*

Proof. (sketch). MQuAcq-2-OM2 will learn $|C_T \setminus C_{OM}|$ constraints and will find $|C_{OM}|$ omission constraints. The constraints from $C_T \setminus C_{OM}$ are learned using the functions *FindScope-NO* and *FindC* while the constraints of $|C_{OM}|$ are found using the functions *FindScope-OM* and *FindC*. Both *FindScope-NO* and *FindScope-OM* need a maximum number of $|S| \cdot \log|Y| = O(\log|X|)$ queries in order to find a scope S in an example e_Y, as they use a process very similar to *FindScope* [5]. *FindC* needs at most $|\Gamma|$ queries to learn a constraint. Also, assuming that each positive query removes only one constraint from $|B|$ it will need to ask a total number of $|B| - |C_T \setminus C_{OM}| - |C_{OM}| = O(|B|)$ queries to prune B and reach convergence. Thus, the total total number of queries in $O(|C_T| \cdot (\log|X| + |\Gamma|) + |B| + l)$, with $l <= |C_{OM}| \cdot (\log|X| + |\Gamma|)$. The above result in $O((|C_T| + |C_{OM}|) \cdot (\log|X| + |\Gamma|) + |B|)$. □

4 Experimental Evaluation

We ran experiments both in the RFT and the CIT models. In the former, as the omissions are not related to missing knowledge, we evaluated only MQuAcq-2-OM1. In the latter we compared our methods to each other. We used MQuAcq-2 without omissions as a reference point. Experiments were run on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 16 GB of RAM. In more detail:

- We used the *max*$_B$ heuristic for query generation [20]. *max*$_B$ generates examples violating as many constraints as possible from B. The best example found within 1 second is returned, even if not proved optimal. If none is found, we continue and return the first suitable (partial) example found. The variable involved in the most constraints in B is chosen during search. Values are chosen randomly.

- In the RFT model, a query is answered by an omission with 20% probability.
- In the CIT model, we used a cutoff for MQuAcq-2-OM1, as the number of omissions can be exponential (it did not complete within 10 hours). So we only present results for MQuAcq-2-OM1 in CIT with a cutoff, which was imposed as follows: the system stops at line 3 when the number of omission $\#omissions$ is more than the 30% of the total number of queries $\#queries$.
- To compare the algorithms on the same scenario, all our experiments concern the extreme case where C_L is initially empty, This results in a number of queries that may seem too large for human users. But in real applications, background knowledge can used by giving a frame of basic constraints or by using other methods, e.g. ModelSeeker [4], to extract some constraints from known solutions. Then, our algorithms can be used to finalize the model.
- We measure the size of the learned network C_L, the size of the learned omission network $C_{L_{OM}}$, the total number of queries $\#queries$, the total number of omission answers $\#omissions$ and the total cpu time T. We present results of MQuAcq-2 without *analyze&Learn*, MQuAcq-2-OM1 and MQuAcq-2-OM2. Each algorithm was run 5 times and the means are presented.

We used the following benchmarks in our study:

Zebra. It consists of 25 variables with domains of maximum size 5. The target network C_T contains five cliques of 10 $\neq$ constraints each and 11 additional constraints. The bias was initialized with 1200 binary constraints from the language $\Gamma = \{=, \neq, >, <, x_i - x_j = 1, |x_i - x_j| = 1\}$. C_{OM} contains the constraints of C_T not belonging to a clique, and 5 randomly chosen constraints from B.

Murder. It has 20 variables with domains of size 5. C_T contains 4 cliques of $\neq$ constraints and 12 additional constraints. The bias was initialized with 760 constraints from the language $\Gamma = \{=, \neq, >, <\}$. C_{OM} contains the constraints of C_T not belonging to a clique, and 10 randomly chosen constraints from B.

Random. We generated a random target network with 50 variables, domains of size 10, and 122 $\neq$ constraints. The bias was initialized with 19,800 constraints, using the language $\Gamma = \{=, \neq, >, <\}$. C_{OM} was created randomly, containing 15 constraints with only 1 belonging to C_T.

Radio Link Frequency Assignment Problem (RLFAP). We use a simplified version of the communication problem from [12], with 50 variables having domains of size 40. C_T contains 125 distance constraints. The bias was built using a language of 2 distance constraints ($\{|x_i - x_j| > y, |x_i - x_j| = y\}$) with 5 different possible values for y. This led to a language of 10 different distance constraints. In total, B contains 12,250 constraints. C_{OM} was created randomly, containing 15 constraints in total with 5 belonging to C_T.

Results from the RFT model presented in Table 2 (see rows for MQuAcq-2-OM1$_{RFT}$), confirm our complexity analysis. When omissions are random events, the queries posted by MQuAcq-2-OM1 do not increase a lot compared to MQuAcq-2 without omissions. We can see that the increase is related to the number of variables of the problem. On the other hand, cpu times increase significantly. This is because most of the (few) additional queries are generated queries and not partial ones (because the system generates a

new query when an omission occurs). As query generation is the most time-consuming process of the algorithm, this affects the run times considerably.

Table 2. Results from the RFT and CIT models.

Benchmark	Algorithm	$\|C_L\|$	$\|C_{L_{OM}}\|$	$\#q$	$\#om$	T
Zebra	MQuAcq-2	61	0	494	0	9.28
	MQuAcq-2-OM1$_{RFT}$	60	0	624	115	48.09
	MQuAcq-2-OM1$_{CIT}$	49	0	534	135	33.07
	MQuAcq-2-OM2	48	12	480	73	8.37
Murder	MQuAcq-2	52	0	384	0	12.70
	MQuAcq-2-OM1$_{RFT}$	52	0	484	101	51.18
	MQuAcq-2-OM1$_{CIT}$	40	0	411	124	48.87
	MQuAcq-2-OM2	39	17	397	78	11.81
Random122	MQuAcq-2	122	0	1031	0	37.48
	MQuAcq-2-OM1$_{RFT}$	122	0	1464	294	139.22
	MQuAcq-2-OM1$_{CIT}$	121	0	1583	475	404.01
	MQuAcq-2-OM2	121	15	1095	68	37.99
RLFAP	MQuAcq2	125	0	1157	0	241.10
	MQuAcq2-OM1$_{RFT}$	125	0	1391	309	459.71
	MQuAcq2-OM1$_{CIT}$	28	0	401	121	28.13
	MQuAcq2-OM2	119	15	1273	115	602.40

Focusing on the CIT model, where the omissions are related to a gap in the user's knowledge (i.e., the "uncertain" constraints of C_{OM}), the results (Table 2) demonstrate that both MQuAcq-2-OM1 with a cutoff (denoted MQuAcq-2-OM1$_{CIT}$) and MQuAcq-2-OM2 achieve a quite good performance in most of the problems. The exception for MQuAcq-2-OM1 is RLFAP where it learns only 23% of $C_T \setminus C_{OM}$, because the cutoff condition is activated too early. On the other hand, MQuAcq-2-OM2 gives very good overall results, with the increase in number of queries being only up to 6.2% compared to MQuAcq-2 without omissions. In addition, the omission answers in MQuAcq-2-OM2 are quite fewer than in MQuAcq-2-OM1 (up to 86% in Random). Also, we observe that in Zebra and Murder the number of queries is very close to that of MQuAcq-2. This happens because most of the "uncertain" constraints are in C_T.

5 Conclusions

One significant issue that has not been addressed in constraint acquisition is the possible presence of uncertainty in the answers of the users. We address this for the first time by introducing Limited Membership Queries in constraint acquisition. We propose two algorithms for handling omissions that correspond to the two models of omissions in concept learning. The first method assumes that omissions are independent events and nothing can be learned from them, while the second assumes that they are related to gaps in the user's knowledge, and can be exploited. Theoretical and experimental results show that both methods perform well when used in their corresponding omission models.

References

1. Angluin, D.: Queries and concept learning. Machine learning **2**(4), 319–342 (1988)
2. Angluin, D., Kriķis, M., Sloan, R.H., Turán, G.: Malicious omissions and errors in answers to membership queries. Machine Learning **28**(2-3), 211–255 (1997)
3. Angluin, D., Slonim, D.K.: Randomly fallible teachers: Learning monotone dnf with an incomplete membership oracle. Machine Learning **14**(1), 7–26 (1994)
4. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Principles and practice of constraint programming. pp. 141–157. Springer (2012)
5. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C.G., Walsh, T., et al.: Constraint acquisition via partial queries. In: IJCAI. vol. 13, pp. 475–481 (2013)
6. Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M., et al.: Query-driven constraint acquisition. In: IJCAI. vol. 7, pp. 50–55 (2007)
7. Bessiere, C., Daoudi, A., Hebrard, E., Katsirelos, G., Lazaar, N., Mechqrane, Y., Narodytska, N., Quimper, C.G., Walsh, T.: New approaches to constraint acquisition. In: Data mining and constraint programming, pp. 51–76. Springer (2016)
8. Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence **244**, 315–342 (2017)
9. Bisht, L., Bshouty, N.H., Khoury, L.: Learning with errors in answers to membership queries. Journal of Computer and System Sciences **74**(1), 2–15 (2008)
10. Bshouty, N.H.: Exact learning from an honest teacher that answers membership queries. Theoretical Computer Science **733**, 4–43 (2018)
11. Bshouty, N.: Exact learning boolean functions via the monotone theory. Information and Computation **123**(1), 146 – 153 (1995)
12. Cabon, B., De Givry, S., Lobjois, L., Schiex, T., Warners, J.P.: Radio link frequency assignment. Constraints **4**(1), 79–89 (1999)
13. Frazier, M., Goldman, S., Mishra, N., Pitt, L.: Learning from a consistently ignorant teacher. In: Proceedings of the seventh annual conference on Computational learning theory. pp. 328–339. ACM (1994)
14. Freuder, E.C.: Modeling: the final frontier. In: The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London. pp. 15–21 (1999)
15. Freuder, E.C.: Progress towards the holy grail. Constraints **23**(2), 158–171 (2018)
16. Freuder, E.C., O'Sullivan, B.: Grand challenges for constraint programming. Constraints **19**(2), 150–162 (2014)
17. Goldman, S.A., Mathias, H.D.: Learning k-term dnf formulas with an incomplete membership oracle. In: COLT. pp. 77–84. Citeseer (1992)
18. Mitchell, T.M.: Version spaces: an approach to concept learning. Tech. rep., STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE (1978)
19. Tsouros, D.C., Stergiou, K., Bessiere, C.: Structure-driven multiple constraint acquisition. In: International Conference on Principles and Practice of Constraint Programming. pp. 709–725. Springer (2019)
20. Tsouros, D.C., Stergiou, K., Sarigiannidis, P.G.: Efficient methods for constraint acquisition. In: 24th International Conference on Principles and Practice of Constraint Programming (2018)