Multiple Constraint Aquisition

Robin Arcangioli, Christian Bessiere, Nadjib Lazaar

Multiple Constraint Acquisition

Robin Arcangioli Christian Bessiere Nadjib Lazaar
CNRS, University of Montpellier, France
{arcangioli,bessiere,lazaar}@lirmm.fr

Abstract

QUACQ is a constraint acquisition system that assists a non-expert user to model her problem as a constraint network by classifying (partial) examples as positive or negative. For each negative example, QUACQ focuses onto a constraint of the target network. The drawback is that the user may need to answer a great number of such examples to learn all the constraints. In this paper, we provide a new approach that is able to learn a maximum number of constraints violated by a given negative example. Finally we give an experimental evaluation that shows that our approach improves on QUACQ.

1 Introduction

Constraint programming is a powerful paradigm for modeling and solving combinatorial problems. However, it has long been recognized that expressing a combinatorial problem as a constraint network requires significant expertise in constraint programming [Freuder, 1999]. To alleviate this issue, several techniques have been proposed to *acquire* a constraint network. For example, the matchmaker agent [Freuder and Wallace, 2002] interactively asks the user to provide one of the constraints of the target problem each time the system proposes an incorrect (negative) solution. In [Beldiceanu and Simonis, 2012], Beldiceanu and Simonis have proposed MODELSEEKER, a system devoted to problems with regular structures, like matrix models. Based on examples of solutions and non-solutions provided by the user, CONACQ.1 [Bessiere *et al.*, 2005; 2015] learns a set of constraints that correctly classifies all the examples given so far. As an active learning version, CONACQ.2 [Bessiere *et al.*, 2007; 2015] proposes examples to the user to classify (i.e., membership queries). In [Shchekotykhin and Friedrich, 2009], the approach used in CONACQ.2 has been extended to allow the user to provide *arguments* as constraints to converge more rapidly.

QUACQ is a recent active learning system that is able to ask the user to classify partial queries [Bessiere *et al.*, 2013]. QUACQ iteratively computes membership queries. If the user says *yes*, QUACQ reduces the search space by removing all constraints violated by the positive example. If the user says *no*, QUACQ focuses onto one, and only one, constraint of the target network in a number of queries logarithmic in the size of the example. This key component of QUACQ allows it to always converge on the target set of constraints in a polynomial number of queries. However, even that good theoretical bound can be hard to put in practice. For instance, QUACQ requires the user to classify more than 8000 examples to get the complete Sudoku model.

An example can be classified as negative because of more than one violated target constraint. In this paper, we extend the approach used in QUACQ to make constraint acquisition more efficient in practice by learning, not only one constraint on a negative example, but a maximum number of constraints violated by this negative example. Inspired by the work on enumerating infeasibility in SAT (Minimal Unsatisfiability Subsets [Liffiton and Sakallah, 2008]), we propose an algorithm that computes all minimal scopes of target constraints violated by a given negative example.

2 Background

The constraint acquisition process can be seen as an interplay between the user and the learner. For that, user and learner need to share a same *vocabulary* to communicate. We suppose this vocabulary is a set of n variables $X = \{x_1, \ldots, x_n\}$ and a domain $D = \{D(x_1), \ldots, D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values for x_i. A constraint c_S is defined as a pair where S is a subset of variables of X, called *the constraint scope*, and c is a relation over D of arity $|S|$. A *constraint network* is a set C of constraints on the vocabulary (X, D). An assignment e_Y on a set of variables $Y \subseteq X$ is *rejected* by a constraint c_S if $S \subseteq Y$ and the projection e_S of e_Y on the variables in S is not in c. An assignment on X is a *solution* of C if and only if it is not rejected by any constraint in C. The set of solutions of C is denoted by $sol(C)$.

In addition to the vocabulary, the learner owns a *language* Γ of bounded arity relations from which it can build constraints on specified sets of variables. Adapting terms from machine learning, the *constraint bias*, denoted by B, is a set of constraints built from the constraint language Γ on the vocabulary (X, D), from which the learner builds the constraint network. A *concept* is a Boolean function over $D^X = \Pi_{x_i \in X} D(x_i)$, that is, a map that assigns to each example $e \in D^X$ a value in $\{0, 1\}$. We call *target concept* the concept f_T that returns 1 for e if and only if e is a solution of the problem the user has in mind. In a constraint programming context, the target concept is represented by a *target network* denoted by C_T.

A *membership query* $ASK(e)$ is a classification question asked to the user, where e is a *complete* assignment in D^X. The answer to $ASK(e)$ is "yes" if and only if $e \in sol(C_T)$. A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, is a classification question asked to the user, where e_Y is a *partial* assignment in $D^Y = \Pi_{x_i \in Y} D(x_i)$. A set of constraints C *accepts* a partial assignment e_Y if and only if there does not exist any constraint $c \in C$ rejecting e_Y. The answer to $ASK(e_Y)$ is "yes" if and only if C_T accepts e_Y. A classified assignment e_Y is called a positive or negative *example* depending on whether $ASK(e_Y)$ is "yes" or "no". For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y.

We now define *convergence*, which is the constraint acquisition problem we are interested in. We are given a set E of (partial) examples labelled by the user as positive or negative. We say that a constraint network C agrees with E if C accepts all the examples labelled as positive in E and does not accept those labelled as negative. The learning process has *converged* on the learned network $C_L \subseteq B$ if C_L agrees with E and for every other network $C' \subseteq B$ agreeing with E, we have $sol(C') = sol(C_L)$. If there does not exist any $C_L \subseteq B$ such that C_L agrees with E, we say that we have *collapsed*. This happens when $C_T \not\subseteq B$.

Finally, we introduce the notion of *minimal scope*, which is similar to the notion of MUS (*Minimal Unsatisfiable Subset*) in SAT [Liffiton and Sakallah, 2008].

Definition 1 (Minimal Scope)**.** *Given a negative example e, a minimal scope is a subset of variables $U \subseteq X$ such that $ASK(e_U) = no$ and for all $x_i \in U, ASK(e_{U\setminus\{x_i\}}) = yes$.*

3 Multiple Constraint Acquisition

In this section, we propose MULTIACQ for Multiple Acquisition. MULTIACQ takes as input a bias B on a vocabulary (X, D) and returns a constraint network C_L equivalent to the target network C_T by asking (partial) queries. The main difference between QUACQ and MULTIACQ is the fact that QUACQ learns and focuses on one constraint each time we have a negative example, whereas MULTIACQ tries to learn more than one explanation (constraint) of why the user classifies a given example as negative.

3.1 Description of MULTIACQ

MULTIACQ (see Algorithm 1) starts by initializing the C_L network to the empty set (line 1). If C_L is unsatisfiable (line 3), the acquisition process will reach a collapse state. At line 4, we compute a complete assignment e satisfying the current learned network C_L and violating at least one constraint in B. If such an example does not exist (line 5), then all the constraints in B are implied by C_L, and we have converged. Otherwise, we call the function `findAllScopes` on the example e (line 8). If e is negative, `findAllScopes` returns the set *MSes* of minimal scopes of e. As each minimal scope in *MSes* represents the scope of a violated constraint that must be learned, the function `findC` is called for each such minimal scope (line 10). It returns a constraint from C_T with the given minimal scope as scope, that rejects e. We do not give the code of function `findC` because it is implemented as in [Bessiere *et al.*, 2013]. If no constraint is returned (line 11), this is a second condition for collapsing as we could not find in the bias B a constraint rejecting the negative example. Otherwise, the constraint returned by `findC` is added to the learned network C_L (line 12).

Algorithme 1 : MULTIACQ

1 $C_L \leftarrow \varnothing$
2 **while** $true$ **do**
3 | **if** $sol(C_L) = \varnothing$ **then return** "collapse"
4 | choose e in D^X accepted by C_L and rejected by B
5 | **if** $e = nil$ **then return** "convergence on C_L"
6 | **else**
7 | | *MSes* $\leftarrow \varnothing$
8 | | `findAllScopes` (e, X, *MSes*)
9 | | **foreach** $Y \in$ *MSes* **do**
10 | | | $c_Y \leftarrow$ `findC`(e, Y)
11 | | | **if** $c_Y = nil$ **then return** "collapse"
12 | | | **else** $C_L \leftarrow C_L \cup \{c_Y\}$

3.2 Description of the function `findAllScopes`

The recursive function `findAllScopes` (see Function 1) takes as input a complete example e and a subset of variables Y (X for the first call). Function `findAllScopes` returns $true$ if and only if there exists a minimal scope of e in Y. But the real aim of function `findAllScopes` is to fill the set *MSes* with all the minimal scopes of e. Function `findAllScopes` starts by checking if the subset Y is already reported as a minimal scope (line 1). If this occurs, we return $true$. As we assume that the bias is expressive enough to learn C_T, when $\kappa_B(e_Y) = \varnothing$ (i.e., there is no violated constraint in B to learn on Y), it implies that $ASK(e_Y) = yes$ and we return $false$ (line 2). As a third check (line 3), we verify if we have not reported a subset of Y as a minimal scope. If that is the case, we are sure that $ASK(e_Y) = no$ and we get into the main part of the algorithm. If not, at line 4, we ask the user to classify the (partial) example e_Y. If it is positive, we remove from B all the constraints rejecting e_Y and we return $false$ (lines 5-6). If $ASK(e_Y) = no$, this means that there exist minimal scopes in Y and in what follows, we try to report them. For that, we call `findAllScopes` on each subset of Y built

Function 1 : `findAllScopes` (e, Y, *MSes*) : **Boolean**

1 **if** $Y \in$ *MSes* **then return** $true$
2 **if** $\kappa_B(e_Y) = \varnothing$ **then return** $false$
3 **if** $\nexists M \in$ *MSes* $\mid M \subset Y$ **then**
4 | **if** $ASK(e_Y) = yes$ **then**
5 | | $B \leftarrow B \setminus \kappa_B(e_Y)$
6 | | **return** $false$
7 $flag \leftarrow false$
8 **foreach** $x_i \in Y$ **do**
9 | $flag \leftarrow$ `findAllScopes`(e, $Y \setminus \{x_i\}$, *MSes*) $\vee$ $flag$
10 **if** $\neg flag$ **then** *MSes* $\leftarrow$ *MSes* $\cup \{Y\}$
11 **return** $true$

by removing one variable from Y (lines 8-9). If any sub-call to `findAllScopes` returns $true$, the Boolean $flag$ will be set to $true$, which means that Y itself is not a minimal scope (line 10) and we return $true$ at line 11. Otherwise, when all the sub-calls of `findAllScopes` on Y subsets return $false$, this means that Y is a minimal scope (line 10) so we add it to the set *MSes*.

3.3 Analysis

We analyze the correctness (i.e., soundness and completeness) of `findAllScopes`. We also give a complexity study of MULTIACQ in terms of the number of queries it can ask of the user.

Lemma 1. *If $ASK(e_Y) = yes$ then for any $Y' \subseteq Y$ we have $ASK(e_{Y'}) = yes$.*

Proof. Assume that $ASK(e_Y) = yes$. Hence, there exists no constraint from C_T violated by e_Y. For any Y' subset of Y, the projection $e_{Y'}$ also does not violate any C_T constraint (i.e., $ASK(e_{Y'}) = yes$). □

Lemma 2. *If $ASK(e_Y) = no$ then for any $Y' \supseteq Y$ we have $ASK(e_{Y'}) = no$.*

Proof. Assume that $ASK(e_Y) = no$. Hence, there exists at least one constraint from C_T violated by e_Y. For any Y' superset of Y, $e_{Y'}$ also violates at least the constraint violated by e_Y (i.e., $ASK(e_{Y'}) = no$). □

Theorem 1. *Given a bias B and a target network C_T representable by B, function `findAllScopes` is correct.*

Proof. Soundness. Assume that we have a complete assignment e_X. We show that any subset Y added to *MSes* by `findAllScopes` is a minimal scope. If Y is added to *MSes* (line 10), this means that $flag = false$, and either there exists $M \subset Y$ in *MSes* or $ASK(e_Y) = no$ (lines 3 and 4 to avoid line 6). As $flag = false$, all sub-calls of `findAllScopes` (line 9) on subsets of Y of size $|Y| - 1$ returned $false$. Hence, there does not exist any $M \subset Y$ in *MSes*. Thus, $ASK(e_Y) = no$. In addition, the fact that `findAllScopes` returns $false$ when called on any $Z \subseteq Y$ implies that $ASK(e_Z) = yes$ or $\kappa_B(e_Z) = \varnothing$. As the target network is representable by the bias B, $\kappa_B(e_Z) = \varnothing \Rightarrow ASK(e_Z) = yes$. As a result, all the sub-calls of `findAllScopes` on the subsets of Y return $false$ because $\forall x_i \in Y$, we have $ASK(e_{Y \setminus \{x_i\}}) = yes$. Then, knowing that $ASK(e_Y) = no$ implies that Y is a minimal scope (Definition 1).

Completeness. Given a complete example e_X, if $ASK(e_X) = yes$, there is no minimal scope of e to find. Suppose that $ASK(e_X) = no$ with a minimal scope M to find. The three conditions at lines 1, 2 and 4 are not satisfied ($X \notin$ *MSes* because $M \subseteq X$, $\kappa_B(e_X) \neq \varnothing$, and $ASK(e_X) = no$). Hence, function `findAllScopes` is called on all the subsets $X'_i \subset X$ such that $X'_i = X \setminus \{x_i\}, x_i \in X$. As $M \subseteq X$ is a minimal scope of e, $ASK(e_M) = no$ and $\forall Y \subseteq X \setminus M$, $ASK(e_{M \cup Y}) = no$ (Lemma 2). Hence, any call of `findAllScopes` on a superset of M will behave exactly as on X. By induction, there will be a call of `findAllScopes` on M. As M is a minimal scope, for any subset $M' \subset M$ we have $ASK(e_{M'}) = yes$ (Definition 1). Thus, M is added to *MSes* by `findAllScopes` at line 10. □

Theorem 2. *Given a bias B built from a language Γ of bounded arity constraints, and a target network C_T,* MULTIACQ *uses $O(|C_T| \cdot (|X| + |\Gamma|) + |B|)$ queries to prove convergence on the target network or to collapse.*

Proof. We first show that the number of queries asked by `findAllScopes` is bounded above by $|C_T| \cdot |X| + |B|$.

Let us start with the number of queries asked by `findAllScopes` and classified as negative by the user. By definition, a query on e_Y where Y is a subset of X is classified as negative if and only if there exists a minimal scope M of e, $M \subseteq Y$. Hence, given a minimal scope $M \subset X$, each time `findAllScopes` asks a query on a superset Y of M, it is classified as negative and `findAllScopes` reduces the size of Y by one. Thus, the number of negative queries asked by `findAllScopes` to go from X to M is bounded above by $|X| - |M|$. The worst case is when $|M| = 1$ where we will have $|X| - 1$ negative queries. Once M is found, the use of Lemma 2 at lines 1 and 3 of `findAllScopes` ensures that we will never ask again a query on a superset of M. With the fact that the total number of minimal scopes is bounded by $|C_T|$, we have that the total number of negative queries asked by `findAllScopes` is bounded above by $|C_T| \cdot |X|$.

We now show that the number of queries asked by `findAllScopes` and classified as positive by the user is bounded above by $|B|$. Let us take two subsets Y and Y' where $\kappa_B(e_{Y'}) \subseteq \kappa_B(e_Y)$. If an ASK on Y is classified as positive, we remove $\kappa_B(e_Y)$ from B (line 5) and therefore $e_{Y'}$ will be discarded without any query because $\kappa_B(e_{Y'})$ became empty (line 2). The worst case is when we remove, at each time, only one constraint from B (i.e., $|\kappa_B(e_Y)| = 1$). In this case, `findAllScopes` asks $|B|$ positive queries.

As a result, the total number of queries asked by `findAllScopes` is bounded above by $|C_T| \cdot |X| + |B|$.

Function `findC` uses $|\Gamma|$ queries to return a constraint from C_T [Bessiere *et al.*, 2013] and thus $|C_T| \cdot |\Gamma|$ queries to return all the constraints. Therefore, the total number of queries asked by MULTIACQ to converge to the target network is bounded above by $|C_T| \cdot |X| + |B| + |C_T| \cdot |\Gamma|$. □

From this complexity analysis we can see that MULTIACQ converges on a constraint of the target network in a number of queries linear in the size of the example whereas QUACQ converges on a constraint in a *logarithmic* number of queries. We will show that this decay in theoretical complexity does not prevent a good experimental behavior.

4 Strategies

Given a negative example, function `findAllScopes` asks partial queries to compute the set of minimal scopes of constraints that explain why the user said *no*. Function `findAllScopes` needs to explore, in the worst case, a search space containing $2^{|X|}$ candidate scopes. Hence, generating a

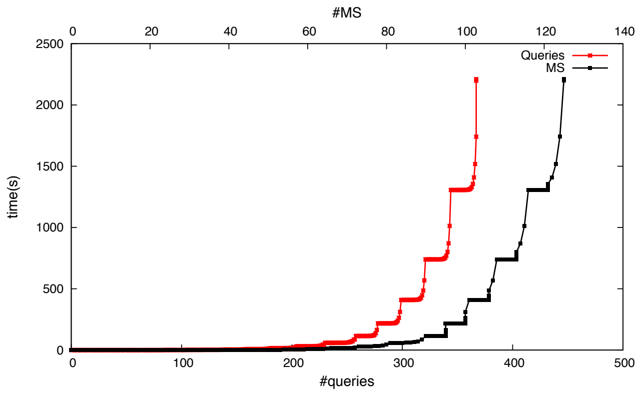

Figure 1: #queries and #minimal scopes (#MS) returned by `findAllScopes` on an RLFAP negative example.

query in `findAllScopes` can be time-consuming. We analyze the behavior of `findAllScopes` on a sample problem. Based on this analysis, we propose strategies of exploration of the search space to get the best tradeoff between time-consumption and number of queries.

4.1 Analyzing the behavior of `findAllScopes`

We take the *Radio Link Frequency Assignment Problem* described in Section 5 as sample problem.

In Figure 1, we report the time needed to compute queries and minimal scopes using `findAllScopes` on a negative example. The first observation we make is that the time needed to compute queries and minimal scopes follows an exponential scale. Generating a query and computing a minimal scope becomes more and more time-consuming as minimal scopes are found. The increasing cost of generating minimal scopes is due to the fact that `findAllScopes` returns quickly most of the minimal scopes just by exploring the first branch (90% of the minimal scopes are found in the first branch for our sample problem). The few remaining minimal scopes are found by exploring the whole remaining branches.

The second observation is that the two curves are almost parallel. Thus, the number of minimal scopes increases quasi-linearly with the number of queries. Hence, the increasing cost of finding minimal scopes is not due to an increasing number of queries required to find a minimal scope. It is due to the increasing cost of generating queries. The increasing cost of generating queries is because during search, `findAllScopes` will apply more and more Lemmas 1 and 2, avoiding to ask question on many branches.

4.2 Heuristics

It is not satisfactory, in an interactive process, to let the human user wait too long between two queries. We then use the observations made in the previous subsection to propose a combination of heuristics to maintain a good trade-off between number of queries and waiting time.

Our first heuristic is merely to use a cutoff on the waiting time between two queries. As a result, we guarantee that the user will not wait too long between two queries. However, if the cutoff is too short, we will not be able to explore the last branches of the search for minimal scopes, those branches including the last variables.

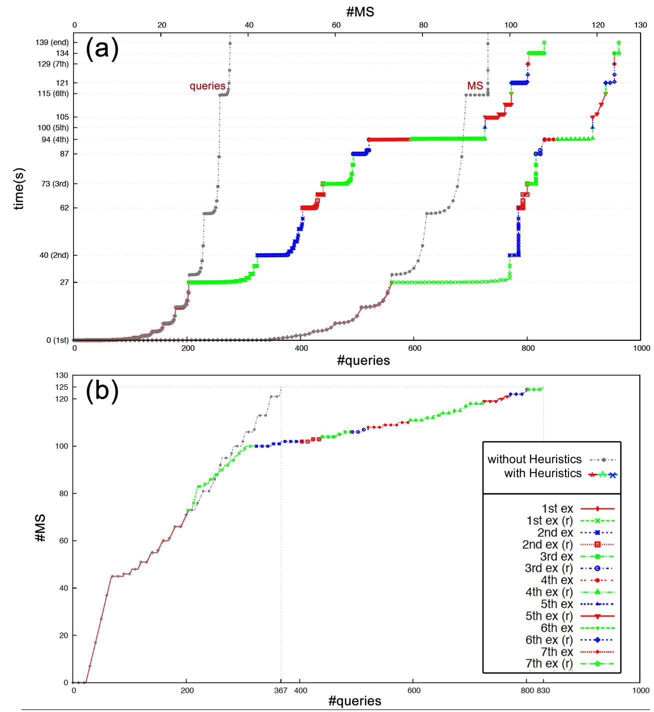

Figure 2: `findAllScopes` with/without heuristics on RLFAP.

We combine the cutoff technique with a second heuristic based on reordering the variables. Given a call to `findAllScopes` on a *complete* (negative) example e, after triggering the cutoff for the first time, we call again `findAllScopes` on the same complete example e, but on a reverse order of the variables. If a second cutoff is triggered, we come back to MULTIACQ, generate a new example and make a shuffle on the variable order. To ensure termination of MULTIACQ, we force `findAllScopes` to return at least a minimal scope before cutting off.

We implemented this combination of heuristics in `findAllScopes` and tested it on our sample problem. The cutoff on the time between two queries has been set to 5 seconds. This is an acceptable waiting time for a human user.

Figure 2(a) shows a comparison on #queries and #minimal scopes (#MS) computed over time by `findAllScopes` with and without heuristics. Without heuristics (grey lines), `findAllScopes` returns its set of minimal scopes on only one negative example generated by MULTIACQ. But the time needed per minimal scope grows during search, as already observed in Figure 1. With heuristics (multicolored lines), `findAllScopes` will cut off the search, reverse the variables, restart, cut off again, and take a new negative example generated by MULTIACQ. This is marked in the figure as 1st negative example, 2nd negative example, and so on. We observe that for a same amount of time, the number of minimal scopes found with heuristics increases and thus, the total time needed to return the whole set of minimal scopes decreases (from 36 minutes without heuristics to 139 seconds with heuristics).

Figure 2(b) shows #minimal scopes over #queries with and without heuristics. With heuristics, we observe a clear decrease of the ratio #minimal scopes/#queries. The total number of queries increases from 500 to more than 800.

This example shows that the use of heuristics allows us to reduce the time needed to compute the total number of minimal scopes by a factor of 44 with an increase by a factor of almost 2 on the number of queries asked.

5 Experimentations

We made experiments to evaluate the performance of MULTIACQ and its **findAllScopes** function compared to QUACQ. We also evaluate the version using cut-offs, that we call MACQ-CO (i.e., **M**ULTI**Acq** with **C**uts-**O**ffs). Our tests were conducted on an 1,6 GHz Intel Core i5 with 4.0GB of RAM (1600 MHz DDR3). We first present the benchmark problems we used for our experiments.

5.1 Benchmark Problems

Murder. Someone was murdered last night. There are 5 suspects, each having an item, an activity and a motive for the crime. Under a set of additional clues, the problem is who was the murderer? The target network has the 20 variables we described (i.e., the 5 suspects and their items, activities and motives) with domains of size 5, and 53 binary constraints. We use a bias of 380 constraints based on the language $\Gamma = \{=, \neq\}$.

Latin Square. A Latin square is an n x n array filled with n different Latin letters, each occurring exactly once in each row and exactly once in each column. Here we take $n = 5$ and the target network is built with 100 binary $\neq$ constraints on rows and columns. We use a bias of 600 contraints built with the language $\Gamma = \{\neq, =\}$.

Golomb Rulers. (prob006 in [Gent and Walsh, 1999]) The problem is to find a ruler where the distance between any two marks is different from that between any other two marks. The target network is encoded with n variables corresponding to the n marks, and constraints of varying arity. For our experiments, we selected the 12-marks ruler instance with a bias of 4356 constraints.

Sudoku. We used the Sudoku logic puzzle with 9×9 grid. The grid must be filled with numbers from 1 to 9 in such a way that all the rows, all the columns and the 9 non overlapping 3×3 squares contain the numbers 1 to 9. The target network has 81 variables with domains of size 9, and 810 binary $\neq$ constraints on rows, columns and squares. We use a bias of 6480 binary constraints built with the language $\Gamma = \{\neq, =\}$.

RLFA Problem. [Cabon *et al.*, 1999] The *Radio Link Frequency Assignment Problem* is to provide communication channels from limited spectral resources. Here we build a simplified version of RLFAP that consists in distributing all the frequencies available on the base stations of the network. We have five stations of five terminals (transmitters/receivers). The target network has 25 variables with domains of size 25, and 125 binary constraints We use a bias of 1800 binary constraints taken from a language of 6 arithmetic and distance constraints.

Langford's Numbers Problem. (prob024 in [Gent and Walsh, 1999]) The Langford's numbers problem is to arrange k sets of numbers 1 to n such that each appearance of the number m is m numbers on from the last. As an instance of this problem, we select $n = 5$ and $k = 5$. Thus, we have 5 different numbers, each one occurring 5 times , which leads to 25 variables. The target network contains 300 constraints with binary constraints of distance and clique of $\neq$ constraints. We use a bias of 3600 binary constraints taken from arithmetic and distance constraints.

Zebra problem. The Lewis Carroll Zebra problem is formulated using 25 variables, with 5 cliques of $\neq$ constraints and 14 additional constraints. We use a bias of 4450 unary and binary constraints taken from a language with 24 basic arithmetic and distance constraints.

Graceful Graphs. (prob053 in [Gent and Walsh, 1999]) A graph of m edges is graceful if all its nodes are assigned to a unique label from $\{0, \ldots, m\}$ and when each edge xy is assigned to $(label(x) - label(y))$. The edge labels are all different. As an instance, we take 9 nodes and 15 edges. The target network contains 156 constraints (binary and ternary constraints). The bias that we use contains 4600 constraints of basic arithmetic (binary and ternary) constraints.

Table 1: QUACQ vs. MULTIACQ

XP	Algorithm	$\|C_L\|$	$\bar{T}$	σ_T	$\#q$	$\bar{q}$	$\#q_c^+$	$\#q_c^-$	$\#q_p^+$
Murder	QUACQ	52	0.01	0.03	518	10.35	1	52	90
	MULTIACQ	52	**0.00**	0.06	**404**	**4.93**	2	3	248
Latin	QUACQ	**90**	0.02	0.03	1078	12.1	4	90	140
	MULTIACQ	100	**0.00**	0.05	**379**	**3.53**	8	1	200
Golomb 12	QUACQ	**323**	**0.53**	1.93	4258	6.45	1	323	61
	MULTIACQ	418	1.20	1.19	**1946**	**5.14**	0	43	567
Sudoku	QUACQ	**622**	**0.08**	1.43	10110	36.24	0	622	1107
	MULTIACQ	810	0.24	0.36	**3821**	**3.50**	10	1	2430

5.2 Results

Table 1 displays the performance of MULTIACQ and QUACQ. We report the size $|C_L|$ of the learned network, the average time $\bar{T}$ needed to generate a query in seconds, the standard deviation σ_T of $\bar{T}$, the total number of queries $\#q$, the average size $\bar{q}$ of all queries, the number of complete positive (resp. negative) queries $\#q_c^+$ (resp. $\#q_c^-$) and the number of partial positive queries $\#q_p^+$.

If we compare MULTIACQ to QUACQ, the main observation is that the use of **findAllScopes** to find all minimal scopes of a negative example reduces significantly the number of queries required for convergence.

Let us take the Murder problem. MULTIACQ exhibits a gain of 22% on the number of queries. The second observation that we can make is the fact that MULTIACQ reduces significantly the average size of the queries (52%), which is probably easier to answer by the user. Another point to stress is that MULTIACQ needs only 3 complete negative examples instead of 52 for QUACQ. This is not surprising as MULTIACQ is dedicated to return the entire set of minimal scopes of a negative example, where QUACQ focuses on one minimal scope each time we feed it with a negative example.

The same observations on the performance of MULTI-

ACQ comparing to QUACQ are true on the other problem instances:

- Number of queries reduction (i.e., gain of 65% on Latin Square, 55% on Golomb Rulers and 73% on Sudoku).
- The average size of the queries (i.e., 71% on Latin-Square, 20% on Golomb Rulers and 90% on Sudoku).
- Obviously, we need less complete negative example (i.e., only one instead of 90 on Latin Square, 43 instead of 323 on Golomb Rulers and only one instead of 622 on Sudoku).

We also observe that the number of constraints learned by MULTIACQ is always greater than or equal to the number of constraints learned using QUACQ ($|C_L|$ column). This can be explained by the fact that MULTIACQ can learn redundant constraints, which is not the case using QUACQ. Take three constraints c_1, c_2 and c_3 such that $c_1 \wedge c_2 \rightarrow c_3$. If we generate a negative example e_1 that violates the three constraints, MULTIACQ will return three minimal scopes corresponding to the three constraints. By contrast, QUACQ will return only one minimal scope, let us say the one of c_2. If in a second iteration QUACQ learns c_1, c_3 is automatically satisfied in any next iteration and will never be learned by QUACQ.

A last observation we can make on Table 1 is related to the average time needed to generate a query. If we take the instances of Golomb Rulers and Sudoku, we observe that MULTIACQ respectively needs twice more time ($1.20s$ instead of $0.53s$) and three times more time ($0.24s$ instead of $0.08s$) than QUACQ. On these two instances, generating a query is starting to become time-consuming.

Table 1 does not report the results of MACQ-CO because it performs exactly the same as the basic version of MULTIACQ. The reason is that the average time (and standard deviation) needed to generate a query is significantly below the value of the cutoff ($5s$ in our case).

Table 2 gives the results on four additional benchmark problems where MULTIACQ takes long in average to compute a query (more than 2 seconds for all four problems). Results are reported for QUACQ, MULTIACQ and MACQ-CO. The results displayed for MACQ-CO are the average over 10 runs because of the shuffle on the variables, which makes it non deterministic.

The observations made on Table 1 remain true on RLFA and Langford instances. The difference is in the average time needed to generate a query. Using MULTIACQ, the time increases significantly. For instance, on Langford, QUACQ takes 0.03 ± 0.13 seconds to generate a query whereas MULTIACQ takes 4.24 ± 23.03 seconds. On Zebra and Graceful Graphs instances, we observe the same behavior as on the previous two instances for the time to generate a query. However, as for the number of queries, QUACQ is better than MULTIACQ. This can be explained by the high number of partial positive queries asked by MULTIACQ (e.g., for Zebra we have $\#q_p^+ = 592$ with MULTIACQ against 255 with QUACQ). By definition, MULTIACQ has a trend to ask small queries and then, needs an important number of partial positive queries to reduce the bias and thus, to converge.

Comparing MULTIACQ and MACQ-CO in Table 2, we observe that MACQ-CO reduces drastically the time $\bar{T}$ needed to

Table 2: QUACQ vs. MULTIACQ vs. MACQ-CO

XP	Algorithm	$\|C_L\|$	$\bar{T}$	σ_T	$\#q$	$\bar{q}$	$\#q_c^+$	$\#q_c^-$	$\#q_p^+$
RLFA	QUACQ	125	**0.02**	**0.17**	1738	11.93	6	125	255
	MULTIACQ	125	2.32	17.38	**621**	**2.63**	3	1	250
	MACQ-CO	**124.7**	0.13	0.72	1094.7	6.51	5.4	10.1	363.2
Langford	QUACQ	**226**	**0.03**	**0.13**	2925	12.03	11	226	496
	MULTIACQ	300	4.24	23.03	**696**	**2.96**	17	1	296
	MACQ-CO	287	0.11	0.28	1867.5	8.95	12.0	30.0	571.0
Zebra	QUACQ	63	**0.03**	**0.10**	771	11.61	1	62	255
	MULTIACQ	**62**	2.08	13.57	823	**4.50**	1	3	592
	MACQ-CO	62.2	0.14	0.63	**748.0**	5.14	1.0	3.8	469.8
GGraphs	QUACQ	**129**	0.11	0.39	1539	10.57	7	129	330
	MULTIACQ	154	5.76	80.85	1706	**5.88**	0	2	1296
	MACQ-CO	155	**0.04**	**0.29**	**1440.4**	6.42	0.0	2.0	1146.5

generate a query as well as the standard deviation σ_T. However, MACQ-CO requires more queries than MULTIACQ to converge, asking almost twice more queries than MULTIACQ on RLFA and three times more on Langford. This is consistent with our analysis in Section 4. Surprisingly, MACQ-CO is better than MULTIACQ in number of queries on Zebra and Graceful Graphs. A possible explanation is the very low number of solutions of these problems that lead MULTIACQ to visit too many partial positive examples that MACQ-CO avoids thanks to the heuristics.

Let us now compare MACQ-CO to QUACQ. The main observation is that MACQ-CO wins in number of queries on all four problems of Table 2. For instance, on RLFA we note a gain of 37%. Concerning the time to generate queries, we observe that MACQ-CO is quite competitive thanks to the use of cutoffs and its different heuristics.

6 Conclusion

We have proposed a new approach to make constraint acquisition more efficient in practice by learning a maximum number of constraints from a negative example. The QUACQ constraint acquisition system focuses on the scope of one target constraint each time it processes a negative example. Our proposed MULTIACQ, with the `findAllScopes` function, reports all minimal scopes of violated constraints. We have also proposed several heuristics, leading to the MACQ-CO version, to obtain a good trade-off between time and number of queries. We have experimentally evaluated the benefit of our algorithm and heuristics on some benchmark problems. The results show that MULTIACQ dramatically improves the basic QUACQ in terms of number of queries. The queries generated are often much shorter than those asked by QUACQ, so are easier to handle for the user. Finally, as MULTIACQ can take too long to generate queries on some problems, MACQ-CO appears as a good compromise between QUACQ and MULTIACQ in terms of time-consumption and number of queries.

References

[Beldiceanu and Simonis, 2012] Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Principles and*

Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings, pages 141–157, 2012.

[Bessiere *et al.*, 2005] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, pages 23–34, 2005.

[Bessiere *et al.*, 2007] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 50–55, 2007.

[Bessiere *et al.*, 2013] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.

[Bessiere *et al.*, 2015] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence, In Press*, 2015.

[Cabon *et al.*, 1999] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

[Freuder and Wallace, 2002] Eugene C. Freuder and Richard J. Wallace. Suggestion strategies for constraint-based matchmaker agents. *International Journal on Artificial Intelligence Tools*, 11(1):3–18, 2002.

[Freuder, 1999] Eugene C. Freuder. Modeling: The final frontier. In *1st International Conference on the Practical Applications of Constraint Technologies and Logic Programming*, pages 15–21, London, UK, 1999.

[Gent and Walsh, 1999] Ian P. Gent and Toby Walsh. CSPLib : a benchmark library for constraints. http://www.csplib.org/, 1999.

[Liffiton and Sakallah, 2008] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[Shchekotykhin and Friedrich, 2009] Kostyantyn M. Shchekotykhin and Gerhard Friedrich. Argumentation based constraint acquisition. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 476–482, 2009.