Multiple Constraint Acquisition *

Robin Arcangioli, Nadjib Lazaar
LIRMM, University of Montpellier, France
{arcangioli,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 the state of the art.

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] 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] 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 learner 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. In the case of a negative example, 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 9000 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 called MUS (Minimal Unsatisfiability Subsets) in SAT [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 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_Y is defined as a pair where Y is a subset of variables of X, called *the constraint scope*, and c is a relation over D of arity $|Y|$. 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_S. An assignment on X is a *solution* of C if and only if it is not rejected by any constraint in C. $sol(C)$ represents the set of solutions of C.

In addition to the vocabulary, the learner owns a *language* Γ of 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. Formally speaking, $B = \{c_S \mid (S \subseteq X) \wedge (c \in \Gamma)\}$. A *concept* is a Boolean function over $D^X = \Pi_{x_i \in X} D(x_i)$, that is, a

*This work has been funded by the EU project ICON (FP7-284715) and the JCJC INS2I 2015 project APOSTROP.

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 No Scope*, noted MNS, which is similar to the notion of MUS (*Minimal Unsatisfiable Subset*) in SAT [Liffiton and Sakallah, 2008].

Definition 1 (Minimal No Scope)**.** *Given a negative example e, an MNS is a subset of variables $U \subseteq X$ s.t. $ASK(e_U) = no$ and $\forall x_i \in U : \ ASK(e_{U \setminus x_i}) = yes$.*

3 Multiple constraint acquisition approach

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 [Bessiere *et al.*, 2013] and MULTIACQ is the fact that QUACQ learns and focuses on one constraint each time we have a negative example, where 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 a 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 a positive example, the bias B will be reduced. However, when e is negative, `findAllScopes` will compute a set of MNS (i.e., the set $MNSes$). Each

Algorithme 1 : MULTIACQ- Multiple acquisition via partial queries.

```
1  C_L ← ∅
2  while true do
3      if sol(C_L) = ∅ 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          MNSes ← ∅
8          findAllScopes (e, X)
9          foreach Y ∈ MNSes do
10             c_Y ← findC(e, Y)
11             if c_Y = nil then return "collapse"
12             else C_L ← C_L ∪ {c_Y}
```

$Y \in MNSes$ represents the scope of a violated constraint that must be learned. The function `findC` is called for each computed MNS (line 10). It takes as parameter the negative example e and an MNS. For each MNS, it returns a constraint from C_T with the given MNS that rejects e. We do not give the code of `findC` function since it is implemented as it appear 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 according negative example. Otherwise, the constraint returned by `findC` is added to the learned network C_L (line 12).

3.2 Description of the function `findAllScopes`

Function 1 : `findAllScopes` $(e,\ Y)$: *boolean*

Output : The set $MNSes$ contains all MNS of e

```
1  if Y ∈ MNSes then return false
2  if κ_B(e_Y) = ∅ then return true
3  if ∄M ∈ MNSes : M ⊂ Y ∧ ASK(e_Y) = yes then
4      B ← B \ κ_B(e_Y)
5      return true
6  isMNS ← true
7  P ← subSets(Y, |Y| − 1)
8  foreach P_i ∈ P do
9      isMNS ← findAllScopes(P_i, e) ∧ isMNS
10 if isMNS then MNSes ← MNSes ∪ {Y}
11 return false
```

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). The function starts by checking if the subset Y is already reported as an MNS (line 1). If this occurs, we return $false$. 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 the fact that $ASK(e_Y) = yes$ and we return $true$ (line 2). As a third check (line 3), we verify if we have not reported a subset of Y as an MNS. 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 3, 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 $true$ (lines 4-5). If $ASK(e_Y) = no$, this means that there is an MNS in Y and in what follows, we try to report it. For that, we compute all subsets of Y of size $|Y| - 1$ (line 7) and we recall `findAllScopes` on each subset. If any sub-call to `findAllScopes` returns false, the boolean $isMNS$ will be set to $false$, which means that Y itself is not an MNS (line 11). Otherwise, when all the sub-calls of `findAllScopes` on Y subsets return $true$, this means that Y is an MNS (line 10). Notice that the returned boolean corresponds to the classification of the partial example e_Y (positive or negative).

3.3 Analysis

We analyze the soundness and completeness of `findAllScopes`. We also give a complexity study of MULTIACQ in terms 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., $ASK(e_{Y'}) = no$). □

Given a bias B and a target network C_T, we say that C_T is learnable if it is representable by B. Thus, $ASK(e_Y) = no \Rightarrow \kappa_B(e_Y) \neq \varnothing$, which is equivalent to $\kappa_B(e_Y) = \varnothing \Rightarrow ASK(e_Y) = yes$.

Theorem 1. *Given any bias B and any target network C_T representable by B, the `findAllScopes` function is sound and complete.*

Proof. (Sketch.)

- *Soundness :* Assume that we have a set of variables $X = \{x_1, \ldots, x_n\}$ and a complete assignment e_X. We show that any subset $Y = \{x_j, \ldots, x_k\}$ added to $MNSes$ by `findAllScopes` is an MNS. If Y is added to $MNSes$ (line 10), this means that $ASK(e_Y) = no$ and the $isMNS = true$. As $isMNS = true$, it means that all sub-calls of `findAllScopes` (line 9) on Y subsets of size $|Y| - 1$ return true. Hence, as we supposed the bias B to be expressive enough, $\forall Y' \subset Y$ of $|Y| - 1$ size, we have $ASK(e_{Y'}) = yes$ and knowing that $ASK(e_Y) = no$ imply that Y is an MNS (def. 1).
- *Completeness :* Assume that we have a set of variables $X = \{x_1, \ldots, x_n\}$:
  i) We have a complete negative assignment e_X ($ASK(e_X) = no$) with an MNS $M = \{x_j, \ldots, x_k\}$. The three conditions at lines 1, 2 and 3 are not satisfied ($ASK(e_X) = no$ and $\kappa_B(e_X) \neq \varnothing$). Then, `findAllScopes` is called on all the subsets of X of size $|X| - 1$. As $M \subseteq X$ is an MNS of e, $ASK(e_M) = no$ and so $\forall Y \subseteq X \backslash M : ASK(e_{M \cup Y}) = no$ (Lemma 2). Hence, any call of `findAllScopes` on a superset of M will behave exactly as on X. That is, by recurrence, there will be a call of `findAllScopes` on M. By assumption, M is an MNS and for any subset M' we have $ASK(e_{M'}) = yes$ (def.1). Thus, M is added to $MNSes$ by `findAllScopes` as an MNS at line 10.
  ii) We have a complete positive assignment e_X ($ASK(e_X) = yes$). Here, `findAllScopes` is called only once and only on X. That is, the third condition at line 3 is satisfied and `findAllScopes` returns true with $MNSes = \varnothing$.

□

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. i) We will show first that the maximal number of queries asked by `findAllScopes` is bounded above by $|C_T| \cdot (|X| - 1) + |B|$.

- Let us start with tablethe number of queries asked by `findAllScopes` and classified as negative by the user. Given an MNS $M \subset X$, we need to ask at most $|X| - 1$ queries that are classified as negative by the user. That is, knowing that `findAllScopes` uses a depth first search, we need to ask queries at each level from the root (i.e., $|X|$) to the node M. Since all visited sets contain M, the queries are classified as negative. Here, the worst case is when $|M| = 1$ where we will have $|X| - 1$ negative queries. Then, the use of Lemma 2 at line 1 and 3 ensures the fact that we will never ask a query on a superset of M. With the fact that the total number of MNS is bounded by $|C_T|$, we can say that the total number of negative queries asked by `findAllScopes` is bounded above by $|C_T| \cdot (|X| - 1)$.

- Let us now see the number of queries asked by `findAllScopes` and classified as positive by the user. This number 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 4) and therefore $e_{Y'}$ will be classified as positive without a query where $\kappa_B(e_{Y'})$ becomes empty (line 2). The worst case is when we remove, at each time, one constraint from B ($|\kappa_B(e_Y)| = 1$) and therefore `findAllScopes` asks $|B|$ queries classified as positive.

As has been shown, the maximal number of queries asked by the `findAllScopes` function is bounded above by $|C_T| \cdot (|X| - 1) + |B|$.

ii) `findC` function uses $O(|\Gamma|)$ queries to return a constraint from C_T [Bessiere *et al.*, 2013] and therefore $O(|C_T| \cdot |\Gamma|)$ queries to return all the constraints.

From i) and ii), the total number of queries asked by MULTIACQ to converge to the target network is bounded above by $|C_T| \cdot (|X| - 1) + |B| + |C_T| \cdot |\Gamma|$.

□

Before addressing our experimental evaluation, it is important to stress here that QUACQ converges on a constraint of the target network in a logarithmic number of queries [Bessiere *et al.*, 2013]. Whereas MULTIACQ converges on a set of constraints of the target network but on a linear number of queries in the size of the example. However, we believe that MULTIACQ can be more efficient than QUACQ in practice.

4 Experimental results

We made some experiments to evaluate the performance of our MULTIACQ algorithm and its `findAllScopes` function. Our tests were conducted on an Intel Core 2 Duo E4400 2x2.0GHz with 2.0GB of RAM. We first present the benchmark problems we used for our experiments.

N-queens. (prob054 in [Gent and Walsh, 1999]) The problem is to place n queens on a $n \times n$ chessboard so that none can attack each other. The target network is encoded with n variables corresponding to the n queens, and binary constraints. For our experiments, we selected the 4-queens instance with a C_T of 18 constraints (6 constraints $\neq$ on rows and 12 constraints $NotInDiagonal$, 6 for each direction) and a bias of 108 constraints.

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 8-marks ruler instance with a C_T of 385 constraints and a bias of 798 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 of the Sudoku 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.

	Algorithm	$\|C_L\|$	$\#q$	$\bar{q}$	$\#q_c^-$	$\#q_p^-$	T_{mns}	T_q
Queens	QUACQ	14	104	2.64	14	77	0.06	0.01
	MULTIACQ	12	**58**	**2.41**	**7**	**37**	**0.04**	**0.01**
Golomb	QUACQ	95	976	4.78	95	847	2.20	**0.21**
	MULTIACQ	146	**666**	**4.69**	**53**	**498**	**2.18**	0.54
Sudoku	QUACQ	622	9732	37.61	622	8003	3.15	**0.20**
	MULTIACQ	810	**3821**	**3.50**	**1**	**1380**	**1.77**	0.38

Table 1: MULTIACQ vs QUACQ

Table 1 displays a comparative performance of MULTIACQ and QUACQ. We report the size $|C_L|$ of the learned network, the total number of queries $\#q$, the average size $\bar{q}$ of all queries, the number of complete (resp. partial) negative queries $\#q_c^-$ (resp. $\#q_p^-$), and the average time T_{mns} (resp. T_q) needed to compute an MNS (resp. a query) in seconds.

If we compare our MULTIACQ to QUACQ, the main observation is that the use of `findAllScopes` to find all MNS of a negative example reduces significantly the number of queries required for convergence. For 4-queens instance and using MULTIACQ we reduce the number of queries by 44%. Here, MULTIACQ needs only 7 complete negative examples to learn 12 constraints that are equivalent to the target network. Whereas QUACQ needs twice as many negatives (14 examples) to converge to the target network.

Let us take Golomb rulers instance. Here the gain in terms of queries is of 31%. The same observation can be made on the number of complete (partial) negative examples. However, the time needed to compute a query is twice the time needed by QUACQ (0.54s instead of 0.20s).

The Sudoku instance is quite interesting where the problem is expressed on 81 variables and we have to learn a network equivalent to 810 $\neq$ constraints. Despite of the fact that our algorithm learns more constraints to converge than the basic QUACQ (810 instead of 622), MULTIACQ needs 60% less queries than QUACQ to converge. Here, MULTIACQ needs only one complete negative example to learn the whole network, where QUACQ needs 622 complete negative examples. Another result relates to the average size of queries. Here, $\bar{q} = 3.5$ is significantly smaller than the number of variables (i.e., 81 variables) and the average size for QUACQ (i.e., 37.61). This is an interesting result where short queries are probably easier to answer by the user. These good results in terms of queries are somewhat to the detriment of time where using MULTIACQ, we need more than $0.38s$ to compute a query (instead of $0.20s$ using QUACQ).

5 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. As a constraint acquisition system, QUACQ focuses on the scope of one target constraint each time we give it a negative example. We have proposed MULTIACQ with a `findAllScopes` function that aims to report all minimal scopes (MNS) of violated constraints. We have experimentally evaluated the benefit of our approach on some benchmark problems. The results show that i) MULTIACQ dramatically improves the basic QUACQ in terms of number of queries ii) the queries are often much shorter than membership queries, so are easier to handle for the user iii) and MULTIACQ could be time-consuming. An interesting direction would be to use a dichotomic search of MNS and the use of cutoffs during search in order to obtain a good tradeoff (#queries/time).

References

[Beldiceanu and Simonis, 2012] Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *CP*, 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 *ECML*, pages 23–34, 2005.

[Bessiere *et al.*, 2007] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In *IJCAI*, 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.

[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] E. 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. Invited Talk.

[Gent and Walsh, 1999] I.P. Gent and T. 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.