Structure-Driven Multiple Constraint Acquisition

Dimosthenis C. Tsouros, Kostas Stergiou, Christian Bessiere

Structure-driven Multiple Constraint Acquisition

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

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

Abstract. MQuAcq is an algorithm for active constraint acquisition that has been shown to outperform previous algorithms such as QuAcq and MultiAcq. In this paper, we exhibit two important drawbacks of MQuAcq. First, for each negative example, the number of recursive calls to the main procedure of MQuAcq can be non-linear, making it impractical for large problems. Second, MQuAcq, as well as QuAcq and MultiAcq, does not take into account the structure of the learned problem. We propose MQuAcq-2, a new algorithm based on MQuAcq that integrates solutions to both these problems. MQuAcq-2 exploits the structure of the learned problem by focusing the queries it generates to quasi-cliques of constraints. When dealing with a negative query, it only requires a linear number of iterations. MQuAcq-2 outperforms MQuAcq, especially on large problems.

1 Introduction

Constraint acquisition learns 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 [1, 2]. Constraint acquisition is an area where constraint programming meets machine learning, as the problem can be formulated as a concept learning task. In *passive* acquisition, examples of solutions and non-solutions are provided by the user. Based on these examples, the system learns a set of constraints that correctly classifies all the given examples [3–6, 1]. A major limitation of passive acquisition is the requirement, from the user's part, to provide diverse examples of solutions and non-solution to the system. In contrast, *active* or *interactive* acquisition systems interact with the user while acquiring the constraint network. This is a special case of query-directed learning, also known as "exact learning" [7, 8]. In such systems, the basic query is to ask the user to classify an example as solution or not solution. This "yes/no" type of question is called membership query [9], and this is the type of query that has received the most attention in active constraint acquisition [10, 1, 11]. The system can also ask the user to classify partial examples [12] or to provide a violated constraint when

a proposed example is considered as incorrect [13]. Other types of queries, e.g. recommendation and generalization ones, have also been considered [14, 15].

Quacq is a state-of-the-art interactive constraint acquisition algorithm that uses partial queries [12]. Given a negative example, QuAcq finds a constraint that is violated by repeatedly posting partial examples to the user. QuAcq needs a number of queries logarithmic in the size of the example to locate the scope of a violated constraint. Another relevant algorithm is MultiAcq [16]. This algorithm learns all the constraints that are violated by a negative example, but it needs a linear number of queries to learn each one. Recently, an algorithm called MQuAcq, that combines the strengths of QuAcq and MultiAcq and outperforms both of them, was proposed [17]. MQuAcq requires a logarithmic number of queries to locate the scope of each violated constraint, and discovers all the violated constraints from a negative example.

In this paper, we further enhance the efficiency of active constraint acquisition by identifying and addressing two important deficiencies of MQuAcq. We first show that there exist negative examples where the process of learning *all* the violated constraints can make $\Omega(|Y|^2)$ recursive calls, where $|Y|$ is the number of variables of the given example. This has important practical implications as MQuAcq becomes unacceptably slow when the size of the problems grows, and as a result it can be outperformed by its generally less efficient predecessor QuAcq. Another deficiency of MQuAcq (and also QuAcq and MultiAcq) is that although non-random problems usually display some structure/patterns in the way their constraints are interleaved, this is ignored by the acquisition process. By identifying and exploiting these patterns we could possibly speed up the process. Such patterns have for instance been exploited to detect types of variables suitable for generalization [18].

Aiming at addressing the above problems, we propose an algorithm called MQuAcq-2 that learns multiple constraints from a negative generated query, but not necessarily all of them as opposed to MQuAcq, and also exploits structure that may be present in the problem to better focus its queries. MQuAcq-2 blends together the following two ideas. First, MQuAcq-2 exploits the structure of the learned network to focus on some of the violated constraints instead of exhaustively searching in the generated example. In our implementation, we used the detection of quasi-cliques in the learned network and then focus on the missing constraints (i.e., the ones required to complete the cliques). Second, when trying to learn constraints from a negative example, the entire scope of a learned constraint is removed from the example as soon as the constraint is acquired. This means that the algorithm no longer guarantees to find all the violated constraints from a negative example, but nevertheless it may find several of them, and crucially, it only requires a linear number of iterations to achieve this. With the integration of these ideas we achieve the benefits of learning several constraints from each generated query and we also avoid the extensive search for scopes of MQuAcq. Experimental results with benchmark problems demonstrate that MQuAcq-2 offers significant improvements compared to MQuAcq, both in

terms of time and number of queries, especially on large problems. Importantly, the new algorithm outperforms MQuAcq even in the absence of structure.

The rest of this paper is organized as follows. In Section 2 the necessary background on interactive constraint acquisition is presented. Section 3 reviews the basics of multiple constraint acquisition with MQuAcq. Section 4 presents the proposed methods. An experimental evaluation is presented in Section 5. Section 6 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 a relation between the variables in var(c) that specifies which of their assignments are allowed. $|var(c)|$ is called the *arity* of the constraint. Two constraints $c1, c2$ are *overlapping* when $var(c1) \cap var(c2) \neq \emptyset$. 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 is rejected by 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 that is accepted by all the constraints in C is a solution to the problem. $sol(C)$ denotes 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 partial solution is not necessarily part of a complete solution. An *implied* constraint c in C is a constraint such that, if removed from the constraint network, the set of solutions remains the same.

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 classification 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

Algorithm 1 The MQuAcq Algorithm

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: **while** true **do**
3: $Scopes.clear()$;
4: **if** $sol(C_L) = \emptyset$ **then return** "collapse";
5: Generate e in D^X accepted by C_L and rejected by B;
6: **if** $e =$ nil **then return** "C_L converged";
7: **if** $\neg findAllCons(e, X, 0)$ **then return** "collapse";

a *membership query* $ASK(e)$. The answer to a membership query is positive if $f_T(e) = 1$ and negative otherwise. 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. Following the literature, we assume that all queries are answered correctly by the user.

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 $sol(C) = sol(C_L)$. If there does not exist a constraint network $C \subseteq B$ such that C agrees with E then the acquisition *collapses*. This happens when the target constraint network is not included in the bias, i.e. $C_T \nsubseteq B$.

3 Multiple Constraint Acquisition

We briefly describe the MQuAcq algorithm for multiple constraint acquisition [17], and we identify an important deficiency of this algorithm. MQuAcq (Algorithm 1) 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. It uses functions *FindScope-2* [17] and *FindC* [12].

MQuAcq starts by initializing the network C_L to the empty set (line 1) and then it enters the main loop (line 2). The array *Scopes*, which is initialized to be empty in line 3, is used within function *FindAllCons* as explained below. If C_L is unsatisfiable, the algorithm collapses (line 4). Otherwise, an assignment e is generated (line 5), satisfying C_L and violating at least one constraint in B. If such an example does not exist then the acquisition process has converged (line 6). Otherwise, it calls the function *FindAllCons* to find all the constraints that are violated by the example e and remove from B those that are surely not in C_T. If *findAllCons* return false then we have collapsed (line 7).

The recursive function *FindAllCons* (Algorithm 2) is used to find all the constraints from C_T that are violated by the generated negative example. It takes as parameters an example e, a set of variables Y, which defines the set of variables to search for the constraints, and an integer variable s, which is an identifier for the scopes. It returns false if collapse has occurred and true otherwise. *FindAllCons* adds to C_L all the constraints from C_T that are violated

Algorithm 2 findAllCons

Input: e, Y, s (e: the example, Y: set of variables, s: scopes identifier)
Output: *not_collapsed* : returns false if collapsed, true otherwise

```
1: function FindAllCons(e, Y, s)
2:     if κ_{B\C_L}(e_Y) = ∅ then return true;
3:     if s < |Scopes| then
4:         for x_i ∈ Scopes[s] do
5:             if ¬findAllCons(e, Y \ {x_i}, s + 1) then return false;
6:     else
7:         if ASK(e_Y) = "yes" then B ← B \ κ_B(e_Y);
8:         else
9:             scope ← FindScope-2(e, ∅, Y, false);
10:            c ← FindC(e, scope);
11:            if c = nil then return false;
12:            else C_L ← C_L ∪ {c}; B ← B \ {c};
13:            Scopes.push(scope);
14:            if ¬findAllCons(e, Y, s) then return false;
15:    return true;
```

by the example e in Y. It uses the array *Scopes* to store all the scopes of the constraints that have been found from the current generated query.

In any recursive call, *FindAllCons* starts by checking if there exists any violated constraint in B, not already in the learned network C_L. If not, it is implied that ASK(e_Y) = "yes" and the function returns true (line 2). After that, at line 3, *FindAllCons* checks if s is smaller than the size of *Scopes* (s acts an identifier of the scopes in which it has already branched). If $s < |Scopes|$, it means that the scope of a found violated constraint still exists in e_Y. Thus, *FindAllCons* is called recursively on each subset of Y created by removing one of the variables of the scope at position s of *Scopes* (lines 4-5), and increasing s by 1 to continue with the next scope in each recursive call.

If $s = |Scopes|$, branching has finished. The system asks the user to classify the partial example (line 7). If the answer is positive then the constraints in B that reject e are removed. Otherwise, function *FindScope-2* is called to find the scope of a violated constraint (line 9). *FindC* will then find a constraint from B with the discovered scope that is violated by e (lines 10-12). In lines 13-14, *FindAllCons* is called recursively to continue searching.

Functions *FindScope-2* and *FindC* are described in [17] and [12] respectively and are not included here due to space limitations.

MQuAcq models the query generation problem in line 5 of Algorithm 1 as an optimization problem that looks for a (partial) solution of C_L that maximizes the number of violated constraints in B. This heuristic is called max_B [17]. We will see in Section 5 that there are some cutoffs imposed.

Although MQuAcq offers improvements over its predecessors QuAcq and MultiAcq, both in terms of queries and cpu time, it still suffers from two weaknesses. We prove in Proposition 1 that the number of recursive calls to function

FindAllCons to learn all the violated constraints from a negative example can be non-linear in the number of variables of the example given. This non-linear number of calls can significantly hinder the cpu time performance of MQuAcq.

Proposition 1 *Given a negative example e_Y, MQuAcq may require a number of recursive calls to function* FindAllCons *in $\Omega(|Y|^2)$ to learn all the constraints of C_T that are violated by e_Y.*

Proof. Consider a constraint network and a negative example e_Y with $|Y| = 2 \cdot p$. Assume that $\kappa_{C_T}(e_Y) = \{c_{1,2}, c_{2,3}, \ldots, c_{p-1,p}\}$, where $c_{i,j}$ denotes a constraint with scope $var(c) = \{i, j\}$. Assume that $\kappa_{B \setminus C_T}(e_Y) = \{c_{1,p+1}, c_{2,p+2}, \ldots, c_{p,2p}\}$. With this pattern, we have $|\kappa_{C_T}(e_Y)| = p - 1$ and $|\kappa_{B \setminus C_T}(e_Y)| = p$.

We know that the branching takes place for each one of the variables in the scope of each learned constraint, meaning that two recursive calls are made to function *FindAllCons* at each branching point. In addition, we know that the depth of the tree of the recursive calls to *FindAllCons* can be up to $|\kappa_{C_T}(e_Y)| + 1 = p$, as it branches once for each scope included in *Scopes* at line 5, whose size in the end will be equal to $|\kappa_{C_T}(e_Y)|$. The maximum depth is reached as all the constraints of $\kappa_{C_T}(e_Y)$ are learned in the first branch.

Due to the structure of $\kappa_{C_T}(e_Y)$ and $\kappa_{B \setminus C_T}(e_Y)$, in each level we will have one more branching point than the previous. This happens because every constraint in $\kappa_{C_T}(e_Y)$ has in common the first variable of its scope with one constraint in $\kappa_{B \setminus C_T}(e_Y)$. Thus, after the first variable removal in each level, one constraint from $\kappa_{B \setminus C_T}(e_Y)$ will not be violated by e'_Y and the algorithm will have to follow the right branch to remove it, adding a new branch to each level. Also, we know that the constraints from $\kappa_{B \setminus C_T}(e_Y)$ will not be removed by the function *FindScope-2*, as a constraint from $\kappa_{C_T}(e_Y)$ will be found first and returned.

Now, let us prove that this results in a total number of nodes $N = 1 + (\frac{Y}{2} - 1) \cdot \frac{Y}{2}$. As in each level l we have one more branching point than the previous, we know that each level l of the tree will have 2 more nodes than the level $l - 1$, without counting the first level with the root node. This results in $N = 1 + 2 + 4 + ... + 2 \cdot (p-1) = 1 + 2 \cdot \sum_{k=1}^{p-1} k = 1 + 2 \cdot \frac{(p-1) \cdot p}{2} = 1 + (\frac{Y}{2} - 1) \cdot \frac{Y}{2}$. Therefore, MQuAcq requires a number of recursive calls to function *FindAllCons* in $\Omega(|Y|^2)$ to learn all the constraints of C_T that are violated by e_Y. □

Another weakness of MQuAcq is that the extensive branching it makes to find all the constraints violated by a negative example yields a lot of (small) partial positive queries. It is better when positive queries violate a large number of constraints from B because we want to prune the bias as much as possible. It would thus be better to focus on asking small partial queries on specific constraints that have greater probability to be included in C_T instead of focusing on all the violated constraints.

4 MQuAcq-2

In this section we propose MQuAcq-2, a new algorithm that acquires multiple constraints from each negative generated example, but not necessarily all of them

Algorithm 3 MQuAcq-2: Quick Acquisition learning multiple scopes

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 ← ∅;
2: while true do
3:     if sol(C_L) = ∅ then return "collapse";
4:     Generate e in D^Y accepted by C_L and rejected by B;
5:     if e = nil then return "C_L converged";
6:     Y' ← Y;
7:     do
8:         if ASK(e_Y') = "yes" then B ← B \ κ_B(e_Y');
9:         else
10:            Scope ← FindScope-2(e_Y', ∅, Y', false);
11:            c ← FindC(e_Y', Scope);
12:            if c = nil then return "collapse";
13:            else C_L ← C_L ∪ {c}; B ← B \ {c};
14:            NScopes ← Scope;
15:            NScopes ← NScopes ∪ analyze&Learn(e_Y);
16:            for Scope ∈ NScopes do
17:                Y' ← Y' \ Scope;
18:    while κ_B(e_Y') ≠ ∅
```

as opposed to MQuAcq. The intuition is to focus on constraints that are more likely to be included in C_T instead of exhaustively searching in the generated example, and thus to decrease the run time as well as the number of queries needed to learn the target network.

4.1 Algorithm description

MQuAcq-2 (Algorithm 3) starts with an empty C_L and a bias B containing constraints that can be built using the constraint language Γ on the vocabulary X, D. MQuAcq-2 returns the learned constraint network C_L, equivalent to the target network C_T. MQuAcq-2 iteratively generates examples and posts them as queries to the user. If the answer from the user is negative (i.e., at least one constraint from C_T is violated from the query posted), it tries to learn multiple constraints. MQuAcq-2 achieves that with the two following steps:

- It exploits the structure of the learned network to focus on specific violated constraints from B,
- In case no more constraints can be learned by exploiting the structure of C_L, it tries to find some non-overlapping constraints of C_T. As we explain below, this allows us to alleviate the high run time that MQuAcq incurs when searching for all the violated constraints from each negative example.

MQuAcq-2 generates a (partial) example e satisfying C_L and rejecting at least one constraint from B (line 4). If it has not converged or collapsed, it tries

to acquire multiple constraints of C_T violating e. At first it posts the example as a query to the user (line 8). In the case the answer of the user is positive then it removes from the bias the set $\kappa_B(e_{Y'})$ of all the constraints from B that reject $e_{Y'}$. If the answer is negative it tries to find a constraint by using the functions *FindScope-2* and *FindC* like in MQuAcq (lines 10-13).

The first novelty is that after a constraint is added to C_L, the system calls the function *analyze&Learn* (line 15) to analyze the structure of C_L and to ask partial queries on scopes of constraints violated by the initial example that seem to fit in that structure. The above steps are done repeatedly, removing from Y' the variables of the scope of each violated constraint it has already learned (lines 16-17) that are stored in the set $NScopes$. When no more constraint from B can be acquired by analyzing the structure of C_L, MQuAcq-2 tries to learn multiple non-overlapping constraints (lines 10-13). The iterative process ends when the example $e_{Y'}$ does not contain any violated constraint from the bias (line 18).

The second novelty is that MQuAcq-2 removes the entire scope of the acquired constraints at lines 16-17 to avoid the exhaustive branching that MQuAcq does by removing one variable in each call to *findAllCons*. We lose the guarantee to learn all the constraints violated by a generated example, as in MQuAcq, but on the other hand we achieve better performance in practice. The fact that MQuAcq-2 does not learn all the violated constraints from a generated example does not mean that it will not learn the entire network. The "missed" constraints will be learned at another example.

4.2 Using the structure of the problem to learn constraints

Function *analyze&Learn* (Algorithm 4) is used to analyze the structure of the learned network and then to focus on some of the violated constraints of the bias that fit in the structure, and thus are likely to be part of C_T. The type of structure that we have investigated so far is that of tightly connected groups of variables that form *quasi-cliques* that are hopefully extendable to complete cliques. Quasi-cliques are subgraphs with an edge density exceeding a threshold parameter [19, 20]. More formally, given a threshold $\gamma \in [0, 1]$, a (sub)graph $G = (V, E)$, with V the set of vertices and E the set of edges, is γ-dense if $|E(G)| \geq \gamma \cdot \frac{|V|*|V|-1}{2}$. If in addition G is connected, it is a quasi-clique. We used quasi-clique detection to focus subsequent queries on the constraints that are still in B and could be included in C_L to possibly complete a detected quasi-clique to form a clique.

The algorithm we use for finding quasi-cliques is similar to the one used in [18]. It is based on the well-known Bron-Kerbosch's [21] algorithm for finding maximal cliques in a graph. It is a recursive backtracking function that searches for maximal quasi-cliques in the graph of constraints of C_L. We consider any type of constraint as an edge, as opposed to the algorithm used in [18], which considers only constraints with same relation.

Function *analyze&Learn* takes only the generated negative example e_Y as a parameter. It returns the set $NScopes$, which contains the scopes of the constraints learned. Function *analyze&Learn* starts by initializing the set $NScopes$

Algorithm 4 *analyze&Learn*

Input: e_Y : the example
Output: $NScopes$: the set of scopes of the constraints that have been learned

```
1: function analyze&Learn(e_Y)
2:     NScopes ← ∅;
3:     QCliques ← FindQCliques(X, ∅, ∅);
4:     C_Q ← {c | c ∈ κ_B(e_Y) \ C_L ∧ ∃q ∈ QCliques | var(c) ⊆ q};
5:     PScopes ← {Y' | c ∈ C_Q ∧ var(c) = Y'};
6:     for Y' ∈ PScopes do
7:         if ASK(e_Y') = "yes" then B ← B \ κ_B(e_Y');
8:         else
9:             Scope ← FindScope-2(e_Y', ∅, Y', false);
10:            c ← FindC(e_Y', Scope);
11:            if c = nil then return "collapse";
12:            else C_L ← C_L ∪ {c}; B ← B \ {c};
13:            NScopes ← NScopes ∪ Scope;
14:    if NScopes ≠ ∅ then
15:        NScopes ← NScopes ∪ analyze&Learn(e_Y);
16:    return NScopes;
```

to the empty set (line 2). At line 3 it finds quasi-cliques in C_L via the function *FindQCliques*. A cutoff is imposed to this function, returning all the quasi-cliques found within this time limit. This is done to avoid the exponential time-complexity of finding all the quasi-cliques. $QCliques$ contains sets of variables where each set forms a quasi-clique in the graph of the already learned network. Using the quasi-cliques found, we fill the set C_Q with the predicted constraints, that is, the constraints of B that have not been already learned (i.e., not in C_L), have a scope that is included in a quasi-clique ($\exists q \in QCliques \mid var(c) \subseteq q$), and are violated by e_Y (are included in $\kappa_B(e_Y)$) (line 4). We only consider constraints violated by the current example to avoid the overhead of generating new examples to learn them. Next, we fill the set $PScopes$ with the scopes of these constraints (line 5). For each scope in $PScopes$ (line 6), the system posts a partial query to the user, focusing on the variables of the scope (line 7). If the answer is positive then the constraints that reject the example are removed from B. Otherwise, function *FindScope-2* is called to find the scope of the violated constraint (line 9). This is done to ensure that the violated constraint the user has in mind is not in a subscope. (As we use only binary problems in the experiments, that means it looks for unary constraints.) Next, *FindC* will select a constraint from B with the discovered scope that is violated by $e_{Y'}$ (line 10). If no constraint is found then the algorithm collapses (line 11). Otherwise, the constraint returned by *FindC* is added to C_L (line 12) and its scope is added to the set of found scopes (line 13). Finally, if any constraint is found (line 14), *analyze&Learn*(e_Y) is recursively called to check if new quasi-cliques have been formed (line 15). The scopes of the constraints learned by this call to *analyze&Learn* are added to $NScopes$, and $NScopes$ is returned (line 16).

Table 1. Execution of MQuAcq-2 at Example 1

repetition	Y	e_Y	*ASK*	Constraint acquired
1	$x_1 - x_8$	$\{1, 2, 2, 2, 3, 3, 4, 4\}$	"no"	$\neq_{23}$
1.1	x_2, x_4	$\{-, 2, -, 2, -, -, -, -\}$	"no"	$\neq_{24}$
1.2	x_3, x_4	$\{-, -, 2, 2, -, -, -, -\}$	"no"	$\neq_{34}$
3	$x_1, x_5 - x_8$	$\{1, -, -, -, 3, 3, 4, 4\}$	"no"	$\neq_{56}$
3	x_1, x_7, x_8	$\{1, -, -, -, -, -, 4, 4\}$	"no"	$\neq_{78}$
3	x_1	$\{1, -, -, -, -, -, -, -\}$	-	-

We chose quasi-clique detection for the analysis of the structure of the network because cliques are a common structure in constraint networks. Function *analyze&Learn* could also look for other types of structures by simply replacing the search for quasi-cliques by any other type of structure (such as [22–25]). The problem of predicting which constraints of B are more likely to be included in C_T can also be seen as a link prediction problem. Any method which deals with this problem can be exploited (e.g., [26–28, 14]).

4.3 Example and analysis of MQuAcq-2

Let us now illustrate the behavior of MQuAcq-2 on a simple example.

Example 1. The vocabulary 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 $\{\neq_{12}, \neq_{13}, \neq_{14}, \neq_{23}, \neq_{24}, \neq_{34}, \neq_{56}, \neq_{78}\}$ and $B = \{\neq_{ij} \mid 1 <= i < 8 \wedge i < j <= 8\}$. Assume that the learned network so far is $C_L = \{\neq_{12}, \neq_{13}, \neq_{14}\}$ and $\gamma = 0.6$ in MQuAcq-2. Also, assume that the current example processed (generated at line 4 of MQuAcq-2) is $e = \{1, 2, 2, 2, 3, 3, 4, 4\}$.

The execution of MQuAcq-2 is presented in Table 1. The first column shows the iteration of the algorithm. In the second column the variables that are considered in Y are given, while in the third column the example e_Y is displayed. Column *ASK* shows the answer of the user to the query posted, if one is posted, − otherwise. Finally, the constraint learned is presented.

MQuAcq-2 will post the example to the user, and after receiving a negative answer it will find the constraint $\neq_{23}$ using functions *FindScope* and *FindC*. After learning this constraint, it detects a possible clique among variables x_1, x_2, x_3, x_4 as shown in Figure 1. So the algorithm will now focus on constraints $\neq_{24}, \neq_{34}$ that are violated by e, and will learn them via the function *analyze&Learn* (iterations 1.1,1.2). As no other quasi-clique (with $\gamma = 0.6$) has been detected, the algorithm continues by removing the entire scope of the constraints learned from Y. In the next iteration, after the negative classification by the user, constraint $\neq_{56}$ will be learned and variables x_5, x_6 will be removed. In the same way, the constraint $\neq_{78}$ will be acquired next. As no other constraint from B rejects the example e_Y after removing the variables from the last constraint learned, MQuAcq-2 will return to the query generation step.

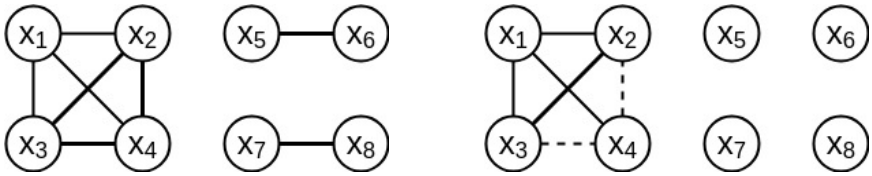

Fig. 1. (a) The target network of the problem. (b) The learned network so far and the predicted constraints

We now study the complexity of MQuAcq-2 in terms of the number of queries it needs to converge to the target network and in terms of the repetitions required to learn multiple violated constraints from a negative example.

Proposition 2 *Given a bias B built from a language Γ, with bounded arity constraints, and a target network C_T, MQuAcq-2 uses $O(|C_T| \cdot (\log |X| + |\Gamma|))$ queries to find the target network or to collapse and $O(|B|)$ queries to prove convergence.*

Proof. MQuAcq-2 learns each constraint from a negative example via the functions *FindScope* and *FindC* at lines 10-13 or via the function *analyze&Learn* using the same functions. We know that *FindScope* needs at most $|S| \cdot \log |Y|$ queries to locate a scope of a constraint from C_T, with $|S|$ being the arity of the scope and $|Y|$ the size of the example given to the function [12]. Since $Y \subseteq X$, *FindScope* needs in the worst case $|S| \cdot \log |X|$ queries to find a scope. In addition, we know that *FindC* needs at most $|\Gamma|$ queries to find a constraint from C_T in the scope it takes as parameter, if one exists [12]. In the case that none exists, the system collapses with the same bound. As a result, the number of queries necessary to find a constraint using the functions *FindScope* and *FindC* is $O(|S| \cdot \log |X| + |\Gamma|)$. Thus, the number of queries required for finding all the constraints in C_T or collapsing is at most $C_T \cdot (|S| \cdot \log |X| + |\Gamma|)$ which is $O(C_T \cdot (\log |X| + |\Gamma|))$ because $|S|$ is bounded. Concerning the convergence problem, it is proved when B is empty or contains only implied constraints. Constraints are removed from B when the answer from the user is "yes" in a query in the above cases. In the worst case, in which each positive query rejects only one constraint from B, it leads to at least one constraint removal in each query. This is because the example generated at line 4 of MQuAcq-2 violates at least one constraint from B and *analyze&Learn* does not ask a query $e_{Y'}$ when $\kappa_B(e_{Y'})$ is empty (lines 4-5 in *analyze&Learn*). This gives a total of $O(|B|)$ queries to converge. □

Therefore, MQuAcq-2 has a logarithmic complexity in terms of the number of queries needed to find the scope of a violated constraint, the same as QuAcq and MQuAcq. Now we turn out attention to the process of learning multiple constraints from a negative example e_Y.

Proposition 3 *The number of iterations needed by MQuAcq-2 to acquire multiple constraints from a given negative example e_Y is bounded above by $O(|Y|)$.*

Proof. Given a negative example e_Y, MQuAcq-2 enters into a do-while loop at line 7 to acquire multiple constraints of C_T. After the acquisition of each constraint, the entire scope (i.e., all the variables of the scope) of the constraint(s) acquired is removed from Y. Thus, assuming unary constraints are included in the target network of the problem, in the worst case only one variable will be removed from Y in each repetition. As a result, the worst case number of iterations made by MQuAcq-2 to acquire multiple constraints of C_T, is equal to $|Y|$. □

Therefore, given a negative example, MQuAcq-2 learns multiple constraints of the target network in a complexity lower than MQuAcq. As we will see in the experiments, it significantly improves its time performance.

5 Experimental Evaluation

To evaluate our proposed algorithm, we ran experiments comparing MQuAcq-2 against MQuAcq. We also ran QuAcq as a reference point. Some more details about our experiments:

- All the experiments were conducted on a system carrying an Intel(R) Xeon(R) CPU E5-2667, 2.9 GHz clock speed, with 8 Gb of RAM.
- The max_B heuristic [17] was used for the generation of the queries by all algorithms. max_B focuses on examples violating as many constraints as possible from B without necessarily building a complete variable assignment. *bdeg* was used for variable ordering, that is the variable with the most constraints in B is chosen. Random value ordering was used.
- For all the algorithms we set some cutoffs in the query generation step. The best (according to max_B) example found within 1 second is returned, even if not proved optimal. If after 5 seconds, not a single example is found, the system takes one by one each constraint c in B and tries to solve $C_L \cup \{\neg c\}$ with a additional cutoff of 5 seconds.
- We do not check for collapse before the generation of the queries, as it can be very time consuming, especially in large problems, with a lot of variables and a large C_T. We assume that the problem the user has in mind is solvable, the user's answers are correct and C_T is representable by B.
- In the function *FindQCliques*, γ was set to 0.8.
- As finding all the quasi-cliques is an NP-hard problem, we have added a cutoff of 1 second in the function *FindQCliques*, which then returns all the quasi-cliques found within this time limit.

We used the following benchmarks in our study:

Sudoku. The Sudoku puzzle is a $n^2 \times n^2$ grid. It must be completed in such a way that all the rows, all the columns and the n^2 non-overlapping $n \times n$ squares contain the numbers 1 to n. We use two variations of the problem with $n = 3$ and $n = 4$. This gives a *vocabulary* having 81 (256 respectively) variables and domains of size 9 (16 respectively). The target networks for the two problems are

of size 810 and 4,992. The bias was initialized with 12,960 and 130,560 binary constraints respectively, using the language $\Gamma = \{=, \neq, >, <\}$.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. That means that the domain is of size n. We use two variations of the problem. The first one with 100 variables (i.e., $n = 10$) having a target network of 900 binary $\neq$ constraints on rows and columns, and the second with 225 variables (i.e., $n = 15$) and a target nework of size 3,150. The language used was $\Gamma = \{=, \neq, >, <\}$, resulting in a bias of 19,800 binary constraints in the first problem and 100,800 constraints in the second.

Random. We used a problem with 100 variables and domains of size 5. We generated a random target network with 495 $\neq$ constraints. The bias was initialized with 19,800 constraints, using the language $\Gamma = \{=, \neq, >, <\}$.

Radio Link Frequency Assignment Problem. The RLFAP is the problem of providing communication channels from limited spectral resources [29]. We use a simplified version which consists of 50 variables with domains of size 40. The target network contains 125 binary distance constraints. We built the bias using a language of 2 basic 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.

AllDiff We used a problem with 50 variables and domains of size 50 with the condition that all variables must take different values. Thus, the target network contains a clique of 1,225 binary $\neq$ constraints. The bias was initialized with 4,900 constraints, using the language $\Gamma = \{=, \neq, >, <\}$.

To compare all the algorithms on the same simple scenario, all our experiments take the extreme case where we start from an empty constraint network. Even for the best algorithms, this scenario leads to an overall number of queries that can be considered as too large when the user is a human. In real applications, the user often has some background knowledge that give a frame of basic constraints. If not, the user may take other methods, such as ModelSeeker [6], to extract constraints from the structure of solutions of the problem. In this case, the interactive acquisition algorithm is only used to finalize the model.

5.1 Results

We measure the size of the learned network C_L, the average waiting time $\bar{T}$ (in seconds) for the user, the total number of queries $\#q$, the average size $\bar{q}$ of all queries, the number of complete queries $\#q_c$, and the total time needed to converge T_{total}. We present results of MQuAcq, QuAcq, MQuAcq-2 without *analyze&Learn* (denoted by MQuAcq-2 w/o $A\&L$) and full MQuAcq-2. Each algorithm was run 5 times and the means are presented in Table 2.

Looking at the time performance of MQuAcq-2 without *analyze&Learn* we see that in all problems except AllDiff (which is relatively small) it decreases the total time of the acquisition process compared to MQuAcq. In the larger problems the decrease in total time is quite significant: MQuAcq is 6.3 times slower on Latin 15x15 and 10.7 times slower on Sudoku 16x16. This confirms our intuition and complexity analysis. In terms of number of queries, we also have an

Table 2. Results of MQuAcq-2

Benchmark	Algorithm	$\|C_L\|$	$\#q$	$\bar{q}$	$\#q_c$	$\bar{T}$	T_{total}
Latin 15x15	MQuAcq	3150	37176	101	27	0.18	6730.88
	QuAcq	3150	31426	117	2345	0.18	5808.12
	MQuAcq-2 w/o $A\&L$	3150	28364	70	42	0.04	1069.80
	MQuAcq-2	3150	25517	61	49	0.11	2757.66
Latin 10x10	MQuAcq	900	8457	46	16	0.02	155.84
	QuAcq	900	7997	53	784	0.13	1026.86
	MQuAcq-2 w/o $A\&L$	900	7265	33	31	0.02	139.58
	MQuAcq-2	900	6133	19	33	0.03	168.88
Sudoku 16x16	MQuAcq	4992	59953	90	18	0.49	29612.70
	QuAcq	4992	42648	52	120	0.33	14092.20
	MQuAcq-2 w/o $A\&L$	4992	43958	61	36	0.06	2771.94
	MQuAcq-2	4992	40905	51	35	0.15	6098.34
Sudoku 9x9	MQuAcq	810	6964	32	14	0.02	124.07
	QuAcq	810	6478	38	518	0.14	880.96
	MQuAcq-2 w/o $A\&L$	810	6136	24	31	0.02	94.12
	MQuAcq-2	810	4912	15	30	0.04	191.63
Random	MQuAcq	495	5959	45	10	0.03	168.52
	QuAcq	495	5500	53	472	0.10	570.96
	MQuAcq-2 w/o $A\&L$	495	4930	34	16	0.02	79.35
	MQuAcq-2	495	4962	34	16	0.02	79.05
RLFAP	MQuAcq	122	1520	22	26	0.14	222.72
	QuAcq	106	1168	25	71	0.23	274.01
	MQuAcq-2 w/o $A\&L$	124	1102	21	27	0.19	218.88
	MQuAcq-2	124	1113	21	23	0.19	237.27
AllDiff	MQuAcq	1225	3912	26	24	0.03	107.82
	QuAcq	1225	5082	34	1116	0.25	1280.27
	MQuAcq-2 w/o $A\&L$	1225	4153	23	51	0.03	135.24
	MQuAcq-2	1225	2774	14	37	0.12	345.65

important decrease compared to MQuAcq in all the problems except AllDiff (up to 26.7% in Sudoku 16x16). This decrease in the number of queries is mainly due to the fact that avoiding the extensive branching that MQuAcq makes, MQuAcq-2 also avoids posting a lot of small positive partial queries. As a result, the pruning of B is achieved with fewer queries. Another observation is that although the number of *complete* queries posted to the user is increased in all problems, the average size of the queries is reduced. This reduced size of the queries is mainly due to the smaller negative queries asked by MQuAcq-2 because, as opposed to MQuAcq, MQuAcq-2 removes the entire scope of the previous constraint learned (line 17) before proceeding with the search for constraints.

When MQuAcq-2 takes into account the structure of the problem by using *analyze&Learn*, we observe that the total time is larger than without using *analyze&Learn*, but it is still significantly faster than MQuAcq on the large problems. The decrease in time is 59% on Latin 15x15, 79.4% on Sudoku 16x16 and 46.9% on Random. In contrast, the run time is increased by 8.4% on Latin 10X10, 54.5% on Sudoku 9x9, 6.5% on RLFAP and 220.6% on AllDiff. The increase in

time in these smaller problems is due to the overhead of *analyze&Learn*. The largest increase is in the AllDiff problem. This is not surprising as the AllDiff problem consists of a single clique of constraints. Focusing on the number of queries, by using *analyze&Learn* to exploit the structure of the learned network, MQuAcq-2 offers significant improvements compared to MQuAcq on all problems. The number of queries is decreased by 31.4% on Latin 15x15, 27.5% on Latin 10x10, 30.4% on Sudoku 16x16, 29.5% on Sudoku 9x9, 17.3% on Random 26.8% on RLFAP and 29.1% on AllDifferent. Interestingly, it seems that the larger the target network, the bigger the gain. We also observe that the more structured the problem is, the more queries full MQuAcq-2 saves compared to MQuAcq-2 without *analyze&Learn*. The average size of the queries is even smaller than with MQuAcq-2 without *analyze&Learn*. This is because *analyze&Learn* focuses on small scopes in the most promising parts of the network, avoiding the large negative queries that FindScope-2 would have asked to find these scopes.

Looking at the performance of full MQuAcq-2 on the Random problem, we see that both in terms of time and number of queries it dominates MQuAcq. Random is a pure random problem without any kind of structure. Thus, although MQuAcq-2 tries to exploit the structure of the problem to enhance the acquisition process, it performs quite well even in the absence of structure. The results with and without *analyze&Learn* are quite similar, confirming that in the absence of structure the overhead of *analyze&Learn* is relatively small. The same occurs on the RLFAP problem, which does not contain any cliques.

Finally, regarding QuAcq, we observe that this algorithm is better than MQuAcq both in run times and in number of queries on the largest problems that had not been considered before. However, it is inferior to MQuAcq-2 both in terms of time and number of queries on all problems.

6 Conclusion

Although the MQuAcq algorithm for constraint acquisition was shown to outperform previous algorithms such as QuAcq and MultiAcq, we have demonstrated that it suffers from two important drawbacks. First, the process of learning a maximum number of constraints from each negative generated example is time-consuming. This makes MQuAcq inefficient for large problems. Second, MQuAcq, as well as QuAcq and MultiAcq, does not take into account the structure revealed as constraints are learned. We have proposed a new algorithm, named MQuAcq-2, that integrates solutions to both of these problems. MQuAcq-2 exploits the structure of the learned problem by focusing the queries it generates to quasi-cliques of constraints that are being revealed. In addition, it alleviates the high cpu time requirements of MQuAcq by acquiring multiple constraints from each generated negative example, but not trying to learn all of them. Experiments with benchmark problems demonstrate that MQuAcq-2 outperforms MQuAcq both in terms of the number of queries and in the total time of the acquisition process, especially on large problems.

References

1. Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence **244** (2017) 315–342
2. 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. Springer (2016) 51–76
3. Bessiere, C., Coletta, R., Freuder, E.C., O'Sullivan, B.: Leveraging the learning power of examples in automated constraint acquisition. In: International Conference on Principles and Practice of Constraint Programming, Springer (2004) 123–137
4. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: A sat-based version space algorithm for acquiring constraint satisfaction problems. In: European Conference on Machine Learning, Springer (2005) 23–34
5. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI). Volume 1., IEEE (2010) 45–52
6. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: International Conference on Principles and Practice of Constraint Programming, Springer (2012) 141–157
7. Bshouty, N.: Exact learning boolean functions via the monotone theory. Information and Computation **123**(1) (1995) 146 – 153
8. Bshouty, N.H.: Exact learning from an honest teacher that answers membership queries. Theoretical Computer Science **733** (2018) 4–43
9. Angluin, D.: Queries and concept learning. Machine learning **2**(4) (1988) 319–342
10. Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M., et al.: Query-driven constraint acquisition. In: International Joint Conference on Artificial Intelligence (IJCAI). Volume 7. (2007) 50–55
11. Shchekotykhin, K., Friedrich, G.: Argumentation based constraint acquisition. In: Ninth IEEE International Conference on Data Mining, IEEE (2009) 476–482
12. 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: International Joint Conference on Artificial Intelligence (IJCAI). Volume 13. (2013) 475–481
13. Freuder, E.C., Wallace, R.J.: Suggestion strategies for constraint-based matchmaker agents. In: International Conference on Principles and Practice of Constraint Programming, Springer (1998) 192–204
14. Daoudi, A., Mechqrane, Y., Bessiere, C., Lazaar, N., Bouyakhf, E.H.: Constraint acquisition using recommendation queries. In: International Joint Conference on Artificial Intelligence (IJCAI). (2016) 720–726
15. Bessiere, C., Coletta, R., Daoudi, A., Lazaar, N., Mechqrane, Y., Bouyakhf, E.H.: Boosting constraint acquisition via generalization queries. In: European Conference on Artificial Intelligence (ECAI). (2014) 99–104
16. Arcangioli, R., Bessiere, C., Lazaar, N.: Multiple constraint aquisition. In: International Joint Conference on Artificial Intelligence (IJCAI). (2016) 698–704
17. 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)
18. Daoudi, A., Lazaar, N., Mechqrane, Y., Bessiere, C., Bouyakhf, E.H.: Detecting types of variables for generalization in constraint acquisition. In: 2015 IEEE 27th

International Conference on Tools with Artificial Intelligence (ICTAI), IEEE (2015) 413–420
19. Pardalos, J., Resende, M.: On maximum clique problems in very large graphs. DIMACS series **50** (1999) 119–130
20. Abello, J., Resende, M.G., Sudarsky, S.: Massive quasi-clique detection. In: Latin American symposium on theoretical informatics, Springer (2002) 598–612
21. Bron, C., Kerbosch, J.: Algorithm 457: Finding all cliques of an undirected graph. Commun. ACM **16**(9) (1973) 575–577
22. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: Proceedings of the 31st International Conference on Very large data bases, VLDB Endowment (2005) 721–732
23. Girvan, M., Newman, M.E.: Community structure in social and biological networks. Proceedings of the national academy of sciences **99**(12) (2002) 7821–7826
24. Newman, M.E.: Modularity and community structure in networks. Proceedings of the national academy of sciences **103**(23) (2006) 8577–8582
25. Papadopoulos, S., Kompatsiaris, Y., Vakali, A., Spyridonos, P.: Community detection in social media. Data Mining and Knowledge Discovery **24**(3) (2012) 515–554
26. Adamic, L.A., Adar, E.: Friends and neighbors on the web. Social networks **25**(3) (2003) 211–230
27. Leicht, E.A., Holme, P., Newman, M.E.: Vertex similarity in networks. Physical Review E **73**(2) (2006) 026120
28. Liben-Nowell, D., Kleinberg, J.: The link-prediction problem for social networks. Journal of the American society for information science and technology **58**(7) (2007) 1019–1031
29. Cabon, B., De Givry, S., Lobjois, L., Schiex, T., Warners, J.P.: Radio link frequency assignment. Constraints **4**(1) (1999) 79–89