Detecting Types of Variables for Generalization in Constraint Acquisition

Abderrazak Daoudi[1,2], Nadjib Lazaar[1], Younes Mechqrane, Christian Bessiere[1], El Houssine Bouyakhf[2]
[1]*CNRS, University of Montpellier*
Montpellier, France
Email: {daoudi, lazaar, bessiere}@lirmm.fr, ymechqrane@gmail.com
[2]*LIMIARF/FSR, Mohammed V University of Rabat*
Rabat, Morocco
Email: bouyakhf@fsr.ac.ma

***Abstract*—During the last decade several constraint acquisition systems have been proposed for assisting non-expert users in building constraint programming models. GENACQ is an algorithm based on generalization queries that can be plugged into many constraint acquisition systems. However, generalization queries require the aggregation of variables into types which is not always a simple task for non-expert users. In this paper, we propose a new algorithm that is able to learn types during the constraint acquisition process. The idea is to infer potential types by analyzing the structure of the current constraint network and to use the extracted types to ask generalization queries. Our approach gives good results although no knowledge on the types is provided.**

***Keywords*-Constraint Acquisition, Generalization Queries, Graph Community Detection**

I. INTRODUCTION

Constraint programming (CP) is a paradigm that allows effective solving of combinatorial problems in many areas, such as planning and scheduling. However, modeling a combinatorial problem using constraints requires significant expertise in CP [10].

To alleviate this issue, several constraint acquisition systems have been introduced [4], [6], [1], [13], [5], [15]. Recently, an active learner system named QUACQ for *Quick Acquisition* [3] has been proposed. QUACQ iteratively generates membership queries and asks the user to classify them. When the answer of the user is *yes* for a given membership query, QUACQ reduces the search space by removing all violated constraints by the positive example. In the case of a negative answer, QUACQ focuses onto a constraint in a number of queries logarithmic in the size of the example. This key component allows QUACQ to converge on the target network in a polynomial number of queries. However, even with that good theoretical bound, QUACQ can require too many queries to be put in practice. For instance, the user has to classify more than 9000 queries using QUACQ to get the complete Sudoku puzzle model.

To make constraint acquisition systems more efficient in practice, an opportunistic kind of query that uses the structure of the problem has been introduced in [2] with the GENACQ algorithm. This kind of query, named “generalization query”, is based on an aggregation of variables into types. The user provides the variable types and based on these types, generalization queries ask the user whether or not a learned constraint can be generalized to other scopes of variables of the same type as those on the learned constraint. By using such queries over existing constraint acquisition systems, the number of queries needed to converge on the target constraint network can be significantly reduced.

Nevertheless, the aggregation of variables into types may not always be a straightforward task for the user especially when the problem under consideration has a hidden structure. In this paper, we propose to learn the potential types during the constraints acquisition process. The idea is to analyze the structure of the partial constraint network learned so far in order to detect potential types and to build generalization queries.

Indeed, when one looks more closely at the constraint network of a given problem, the variables of the same type are often tightly connected with similar constraints whereas the variables of different types are connected in a weaker way. To illustrate this point, let us consider the well known Lewis Carroll’s Zebra problem. The constraint network of this problem is usually formulated using 25 variables, partitioned into 5 types of 5 variables each. The types are color, nationality, drink, cigaret and pet. There is a clique of $\neq$ constraints on all pairs of variables of the same type and 14 additional constraints, among which 3 are unary, given in the description of the problem. Figure 1 shows the constraint network of the Zebra problem. In this example, it is clear that types have dense internal links but there are only a lower density of external links between different types.

The idea of detecting tightly connected sub-graphs arose in the study of networks such as social networks [16] and biochemical networks [12]. An important characteristic that commonly occurs in such networks is *community* structure. Informally, a network or a graph is said to have community structure, if the nodes of the network can be easily grouped into (potentially overlapping) sets of nodes such that the groups have more internal edges than outgoing edges. Each such group is called a community.

Given the similarity between the structure of a type and that of a community, we propose in this paper to detect

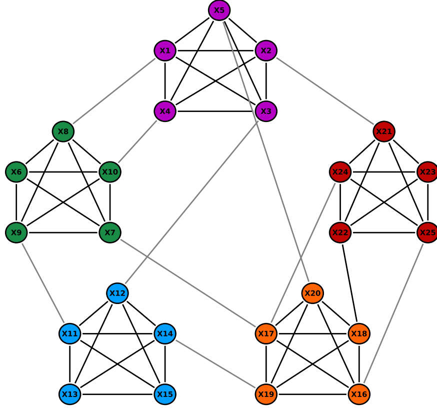

Figure 1. The constraint graph of the Zebra problem.

potential types by finding communities in the constraint network during the constraint acquisition process. Several methods for community finding have been proposed in the literature. We have considered three different techniques in this paper. The first one is based on the concept of modularity [14] which provides information on the strength of division of a network into communities. Networks with high modularity have dense connections between the nodes within communities but sparse connections between nodes in different communities. The second technique exploits the notion of edge betweenness [11], which is a measure that assigns a score to each edge. The edges that lie "between" many pairs of nodes have high scores which enables their easier identification. Removing these edges will leave behind just the communities themselves. The third technique is more straightforward. It is based on the assumption that the variables of the same type will tend to form quasi-cliques during the constraint acquisition process. That is, this technique finds sub-graphs with an edge density exceeding a threshold parameter.

In this paper we propose an algorithm, named MINE&ASK, that makes use of the extracted potential types to ask the user to classify generalization queries. We plugged MINE&ASK into the QUACQ constraint acquisition system, to obtain the boosted version M-QUACQ algorithm. We experimentally evaluate the benefit of our technique on several benchmark problems. The results show that M-QUACQ improves the basic QUACQ algorithm in terms of number of queries although no knowledge on the types is provided.

The outline of this paper is as follows. Section II gives the necessary definitions to understand the technical presentation. The algorithm MINE&ASK is presented in Section III. We illustrate the idea behind our approach with an example in section IV. Section V gives more details on the techniques used to extract potential types. The experimental results we obtained when comparing MINE&ASK with the different techniques to the basic QUACQ are given in section VI. Section VII concludes the paper.

II. BACKGROUND

The constraint acquisition process can be viewed as an interplay between the user who knows the problem and the learner that aims at solving the problem. The common knowledge shared between the user and the learner is a *vocabulary*. This vocabulary is represented by a (finite) set of variables X and domains $D = \{D(x_1), \ldots, D(x_n)\}$ over $\mathbb{Z}$. A constraint $(var(c), r)$ represents a relation $rel(c)$ on a subset of variables $var(c) \subseteq X$ (called the *scope* of c) that specifies which assignments of $var(c)$ are allowed. The arity of the relation r, denoted by $|r|$, is the number of variables involved by r. Combinatorial problems are represented with *constraint networks*. A constraint network is a set C of constraints on the vocabulary (X, D). An example e is a (partial/complete) assignment on a set of variables $var(e) \subseteq X$. e is rejected by a constraint c (i.e., $e \not\models c$) iff $var(c) \subseteq var(e)$ and the projection $e[var(c)]$ of e on $var(c)$ is not in c. A complete assignment e of X is a solution of C iff for all $c \in C$, c does not reject e. We denote by $sol(C)$ 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. A *constraint basis* is a set B of constraints built from the constraint language Γ on the vocabulary (X, D). Formally speaking, $B = \{c \mid (var(c) \subseteq X) \wedge (rel(c) \in \Gamma)\}$.

In terms of machine learning, 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 *query* $Ask(e)$, with $var(e) \subseteq X$, is a classification question asked to the user, where e is an assignment in $D^{var(e)} = \Pi_{x_i \in var(e)} D(x_i)$. A set of constraints C *accepts* an assignment e if and only if there does not exist any constraint $c \in C$ rejecting e. The answer to $Ask(e)$ is *yes* if and only if C_T accepts e.

A *type* T is a subset of variables that have a common property. A variable x is of type T iff $x \in T$. We denote by var a tuple of variables of the same type with the constraint that each variable appears only once in this tuple. A relation r *holds* on a type T if and only if $(var, r) \in C_T$ for all $var \in T^{|r|}$ where $T^{|r|}$ is the Cartesian power of T. A generalization query $AskGen(T, r)$ is a classification question asked to the user. $AskGen(T, r)$ is answered *yes* by the user if and only if r holds on T.

III. MINE&ASK ALGORITHM

In this section we present the MINE&ASK algorithm. The idea behind this algorithm is to mine the partial graph of the current constraint network in order to get potential types and then ask the user to classify generalization queries.

A. Description of MINE&ASK

Algorithm 1: MINE&ASK

Input: C: a set of constraints, r: a relation, $mine \in$ {modularity, betweenness, γ-clique}: a mine strategy, GQ_{max}: the maximum number of generalization queries

Output: L: a set of learned constraints

1. $L \leftarrow \varnothing$; $\#GQ \leftarrow 0$
2. $X' \leftarrow \bigcup var(c)$ s.t. $c \in C \wedge rel(c) = r$
3. $G_C \leftarrow G(X', E)$, $E = \{\{x, y\} \mid x, y \in var(c) \wedge x \neq y \wedge c \in C \wedge rel(c) = r\}$
4. $Table \leftarrow \{Y \mid Y \in component(G_C) \wedge \neg isClique(G_C(Y))\}$
5. **while** $Table \neq \varnothing \wedge \#GQ \leq GQ_{max}$ **do**
6. pick Y in $Table$
7. $generalized \leftarrow false$
8. **if** $(\nexists (Y', r') \in NegativeQ \mid Y' \subseteq Y \wedge r \subseteq r') \wedge (\nexists var \in Y^{|r|} \mid (var, r) \notin B)$ **then**
9. **if** $Sol(C_L \cup \{(var, r) \mid var \in Y^{|r|}\}) \neq \varnothing$ and $AskGen(Y, r) = Yes$ **then**
10. $L \leftarrow L \cup \{(var, r) \mid var \in Y^{|r|}\}$
11. $generalized \leftarrow true$
12. **else** $NegativeQ \leftarrow NegativeQ \cup \{(Y, r)\}$
13. $\#GQ++$
14. **if** $\neg generalized$ **then**
15. $Table \leftarrow Table \cup mine(G(Y))$
16. **return** L;

The algorithm MINE&ASK takes as argument the set of constraints C learned so far, a relation r, the operator $mine$ that corresponds to the strategy used for extracting potential types and the maximum number of generalization queries GQ_{max} that we are allowed to ask. The algorithm uses a global data structure $NegativeQ$, which is a set of pairs (Y, r) for which we know that r does not hold. MINE&ASK also uses the local data structure $Table$ which contains all potential types that are candidates for generalization.

MINE&ASK starts by initializing L to the empty set and the number of generalization queries $\#GQ$ to zero (line 1). The set L will contain the output of MINE&ASK, that is all learned constraints. In line 3, we build the constraint graph $G(X', E)$, noted G_C and restricted to the relation r. This step is important because it helps to reveal the structure of the network. To illustrate what this means, let us consider the example given in Figure 2(a). This problem consists of 12 variables and 66 binary constraints which means that the constraint network of this problem is a complete graph (i.e. clique). Two relations are used in the constraints namely $\neq$ and $\geq$. Although the constraint network is a clique, no learned constraint can be generalized to the other scopes in this clique as the links connecting the variables of the clique come from different relations. On the contrary, as shown in Figure 2(b), when we focus on the only $\neq$ relation, useful types can be easily detected.

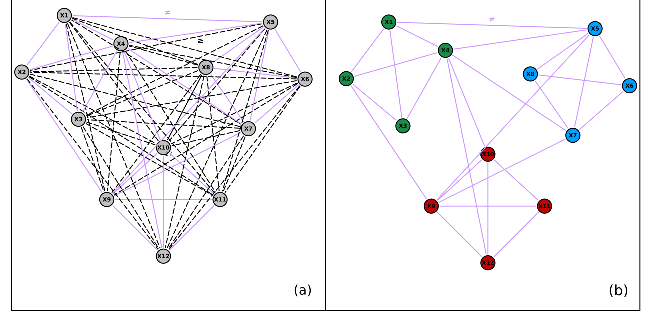

Figure 2. A constraint network with a hidden structure

Afterwards, we put in $Table$ all connected components of G whilst excluding the already formed cliques (line 4). At each iteration, MINE&ASK picks a potential type Y from this table (line 6) and asks a generalization query on (Y, r) (line 9) if the answer cannot be deduced. That is, the answer to an $AskGen(Y, r)$ is negative if the user already classifies as negative a query on a sub-type of Y (line 8). A negative answer can also be deduced in the case of the unsatisfiability of the resulting network where in such case the answer cannot be positive (line 9). Such satisfiability tests allow us to avoid asking unnecessary queries. For instance, trying to generalize a non-commutative relation to a clique, or asking if a clique of difference applies on a set of 4 variables with the same domain of size 3.

Now, if the answer of the user to $AskGen(Y, r)$ is *yes*, this means that r holds on the type Y. Thus, we add all inferred constraints on Y to the set L (line 10). In the case of a negative answer, we add (Y, r) to $NegativeQ$ with intent to avoid asking redundant queries afterwards. When no generalization happened on (Y, r) (line 15), this means that Y is not a type on which r can be generalized. Here, we call the operator $mine$ to extract potential types from the subgraph $G(Y)$. The resulting potential types are added to $Table$ to be taken into account later. The main loop (line 5) terminates when all potential types in $Table$ are processed, or when the number of generalization queries exceeds a given threshold GQ_{max}.

B. M-QUACQ Algorithm

MINE&ASK is a generic technique that can be plugged into any constraint acquisition system. In this section

we present M-QUACQ (Algorithm 2) where we plugged MINE&ASK into the QUACQ system [3].

M-QUACQ initializes the constraint network C_L to the empty set (line 1). When C_L is unsatisfiable (line 3), the space of possible networks collapses because there does not exist any subset of the given basis B that is able to correctly classify the examples already asked to the user. In line 4, M-QUACQ computes a complete assignment e satisfying C_L and violating at least one constraint from B. If such an example does not exist (line 5), then all constraints in B are implied by C_L, and the algorithm has converged. Otherwise, we propose the example e to the user, who will answer by *yes* or *no* (line 6). If the answer is *yes*, we can remove from B the set $\kappa_B(e)$ of all constraints in B that reject e (line 7). If the answer is *no*, we are sure that e violates at least one constraint of the target network C_T. We then call the function `FindScope` to discover the scope of one of these violated constraints. Here, `FindScope` acts in a dichotomous manner and asks a number of queries logarithmic in the size of the example. Afterwards, `FindC` will select which constraint with the given scope is violated by e (line 9). If no constraint is returned (line 10), this is a condition for collapsing as we could not find in B a constraint rejecting one of the negative examples. Otherwise, we know that the constraint c returned by `FindC` belongs to the target network C_T, then we add it to the learned network C_L (line 11). Note that `FindScope` and `FindC` functions are used exactly as they appear in [3]. Afterwards, we call MINE&ASK to mine the learned constraint network C_L in order to extract potential types and to ask generalization queries. M-QUACQ updates C_L by adding all learned constraints (line 12).

IV. AN ILLUSTRATIVE EXAMPLE

In this section we illustrate on an example the idea of extracting potential types during the constraint acquisition process. Let us consider the example given in Figure 3. The **part (a)** of Figure 3 shows the constraint network of the problem that the user has in mind. This problem consists in 15 variables and 39 binary constraints. Two relations are used, noted r_1 and r_2 in Figure 3. The **part (b)** of Figure 3 shows the constraint network learned, at a given point, using QUACQ system. Suppose that the last constraint learned using QUACQ was $((x_1, x_2), r_1)$. Now, we want to extract potential types on which the relation r_1 can be generalized. To this end, MINE&ASK restricts the constraint network to the constraints that use r_1 (**part (c)** in Figure 3). Suppose now that MINE&ASK algorithm finds three potential types $T_1 = \{x_1, \ldots x_5\}$, $T_2 = \{x_6, \ldots x_{10}\}$ and $T_3 = \{x_{11}, \ldots x_{15}\}$. According to what the user has in mind (**part (a)** of Figure 3), a generalization query on T_3 will be classified as negative whereas the ones on T_1 and T_2 will be classified as positive. Nine constraints will be, in one

Algorithm 2: M-QUACQ = QUACQ + MINE&ASK

Input: $mine \in$ {`modularity`, `betweenness`, γ-`clique`}: a mine strategy, GQ_{max}: the maximum number of generalization queries
Output: C_L: a set of learned constraints

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   **if** $Ask(e) = yes$ **then**
7     $B \leftarrow B \setminus \kappa_B(e)$;
8   **else**
9     $c \leftarrow$ `FindC`(e, `FindScope`($e, \varnothing, X,$ **false**));
10   **if** $c = nil$ **then return** "collapse" **else**
11       $C_L \leftarrow C_L \cup \{c\}$;
12       $C_L \leftarrow C_L \cup$ MINE&ASK($C_L, mine, rel(c), GQ_{max}$);
13 **return** C_L;

shot, added to the current constraint network (see **part (d)** of Figure 3).

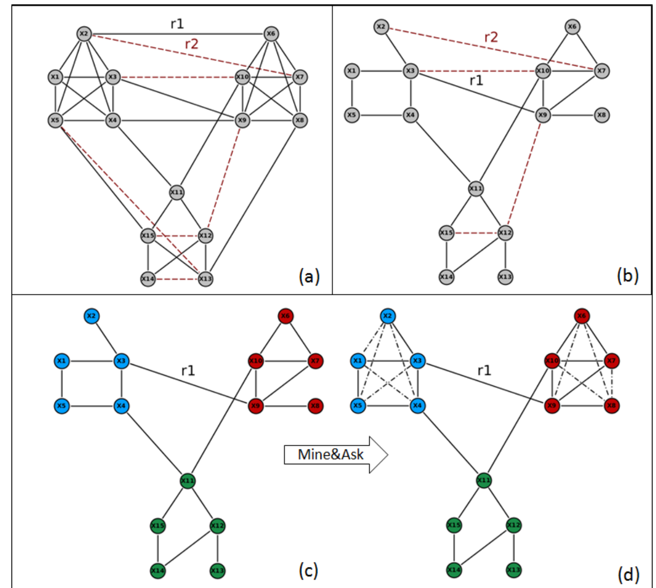

Figure 3. Illustrative Example

V. EXTRACTION OF POTENTIAL TYPES

MINE&ASK extracts variable types by finding communities in the current graph of learned constraints. The way in which the operator $mine$ finds communities at line 15 of Algorithm 1 is described in this section.

A. Optimizing modularity

One of the most effective approaches for detecting communities in networks is based on the optimization of the measure known as modularity [14]. Given a partition of vertices of a network into disjoint communities, modularity reflects the concentration of edges within communities compared with random distribution of links between all nodes regardless of communities.

More formally, let $G = (X, E)$ be a graph, with $X = \{x_1, \dots, x_n\}$ the set of vertices and let A be the adjacency matrix of G. That is, $A_{ij} = 1$ if there exists an edge between vertices x_i and x_j and $A_{ij} = 0$ otherwise. Suppose the vertices are divided into communities such that vertex x_i belongs to community $c(x_i)$ and let $deg(x_i)$ denotes the degree of x_i. Then the modularity Q is given by the following formula:

$$Q = \sum_{i,j} \left[\frac{A_{ij}}{2m} - \frac{deg(x_i) \times deg(x_j)}{4m^2} \right] \delta(c(x_i), c(x_j))$$

where m is the total number of edges in the network, and $\delta(c(x_i), c(x_j)) = 1$ if x_i and x_j belong to the same community (i.e., $c(x_i) = c(x_j)$) and 0 otherwise.

High values of the modularity correspond to good partitions of a network into communities [14]. Hence one should be able to find such good partitions by searching through the possible candidates for ones with high modularity. Unfortunately, finding the global maximum modularity over all possible divisions is NP-hard, but reasonably good solutions can be found with approximate optimization techniques. In this paper, we have used the algorithm introduced in [8] and implemented in the $igraph$ software package [9]. This algorithm uses a greedy optimization where, starting with a partition where each vertex is the unique member of a community, it repeatedly joins together the two communities whose fusion produces the largest increase in Q.

B. Edge betweenness centrality

Recently, the concept of edge betweenness was introduced [11] as a measure that provides information on edges centrality in networks. This measure can be implemented in several ways but the most common way is the one based on shortest paths. Formally speaking, let x_i and x_j be two nodes in the network. Let σ_{ij} denotes the number of shortest paths between nodes x_i and x_j and $\sigma_{ij}(e)$ denotes the number of shortest paths between x_i and x_j which go through the edge e. The Betweenness centrality of e, denoted by $B(e)$, is defined as follows:

$$B(e) = \sum_{ij} \frac{\sigma_{ij}(e)}{\sigma_{ij}}$$

If two communities are joined by only a few inter-community edges, then all paths through the network from vertices of one community to vertices of the other must pass through one of those few edges. Thus, the edge betweenness scores for inter-community edges are expected to be larger than the ones for intra-community edges. The betweeness based algorithm to find community structure is used as it appears in [11]. The idea behind this algorithm is to iteratively calculate the betweenness score for each edge and to remove the one with the highest score. That is, removing edges with high betweenness scores allows us to isolate the communities. This algorithm is also available in the $igraph$ software package [9].

C. Quasi-cliques detection

Mining the constraint network for dense subgraphs may be a possible way for discovering communities. Cliques are the densest form of subgraphs. A graph is a clique if there is an edge between every pair of the vertices. This requirement is not desirable in our case because the idea is to anticipate the formation of complete cliques in order to be able to infer some constraints. Therefore, instead of mining cliques, our goal is to extract $\gamma - cliques$ (i.e. quasi-cliques), which are sub-graphs with an edge density exceeding a given threshold parameter $\gamma \in [0, 1]$.

Definition 1: ($\gamma -$ **clique**) Let $G = (X, E)$ be a graph with X the set of vertices, E the set of edges, and a parameter $\gamma \in [0, 1]$. A $\gamma - clique$ is a subset of vertices $K \subseteq X$ such that the induced subgraph $G(K)$ is a connected component and $|E \cap K \times K| \geq \gamma \frac{q(q-1)}{2}$, with $q = |K|$.

Algorithm 3: `FindQCliques` (G, A, B, K, γ)

```
1  if A = ∅ then
2      report K as a quasi-clique;
3  while A ≠ ∅ do
4      choose x ∈ A;
5      K' ← K ∪ {x};
6      A' ← {y | y ∈ X \ K' ∧ K' ∪ {y} is a γ − clique};
7      A' ← A' \ A' ∩ B;
8      FindQCliques(G, A', B, K', γ) ;
9      A ← A \ {x};
10     B ← B ∪ {x};
```

We propose an incomplete recursive algorithm (Algorithm 3) for finding $\gamma - cliques$ in an undirected graph G. This algorithm is an adaptation of the basic form of the well-known Bron-Kerbosch's algorithm [7] for finding maximal cliques in a graph. Algorithm 3 is based on the recursive function `FindQCliques` that takes as arguments an undirected graph G, a set A of candidates, a set B of vertices to exclude from consideration to avoid generating the same quasi-clique several times, the quasi-clique K being constructed and a parameter $\gamma \in]0, 1[$ which specifies the minimum edge density of quasi-cliques. The recursion is initiated by setting B and K to be the empty set and A to be the vertex set of the graph. Each time a new element is added to the current quasi-clique (line 5), we

calculate a new set A' of candidates. A vertex y is an element of A' if and only if when it is added to the current quasi-clique we obtain a new quasi-clique (line 6). This condition is not a necessary condition to lead to a new quasi-clique. Consequently, Algorithm 3 is incomplete. When the candidate set becomes empty (line 1), a new quasi-clique is reported and a backtrack to the last choice is performed. Then, the last choice is added to the set B to exclude it from consideration in future quasi-cliques.

VI. EXPERIMENTATIONS

We performed some experiments to evaluate the impact of using MINE&ASK in constraint acquisition. We implemented MINE&ASK and plugged it in QUACQ system, leading to the M-QUACQ version. We first present the benchmark problems we used for our experiments. Then, we report the results of acquiring these problems with the basic version of QUACQ [3], our version M-QUACQ and G-QUACQ version. The G-QUACQ version includes the generalization process with GENACQ algorithm and the user provides all variable types [2]. The experiments evaluate also the different ways in which our approach extracts potential types, namely the `modularity`, the `betweenness` and the γ-`clique`. Our tests were conducted on an Intel Core i5-3320M CPU @ 2.60GHz $\times$ 4 with 4 Gb of RAM.

A. Benchmark Problems

Zebra problem. The Lewis Carroll Zebra problem is formulated using 5 types of 5 variables each, with 5 cliques of $\neq$ constraints and 14 additional constraints given in the description of the problem. We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 4450 unary and binary constraints taken from a language with 24 basic arithmetic and distance constraints.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. For this problem, we use 36 variables with domains of size 6 and 180 binary $\neq$ constraints on rows and columns. Rows and columns are the types of variables (10 types). We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 1260 constraints based on the language $\Gamma = \{=, \neq\}$.

Purdey. Like Zebra, this problem has a single solution. Four families have stopped by Purdey's general store, each to buy a different item and paying differently. Under a set of additional constraints given in the description, the problem is how can we match family with the item they bought and how they paid for it. The target network of Purdey has 12 variables with domains of size 4 and 30 binary constraints. Here we have three types of variables, which are *family*, *bought* and *paid*, each of them contains four variables. We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 396 constraints based on the language $\Gamma = \{=, \neq\}$.

PlaceNumPuzzle. The PlaceNumPuzzle problem is to place numbers 1 through N on nodes of a given graph such that each number appears exactly once and no connected nodes have consecutive numbers. For this problem, we use 25 variables with domains of size 25 and 64 binary constraints. The problem has three types which are the cliques of the graph. We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 1260 binary constraints taken from a language of 4 arithmetic and distance constraints.

Murder. Someone was murdered last night, and you are summoned to investigate the murder. The objects found on the spot that do not belong to the victim include: a pistol, an umbrella, a cigarette, a diary, and a threatening letter. There are also witnesses who testify that someone had argued with the victim, someone left the house, someone rang the victim, and some walked past the house several times about the time the murder occurred. The suspects are: Miss Linda Ablaze, Mr. Tom Burner, Ms. Lana Curious, Mrs. Suzie Dulles, and Mr. Jack Evilson. Each suspect has a different motive for the murder, including: being harassed, abandoned, sacked, promotion and hate. Other clues are given below. Under a set of additional clues given in the description, the problem is who was the Murderer? And what was the motive, the evidence-object, and the activity associated with each suspect. The target network of Murder has 20 variables with domains of size 5 and 53 binary constraints. Here we have four types of variables, which are *suspect*, *motive*, *object*, and *activity*, each of them contains five variables. We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 380 constraints based on the language $\Gamma = \{=, \neq\}$.

Sudoku. The Sudoku model is expressed using 81 variables with domains of size 9, and 810 $\neq$ binary constraints on rows, columns and squares. In this problem, the types are the 9 rows, 9 columns and 9 squares, of 9 variables each. We fed QUACQ, G-QUACQ and M-QUACQ with a basis B of 6480 binary constraints from the language $\Gamma = \{=, \neq\}$.

B. Results

For all our experiments we report, the total number $\#Ask$ of standard queries asked by the basic QUACQ, the total number $\#AskGen$ of generalization queries, and the numbers $\#no$ and $\#yes$ of negative and positive generalization queries, where $\#AskGen = \#no + \#yes$. The time overhead of using M-QUACQ rather than QUACQ is not reported since computing a generalization query takes a few milliseconds.

First of all, it should be noted that the parameter γ specifying the minimum edge density of quasi-cliques may have an influence on the performance of M-QUACQ. Indeed, the lower γ, the greater the number of extracted quasi-cliques. This means that the probability that extracted types do not correspond to real types increases when γ is small and therefore, the number of negative answers to generalization queries may become important. This phenomenon can be more or less controlled by adjusting the value of γ. That

being said, the value of γ was fixed to 0.8 after a few preliminary tests.

Now, with the results reported in table I, we aim at comparing the performance of the basic QUACQ, G-QUACQ where the types are provided by the user and the three versions of M-QUACQ where types are learned during the acquisition process.

Not surprisingly, we notice that M-QUACQ is always better than QUACQ but still less efficient than G-QUACQ. Furthermore, the number of queries ($\#Ask + \#AskGen$) that were asked using M-QUACQ is often closer to the number of queries asked using G-QUACQ. For instance, to learn the *PlaceNumPuzzle* QUACQ needs 3746 queries. Providing the types to G-QUACQ reduced the number of queries to 390. Now, using our approach, M-QUACQ needed only 662 queries although no knowledge on types was provided.

In addition, Table I reports the performance of the three versions of M-QUACQ. What is noticeable about the three different ways for extracting potential types is that `modularity` clearly outperforms the two other techniques and improves significantly the performance of M-QUACQ on all considered problems. Indeed, the combination of M-QUACQ with `modularity` leads to tremendous savings in the number of queries compared to QUACQ: 627+35 (-82%) instead of 3746 on PlaceNumPuzzle, 272+12 (-41%) instead of 483 on Murder, 410+14 (-39%) instead of 694 on Zebra, 140+8 (-28%) instead of 205 on Purdey, 7963+57 (-28%) instead of 9593 on Sudoku.

We also notice that the `betweenness` outperforms the γ`-clique` on all problems except for the Latin Square. This can be explained by the fact that the types overlap (rows and columns) in Latin Square and that γ`-clique` is likely more able to detect overlapping types than `betweenness`.

In conclusion we can say that when `modularity` is used to extract the types, the algorithm M-QUACQ performs very well and is very close to G-QUACQ although no knowledge on types is provided. Furthermore, since `modularity` was introduced to find communities in very large networks, we think that the performance of M-QUACQ with `modularity` can be more significant when the size of the problem increases. For instance, Figure 4 shows the gain of M-QUACQ with `modularity` compared to QUACQ on the Latin Square when fed with an increasing number of variables. It is clear in this figure that the gain significantly increases with the size of the problem.

Table I
M-QUACQ WITH MODULARITY, BETWEENNESS AND γ-CLIQUE ON PLACENUMPUZZLE, MURDER, ZEBRA, PURDEY AND SUDOKU.

	QUACQ	G-QUACQ		M-QUACQ			
Strategies	$\#Ask$	$\#Ask$	$\#AskGen$	$\#Ask$	$\#AskGen$	$\#no$	$\#yes$
Latin Square							
modularity	2058	129	68	987	61	26	35
betweenness				1674	22	5	17
γ**-clique**				1172	35	1	34
PlaceNumPuzzle							
modularity	3746	351	39	627	35	4	31
betweenness				655	33	2	31
γ**-clique**				688	33	2	31
Murder							
modularity	483	230	55	272	12	2	10
betweenness				272	12	2	10
γ**-clique**				342	13	3	10
Zebra							
modularity	694	257	67	410	14	0	14
betweenness				410	14	0	14
γ**-clique**				410	14	0	14
Purdey							
modularity	205	93	39	140	8	0	8
betweenness				140	8	0	8
γ**-clique**				140	8	0	8
Sudoku							
modularity	9593	260	166	7963	57	20	37
betweenness				8960	50	18	32
γ**-clique**				9461	117	104	13

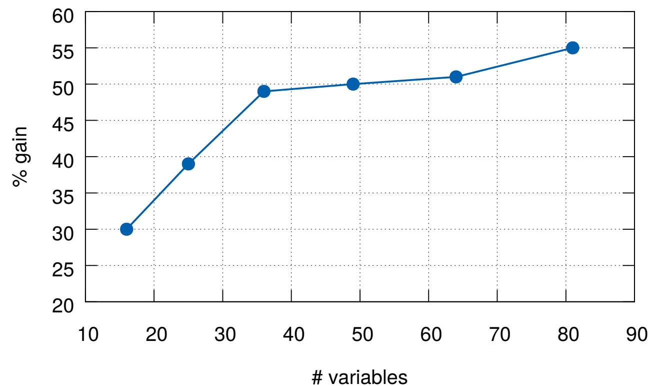

Figure 4. The gain of M-QUACQ on Latin Square when the number of variables increases.

VII. CONCLUSION

We have proposed MINE&ASK, a generalization based algorithm that is able to mine partial graphs of constraint networks and to generalize, on potential types, constraints learned by any constraint acquisition system. MINE&ASK acts when no knowledge is provided on the variable types. We have detailed and tested three techniques to extract potential types, namely the `modularity`, the `betweenness` and the γ`-clique` techniques. We have plugged our MINE&ASK into the QUACQ constraint acquisition system, leading to the M-QUACQ algorithm. We have experimentally evaluated the benefit of our approach on several benchmark problems. The results show that M-QUACQ significantly improves the basic QUACQ algorithm and they are quite close to the results when variable types are provided.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.

[2] Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. Boosting constraint acquisition via generalization queries. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 99–104, 2014.

[3] 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.

[4] Christian Bessière, 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.

[5] Christian Bessiere, Remi Coletta, and Nadjib Lazaar. Solve a constraint problem without modeling it. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, pages 1–7, 2014.

[6] Christian Bessière, 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.

[7] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[8] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70:066111, 2004.

[9] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[10] 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. Invited Talk.

[11] Michelle Girvan and Mark E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[12] Takashi Ito, Tomoko Chiba, Ritsuko Ozawa, Mikio Yoshida, Masahira Hattori, and Yoshiyuki Sakaki. A comprehensive two-hybrid analysis to explore the yeast protein interactome. *Proceedings of the National Academy of Sciences*, 98(8):4569–4574, 2001.

[13] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 45–52, 2010.

[14] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[15] K.M. Shchekotykhin and G. Friedrich. Argumentation based constraint acquisition. In *Proceedings of the Ninth IEEE International Conference on Data Mining (ICDM'09)*, pages 476–482, Miami, Florida, 2009.

[16] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.