Boosting Constraint Acquisition via Generalization Queries[1]

Christian Bessiere[2] and **Remi Coletta**[2] and **Abderrazak Daoudi**[2,3] and **Nadjib Lazaar**[2] and **Younes Mechqrane**[2] and **El Houssine Bouyakhf** [3]

Abstract. Constraint acquisition assists a non-expert user in modeling her problem as a constraint network. In existing constraint acquisition systems the user is only asked to answer very basic questions. The drawback is that when no background knowledge is provided, the user may need to answer a great number of such questions to learn all the constraints. In this paper, we introduce the concept of *generalization query* based on an aggregation of variables into types. We present a constraint generalization algorithm that can be plugged into any constraint acquisition system. We propose several strategies to make our approach more efficient in terms of number of queries. Finally we experimentally compare the recent QUACQ system to an extended version boosted by the use of our generalization functionality. The results show that the extended version dramatically improves the basic QUACQ.

1 INTRODUCTION

Constraint programming (CP) is used to model and solve combinatorial problems in many application areas, such as resource allocation or scheduling. However, building a CP model requires some expertise in constraint programming. This prevents the use of this technology by a novice and thus this has a negative effect on the uptake of constraint technology by non-experts.

Several techniques have been proposed for assisting the user in the modeling task. In [10], Freuder and Wallace proposed the matchmaker agent, an interactive process where the user is able to provide one of the constraints of her target problem each time the system proposes a wrong solution. In [12], Lallouet et al. proposed a system based on inductive logic programming that uses background knowledge on the structure of the problem to learn a representation of the problem correctly classifying the examples. In [4, 6], Bessiere et al. made the assumption that the only thing the user is able to provide is examples of solutions and non-solutions of the target problem. Based on these examples, the *Conacq.1* system learns a set of constraints that correctly classifies all examples given so far. This type of learning is called *passive* learning. In [3], Beldiceanu and Simonis proposed *Model Seeker*, another passive learning approach. Positive examples are provided by the user. The system arranges these examples as a matrix and identifies constraints in the global constraints catalog ([2]) that are satisfied by rows or columns of all examples.

By contrast, in an active learner like *Conacq.2*, the system proposes examples to the user to classify as solutions or non solutions [7]. Such questions are called membership queries [1]. CONACQ introduces two computational challenges. First, how does the system generate a useful query? Second, how many queries are needed for the system to converge to the target set of constraints? It has been shown that the number of membership queries required to converge to the target set of constraints can be exponentially large [8].

QUACQ is a recent active learner system that is able to ask the user to classify *partial* queries [5]. Using partial queries and given a negative example, QUACQ is able to find a constraint of the problem the user has in mind in a number of queries logarithmic in the size of the example. This key component of QUACQ allows it to always converge on the target set of constraints in a polynomial number of queries. However, even that good theoretical bound can be hard to put in practice. For instance, QUACQ requires the user to classify more than 8000 examples to get the complete Sudoku model.

In this paper, we propose a new technique to make constraint acquisition more efficient in practice by using variable types. In real problems, variables often represent components of the problem that can be classified in various types. For instance, in a school timetabling problem, variables can represent teachers, students, rooms, courses, or time-slots. Such types are often known by the user. To deal with types of variables, we introduce a new kind of query, namely, *generalization query*. We expect the user to be able to decide if a learned constraint can be generalized to other scopes of variables of the same type as those in the learned constraint. We propose an algorithm, GENACQ for *generalized acquisition*, that asks such generalization queries each time a new constraint is learned. We propose several strategies and heuristics to select the good candidate generalization query. We plugged our generalization functionality into the QUACQ constraint acquisition system, leading to the G-QUACQ algorithm. We experimentally evaluate the benefit of our technique on several benchmark problems. The results show that G-QUACQ dramatically improves the basic QUACQ algorithm in terms of number of queries.

The rest of the paper is organized as follows. Section 2 gives the necessary definitions to understand the technical presentation. Section 3 describes the generalization algorithm. In Section 4, several strategies are presented to make our approach more efficient. Section 5 presents the experimental results we obtained when comparing G-QUACQ to the basic QUACQ and when comparing the different strategies in G-QUACQ. Section 6 concludes the paper and gives some directions for future research.

[1] This work has been funded by the ANR project BR4CP (ANR-11-BS02-008) and by the EU project ICON (FP7-284715).

[2] University of Montpellier, France, email: {bessiere, coletta, daoudi, lazaar, mechqrane}@lirmm.fr

[3] LIMIARF/FSR, University Mohammed V Agdal, Rabat, Morocco, email: bouyakhf@fsr.ac.ma

2 BACKGROUND

We introduce some useful notions in constraint programming and concept learning. The common knowledge shared between a learner that aims at solving the problem and the user who knows the problem is a *vocabulary*. This vocabulary is represented by a (finite) set of variables X and domains $D = \{D(x_1), \dots, D(x_n)\}$ over $\mathbb{Z}$. A constraint c 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. 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_i is a subset of variables defined by the user as having a common property. A variable x is of type T_i iff $x \in T_i$. A scope $var = (x_1, \dots, x_k)$ of variables *belongs* to a sequence of types $s = (T_1, \dots, T_k)$ (denoted by $var \in s$) if and only if $x_i \in T_i$ for all $i \in 1..k$. Consider $s = (T_1, T_2, \dots, T_k)$ and $s' = (T'_1, T'_2, \dots, T'_k)$ two sequences of types. We say that s' *covers* s (denoted by $s \sqsubseteq s'$) iff $T_i \subseteq T'_i$ for all $i \in 1..k$. A relation r *holds* on a sequence of types s if and only if $(var, r) \in C_T$ for all $var \in s$. A sequence of types s is *maximal* with respect to a relation r if and only if r holds on s and there does not exist s' covering s on which r holds.

3 GENACQ ALGORITHM

In this section we present GenAcq, a *generalized acquisition* algorithm, The idea behind this algorithm is, given a constraint c learned on $var(c)$, to generalize this constraint to sequences of types s covering $var(c)$ by asking generalization queries $AskGen(s, r)$. A generalization query $AskGen(s, r)$ is answered *yes* by the user if and only if for every sequence var of variables covered by s the relation r holds on var in the target constraint network C_T.

3.1 Description of GenAcq

The algorithm GenAcq (see Algorithm 1) takes as input a target constraint c that has already been learned and a set $NonTarget$ of constraints that are known not to belong to the target network. It also uses the global data structure $NegativeQ$, which is a set of pairs (s, r) for which we know that r does not hold on all sequences of variables covered by s. c and $NonTarget$ can come from any constraint acquisition mechanism or as background knowledge. $NegativeQ$ is built incrementally by each call to GenAcq. GenAcq also uses the set $Table$ as local data structure. $Table$ will contain all sequences of types that are candidates for generalizing c.

Algorithm 1: GenAcq (c, $NonTarget$)

```
1  $Table \leftarrow \{s \mid var(c) \in s\} \setminus \{var(c)\}$
2  $G \leftarrow \emptyset$
3  $\#NoAnswers \leftarrow 0$
4  foreach $s \in Table$ do
5      if $\exists (s', r) \in NegativeQ \mid rel(c) \subseteq r \wedge s' \sqsubseteq s$ then
6          $Table \leftarrow Table \setminus \{s\}$
7      if $\exists c' \in NonTarget \mid rel(c') = rel(c) \wedge var(c') \in s$
8      then $Table \leftarrow Table \setminus \{s\}$
9  while $Table \neq \emptyset \wedge \#NoAnswers <$ cutoffNo do
10     pick $s$ in $Table$
11     if $AskGen(s, rel(c)) = yes$ then
12         $G \leftarrow G \cup \{s\} \setminus \{s' \in G \mid s' \sqsubseteq s\}$
13         $Table \leftarrow Table \setminus \{s' \in Table \mid s' \sqsubseteq s\}$
14         $\#NoAnswers \leftarrow 0$
       else
15         $Table \leftarrow Table \setminus \{s' \in Table \mid s \sqsubseteq s'\}$
16         $NegativeQ \leftarrow NegativeQ \cup \{(s, rel(c))\}$
17         $\#NoAnswers + +$
18 return $G$;
```

In line 1, GenAcq initializes the set $Table$ to all possible sequences s of types that contain $var(c)$. In line 2, GenAcq initializes the set G to the empty set. G will contain the output of GenAcq, that is, the set of maximal sequences from $Table$ on which $rel(c)$ holds. The counter $\#NoAnswers$ counts the number of consecutive times generalization queries have been classified negative by the user. It is initialized to zero (line 3). $\#NoAnswers$ is not used in the basic version of GenAcq but it will be used in the version with cutoffs. (In other words, the basic version uses cutoffNo $= +\infty$ in line 9).

The first loop in GenAcq (line 4) eliminates from $Table$ all sequences s for which we already know the answer to the query $AskGen(s, rel(c))$. In lines 5-6, GenAcq eliminates from $Table$ all sequences s such that a relation r entailed by $rel(c)$ is already known not to hold on a sequence s' covered by s (i.e., (s', r) is in $NegativeQ$). This is safe to remove such sequences because the absence of r on some scope in s' implies the absence of $rel(c)$ on some scope in s (see Lemma 1). In lines 7-8, GenAcq eliminates from $Table$ all sequences s such that we know from $NonTarget$ that there exists a scope var in s such that $(var, rel(c)) \notin C_T$.

In the main loop of GenAcq (line 9), we pick a sequence s from $Table$ at each iteration and we ask a generalization query to the user (line 11). If the user says *yes*, s is a sequence on which $rel(c)$ holds. We put s in G and remove from G all sequences covered by s, so as to keep only the maximal ones (line 12). We also remove from $Table$ all sequences s' covered by s (line 13) to avoid asking redundant questions later. If the user says *no*, we remove from $Table$ all sequences s' that cover s (line 15) because we know they are no longer candidate for generalization of $rel(c)$ and we store in $NegativeQ$ the fact that $(s, rel(c))$ has been answered *no*. The loop finishes when $Table$ is empty and we return G (line 18).

3.2 Completeness and Complexity

We analyze the completeness and complexity of GENACQ in terms of number of generalization queries.

Lemma 1. *If $AskGen(s, r) = no$ then for any (s', r') such that $s \sqsubseteq s'$ and $r' \subseteq r$, we have $AskGen(s', r') = no$.*

Proof. Assume that $AskGen(s, r) = no$. Hence, there exists a sequence $var \in s$ such that $(var, r) \notin C_T$. As $s \sqsubseteq s'$ we have $var \in s'$ and then we know that $(var, r) \notin C_T$. As $r' \subseteq r$, we also have $(var, r') \notin C_T$. As a result, $AskGen(s', r') = no$. □

Lemma 2. *If $AskGen(s, r) = yes$ then for any s' such that $s' \sqsubseteq s$, we have $AskGen(s', r) = yes$.*

Proof. Assume that $AskGen(s, r) = yes$. As $s' \sqsubseteq s$, for all $var \in s'$ we have $var \in s$ and then we know that $(var, r) \in C_T$. As a result, $AskGen(s', r) = yes$. □

Proposition 1 (Completeness). *When called with constraint c as input, the algorithm* GENACQ *returns all maximal sequences of types covering $var(c)$ on which the relation $rel(c)$ holds.*

Proof. All sequences covering $var(c)$ are put in $Table$. A sequence in $Table$ is either asked for generalization or removed from $Table$ in lines 6, 8, 13, or 15. We know from Lemma 1 that a sequence removed in line 6, 8, or 15 would necessarily lead to a *no* answer. We know from Lemma 2 that a sequence removed in line 13 is subsumed and less general than another one just added to G. □

Proposition 2. *Given a learned constraint c and its associated $Table$,* GENACQ *uses $O(|Table|)$ generalization queries to return all maximal sequences of types covering $var(c)$ on which the relation $rel(c)$ holds.*

Proof. For each query on $s \in Table$ asked by GENACQ, the size of $Table$ strictly decreases regardless of the answer. As a result, the total number of queries is bounded above by $|Table|$. □

3.3 Illustrative Example

Let us take the Zebra problem to illustrate our generalization approach. The Lewis Carroll's Zebra problem has a single solution. The target network is formulated using 25 variables, partitioned in 5 types of 5 variables each. The types are thus *color*, *nationality*, *drink*, *cigaret*, *pet*, and the trivial type X of all variables. There is a clique of $\neq$ constraints on all pairs of variables of the same non trivial type and 14 additional constraints given in the description of the problem.

Figure 1 shows the variables of the Zebra problem and their types. In this example, the constraint $x_2 \neq x_5$ has been learned between the two color variables x_2 and x_5. This constraint is given as input of the GENACQ algorithm. GENACQ computes the $Table$ of all sequences of types covering the scope (x_2, x_5). $Table = \{(x_2, color)$, (x_2, X), $(color, x_5)$, $(color, color)$, $(color, X)$, (X, x_5), $(X, color)$, $(X, X)\}$. Suppose we pick $s = (X, x_5)$ at line 10 of GENACQ. According to the user's answer (*no* in this case), the $Table$ is reduced to $Table = \{(x_2, color), (x_2, X), (color, x_5), (color, color)$, $(color, X)\}$. As next iteration, let us pick $s = (color, color)$. The user will answer *yes* because there is indeed a clique of $\neq$ on the *color* variables. Hence, $(color, color)$ is added to G and the $Table$ is reduced to $Table = \{(x_2, X), (color, X)\}$. If we pick (x_2, X), the user answers *no* and we reduce the $Table$ to the empty set and return $G = \{(color, color)\}$, which means that the constraint $x_2 \neq x_5$ can be generalized to all pairs of variables in the sequence $(color, color)$, that is, $(x_i \neq x_j) \in C_T$ for all $(x_i, x_j) \in (color, color)$.

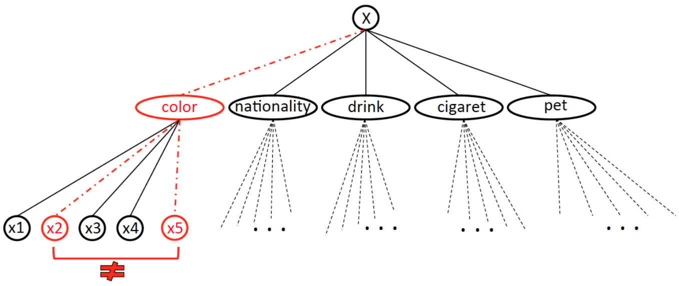

Figure 1. Variables and types for the Zebra problem.

3.4 Using Generalization in QUACQ

GENACQ is a generic technique that can be plugged into any constraint acquisition system. In this section we present G-QUACQ, a constraint acquisition algorithm obtained by plugging GENACQ into QUACQ, the constraint acquisition system presented in [5],

G-QUACQ is presented in Algorithm 2. We do not give the code of functions `FindScope` and `FindC` as we use them exactly as they appear in [5]. But let us say a few words on how they work. Given sets of variables S_1 and S_2, `FindScope`$(e, S_1, S_2, false)$ returns the subset of S_2 that, together with S_1 forms the scope of a constraint in the basis of possible constraints B that rejects e. Inspired from a technique used in QUICKXPLAIN [11], `FindScope` requires a number of queries logarithmic in $|S_2|$ and linear in the size of the final scope returned. The function `FindC` takes as parameter the negative example e and the scope returned by `FindScope`. It returns a constraint from C_T with the given scope that rejects e. For any assignment e, $\kappa_B(e)$ denotes the set of all constraints in B rejecting e.

Algorithm 2: G-QUACQ

```
$C_L \leftarrow \emptyset$, $NonTarget \leftarrow \emptyset$;
while true do
    if $sol(C_L) = \emptyset$ then return "collapse"
    choose $e$ in $D^X$ accepted by $C_L$ and rejected by $B$
    if $e = nil$ then return "convergence on $C_L$"
    if $Ask(e) = yes$ then
        $B \leftarrow B \setminus \kappa_B(e)$
        $NonTarget \leftarrow NonTarget \cup \kappa_B(e)$
    else
        $c \leftarrow$ FindC($e$, FindScope($e, \emptyset, X$, false))
        if $c = nil$ then return "collapse"
        else
            $G \leftarrow$ GENACQ($c, NonTarget$)
            foreach $s \in G$ do
                $C_L \leftarrow C_L \cup \{(var, rel(c)) \mid var \in s\}$
```

G-QUACQ has a structure very similar to QUACQ. It initializes the set $NonTarget$ and the network C_L it will learn to the empty set (line 1). If 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 of the user. In line 4, QUACQ computes a complete assignment e satisfying C_L but 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 we have converged. If we have not converged, 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) and we add all these ruled out constraints to the set $NonTarget$ to be used in GENACQ (line 8). 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. `FindC` will select which one with the given scope is violated by e (line 10). If no constraint is returned (line 11), this is again 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. This is here that the algorithm differs from QUACQ as we call GENACQ to find all the maximal sequences of types covering $var(c)$ on which $rel(c)$ holds. They are returned in G (line 13). Then, for every sequence of variables var belonging to one of by these sequences in G, we add the constraint $(var, rel(c))$ to the learned network C_L (line 14).

4 STRATEGIES

GENACQ learns the maximal sequences of types on which a constraint can be generalized. The order in which sequences are picked from $Table$ in line 10 of Algorithm 1 is not specified by the algorithm. As shown on the following example, different orderings can lead more or less quickly to the good (maximal) sequences on which a relation r holds. Let us come back to our example on the Zebra problem (Section 3.3). In the way we developped the example, we needed only 3 generalization queries to empty the set $Table$ and converge on the maximal sequence $(color, color)$ on which $\neq$ holds:

1. $AskGen((X, x_5), \neq) = no$
2. $AskGen((color, color), \neq) = yes$
3. $AskGen((x_2, X), \neq) = no$

Using another ordering, GENACQ needs 8 generalization queries:

1. $AskGen((X, X), \neq) = no$
2. $AskGen((X, color), \neq) = no$
3. $AskGen((color, X), \neq) = no$
4. $AskGen((X, x_5), \neq) = no$
5. $AskGen((x_2, X), \neq) = no$
6. $AskGen((x_2, color), \neq) = yes$
7. $AskGen((color, x_5), \neq) = yes$
8. $AskGen((color, color), \neq) = yes$

If we want to reduce the number of generalization queries, we may wonder which strategy to use. In this section we propose two techniques. The first idea is to pick sequences in the set $Table$ following an order given by a heuristic that will try to minimize the number of queries. The second idea is to put a cutoff on the number consecutive negative queries we accept to face, leading to a non complete generalization startegy: the output of GENACQ will no longer be guaranteed to be the *maximal* sequences.

4.1 Query Selection Heuristics

We propose some query selection heuristics to decide which sequence to pick next from $Table$. We first propose *optimistic* heuristics, which try to take the best from positive answers:

- **`max_CST`:** This heuristic selects a sequence s maximizing the number of possible constraints (var, r) in the basis such that var is in s and r is the relation we try to generalize. The intuition is that if the user answers *yes*, the generalization will be maximal in terms of number of constraints.
- **`max_VAR`:** This heuristic selects a sequence s involving a maximum number of variables, that is, maximizing $|\bigcup_{T \in s} T|$. The intuition is that if the user answers *yes*, the generalization will involve many variables.

Dually, we propose *pessimistic* heuristics, which try to take the best from negative answers:

- **`min_CST`:** This heuristic selects a sequence s minimizing the number of possible constraints (var, r) in the basis such that var is in s and r is the relation we try to generalize. The intuition is to maximize the chances to receive a *yes* answer. If, despite this, the user answers *no*, a great number of sequences are removed from $Table$ (see Lemma 1).
- **`min_VAR`:** This heuristic selects a sequence s involving a minimum number of variables, that is, minimizing $|\bigcup_{T \in s} T|$. The intuition is to maximize the chances of a *yes* answer while focusing on smaller sets of variables than `min_CST`. Again, a *no* answer leads to a great number of sequences removed from $Table$.

As a baseline for comparison, we define a random selector.

- **`random`:** It picks randomly a sequence s in $Table$.

4.2 Using Cutoffs

The idea here is to exit GENACQ before having proved the maximality of the sequences returned. We put a threshold cutoffNo on the number of consecutive negative answers to avoid using queries to check unpromising sequences. The hope is that GENACQ will return near-maximal sequences of types despite not proving maximality. This cutoff strategy is implemented by setting the variable cutoffNo to a predefined value. In lines 14 and 17 of GENACQ, a counter of consecutive negative answers is respectively reset and incremented depending on the answer from the user. In line 9, that counter is compared to cutoffNo to decide to exit or not.

5 EXPERIMENTATIONS

We made some experiments to evaluate the impact of using our generalization functionality GENACQ in the QUACQ constraint acquisition system. We implemented GENACQ and plugged it in QUACQ, leading to the G-QUACQ version. We first present the benchmark problems we used for our experiments. Then, we report the results of several experiments. The first one compares the performance of G-QUACQ to the basic QUACQ. The second reports experiments evaluating the different strategies we proposed (query selection heuristics and cutoffs) on G-QUACQ. The third evaluates the performance of our generalization approach when our knowledge of the types of variables is incomplete.

5.1 Benchmark Problems

Zebra problem. As introduced in section 3.3, the Lewis Carroll's 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 and G-QuAcq with a basis B of 4450 unary and binary constraints taken from a language with 24 basic arithmetic and distance constraints.

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 and G-QuAcq with a basis B of 6480 binary constraints from 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. For this problem, we use 25 variables with domains of size 5 and 100 binary $\neq$ constraints on rows and columns. Rows and columns are the types of variables (10 types). We fed QuAcq and G-QuAcq with a basis of constraints based on the language $\Gamma = \{=, \neq\}$.

Radio Link Frequency Assignment Problem. The RLFAP problem is to provide communication channels from limited spectral resources [9]. Here we build a simplified version of RLFAP that consists in distributing all the frequencies available on the base stations of the network. The constraint model has 25 variables with domains of size 25 and 125 binary constraints. We have five stations of five terminals (transmitters/receivers), which form five types. We fed QuAcq and G-QuAcq with a basis of 1800 binary constraints taken from a language of 6 arithmetic and distance constraints

Purdey. Like Zebra, this problem has a single solution. Four families have stopped by Purdeys 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.

5.2 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, respectively, where $\#AskGen = \#no + \#yes$. The time overhead of using G-QuAcq rather than QuAcq is not reported. Computing a generalization query takes a few milliseconds.

Our first experiment compares QuAcq and G-QuAcq in its baseline version, G-QuAcq +rand, on our benchmark problems. Table 1 reports the results. We observe that the number of queries asked by G-QuAcq is dramatically reduced compared to QuAcq. This is especially true on problems with many types involving many variables, such as Sudoku or Latin square. G-QuAcq acquires the Sudoku with 260 standard queries plus 166 generalization queries, when QuAcq acquires it in 8645 standard queries.

Table 1. QuAcq vs G-QuAcq.

	QuAcq	G-QuAcq +random	
	$\#Ask$	$\#Ask$	$\#AskGen$
Zebra	638	257	67
Sudoku	8645	260	166
Latin square	1129	117	60
RFLAP	1653	151	37
Purdey	173	82	31

Let us now focus on the behavior of our different heuristics in G-QuAcq. Table 2(top) reports the results obtained with G-QuAcq using min_VAR, min_CST, max_VAR, and max_CST to acquire the Sudoku model. (Other problems showed similar trends.) The results clearly show that max_VAR, and max_CST are very bad heuristics. They are worse than the baseline random. On the contrary, min_VAR and min_CST significantly outperform random. They respectively require 90 and 132 generalization queries instead of 166 for random. Notice that they all ask the same number of standard queries (260) as they all find the same maximal sets of sequences for each learned constraint.

Table 2. G-QuAcq with heuristics and cutoff strategy on Sudoku.

	cutoff	$\#Ask$	$\#AskGen$	$\#yes$	$\#no$
random	$+\infty$	260	166	42	124
min_VAR			90	21	69
min_CST			132	63	69
max_VAR			263	63	200
max_CST			247	21	226
min_VAR	3	260	75	21	54
	2		57	21	36
	1		**39**	21	18
min_CST	3	626	238	112	126
	2	679	231	132	99
	1	837	213	153	60

At the bottom of Table 2 we compare the behavior of our two best heuristics (min_VAR and min_CST) when combined with the cutoff strategy. We tried all values of the cutoff from 1 to 3. A first observation is that min_VAR remains the best whatever the value of the cutoff is. Interestingly, even with a cutoff equal to 1, min_VAR requires the same number of standard queries as the versions of G-QuAcq without cutoff. This means that using min_VAR as selection heuristic in $Table$, G-QuAcq is able to return the maximal sequences despite being stopped after the first negative generalization answer. We also observe that the number of generalization queries with min_VAR decreases when the cutoff becomes smaller (from 90 to 39 when the cutoff goes from $+\infty$ to 1). By looking at the last two columns we see that this is the number $\#no$ of negative answers which decreases. The good performance of min_VAR + cutoff=1 can thus be explained by the fact that min_VAR selects first queries that cover a minimum number of variables, which increases the chances to have a *yes* answer. Finally, we observe that the heuristic min_CST does not have the same nice characteristics as min_VAR. The smaller the cutoff, the more standard queries are needed, not compensating for the saving in number of generalization queries (from 260 to 837 standard queries for min_CST when the cutoff goes from $+\infty$ to 1). This means that with min_CST, when the cutoff becomes too small, GenAcq does not return the maximal sequences of types where the learned constraint holds.

In Table 3, we report the performance of G-QuAcq with random, min_VAR and min_VAR +cutoff=1 on all the other problems. We see that min_VAR +cutoff=1 significantly improves the performance of G-QuAcq on all problems. As in the case of Sudoku, we observe that min_VAR +cutoff=1 does not lead to an increase in the number of standard queries. This means that on all these problems min_VAR +cutoff=1 always returns the maximal sequences while asking less generalization queries with negative answers.

From these experiments we see that G-QuAcq with min_VAR +cutoff=1 leads to tremendous savings in number of queries compared to QuAcq: 257+23 instead of 638 on Zebra, 260+39 instead

Table 3. G-QuAcq with `random`, `min_VAR`, and cutoff=1 on Zebra, Latin square, RLFAP, and Purdey.

	$\#Ask$	$\#AskGen$	$\#yes$	$\#no$
Zebra				
Random	257	67	10	57
`min_VAR`		48	5	43
`min_VAR` +cutoff=1		**23**	5	18
Latin square				
Random	117	60	16	44
`min_VAR`		34	10	24
`min_VAR` +cutoff=1		**20**	10	10
RLFAP				
Random	151	37	16	21
`min_VAR`		41	14	27
`min_VAR` +cutoff=1		**22**	14	8
Purdey				
Random	82	31	5	26
`min_VAR`		24	3	21
`min_VAR` +cutoff=1		**12**	3	9

Table 4. G-QuAcq when the percentage of provided types increases.

	% of types	$\#Ask$	$\#AskGen$
Zebra	0	638	0
	20	619	12
	40	529	20
	60	417	27
	80	332	40
	100	257	48
Sudoku 9×9	0	8645	0
	33	3583	232
	66	610	60
	100	260	39
Latin Square	0	1129	0
	50	469	49
	100	117	20
Purdey	0	173	0
	33	111	8
	66	100	10
	100	82	12

of 8645 on Sudoku, 117+20 instead of 1129 on Latin square, 151+22 instead of 1653 on RLFAP, 82+12 instead of 173 on Purdey.

In our last experiment, we show the effect on the performance of G-QuAcq of a lack of knowledge on some variable types. We took again our 5 benchmark problems in which we have varied the amount of types known by the algorithm. This simulates a situation where the user does not know that some variables are from the same type. For instance, in Sudoku, the user could not have noticed that variables are arranged in columns. Figure 2 shows the number of standard queries and generalization queries asked by G-QuAcq with `min_VAR` +cutoff=1 to learn the RLFAP model when fed with an increasingly more accurate knowledge of types. We observe that as soon as a small percentage of types is known (20%), G-QuAcq reduces drastically its number of queries. Table 4 gives the same information for all other problems.

6 CONCLUSION

We have proposed a new technique to make constraint acquisition more efficient by using information on the types of components the variables in the problem represent. We have introduced generalization queries, a new kind of query asked to the user to generalize a constraint to other scopes of variables of the same type where this constraint possibly applies. Our new technique, GenAcq, can be called to generalize each new constraint that is learned by any constraint acquisition system. We have proposed several heuristics and strategies to select the good candidate generalization query. We have plugged GenAcq into the QuAcq constraint acquisition system, leading to the G-QuAcq algorithm. We have experimentally evaluated the benefit of our approach on several benchmark problems, with and without complete knowledge on the types of variables. The results show that G-QuAcq dramatically improves the basic QuAcq algorithm in terms of number of queries.

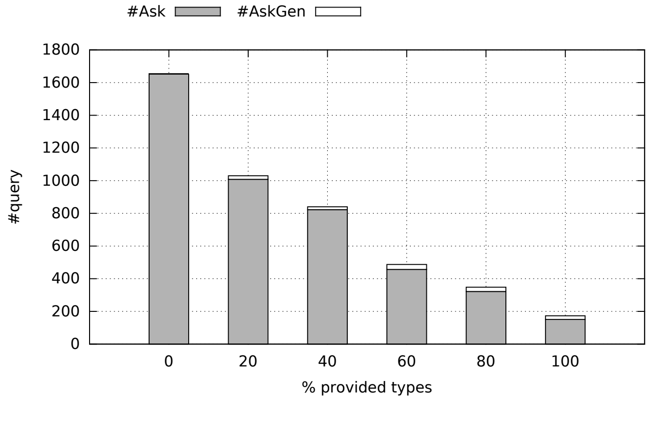

Figure 2. G-QuAcq on RLFAP when the percentage of provided types increases.

REFERENCES

[1] Dana Angluin, 'Queries and concept learning.', *Machine Learning*, 319–342, (1987).

[2] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit, 'Global constraint catalogue: Past, present and future', *Constraints*, **12**(1), 21–62, (2007).

[3] Nicolas Beldiceanu and Helmut Simonis, 'A model seeker: Extracting global constraint models from positive examples.', in *CP*, pp. 141–157, (2012).

[4] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan, 'Leveraging the learning power of examples in automated constraint acquisition', in *CP*, pp. 123–137, (2004).

[5] 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).

[6] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan, 'A sat-based version space algorithm for acquiring constraint satisfaction problems', in *ECML*, pp. 23–34, (2005).

[7] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin, 'Query-driven constraint acquisition.', in *IJCAI*, pp. 50–55, (2007).

[8] Christian Bessiere and Frédéric Koriche, 'Non learnability of constraint networks with membership queries', Technical report, Coconut, Montpellier, France, (February, 2012).

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

[10] 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).

[11] Ulrich Junker, 'Quickxplain: Preferred explanations and relaxations for over-constrained problems', in *AAAI*, pp. 167–172, (2004).

[12] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain, 'On learning constraint problems.', in *ICTAI*, pp. 45–52, (2010).