Solve a Constraint Problem without Modeling It

Christian Bessiere, Remi Coletta, Nadjib Lazaar

Solve a Constraint Problem Without Modeling It

Christian Bessiere, Remi Coletta, Nadjib Lazaar
CNRS, University of Montpellier
Montpellier, France
Email: {bessiere, coletta, lazaar}@lirmm.fr

***Abstract*—We study how to find a solution to a constraint problem without modeling it. Constraint acquisition systems such as *Conacq* or *ModelSeeker* are not able to solve a single instance of a problem because they require positive examples to learn. The recent *QuAcq* algorithm for constraint acquisition does not require positive examples to learn a constraint network. It is thus able to solve a constraint problem without modeling it: we simply exit from *QuAcq* as soon as a complete example is classified as positive by the user. In this paper, we propose ASK&SOLVE, an elicitation-based solver that tries to find the best tradeoff between learning and solving to converge as soon as possible on a solution. We propose several strategies to speed-up ASK&SOLVE. Finally we give an experimental evaluation that shows that our approach improves the state of the art.**

***Keywords*-Elicitation based resolution; Constraint acquisition; Constraint Learning**

I. INTRODUCTION

The success of constraint programming comes from its efficiency to solve combinatorial problems once they are represented as a constraint network. However, the design of the original model, that is, the construction of the set of constraints defining the problem, remains a delicate task that requires some expertise in constraint programming. This is a bottleneck in the use of constraint solvers by non-experts.

There already exist several techniques to tackle this bottleneck. In [8], the matchmaker agent is an interactive process where the user is able to provide one of the constraints of her target problem each time the system proposes an incorrect solution. In [9], a preference elicitation mechanism solves soft constraint problems where some of the preferences are unspecified. The idea is to combine a *branch&bound* search with elicitation steps where candidate solutions are presented to the user who provides missing preferences on the variables. When restricted to hard constraint problems (that is, preferences are either accept or reject), this approach becomes very close to the matchmaker agent, requiring the user to be able to express her constraints one by one. In *Conacq* and *ModelSeeker* the assumption is made that the only thing the user is able to provide is examples of solutions and non-solutions of the target problem or to answer *membership queries*, that is, classify as positive or negative examples proposed by the system [2], [4], [5], [13], [1]. Based on these classified examples, the system learns a set of constraints that correctly classifies all examples given so far. A common feature of these techniques is that they require positive examples, otherwise they can learn the (trivial) inconsistent constraint network. *QuAcq* is a recent learning system that is able to ask the user to classify *partial* queries [3]. As opposed to membership queries, partial queries are assignments of only part of the variables of the problem. An interesting property of *QuAcq* is that it is not necessary that the user provides positive examples to converge. This property opens the door to a new type of use: solve a problem without modeling it even if the user does not have examples of past solutions.

In this paper, we propose ASK&SOLVE, an elicitation-based algorithm that solves an unknown constraint network by asking queries to the user. ASK&SOLVE tries to find the best tradeoff between learning and solving to converge on a solution with as few queries to the user as possible. Based on the ASK&SOLVE algorithm we propose several strategies (restart policies / variable ordering) to speed-up even more convergence on a solution. We experimentally evaluate our approach on several benchmark problems. The results show that ASK&SOLVE improves the basic technique based on *QuAcq*.

The rest of the paper is organized as follows. Section II gives the necessary definitions to understand the technical presentation. Section III describes the basic algorithm ASK&SOLVE. In Section IV, several strategies (restart policies and variable ordering) are presented. Section V presents the experimental results we obtained when comparing ASK&SOLVE to existing techniques and when comparing the different restart policies / variable ordering in ASK&SOLVE. Section VI concludes the paper and gives some directions for future research.

II. 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 $D = \{D(x_i)\}_{x_i \in X}$ over $\mathbb{Z}$. A constraint c represents a relation on a subset of variables $var(c) \subseteq X$ that specifies which assignments of $var(c)$ are allowed. Combinatorial problems are represented with *constraints networks*. A constraint network is a set C of constraints on the vocabulary (X, D). An example e is a (partial) assignment on a set of variables $var(e) \subseteq X$. e is rejected by a constraint 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. The projection $C[Y]$ of C on a subset Y of X is the set $\{c \in C \mid var(c) \subseteq Y\}$.

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 bias* is a collection B of constraints built from the constraint language Γ on the vocabulary (X, D).

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* $Ask(e_Y)$, with $Y \subseteq X$, is a classification question asked to the user, where e_Y is an assignment in $D^Y = \Pi_{X_i \in Y} D(X_i)$. A set of constraints C *accepts* a an assignment e_Y if and only if there does not exist any constraint $c \in C$ rejecting e_Y. The answer to $Ask(e_Y)$ is "yes" if and only if C_T accepts e_Y. For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y.

III. Solve by Asking and Learning

In this section we present Ask&Solve, an algorithm for solving a problem without having a constraint network describing it. The idea in Ask&Solve is to try to extend step by step a scope on which we know at least one assignment accepted by the target network C_T. Each time there is a chance that the assignment generated by Ask&Solve violates one of the constraints in the bias B, Ask&Solve asks the query to the user. If the answer to the query is negative, Ask&Solve immediately launches a procedure that learns a culprit constraint to avoid generating again an assignment rejected for the same reason.

A. Description of Ask&Solve

In lines 1 to 5, Ask&Solve initializes several variables that will be used in the main loop. The learned network C_L and the scope *scp* to be extended during search for solution are initialized to the empty set. The set guilty will contain the new variable to be added to the scope *scp* each time the assignment produced on *scp* was accepted by the user. guilty is also initialized to the empty set. The counter #negative-answers counts the number of queries classified negative by the user. It is initialized to zero. #negative-answers is useless in the basic version of Ask&Solve but it will be used in versions that implement a restart policy. Finally, the flag $Found$, that will break the loop once a solution is found, is set to false.

The main loop in Ask&Solve (line 6) starts by computing an assignment e on scope *scp* (line 7). We look for an assignment satisfying all constraints already learned (in C_L) and hopefully satisfying also all constraints from the bias B that do not involve the new variable guilty. Among the possible candidates, we choose the e that maximizes the number of constraints involving the variable guilty that are satisfied. The reason for trying to satisfy all constraints of B on *scp* is that if e is classified as negative, we know the variable guilty is involved in a violated constraint. The reason for maximizing the remaining constraints from B is that if C_T is representable by the initial bias and if e is classified positive, e is an assignment as close as possible to the solution. However, it is not always possible to generate such an assignment. If not possible (line 8), we compute an assignment that satisfies the current C_L and maximizes the satisfied constraints in B (line 10). Satisfying C_L is always possible by extending the previous example with any variable assigned to any value. We also set guilty to the empty set because we are no longer sure the variable in guilty will belong to a violated constraint in case e is classified negative. If there are constraints in B able to reject e (i.e., $\kappa_B(e) \neq \emptyset$), the query $Ask(e)$ is presented to the user. If her answer is 'no', we know that e violates at least one constraint in the target problem C_T. The functions `FindC` and `FindScope` are called to find such a constraint.

We do not give the code of functions `FindScope` and `FindC`. They are implemented as they appear in [3], and they are instrumented so that we increment the variable #negative-answers each time a query is answered negatively by the user. 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 B that rejects e. Inspired from a technique used in QuickXplain [12], `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.

Now, coming back to line 12, the call to `FindScope` returns the set of variables that together with guilty form the scope of one of the violated constraints. Remember that if e comes from line 7, guilty contains the new variable that belongs to the scope of a violated constraint whereas if e comes from line 10 we are not sure that guilty belongs to a constraint in B violated by e. But this is fine because in this case guilty is empty. Once a scope is returned by `FindScope`, `FindC` returns a constraint from B with that scope that rejects e. The constraint found is added to the learned network C_L (line 14). If `FindC` has not found any constraint in B rejecting e, this means that our initial

Algorithm 1: ASK&SOLVE: Solving an unknown constraint network C_T by asking user queries

```
1  $C_L \leftarrow \emptyset$;
2  $scp \leftarrow \emptyset$;
3  guilty $\leftarrow \emptyset$;
4  #negative-answers $\leftarrow 0$;
5  $Found \leftarrow false$;
6  while $\neg Found$ do
7      select $e$ in $sol(C_L[scp] \cup B[scp \setminus$ guilty$])[scp]$
       maximizing satisfaction of $B[scp]$;
8      if $e = nil$ then
9          guilty $\leftarrow \emptyset$;
10         select $e$ in $sol(C_L[scp])[scp]$ maximizing
           satisfaction of $B[scp]$;
11     if $\kappa_B(e) \neq \emptyset$ and $Ask(e) = no$ then
12         $c \leftarrow$ FindC($e$, FindScope($e$, guilty,
           $scp \setminus$guilty, $false$) $\cup$ guilty);
13         if $c = nil$ then return "collapse";
14         else $C_L \leftarrow C_L \cup \{c\}$;
15         if #negative-answers $\geq restart()$ then
16             $scp \leftarrow \emptyset$; #negative-answers $\leftarrow 0$;
       else
17         #negative-answers $\leftarrow 0$;
18         if $scp = X$ then $Found \leftarrow true$;
           else
19             guilty $\leftarrow$ {a variable in $X \setminus scp$};
20             $scp \leftarrow scp \cup$ guilty;
21             $B \leftarrow B \setminus \kappa_B(e)$;
22 return $e$;
```

bias was not able to represent C_T. ASK&SOLVE returns a collapse (line 13).

We do not describe lines 15-17 as we are in the basic version of the algorithm where $restart()$ always returns $+\infty$. In line 18 we are in the case where the user answered 'yes' to the query. We first test if scp was complete. If yes, this means that e is a solution and we return it (lines 18 and 22). If not, we extend scp with a new variable guilty chosen in the remaining variables (lines 19-20) and we remove all constraints violated by e from the bias (line 21).

IV. STRATEGIES

ASK&SOLVE learns constraints by asking queries during search. Solving an instance by asking as few queries as possible requires to find the good tradeoff between learning and solving. If we try to go fast to a complete assignment, this can lead to expensive constraint elicitation steps on long negative assignments when the problem is critically constrained. If we promote exploring first plenty of short assignments, this allows a fast and cheap learning of constraints, but it can be a waste of time if the problem is easy to solve.

We first analyse the behavior of ASK&SOLVE on a sample problem. Based on this analysis, we propose several strategies of exploration of the search space that try to get the best tradeoff between learning and solving and thus to reach a solution as fast as possible.

A. Analysing ASK&SOLVE *behavior*

We take the well-known zebra problem from Lewis Carroll [7]. The zebra problem has a single solution, which makes it challenging to solve by hand. (It is claimed that only 2% of the population can solve it.) This is thus a good candidate to push our algorithm to the edge. We solved the zebra with our basic version of ASK&SOLVE presented in Section III (that is, with the function $restart()$ in line 15 returning $+\infty$). Figure 1(a) shows how the size scp of the variable assignment evolves during search in ASK&SOLVE (the x-axis is the number of iterations of the while loop in line 6 of ASK&SOLVE). We see that the size of the assignment grows with the number of iterations. This means that, in case of negative answer, the space in which we seek a culprit constraint grows, and thus the cost to find it grows as well. To validate this guess we can observe Figure 1(b). It shows how many queries are asked each time the example generated at the beginning of the while loop of ASK&SOLVE is negative. As the plot is sawtoothed, we drew its approximation of Bezier as interpolation to exhibit the main trend (thick line in Figure 1(b)). We observe that the average number of queries per iteration indeed increases with the number of iterations.

Figure 1 shows that extending too fast the scope on which we look for an assignment leads to a higher cost in number of queries. A technique to avoid remaining on long scopes with long sequences of negative answers is to incorporate a *restart* policy in ASK&SOLVE. The idea is to count how many negative examples were generated in a row and to restart when a cutoff value is reached.

B. Restart policies

Restart policies have been a long-held goal in AI and they are one of the approaches developed to boost combinatorial search. We propose to use three restart policies that triggers a restart in line 16 of Algorithm 1 when the number of negative answers is greater than or equal to the cutoff value returned by $restart()$.

- **Fixed cutoff.** The first policy uses a fixed cutoff, noted (FC). This is done by implementing the function $restart()$ so that it always returns $|X|$.
- **Geometric.** The second policy is the Geometric strategy [16]. The cutoff value returned by $restart()$ grows geometrically by a factor of 1.5. We use an initial cutoff size of $|X|$.
- **Luby.** The third policy we selected is more elaborated. In this case the $restart()$ function implements the universal Luby-restarts policy defined in [14].

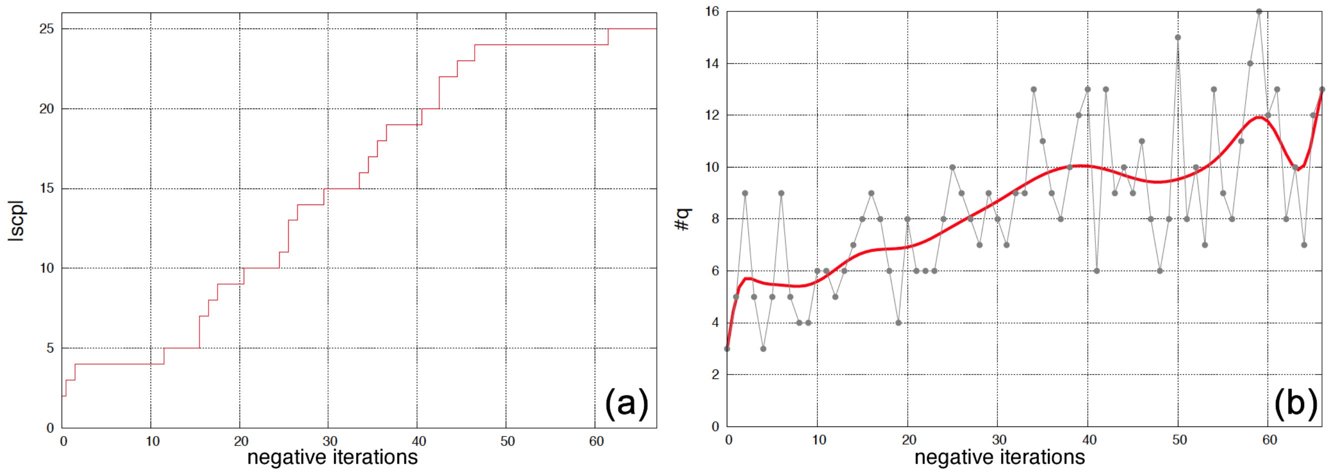

Figure 1: Variation of the size of the examples and of the number of queries per iteration of the while loop in ASK&SOLVE (Zebra problem).

The Luby-restarts policy is given by the sequence $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \ldots)$ that is multiplied by the factor $|X|$. Such a policy has been used with great success in SAT solvers [11]. The goal is, on the one hand to cut the search early enough to escape from bad subtrees and to produce short nogoods, and on the other hand to let the solver the chance to go to a solution. The similarity of this goal and ours led us to try Luby in our setting.

The number #negative-answers of negative answers from the user is reset either when we trigger a restart in line 16, or in line 17 when the user has classified as positive (in line 11) an assignment e computed in line 7 or in line 10.

C. Variable ordering heuristics

When we use a restart policy, the question raises of the order in which to select variables in the search following immediately the restart. We tested and compared the following variable ordering heuristics.

- **random** (RAND)**.** At each restart event, we reorder the variables randomly.
- **lexicographic** (LEX)**.** Here we use the basic lexicographic order. When a restart occurs, we restart on the same variable order:

$$x_1, x_2, x_3 \xrightarrow{\text{restart}} x_1, x_2, x_3, x_4 \ldots$$

- **reverse-lex** (R-LEX)**.** The algorithm is initialized with a LEX order on the variables. Once a restart event occurs, the variables in the scope scp are reversed: we start by selecting the last variable that was added to scp and continue until the first. If we reach the first (that is, we exhausted all variables from that previous scope), we select the remaining variables in the same order as they were ordered before the restart. And so on.

$$x_1, x_2 \xrightarrow{\text{restart}} x_2, x_1, x_3, x_4 \xrightarrow{\text{restart}} x_4, x_3, x_1 \ldots$$

- **countinuous-lex** (C-LEX)**.** Here we use a variant of the lexicographic order. At the beginning we have a LEX order on the variables. Once a restart event occurs, we select the last variable in scp and we continue on the *same* LEX order. (This LEX order has to be circular: x_n is followed by x_1.) This heuristic allows the search process to explore the variables in a balanced way.

$$x_1, x_2, x_3 \xrightarrow{\text{restart}} x_3, x_4 \xrightarrow{\text{restart}} x_4, x_5, x_6 \ldots$$

V. EXPERIMENTAL EVALUATION

We made some experiments to evaluate the performance of our ASK&SOLVE algorithm and the restart policies we proposed. The first part of this section is devoted to the comparison of existing techniques to our baseline version of ASK&SOLVE presented in Algorithm 1 (with $restart() = +\infty$). The second part of the section evaluates the different restart policies and variable orderings we proposed.

Our tests were conducted on an Intel Core i7 @ 2.9 GHz with 8 Gb of RAM. We used the following benchmarks:

- **Golomb Rulers.** (prob006 in [10]) The problem aims at finding a ruler where the distance between any two marks is different from that between any other two marks. The target network is encoded with m variables corresponding to the m marks, and constraints of varying arity. For our experiments, we selected the 8-marks ruler.
- **Zebra Problem.** Lewis Carroll's zebra problem has a single solution. The target network is formulated using 25 variables of domain size of 5 with 5 cliques of $\neq$ constraints and 14 additional constraints given in the description of the problem [7].

- **Purdey's general store.** This problem is described by Jo Mason in [15]. It has a single solution. Four families 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.

A. *Basic* ASK&SOLVE *against other techniques*

We first want to confirm that our ASK&SOLVE architecture goes in a direction that makes it a good candidate for solving unknown constraint problems without having to model them. We compared ASK&SOLVE to three other techniques that are able to solve by asking queries.

The first one that we can use is standard *QuAcq* as presented in [3] that we stop as soon as a solution is found (called QUACQ&SOLVE below).

The second one is a simple backtrack search procedure that asks the user about the validity of the assignment generated at a node of the search tree each time the currently learned constraints do not allow it to infer the answer. If the query is classified as positive we reduce the bias B, otherwise we learn a constraint using the *QuAcq* principle (i.e., we call a `FindScope` and a `FindC` functions on the negative –partial– example). This technique is denoted by BACKTRACK-E.

The last technique, BRANCH&LEARN, is the backtrack search based on elicitation proposed in [6]. The approach is very similar to BACKTRACK-E in the way it explores assignments. However, it differs in the way it learns the constraints. BRANCH&LEARN learns constraints using *Conacq* instead of *QuAcq*. Each time an example e is classified positive, BRANCH&LEARN reduces the bias B as in BACKTRACK-E. But each time e is classified negative, BRANCH&LEARN does not ask anything to the user. It simply stores a disjunction representing the set of candidate constraints in B rejecting e. At least one of them has to be in the target network.

Table I: ASK&SOLVE vs existing elicitation-based solvers

		$\|C_L\|$	$\#q$	time
Golomb	QUACQ&SOLVE	111	548	0.21
	BACKTRACK-E	46	432	0.16
	BRANCH&LEARN	–	389	76.01
	ASK&SOLVE	21	**179**	0.35
Zebra	QUACQ&SOLVE	58	623	0.02
	BACKTRACK-E	51	528	0.06
	BRANCH&LEARN	–	—	—
	ASK&SOLVE	60	**509**	0.02
Purdey	QUACQ&SOLVE	18	157	0.01
	BACKTRACK-E	15	119	0.01
	BRANCH&LEARN	–	109	0.61
	ASK&SOLVE	14	**103**	0.01

Table I displays the comparative performance of QUACQ&SOLVE, BACKTRACK-E, BRANCH&LEARN, and ASK&SOLVE. We report the size $|C_L|$ of the partially learned network (which represents the set of constraints learned during search), the total number $\#q$ of queries, and the average time needed to compute a query (in seconds).

The first observation we can make when comparing QUACQ&SOLVE and BACKTRACK-E is that QUACQ&SOLVE learns more constraints than BACKTRACK-E (column $|C_L|$ in Table I). This is because QUACQ&SOLVE promotes learning constraints and then finds the solution by chance whereas BACKTRACK-E promotes searching and is forced to learn when it fails to extend its assignment. This observation explains why BACKTRACK-E needs less queries than QUACQ&SOLVE.

Concerning BRANCH&LEARN, we cannot report the size of the learned network C_L because it stores a set of disjunctions of constraints without being able to choose a culprit among them. The number of queries needed by BRANCH&LEARN to find a solution is better than that of BACKTRACK-E on the instances it could solve in less than 10 hours. This lower number of queries is due to the lazy strategy used in BRANCH&LEARN: after a negative answer, BRANCH&LEARN does not elucidate a culprit constraint, as done in BACKTRACK-E with the call to `FindScope` and `FindC`. The counterpart to this lazy behavior is that BRANCH&LEARN explores a lot of nodes, leading to prohibitive cpu times when the problem is not small enough. For instance, on Golomb, the average time between two queries to the user is 76 seconds. On Zebra, the whole process had not finished in 10 hours.

Concerning ASK&SOLVE, we see that it is incomparable to QUACQ&SOLVE and BACKTRACK-E in terms of the number of learned constraints. It generates the smallest C_L on Golomb but the largest on Zebra. However, ASK&SOLVE is consistently faster than the other techniques in terms of the number of queries needed to find a solution. This tends to show that ASK&SOLVE promotes the learning side or the solving side depending on the difficulty of the instance to solve, and then reaches a better tradeoff than QUACQ&SOLVE and BACKTRACK-E.

B. *Evaluation of the strategies*

In this section we assess the performance of our restart policies and the variable orderings described in Section IV. To speed-up ASK&SOLVE in terms of number of queries, we implemented the three restart policies:

- Restarting after $|X|$ negative answers (FC).
- The geometric restart policy with an initial restart of $|X|$ negative answers and a growth coefficient of 1.5.
- A luby restart policy with an initial restart of $|X|$ negative answers.

We also implemented the different variable ordering heuristics (RAND, LEX, R-LEX and C-LEX).

Table II displays the comparative performance of all possible strategies on the same three problems as in Section V-A: Golomb, Zebra and Purdey's general store problems. A strategy is a combination (restart policy × variable ordering). Each strategy has a row in Table II.

The first observation that is obvious from Table II is that all techniques are very fast. They can easily be used in an interactive process, as opposed to BRANCH&LEARN, as seen in the experiments in Table I.

The second information we can draw from Table II is that the choice of the order in which variables are selected after a restart can noticeably affect the efficiency of ASK&SOLVE. RAND and LEX are not efficient variable ordering heuristics to be used by our restart policies. This is obvious for RAND, which is almost always much worse than the baseline ASK&SOLVE with no restart policy. This is also true for LEX, to a lesser extent. Among R-LEX and C-LEX, we see that R-LEX wins more often and with greater margins than C-LEX.

Comparing the three restart policies, ASK&SOLVE with Geometric or Luby is significantly better than with FC. This is true on all problems. This confirms what we expected from the observation of Figure 1: increasing the scope too fast and spending time on long scopes is a waste of time when answers are repeatedly negative.

Finally, if we look at the combinations restart policy+variable ordering, the best compromise is Geometric+R-LEX.

When associated with Luby, C-LEX requires less queries on Golomb, but the difference with Luby+R-LEX is tiny: only 2 more queries for Luby+R-LEX.

Luby with C-LEX seems to be the best deal to solve Golomb rulers. This is related to the nature of the Golomb rulers. In Golomb rulers, we have an order on the marks. Using C-LEX, ASK&SOLVE can learn this order with less queries. Furthermore, Golomb rulers contain a set of not-equal constraints on distances. Here also, the use of small restarts as in Luby can speed-up the resolution.

On Zebra (resp. Purdey), Geometric+R-LEX requires 35 (resp. 27) less queries than +C-LEX. As a general conclusion of these experiments, we see that using strategies in ASK&SOLVE improves the performance significantly compared to the baseline ASK&SOLVE without restart. If we compare Geometric+R-LEX to baseline ASK&SOLVE, the saving in number of queries is 7% on Golomb, 32% on Zebra, and 64% on Purdey.

Table II: Various strategies for boosting ASK&SOLVE

	RESTART	VAR-ORDER	$\|C_L\|$	$\#q$	time
Golomb	none	LEX	21	179	0.35
	FC	RANDOM	48	435	0.24
		LEX	21	174	0.34
		R-LEX	30	232	0.30
		C-LEX	28	203	0.35
	Geometric	RANDOM	56	527	0.27
		LEX	21	202	0.33
		R-LEX	21	166	0.28
		C-LEX	21	162	0.31
	Luby	RANDOM	45	402	0.31
		LEX	21	161	0.34
		R-LEX	21	160	0.33
		C-LEX	11	**158**	0.32
Zebra	none	lex	60	509	0.02
	FC	RANDOM	57	560	0.05
		LEX	63	558	0.02
		R-LEX	53	452	0.05
		C-LEX	59	459	0.03
	Geometric	RANDOM	59	503	0.02
		LEX	60	482	0.05
		R-LEX	48	**346**	0.03
		C-LEX	59	381	0.04
	Luby	RANDOM	57	484	0.05
		LEX	60	537	0.03
		R-LEX	41	356	0.03
		C-LEX	57	465	0.02
Purdey	none	LEX	14	103	0.01
	FC	RANDOM	16	106	0.01
		LEX	13	108	0.01
		R-LEX	11	88	0.02
		C-LEX	12	82	0.01
	Geometric	RANDOM	16	99	0.02
		LEX	12	77	0.01
		R-LEX	8	**37**	0.02
		C-LEX	15	64	0.01
	Luby	RANDOM	16	123	0.01
		LEX	12	86	0.01
		R-LEX	9	62	0.02
		C-LEX	11	83	0.01

VI. CONCLUSION

We have proposed ASK&SOLVE, an elicitation based algorithm that solves a constraint problem without the need of a constraint network representing it. The elicitation here consists in asking the user to classify partial assignments as positive or negative. We present several strategies (restart policies and variable orderings) to improve the behavior of ASK&SOLVE. Our experimental evaluation shows that ASK&SOLVE presents a good tradeoff between learning and searching for solutions. It outperforms the other techniques by solving problems with less queries. When enhanced with the right strategy, its performance becomes even better.

These results are promising for the use of ASK&SOLVE on real problems because there exist several other techniques that could be plugged in ASK&SOLVE to decrease even more the number of queries. We could for instance embed *ModelSeeker* as an internal stepofASK&SOLVE to quickly learn global constraints each time we get a partial positive.

ACKNOWLEDGMENT

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

References

[1] N. Beldiceanu and H. Simonis. A model seeker: Extracting global constraint models from positive examples. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'12), LNCS 7514, Springer–Verlag*, pages 141–157, Quebec City, Canada, 2012.

[2] C. Bessiere, R. Coletta, E. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04), LNCS 3258, Springer–Verlag*, pages 123–137, Toronto, Canada, 2004.

[3] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.G. Quimper, and T. Walsh. Constraint acquisition via partial queries. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 475–481, Beijing, China, 2013.

[4] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of the European Conference on Machine Learning (ECML'05), LNAI 3720, Springer–Verlag*, pages 23–34, Porto, Portugal, 2005.

[5] C. Bessiere, R. Coletta, B O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 44–49, Hyderabad, India, 2007.

[6] Christian Bessiere, Remi Coletta, Frdric Koriche, Arnaud Lallouet, and Matthieu Lopez. Branch and learn pour l'acquisition de csp. In *JFPC*, 2012.

[7] L. Carroll. *Life International Magazine*, (December 17th, 1962), december 1962.

[8] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98), LNCS 1520, Springer–Verlag*, pages 192–204, Pisa, Italy, 1998.

[9] M. Gelain, M.S. Pini, F. Rossi, K.B. Venable, and T. Walsh. Elicitation strategies for soft constraint problems with missing preferences: Properties, algorithms and experimental studies. *Artif. Intell.*, 174(3-4):270–294, 2010.

[10] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. http://www.csplib.org/, 1999.

[11] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.

[12] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, San Jose CA, 2004.

[13] A. Lallouet, M. Lopez, L. Martin, and C. Vrain. On learning constraint problems. In *Proceedings of the 22nd IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'10)*, pages 45–52, Arras, France, 2010.

[14] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[15] J. Mason. Purdey's general store. *Dell Magazine*, (April 1997), april 1997.

[16] Toby Walsh. Search in a small world. In *IJCAI*, pages 1172–1177, 1999.