Learning Max-CSPs via Active Constraint Acquisition

Dimosthenis C. Tsouros
Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Kostas Stergiou
Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Abstract

Constraint acquisition can assist non-expert users to model their problems as constraint networks. In active constraint acquisition, this is achieved through an interaction between the learner, who posts examples, and the user who classifies them as solutions or not. Although there has been recent progress in active constraint acquisition, the focus has only been on learning satisfaction problems with hard constraints. In this paper, we deal with the problem of learning soft constraints in optimization problems via active constraint acquisition, specifically in the context of the Max-CSP. Towards this, we first introduce a new type of queries in the context of constraint acquisition, namely *partial preference queries*, and then we present a novel algorithm for learning soft constraints in Max-CSPs, using such queries. We also give some experimental results.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint acquisition, modeling, learning

Digital Object Identifier 10.4230/LIPIcs.CP.2021.54

1 Introduction

Constraint programming (CP) is a powerful paradigm for solving combinatorial problems, with successful applications in various domains. The basic assumption in CP is that the user models the problem and a solver is then used to solve it. One of the major challenges that CP has to deal with is that of efficiently obtaining a good model of a real problem without relying on experts [21, 33, 23, 22]. As a result, automated modeling and constraint learning technologies attract a lot of attention nowadays, and a number of approaches based on Machine Learning have been developed [31, 18, 17].

An area of research in CP towards this direction is that of *constraint acquisition* where the model of a constraint problem is acquired (i.e. learned) using examples of solutions and non-solutions [8, 7, 2, 32]. Constraint Acquisition can be *passive* or *active*. In *passive* acquisition, examples of solutions and non-solutions are provided by the user and based on these examples, the goal is to learn a set of constraints that correctly classifies the given examples [3, 5, 29, 2, 8]. In *active* or *interactive* acquisition the system interacts with an oracle, e.g. a human user, while acquiring the constraint network [24, 6, 37, 8]. State-of-the-art active constraint acquisition systems like QuAcq [4], MQuAcq [42] and MQuAcq-2 [41] use the version space learning paradigm [30], extended for learning constraint networks. They learn the target constraint network by proposing examples to the user to classify them as solutions or not [6, 8, 37]. These questions are called membership queries [1].

Although constraint learning has focused on satisfaction problems, soft constraints, within constraint optimization frameworks such as (weighted) Max-CSP, have also been considered [15, 43, 18, 12] as part of the wider literature on learning preferences [19, 36].

In [34, 9] ML techniques are exploited in order to infer constraint preferences from given solution ratings. Rossi and Sperduti [35] extend these methods so that the scoring function is estimated via an interactive process where high-scoring assignments are posted as queries to the user, who then ranks them according to her preferences. Campigotto et al. [14] consider combinatorial utility functions expressed as weighted combinations of terms.

Techniques for computing minimax optimal decisions have also been developed [10, 11, 44]. [27] uses weighted first-order logical theories to represent constrained optimization problems. Dragone et al. [20] exploits comparison queries in a context where preferences are modeled by individual variables. In [28] MAX-SAT models that can be probably approximately correct (PAC) are learned for combinatorial optimization. Preference elicitation methods for Incomplete Soft Constraint Problems (ISCPs) [25] and Distributed Constraint Optimization Problems (DCOPs) [39, 38] have also been studied.

More closely related to the framework of constraint acquisition are the works of Vu and O'Sullivan [45, 46, 47]. However, all these works concern passive constraint acquisition. In this paper, we deal with the problem of learning Max-CSPs via active constraint acquisition. To the best of our knowledge, there is no study on learning soft constraints in this context. We first introduce a new type of queries in this context, namely *(partial) preference queries*, inspired by works on preference elicitation [16]. Such a query posts two examples to the user and asks her to specify if either of them is preferable or if she is indifferent between the two. We then describe an algorithm that, driven by the user's replies to preference queries, is able to learn all the soft constraints appearing in a Max-CSP. We highlight the differences between learning soft constraints within our proposed framework and standard constraint acquisition of hard constraints and give preliminary experimental results.

2 Background

The *vocabulary* (X, D) is the common knowledge shared by the user and the system. It is a finite set of n variables $X = \{x_1, ..., x_n\}$ and a set of domains $D = \{D(x_1), ..., D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the set of values for x_i.

A *constraint* c is a pair $(\text{rel}(c), \text{var}(c))$, where $\text{var}(c) \subseteq X$ is the *scope* of the constraint, while $\text{rel}(c)$ is a relation between the variables in $\text{var}(c)$ that specifies which of their assignments satisfy c. $|var(c)|$ is called the *arity* of the constraint. A *constraint network* is a set C of constraints on the vocabulary (X, D). A constraint network that contains at most one constraint on each subset of variables (i.e. for each scope) is called *normalized*. Following the literature on constraint acquisition, we will assume that the target constraint network is normalized.

An example e_Y is an assignment on a set of variables $Y \subseteq X$ and it belongs to $D^Y = \prod_{x_i \in Y} D(x_i)$. If $Y = X$, the example e is called a *complete* example. Otherwise, it is called a *partial* example. An example e_Y is rejected (or accepted) by a constraint c iff $\text{var}(c) \subseteq Y$ and the projection $e_{var(c)}$ of e_Y is not in (or is in) $\text{rel}(c)$. A complete assignment that is accepted by all the constraints in C is a solution of C. $sol(C)$ denotes the set of solutions of C. An assignment e_Y is a partial solution of C iff it is not rejected by any constraint in C. Note that such a partial assignment is not necessarily part of a complete solution. $\kappa_C(e_Y)$ denotes the set of constraints in C that reject e_Y, while $\lambda_C(e_Y)$ denotes the set of constraints in C that satisfy e_Y.

Besides the vocabulary, the learner is given a *language* Γ consisting of *bounded arity* constraints. The *constraint bias* B is a set of constraints on the vocabulary (X, D), built using the constraint language Γ. The bias is the set of all candidate constraints from which the system can learn the target constraint network.

In ML, the classification question asking the user to determine if an example e_X is a solution to the problem or not, is called a *membership query* $ASK(e)$. The answer to such a query is positive if e is a solution and negative otherwise. A *partial membership query* $ASK(e_Y)$, with $Y \subset X$, asks the user to determine if $e_Y \in D^Y$ is a partial solution or not. Following the literature on constraint acquisition, we assume that all queries are answered correctly by the user.

In active constraint acquisition, the system iteratively generates a set E of complete or partial examples, which are labelled by the user as positive or negative. A constraint network C agrees with E if C accepts all examples in E labelled as positive and rejects those labelled as negative. The acquisition process has *converged* on the learned network of constraints $C_L \subseteq B$ iff C_L agrees with E and for every other network $C \subseteq B$ that agrees with E, we have $sol(C) = sol(C_L)$.

2.1 Max-CSP

A *Max-CSP* is a quadruple $P = (X, D, Ch, Cs)$, with X being the set of variables, D the set of domains , Ch being the set of *hard* constraints that have to be satisfied mandatorily, and Cs being the set of *soft* constraints whose satisfaction should be maximized, called *soft* constraints. The optimal solution to a Max-CSP maximizes the number of satisfied soft constraints, while satisfying all the hard constraints. In a *weighted Max-CSP* each soft constraint $c_i \in Cs$ is associated with a positive real value (a weight) w_i and the optimal solution maximizes the total sum of the satisfied constraints' weights.

As in learning a standard CSP, it is important to be able to determine whether the version space has converged, or not. If this is indeed the case, the learning system will stop posting queries as the user has an exact characterization of her target problem. But convergence must be defined in a different way compared to the standard case. We now define the target constraint network and the convergence problem in the context of constraint acquisition of soft constraints in Max-CSPs.

▶ **Definition 1.** The *target soft constraint network* Cs_T is the constraint network that correctly states the preferences of the user in the problem she has in mind.

▶ **Definition 2.** Given a bias B being able to represent the target soft constraint network Cs_T, the system has *converged* to Cs_T iff $\forall c \in B$, $Ch_L \models c \vee \exists c' \in Cs_L \mid c' \models c$ w.r.t. Ch_L, with Ch_L and Cs_L being the learned networks of hard and soft constraints respectively.

Hence, the system converges to the target network Cs_T when all the constraints that are still in the bias B are implied by Ch_L or by a constraint we have already learned w.r.t. Ch_L.

3 Partial Preference Queries

In active constraint acquisition, the interaction between the learner and the user is established via membership queries. This process can be used while learning Max-CSPs to acquire any hard constraints that may be present in the problem, but membership queries cannot be used to acquire soft constraints, as such constraints are allowed to be violated in both solutions and non-solutions.

As a result, several other types of queries have been considered in preference learning. For example, the user can be asked to associate a precise desired value to each presented solution [35]. As another example, a comparison query posts two examples to the user and asks her to state which of them she prefers [16, 26]. To be precise, a comparison query posts two complete assignments e_X and e'_X to the user, and the possible answers to such a query are:

1. $e_X \succ e'_X$: the user prefers e_X to e'_X,
2. $e_X \prec e'_X$: the user prefers e'_X to e_X,
3. $e_X \sim e'_X$: the user is indifferent between e_X and e'_X.

Such queries are easier for the user to answer compared to other types used in preference elicitation. We now introduce *partial preference queries* as a variant of comparison queries. Specifically, in a partial preference query, denoted as PrefAsk(), we can have the following cases regarding the two examples included in the query:

1. Both examples are (partial) assignments e_Y, e'_Y over the same set of variables $Y \subseteq X$. In this case, the query is similar to a comparison query, generalized so that the assignments can be partial.
2. One of the examples is e_Y, with $Y \subseteq X$, and the other one is $e_{Y'}$, with $Y' \subset Y$. That is, the second example is a projection of the first one on some of its variables, which means that both examples share the same assignment in the variables in Y', while the first example includes additional variable assignments (the variables in $Y \setminus Y'$). Hence, the answer of the user in this case can either be $e_Y \sim e_{Y'}$ or $e_Y \succ e_{Y'}$.

Let us now demonstrate the use of preference queries through a typical scenario from the literature [14]. Consider a house sales system suggesting candidate houses according to their characteristics. Assume that we have several variables, including the price of the house, its total area, whether it has a garden, whether it has a parking spot, the construction year, etc. Now assume that the preferences of the user are: 1. to have a parking spot, 2. the total area of the house to be $\geq 100m^2$. Now consider a partial preference query consisting of the following examples:

- House #1: Construction year = 2000, parking spot = "yes", garden = "no"
- House #2: Construction year = 2004, parking spot = "no", garden = "no"

In this case the examples in the query are both partial assignments on the same variables, and the user would prefer the first one (i.e. House #1), because it satisfies the requirement to have a parking spot. Now consider the following partial preference query:

- House #1: Construction year = 2000, parking spot = "yes", garden = "no"
- House #2: Construction year = 2000, garden = "no"

This is a case where the second example is a projection on the assignment of the first one. Again, the user would prefer House #1, because it satisfies the requirement to have a parking spot. Hence, it "offers greater satisfaction" of the preferences compared to House #2.

Now assume we have:

- House #3: Construction year = 2000, parking spot = "no", garden = "no"

If the system asks the user to compare House #2 and House #3 the user would answer that she is indifferent, as no additional requirement is satisfied by House #3. This is due to the fact that this type of preference queries is asking the user to state if the additional information offered by House #3 helps satisfy the preferences to a greater degree, and is not a comparison between 2 different examples (i.e. Houses).

A query posted to the user must give the system more information that it already has. So now we define the notion of *informative* queries.

▶ **Definition 3.** *A (partial) preference query q is called irredundant (or informative) iff the answer of the user to q is not predictable. Otherwise, it is called* redundant. *The answer of the user to a query is predictable when the satisfied constraints from Cs_L and B by the two examples imply that $\lambda_{Cs_T}(e) \supset \lambda_{Cs_T}(e')$ or $\lambda_{Cs_T}(e) = \lambda_{Cs_T}(e')$.*

4 Learning Soft Constraints

In this section, we present our proposed approach for learning Max-CSPs. We first detail the differences between the acquisition of soft constraints within our framework and standard active constraint acquisition, and then present our algorithm. We then present our proposed algorithm, PrefAcq, in detail.

4.1 Differences with Constraint Acquisition of hard constraints

In our proposed method the entire network is learned in two separate steps:

1. The hard constraints (if any) are learned via a standard constraint acquisition algorithm. Only membership queries are used in this step, while the soft constraints do not affect the answers of the user.
2. The soft constraints representing the preferences of the user are learned via our proposed algorithm. Only preference queries are used in this step, but as we explain, the examples generated must satisfy the already learned hard constraints.

Although in the context of standard constraint acquisition, learning hard constraints is well defined, some things differ when acquiring soft constraints. Let us first recall how active constraint acquisition algorithms operate. They typically comply with the following generic procedure:

1. Generate an example e_Y in D^Y and post it as a query to the user.
   a. If the answer is positive, update the version space, removing from the bias B the constraints rejecting the example.
   b. If the answer is negative, search for one or more constraints of C_T that reject the example e_Y, via partial membership queries.
2. If not converged, return to step 1.

In more detail, once a generated example e_Y is classified as negative, the system discovers the scope of one of the violated constraints, as follows. It successively decomposes e_Y to a simpler problem by removing entire blocks of variables from the example while posting partial queries to the user. If after the removal of some variables the answer of the user to the partial query posted is "yes", then it has discovered that the removed block contains at least one variable from the scope of a violated constraint. Then the acquisition system focuses on this block. When, after repeatedly removing variables, the size of the considered block is 1, then this variable surely belongs to the scope of a violated constraint. A logarithmic complexity in terms of the number of queries posted to the user is achieved by splitting. In each decomposition step the set of variables is approximately split in half.

Our proposed approach uses a similar technique. We exploit the 2nd type of partial preference queries described above to locate the scope of satisfied soft constraints in a generated example. That is, we repeatedly post a query comparing an example e_Y with its projection on a subset of variables $Y' \subset Y$. The 1st type of partial preference queries is used to find the specific relation of the constraint, after the scope has been located.

Let us now detail the differences between standard learning of hard constraints and learning of soft constraints.

4.1.1 Violation vs. satisfaction of constraints

A main difference is that in standard constraint acquisition, when trying to find a hard constraint via membership queries, it is the violation of constraints that drives the search. This is because the violation of a constraint results in a negative answer by the user, which

forces the system to continue searching. On the other hand, in a preference query, it is the satisfaction of constraints that drives the search process. This is because the preference of one example over another means that more constraints are satisfied in the former compared to the latter, which will force the system to continue searching for these constraints.

4.1.2 Information derived from user answers

In standard constraint acquisition, the procedure to find the scope of one or more violated constraints exploits the fact that the information that is derived from the answer to a membership query $ASK(e_Y)$ concerns the variables in Y, and only them. That is, the answer will inform us about the existence or not of a violated constraint c with $var(c) \subset Y$. In a preference query, when comparing an example e_Y to its projection on a subset of variables $Y' \subset Y$, the answer of the user gives information about the variables in $Y \setminus Y'$. That is, the answer will inform us about the existence or not of a satisfied soft constraint c with $var(c) \subset Y \wedge \exists x \in var(c) \mid x \in Y \setminus Y'$. Hence, if the answer is that the user is indifferent between the examples, any constraint c' with $var(c') \subset Y \wedge \exists x \in var(c') \mid x \in Y \setminus Y'$ has to be removed from B because it certainly does not belong to Cs_T (if it did belong then the user would have preferred e_Y).

4.1.3 Top-down vs. bottom-up

Because of the above, another important difference lies in the algorithmic approach. Standard constraint acquisition algorithms follow a top-down procedure when searching for constraints to learn. They post membership queries to the user while successively decomposing the initial example. As we will explain in the next section, our algorithm for Max-CSPs also performs a top-down decomposition of the initial query, but crucially, no queries are posted while this decomposition takes place. Once this process is finished, having decomposed the query as much as possible, the algorithm continues in a bottom-up fashion, with preference queries being posted to guide the search for satisfied soft constraints.

Example 1 shows the series of queries posted by our method to locate the scope of a constraint.

▶ **Example 1.** Assume that the vocabulary (X, D) given to the system is $X = \{x_1, ..., x_8\}$ and $D = \{D(x_1), ..., D(x_8)\}$ with $D(x_i) = \{1, ..., 8\}$, the target network of soft constraints Cs_T is the set $\{c_{37}, c_{38}\}$ and $B = \{c_{ij} \mid 1 \leq i < j \leq 8\}$, with $|B| = 28$. Also, assume that we have a complete example which satisfies all the constraints in B. Table 1 shows the preference queries posted to the user until the scope of one of the two constraints in Cs_T has been found.

In a process explained below, our method recursively creates sub-examples by splitting the example, approximately in half, until it creates a sub-example with only one variable assignment. Assuming that this sub-example is $e_{\{x_1\}}$, we will now search for a constraint that is satisfied by $e_{\{x_1\}}$. As no constraint c exists in B with $var(c) = \{x_1\}$, we will go back to search in $e_{\{x_1,x_2\}}$, by posting the query PrefAsk($e_{\{x_1\}}$,$e_{\{x_1,x_2\}}$) to the user. As the user will answer that she is indifferent between the two examples (because none of the target constraints is satisfied by them), we will first remove c_{12} from B, as it is definitely not in Cs_T, and then will continue adding variables to the examples and posting queries. The user's answer to the third query will be that the second example is preferable to the first (because both target constraints are satisfied by $e_{\{x_1,...,x_8\}}$). Hence, we find that at least one variable of the scope of a satisfied constraint from Cs_T is in $Y \setminus Y'$, i.e. in $\{x_5, ..., x_8\}$.

Table 1 Searching for a soft constraint via preference queries, in Example 1.

#	Preference query	answer	information we get
1.	$e_{\{x_1\}}$ vs $e_{\{x_1,x_2\}}$	$\sim$	no satisfied constraints, c_{12} removed from B
2.	$e_{\{x_1,x_2\}}$ vs $e_{\{x_1,...,x_4\}}$	$\sim$	no satisfied constraints, $c_{13}, c_{14}, c_{23}, c_{24}$ removed from B
3.	$e_{\{x_1,...,x_4\}}$ vs $e_{\{x_1,...,x_8\}}$	$\prec$	at least one satisfied constraint with $var(c) \subset \{x_1, ..., x_8\} \wedge \exists x \in var(c) \mid x \in \{x_5, ..., x_8\}$
4.	$e_{\{x_1,...,x_4\}}$ vs $e_{\{x_1,...,x_5\}}$	$\sim$	no satisfied constraints here, $c_{15}, c_{25}, c_{35}, c_{45}$ removed from B
5.	$e_{\{x_1,...,x_5\}}$ vs $e_{\{x_1,...,x_6\}}$	$\sim$	no satisfied constraints, $c_{16}, c_{26}, c_{36}, c_{46}, c_{56}$ removed from B
6.	$e_{\{x_1,...,x_6\}}$ vs $e_{\{x_1,...,x_8\}}$	$\prec$	at least one satisfied constraint with $var(c) \subset \{x_1, ..., x_8\} \wedge \exists x \in var(c) \mid x \in \{x_7, x_8\}$
7.	$e_{\{x_1,...,x_6\}}$ vs $e_{\{x_1,...,x_7\}}$	$\prec$	$x_7 \in var(c)$
8.	$e_{\{x_7\}}$ vs $e_{\{x_1,x_7\}}$	$\sim$	no satisfied constraints, c_{17} removed from B
9.	$e_{\{x_1,x_7\}}$ vs $e_{\{x_1,...,x_3,x_7\}}$	$\prec$	at least one satisfied constraint with $var(c) \subset \{x_1, ..., x_3, x_7\} \wedge \exists x \in var(c) \mid x \in \{x_2, x_3\}$
10.	$e_{\{x_1,x_7}vse_{\{x_1,x_2,x_7\}}$	$\sim$	no satisfied constraints, c_{27} removed from B
11.	$e_{\{x_1,x_2,x_7}vse_{\{x_1,...,x_3,x_7\}}$	$\prec$	$x_3 \in var(c)$

Then, trying to discover the complete scope, again we will decompose the projection of the example on this discovered set of variables $\{x_5, ..., x_8\}$ until we reach a sub-example with the fewest variables possible. However, in each query posted now, both examples will include the variables that have already been searched, because one (or more) variable(s) of the sought scope may be among them. For instance, in the 4th query both the examples include the assignments of variables $\{x_1, ..., x_4\}$ in which we have already searched. But although we know that there is no satisfied constraint $c \in Cs_T$ with $var(c) \in \{x_1, ..., x_4\}$, it is possible that one or more variables among $\{x_1, ..., x_4\}$ participate in the sought scope (as is actually the case with both c_{37} and c_{38}).

In query #6 the set of variables to search in is narrowed to $\{x_7, x_8\}$. As we will explain, this query (and query #11) will not be actually posted because it is redundant. We only include it here to make the example easier to understand. With query #7, we will find that x_7 is in the scope of a constraint. Then we will start searching again, with the same reasoning as before, in order to find the remaining variables of the scope, knowing that they are in $\{x_1, ..., x_6\}$.

During this process, each query includes, in both examples, the assignment of the variable in the sought scope that we have already found (i.e. the assignment of x_7). Including the assignment of x_7 means that the answer of the user to the query posted will now depend only on the presence or absence of the other variable of the scope in the two examples. Hence, if x_3 is present in one of the examples and absent from the other, the former example will be preferred. After a few queries we will find scope $\{x_3, x_7\}$ and then we will continue searching for more constraints.

4.2 Description of PrefAcq

PrefAcq (Algorithm 1) is a novel active learning algorithm for soft constraints. It starts by setting the learned network Cs_L equal to the empty set (line 1). Then, it iteratively generates examples (line 3) in which it will search for satisfied soft constraints via the function

SearchSC (line 5) until it detects convergence at line 4. Each example generated must be a solution to the problem, i.e. to satisfy all the hard constraints. Also, it must satisfy at least one of the candidate soft constraints in the version space, i.e. at least one constraint from B, so that the version space is reduced with each generated example.

Algorithm 1 The PrefAcq Algorithm.

Input: Ch, B, X, D (Ch: The set of the hard constraints, B: the bias, X: the set of variables, D: the set of domains)
Output: Cs_L: a constraint network
1: $Cs_L \leftarrow \emptyset$;
2: **while** true **do**
3: Generate e in D^Y, with $Y \subseteq X$, accepted by Ch s.t. $\lambda_B(e_Y) \neq \emptyset$;
4: **if** $e =$ nil **then return** "Cs_L converged";
5: SearchSC(e, $\emptyset$, Y, $\emptyset$, true);

Function *SearchSC* (Algorithm 2) is used to search for satisfied soft constraints in the example generated. It finds *all* the constraints from Cs_T that are satisfied in the example given. It recursively decomposes the example as much as possible, until a sub-example with the minimum number of variables (typically just one) is reached, as in Example 1.

Then *SearchSC* starts the bottom-up search for satisfied constraints that rewinds the recursive decomposition of the initial example, with more variables taken into account step by step. In this way, it exploits the information that can be derived via the preference queries, i.e. for a preference query PrefAsk(e_Y, $e_{Y'}$), with $Y' \subset Y$, the answer of the user will reveal if $\exists c \in Cs_T \mid var(c) \in Y \setminus Y'$. *SearchSC* starts posting queries when the example with minimum number of variables is reached and then goes bottom-up so that in any subsequent query we will already have derived all the information we can in $e_{Y'}$ before we search in $Y \setminus Y'$. For example, in Example 1, when query 3 is posted to the user, we already know that there are no satisfied constraints from Cs_T in $\{x_1, x_2, x_3, x_4\}$.

In more detail, *SearchSC* takes as input an example e, three sets of variables R, Y, S and a Boolean variable ask_query. In each call, S contains variables which we have found to be in the scope of a satisfied constraint, for which we seek the rest of the variables. The set Y is the one in which we will search for satisfied constraints. R contains the variables that *SearchSC* has already searched in previous calls. The Boolean variable ask_query is set to true if a query is needed to be posted and to false if the query may be redundant. True is returned if a constraint has been found and false otherwise. In the first call to *SearchSC* in PrefAcq, we have $R = S = \emptyset$. Also, Y is set to the assigned variables in the example generated and $ask\_query = true$.

First, *SearchSC* initializes the boolean variable $found\_flag$ to false and the set Q to $R \cup S$ (lines 2-3), i.e. the variables in which we have already searched in previous recursive calls (R) and the variables we have found to be in the scope we seek (S). The set Q stores the variables that will be present in both the partial examples posted to be compared by the user. It is empty in the first call. If the projection of the example e on $Q \cup Y$ (i.e. the variables in which we have already searched and the ones we are searching in the current recursive call) does not satisfy more constraints from B than its projection on Q, then there is no point searching for a satisfied constraint with at least one variable of its scope in Y. Hence, false is returned in this case (line 4). Otherwise, at line 5 the set Y is split in two balanced parts, with $|Y_1| = \lfloor |Y|/2 \rfloor$ and then the function recursively calls itself with $Y = Y_1$, reducing the set of variables to be searched. Notice that when splitting Y at line 5 we ensure that if $|Y|$ is not even, Y_2 will contain one more variable than Y_1. This is important because otherwise the algorithm would never terminate because of the recursive call at line 6.

Algorithm 2 SearchSC: Searching for soft constraints.

Input: e, R, Y, S, ask_query (e: the example, R, Y, S: sets of variables, ask_query: a boolean variable)
Output: $found\_flag$: returns true if a constraint is found, false otherwise

1: **function** SearchSC(e, R, Y, S, ask_query)
2: $found\_flag \leftarrow false$;
3: $Q \leftarrow R \cup S$;
4: **if** $\lambda_B(e_Q) = \lambda_B(e_{Q \cup Y})$ **then return** $false$;
5: split Y into $< Y_1, Y_2 >$ such that $|Y_1| = \lfloor |Y|/2 \rfloor$ and $Y_2 = Y \setminus Y_1$;
6: **if** $|Y_1| > 0$ **then** $found\_flag \leftarrow$ SearchSC(e, R, Y_1, S, true);
7: **if** $\exists c \in \lambda_{Cs_L(e_{Q \cup Y})} \mid \exists var(c) \cap Y_2 \neq \emptyset$ **then**
8: $c \leftarrow$ pick random $c \in \lambda_{Cs_L(e_{Q \cup Y})}$ s.t.$var(c) \cap Y_2 \neq \emptyset$;
9: **for each** $x_i \in var(c)$ **do**
10: $found\_flag \leftarrow$ SearchSC(e, R, $Y \setminus \{x_i\}$, $found\_flag \vee ask\_query$) $\vee found\_flag$;
11: **return** $found\_flag$;
12: **if** $\lambda_B(e_{Q \cup Y_1}) = \lambda_B(e_{Q \cup Y})$ **then return** $found\_flag$;
13: $ask\_query \leftarrow (ask\_query \vee found\_flag)$;
14: **if** $ask\_query \wedge$ PrefAsk($e_{Q \cup Y}$, $e_{Q \cup Y_1}$) $= (e_{Q \cup Y} \sim e_{Q \cup Y_1})$ **then**
15: $B \leftarrow B \setminus \lambda_B(e_{Q \cup Y})$;
16: **return** $found\_flag$;
17: **if** $|Y_2| > 1$ **then** SearchSC(e, $R \cup Y_1$, Y_2, S, false);
18: **else**
19: **if** SearchSC(e, $\emptyset$, $R \cup Y_1$, $S \cup Y_2$, true) = false **then**
20: $Cs_L \leftarrow Cs_L \cup$ FindSC(e, $S \cup Y_2$);
21: **return** true;

In case the example $e_{Q \cup Y}$ satisfies constraints that are already in Cs_L, with at least one variable in Y_2, we call *SearchSC* recursively for each subset of Y created by removing one of the variables of the scope of such a constraint (lines 7-11). This is done to ensure soundness, as we now explain.

Ensuring Soundness. Before continuing with the description of the algorithm, let us clarify an important issue. It is possible that when *SearchSC* has focused on a partial example in a set of variables Y and posts a preference query including this example, there may already exist satisfied constraints from Cs_L (i.e. constraints we have already learned) with scopes having at least one variable in Y_2, i.e. in the set of variables in which we search. Consider the running example. After the system finds the constraint c_{37} from example e, it will continue searching. After finding that x_8 is in the scope of a constraint we seek, it will now search in $Y_2 = \{x_1, ..., x_7\}$ for the rest of the variables in the same scope. However, the already learned constraint c_{37} is satisfied in the projection of e on $\{x_1, ..., x_7\}$. This will affect the answers of the user in the subsequent queries and will mislead the algorithm. As a result, instead of learning scope $\{x_3, x_8\}$, it will also include x_7 in the learned scope, which is incorrect, making the algorithm unsound.

To resolve this problem, *SearchSC* does the following: For each satisfied constraint in Cs_L, with $var(c) \subset Y \wedge var(c) \cap Y_2 \neq \emptyset$, it recursively searches in partial examples created by removing one of the variables in $var(c)$. This guarantees that any constraint c', with

$var(c') \subset Y \wedge var(c') \cap Y_2 \neq \emptyset$ that will be learned is indeed in Cs_T. As our assumption is that the constraint network is normalized, at least one variable appears in $var(c)$ but not in $var(c')$, meaning that by removing this variable, the algorithm will be able to search in an example where c is not satisfied while c' is. Therefore, it will be able to learn c', without c affecting the answers of the user to the preference queries posted.

With this method, the system exploits the fact that for any satisfied constraint $c' \in Cs_T$ which we have not already learned, we have $var(c) \setminus var(c') \neq \emptyset$ for every $c \in Cs_L$. This is true because of the assumption that the target constraint network is normalized. In our example, *SearchSC* will search in the two sets of variables created by removing one of the variables in the scope of the learned constraint c_{37}, i.e. $\{x_1, x_2, x_4, ..., x_7\}$ and $\{x_1, ..., x_6\}$. Constraint c_{37} is not satisfied in the projection of e in either of these sets of variables, so eventually the algorithm will discover that x_3 is the other variable it seeks, and learn c_{38}.

We now continue with the description of the algorithm. A preference query is posted to the user at line 14, when the example cannot be simplified any more. The examples compared are $e_{Q \cup Y}$ and $e_{Q \cup Y_1}$, meaning that the information we get from the answer of the user regards the variables in Y_2, which is the set of variables that belong to Y but not to Y_1. As Y_1 is previously given as Y to the recursive call at line 6, we definitely have already searched in there. So, searching now in Y_2, we finish searching in Y. The preference query is posted only if it is not redundant (checks at lines 12,13), e.g. query 6 in the running example will not be actually posted because the answers to previous queries (queries 3-5) imply the user's answer. From query 3 we know that there is at least one satisfied constraint with a variable of its scope in $\{x_5, ..., x_8\}$, and from queries 4-5 we know that this variable is not in $\{x_5, x_6\}$, so it is certain that it is in $\{x_7, x_8\}$, and the relevant query can be avoided. This is what the check in line 13 does, with the Boolean variable ask_query (given as a parameter) denoting whether we know that at least one satisfied constraint exists in Y and $found\_flag$ specifying whether a constraint has been already found in any recursive call until now or not (i.e. if a satisfied constraint was found in Y_1).

If the answer to the query is that the user does not prefer any example, then the satisfied constraints in $B[Q \cup Y]$ are removed from B and false is returned (lines 15,16). Otherwise, if we have reached line 17, we know that Y_2 contains at least one variable from a satisfied constraint from Cs_T that we have not learned yet, because the preference query of type 2 can only have two answers: Either the user is indifferent, or she prefers the example containing more variables. If $|Y_2| > 1$, then *SearchSC* is recursively called with $R = R \cup Y_1$ and $Y = Y_2$, to continue searching in Y_2 (line 17). Notice that ask_query is set to false, because we now know that a constraint certainly exists in Y_2. So, in the next recursive call (where Y_2 will be given as Y) we know that if no constraint is found in any sub-call, then the query does not have to be posted. With ask_query set to false in a recursive call, we know that Y contains at least one variable of the scope. So, if the variable is not found in Y_1, we know it is in Y_2. $found\_flag$ will show us if it is found in Y_1 or not in the check of line 13, as in query 6 of the running example where we know that there is a variable of the scope we seek in $Y = \{\{x_5, ..., x_8\}\}$ (so in this recursive call we have ask_query = false) and no constraint has been found in Y_1 because of queries 4-5. In case $|Y_2| = 1$, we know that it is in the scope of the constraint we seek because the user answered that she prefers the example having Y_2 instantiated. Thus, *SearchSC* is recursively called with $R = \emptyset$, $Y = R \cup Y_1$ and $S = S \cup Y_2$, to search for more variables of the scope in $R \cup Y_1$ (line 19).

If no more variables are discovered at a recursive call, then $S \cup Y_2$ is the scope we seek, so *FindSC* is called to find the specific constraint, which is then added to Cs_L (line 20). Finally, having reached this point means that a constraint has been found either by this call or by a recursive call, so true is returned.

4.2.1 Finding the specific relation

In order to find the specific relation of the constraint sought, in the scope found by *SearchSC*, two functions are used, *FindSC* and *FindSC-2*. *FindSC* is the main function used to find the specific constraint, after a scope has been located, while *FindSC-2* is used under certain circumstances, in case there is any constraint c in Cs_L, with $var(c) \subset S$, that is satisfied by the examples posted to the user, affecting her answers and preventing *FindSC* from learning the specific relation. If this is the case, the constraint may not be learned via the main loop of *FindSC*, so another technique is used in *FindSC-2*.

FindSC (Algorithm 3) takes as parameters e and S, where e is the example in which *SearchSC* located the scope of a satisfied constraint from Cs_T, and S is that scope. It posts partial preference queries of the 1st type to find the specific relation. It returns the soft constraint found to be in Cs_T. The main idea is to compare example e to an example e' that satisfies at least one candidate constraint that e also satisfies, but not all such constraints. In this way, we can shrink the set with the candidate constraints after each query.

Algorithm 3 FindSC:

Input: e, S (e: the example, S: the scope of the soft constraint we seek)
Output: c: the constraint found

```
1:  function FindSC(e, S)
2:      Δ ← {c ∈ B | var(c) = S};
3:      B ← B \ Δ;
4:      Δ ← λ_Δ(e_S);
5:      while true do
6:          Generate e' in D^S accepted by Ch, s.t. λ_{Cs_L}(e') = λ_{Cs_L}(e) ∧ λ_Δ(e') ≠ λ_Δ(e) ∧
        λ_Δ(e') ≠ ∅;
7:          if e' ≠ nil then
8:              answer ← PrefAsk(e', e);
9:              if answer = (e ≻ e') then Δ ← κ_Δ(e');
10:             else
11:                 found ← false;
12:                 if ∃c ∈ λ_B(e') | var(c) ⊂ S then
13:                     for each x_i ∈ S do
14:                         found ← SearchSC(e, ∅, S \ {x_i}, true ) ∨found_flag;
15:                 if found = false then
16:                     Δ ← λ_Δ(e');
17:         else break;
18:     if ∃c ∈ Cs_L | var(c) ⊂ S ∧ |Δ| > 1 then Δ ← FindSC-2(S, Δ);
19:     pick random c ∈ Δ; return c;
```

FindSC first initializes the set Δ to the candidate constraints, i.e. the constraints from B with scope S that are satisfied by e, and removes them from B (lines 2-4) because after the constraint is found no other constraint with scope S can exist in Cs_T, given our normalization assumption. In line 5, *FindSC* enters its main loop in which it posts preference queries to the user. In line 6, a partial example e' is generated, to be compared to e_S, that is accepted by Ch and satisfies fewer constraints from Δ than e_S, but at least one. On the way example e'_S is generated, if there is no satisfied constraint $c \in Cs_T$ with $var(c) \subset S$ that will affect the answer of the user, then whatever the answer of the user is, at least one candidate

constraint will be eliminated. This is because if the user is indifferent between the examples, the satisfied constraints from e_S that are not satisfied by e'_S are removed (line 16), while if the user prefers e_S, the satisfied constraints by e'_S cannot contain the one we seek, so they are removed (line 9).

In order to make sure that there is no satisfied constraint $c \in Cs_T$ with $var(c) \subset S$ in the examples e_S and e'_S that will affect the answer of the user on which example is preferable, the system tries to generate an example that satisfies the same constraints in Cs_L as e_S. However, there are two cases that this may not be possible and special handling is required.

First, when the generated example e'_S may satisfy a not yet learned constraint in Cs_T. To handle this, if there are candidate constraints from B, with a scope subset of S, that are satisfied by e'_S (line 12), *FindSC* checks if any of them is in the target network, in lines 13,14. This is done, by calling *SearchSC* in all the examples $e_{S\setminus x_i}$, $\forall x_i \in S$, in a technique similar to the one used in lines 6-10 of *SearchSC*. If at least one constraint is found to be in the target network, the system returns on generating a new example in line 6, as no useful information can be derived from the answers of the user with the current example.

The second case is when there exists an already learned constraint c (i.e. in Cs_L), with $var(c) \subset S$. This may affect the answers of the user, preventing *FindSC* from generating an example e'_S with the necessary properties, and thus making it not possible to learn the specific relation. Specifically, the reason that such an example may not be generated in line 6, although all the constraints in Δ are not yet equivalent w.r.t Ch, is the existence of constraint(s) $c \in Cs_L$ with a scope $var(c) \subset S$, that may not allow the generation of an example e' accepted by Ch, with $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e) \wedge \lambda_\Delta(e') \neq \lambda_\Delta(e) \wedge \lambda_\Delta(e') \neq \emptyset$.

Let us give an example: Assume that $\Delta = \{c1, c2\}$, and thus $e \in sol(c1) \cap sol(c2)$. Also, we have constraints $c3, c4 \in Cs_L$ with $var(c3), var(c4) \subset S$ and we have $sol(c3) = sol(c1) \setminus sol(c2)$ and $sol(c4) = sol(c2) \setminus sol(c1)$

In this case, if we want to generate an example e' with $\lambda_\Delta(e') \neq \lambda_\Delta(e) \wedge \lambda_\Delta(e') \neq \emptyset$, e' will satisfy either $c1$ or $c2$, and this will affect the user preferences in the query in line 8, which makes it not possible to use the answer to find which of $c1, c2$ is in Cs_T. So, in order to have an informative query in line 8, we must be able to generate an example e' s.t. $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e)$ along with the rest of the properties, which we cannot in this case.

Also, another case is where there may be a constraint $c \in Cs_L$ accepted by e that cannot by accepted by any example e' with the desired properties.

To deal with such cases, function *FindSC-2* is called in line 18, to find the constraint we seek, or to prove that all constraints in Δ are equivalent w.r.t the hard constraints.

FindSC-2 (Algorithm 4) has two main loops, to cover all different cases where a constraint with a scope subset of S exists in Cs_L. The first loop is in lines 2-16 and the second loop is in lines 17-31. Each one deals with different cases. They both remove constraints from Δ that provably cannot be in Cs_T. *FindSC-2* returns the final Δ with all constraints being equivalent w.r.t. the hard constraints of the problem.

Algorithm 4 FindSC-2:

Input: S, Δ (S: the scope of the soft constraint we seek, Δ: the set containing the constraints that may belong to Cs_T)
Output: Δ: the set containing the constraints that may belong to Cs_T

1: **function** FindSC-2(S, Δ)
2: **while** true **do**
3: Generate e, e' in D^S accepted by Ch, s.t. $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e) \wedge \lambda_\Delta(e') \neq \lambda_\Delta(e) \wedge \lambda_\Delta(e'), \lambda_\Delta(e) \neq \emptyset$;
4: **if** $e' = nil \vee e = nil$ **then** break;
5: $found \leftarrow false$;
6: **if** $\exists c \in \lambda_B(e) \mid var(c) \subset S$ **then**
7: **for each** $x_i \in S$ **do**
8: $found \leftarrow$ SearchSC(e, $\emptyset$, $S \setminus \{x_i\}$, true) $\vee found\_flag$;
9: **if** $\exists c \in \lambda_B(e') \mid var(c) \subset S$ **then**
10: **for each** $x_i \in S$ **do**
11: $found \leftarrow$ SearchSC(e', $\emptyset$, $S \setminus \{x_i\}$, true) $\vee found\_flag$;
12: **if** $found = false$ **then**
13: $answer \leftarrow PrefAsk(e', e)$;
14: **if** answer $= (e \sim e')$ **then** $\Delta \leftarrow \Delta \setminus ((\lambda_\Delta(e) \cup \lambda_\Delta(e')) \setminus (\lambda_\Delta(e) \cap \lambda_\Delta(e'))$;
15: **else if** answer $= (e \succ e')$ **then** $\Delta \leftarrow \lambda_\Delta(e) \setminus \lambda_\Delta(e')$;
16: **else** $\Delta \leftarrow \lambda_\Delta(e') \setminus \lambda_\Delta(e)$;
17: **while** true **do**
18: Generate e in D^S accepted by Ch, s.t. $\emptyset \subset \lambda_\Delta(e) \subset \Delta \wedge \lambda_{Cs_L}(e) \neq \emptyset$;
19: **if** $e \neq nil$ **then**
20: $answer \leftarrow PrefAsk(e_S, e_{var(\lambda_{Cs_L}(e))})$;
21: **if** answer $= (e_S \sim e_{var(\lambda_{Cs_L}(e))})$ **then** $\Delta \leftarrow \kappa_\Delta(e_S)$;
22: **else**
23: $found \leftarrow false$;
24: **if** $\exists c' \in \lambda_B(e) \mid var(c') \subset S \wedge var(c') \supset var(c)$ **then**
25: **for each** $x_i \in S$ **do**
26: $found \leftarrow$ SearchSC(e, $\emptyset$, $S \setminus \{x_i\}$, true) $\vee found\_flag$;
27: **if** $found = false$ **then**
28: $\Delta \leftarrow \lambda_\Delta(e)$;
29: **else** break;
return Δ;

In more details, in the first loop, a partial example e_S is generated (line 3), that is accepted by Ch and satisfies fewer constraints from Δ than e_S, but at least one.

The main idea is the following. The system generates two new examples e, e' s.t. each one satisfies a different *subset* of Δ and the same subset of Cs_L. Next, these examples are compared, eliminating parts from Δ from the candidate constraints in a repetitive process, depending on the answer of the user on which example is preferable. If one is preferable then only the constraints satisfied only by that example stay in Δ. If the user is indifferent between the two, then the examples satisfied by one example and not by the other are removed from Δ.

If the system cannot generate such a pair of examples, because e.g. all examples satisfying a (different) subset of Δ also satisfy a (different) subset of Cs_L, then it tries to generate a new example e_S, satisfying at least one constraint from Δ but not all, satisfying also a constraint

$c \in Cs_L$ with $var(c) \subset S$, if one exists. Then it posts a preference query comparing the example e_S with its projection in $var(c)$, in order to find out if the constraint(s) in Δ that are satisfied are in the target network. If the user is indifferent between the examples, it means that the constraints in Δ satisfied by e_S are not in the target network and are removed. Otherwise, only these constraints remain in Δ.

If no example can be generated, then the system can return randomly a constraint from Δ in line 19 of *FindSC*, because they are all equivalent w.r.t. Ch.

5 Experimental Evaluation

We first detail the experimental setting.

- Experiments were run on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 16 GB of RAM
- The max_B heuristic [42] was used for query generation, altered to fit the context of soft constraints. max_B normally focuses on examples violating as many constraints from B as possible. But it now focuses on examples satisfying as many constraints from B as possible. The best such example (according to max_B) found within 1 second is returned, even if not proved optimal. Also, *bdeg* was used for variable ordering. The variable appearing in the most constraints in B is chosen [40]. Random value ordering was used.
- We evaluated our algorithm in the extreme case where Cs_L is initially empty, meaning that we have no background knowledge. This results in a large number of queries. However, in real applications it is common to have background knowledge and use it e.g. by giving a frame of basic constraints.
- We evaluate our algorithm on learning the soft constraints, while hard constraints are given by the user. If the hard constraints are unknown too, a constraint acquisition algorithm like QuAcq [4], MQuAcq [42] or MQuAcq-2 [41] can be used to learn them before using PrefAcq to learn the soft constraints.
- We measure the size of the learned network Cs_L, the total number of queries $\#queries$, the average time per query $\bar{T}$ and the total cpu time of the acquisition process T. The time measured is in seconds. PrefAcq was run 5 times on each benchmark. The means are presented along with the standard deviation .

We used the following benchmarks:

Random. We generated two classes of random Max-CSPs with 50 variables and domains of size 10. The first instance consists of 12 hard and 100 soft constraints, while the second consists of 122 hard and 30 soft constraints. All the hard ones are $\neq$ constraints, while the soft are among $\{\neq, >, <\}$. The bias was initialized with 7,350 constraints, using the language $\Gamma = \{=, \neq, >, <, \geq, \leq\}$.

Radio Link Frequency Assignment Problem. The RLFAP is the problem of providing communication channels from limited spectral resources [13]. We used a simplified version of the RLFAP [13], with 50 variables having domains of size 15. The target network contains 100 hard and 25 soft constraints. B contains 12,250 constraints from the language of 2 distance constraints ($\{|x_i - x_j| > y, |x_i - x_j| = y\}$) with 5 different possible values for y.

Exam Timetabling Problem. We used a simplified version of the exam timetabling problem from the Elect. Eng. Dept. of UOWM, Greece. We considered 24 courses, and 2 weeks of exams, meaning that there are 10 possible days for each course to be assigned. We assumed

that there are 3 timeslots in each day. This resulted in a model with 24 variables and domains of size 30. There are hard $\neq$ constraints between any two courses, assuming that only one course is examined during each time slot. Also, hard constraints prohibit courses of the same semester being examined on the same day. 30 soft constraints from the language $\Gamma = \{\neq, >, <, |x_i - x_j| > y, |x_i - x_j| < y\}$, capture the lecturers' and the students' preferences about the examination of specific courses. We built a bias with 5,796 constraints from the language($\Gamma = \{\neq, =, >, <, \geq, \leq, |x_i - x_j| > y, |x_i - x_j| < y, ||\lfloor x_i/3 \rfloor - \lfloor x_j/3 \rfloor| > y\}$) with 5 different possible values for y. This resulted in a bias containing 5,796 constraints in total.

Note that although all the benchmarks we used are binary (i.e. their constraint network consists of binary constraints only), our proposed method works of constraints of any arity, as long as it is bounded, as mentioned in Section 2. However, it does not work on global constraints, but neither does any active constraint acquisition algorithm, as they are unbounded, which means the bias should have an exponential size on the number of variables of the problem.

Table 2 Results of PrefAcq.

Benchmark	$\lvert C_s L \rvert$	$\#q$	$\bar{T}$	T_{total}
Random122-30	30 ± 0	859 ± 25	0.03 ± 0.001	27.29 ± 0.78
Random12-100	100 ± 0	2234 ± 65	0.02 ± 0.001	39.43 ± 1.06
RLFAP	25 ± 0	621 ± 26	0.33 ± 0.02	209.26 ± 12.56
Exam TT	30 ± 0	751 ± 13	0.19 ± 0.02	146.54 ± 16.30

Table 2 presents the results of PrefAcq. We see that the number of queries is proportional to the number of constraints learned. Also, comparing the two random problems, we see that although the number of queries increases when learning more soft constraints, the average time between two queries is about the same. This is because the number of queries depends only on the number of soft constraints we have to learn and on the size of B, while the waiting time depends on the time taken for query generation. As both random problems are easy to solve, queries are generated very fast. In contrast, in the RLFAP and the Exam Timetabling problem (denoted as Exam TT), which are harder to solve, the system takes more time to generate examples that do not violate any hard constraints and satisfy at least one $c \in B$, at line 3 of PrefAcq. But still, the average waiting time for the user is under 1 second.

6 Conclusion

We have extended the framework of active constraint acquisition to the learning of soft constraints in Max-CSPs. Based on a type of query used in preference elicitation, we introduced partial preference queries. Then we presented PrefAcq, a novel algorithm for learning soft constraints in Max-CSPs using such queries. Finally, we give some preliminary experimental results. Our method can be extended to weighted Max-CSP if PrefAcq is used to learn the constraints and a method such as the one in [35] is then used the learn their weights. The existence of weights in constraints does not affect the procedure followed by our method.

References

1 Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.
2 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, pages 141–157. Springer, 2012.

3 Christian Bessiere, Remi Coletta, Eugene C Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 123–137. Springer, 2004.

4 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Toby Walsh, et al. Constraint acquisition via partial queries. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 13, pages 475–481, 2013.

5 Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *European Conference on Machine Learning*, pages 23–34. Springer, 2005.

6 Christian Bessiere, Remi Coletta, Barry O'Sullivan, Mathias Paulin, et al. Query-driven constraint acquisition. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 50–55, 2007.

7 Christian Bessiere, Abderrazak Daoudi, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Younes Mechqrane, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. New approaches to constraint acquisition. In *Data mining and constraint programming*, pages 51–76. Springer, 2016.

8 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

9 Alessandro Biso, Francesca Rossi, and Alessandro Sperduti. Experimental results on learning soft constraints. *KR*, 2:435–444, 2000.

10 Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Regret-based utility elicitation in constraint-based decision problems. In *IJCAI*, volume 9, pages 929–934, 2005.

11 Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Constraint-based optimization and utility elicitation using the minimax decision criterion. *Artificial Intelligence*, 170(8-9):686–713, 2006.

12 Céline Brouard, Simon de Givry, and Thomas Schiex. Pushing data into cp models using graphical model learning and solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 811–827. Springer, 2020.

13 Bertrand Cabon, Simon De Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

14 Paolo Campigotto, Andrea Passerini, and Roberto Battiti. Active learning of combinatorial features for interactive optimization. In *International Conference on Learning and Intelligent Optimization*, pages 336–350. Springer, 2011.

15 Li Chen and Pearl Pu. Survey of preference elicitation methods. Technical report, EPFL, 2004.

16 Vincent Conitzer. Eliciting single-peaked preferences using comparison queries. *Journal of Artificial Intelligence Research*, 35:161–191, 2009.

17 Luc De Raedt, Anton Dries, Tias Guns, and Christian Bessiere. Learning constraint satisfaction problems: An ilp perspective. In *Data Mining and Constraint Programming*, pages 96–112. Springer, 2016.

18 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

19 Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in ai: An overview, 2011.

20 Paolo Dragone, Stefano Teso, and Andrea Passerini. Constructive preference elicitation. *Frontiers in Robotics and AI*, 4:71, 2018.

21 Eugene C Freuder. Modeling: the final frontier. In *The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London*, pages 15–21, 1999.

22 Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

23 Eugene C Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014.

24 Eugene C Freuder and Richard J Wallace. Suggestion strategies for constraint-based matchmaker agents. In *International Conference on Principles and Practice of Constraint Programming*, pages 192–204. Springer, 1998.

25 Mirco Gelain, Maria Silvia Pini, Francesca Rossi, K Brent Venable, and Toby Walsh. Elicitation strategies for soft constraint problems with missing preferences: Properties, algorithms and experimental studies. *Artificial Intelligence*, 174(3-4):270–294, 2010.

26 Shengbo Guo and Scott Sanner. Real-time multiattribute bayesian preference elicitation with pairwise comparison queries. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 289–296, 2010.

27 Samuel M Kolb. Learning constraints and optimization criteria. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

28 Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. Learning max-sat from contextual examples for combinatorial optimisation. In *AAAI*, pages 4493–4500, 2020.

29 Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 45–52. IEEE, 2010.

30 Tom Michael Mitchell. Version spaces: an approach to concept learning. Technical report, Stanford Univ Calif Dept of Computer Science, 1978.

31 Barry O'Sullivan. Automated modelling and solving in constraint programming. In *AAAI Conference on Artificial Intelligence*, pages 1493–1497, 2010.

32 Steven D Prestwich. Robust constraint acquisition by sequential analysis. *Frontiers in Artificial Intelligence and Applications*, 325:355–362, 2020.

33 Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In *International Conference on Principles and Practice of Constraint Programming*, pages 5–8. Springer, 2004.

34 Francesca Rossi and Alessandro Sperduti. Learning solution preferences in constraint problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 10(1):103–116, 1998.

35 Francesca Rossi and Allesandro Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.

36 Francesca Rossi, Kristen Brent Venable, and Toby Walsh. Preferences in constraint satisfaction and optimization. *AI magazine*, 29(4):58–58, 2008.

37 Kostyantyn Shchekotykhin and Gerhard Friedrich. Argumentation based constraint acquisition. In *Ninth IEEE International Conference on Data Mining*, pages 476–482. IEEE, 2009.

38 Atena M Tabakhi. Preference elicitation in dcops for scheduling devices in smart buildings. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

39 Atena M Tabakhi, Tiep Le, Ferdinando Fioretto, and William Yeoh. Preference elicitation for dcops. In *International Conference on Principles and Practice of Constraint Programming*, pages 278–296. Springer, 2017.

40 Dimosthenis C Tsouros and Kostas Stergiou. Efficient multiple constraint acquisition. *Constraints*, 25(3):180–225, 2020.

41 Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 709–725. Springer, 2019.

42 Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *24th International Conference on Principles and Practice of Constraint Programming*, 2018.

43 Paolo Viappiani. Preference modeling and preference elicitation: An overview. In *DMRS*, pages 19–24, 2014.

44 Paolo Viappiani and Christian Kroer. Robust optimization of recommendation sets with the maximin utility criterion. In *International Conference on Algorithmic DecisionTheory*, pages 411–424. Springer, 2013.

45 Xuan-Ha Vu and Barry O'Sullivan. Semiring-based constraint acquisition. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, volume 1, pages 251–258. IEEE, 2007.

46 Xuan-Ha Vu and Barry O'Sullivan. Generalized constraint acquisition. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 411–412. Springer, 2007.

47 Xuan-Ha Vu and Barry O'Sullivan. A unifying framework for generalized constraint acquisition. *International Journal on Artificial Intelligence Tools*, 17(05):803–833, 2008.