Learning Constraints and Optimization Criteria

Samuel Kolb
KU Leuven, Leuven, Belgium
samuel.kolb@cs.kuleuven.be

Abstract

While there exist several approaches in the constraint programming community to learn a constraint theory, few of them have considered the learning of constraint optimization problems. To alleviate this situation, we introduce an initial approach to learning first-order weighted MAX-SAT theories. It employs inductive logic programming techniques to learn a set of first-order clauses and then uses preference learning techniques to learn the weights of the clauses. In order to learn these weighted clauses, the clausal optimization system uses examples of possible worlds and a set of preferences that state what examples are preferred over others. The technique is also empirically evaluated on a number of examples.

Introduction

An important class of declarative models is concerned with (discrete) constrained optimization problems. These combine constraint satisfaction problems with an optimization function that specifies which solutions are optimal. While these models are powerful and elegant, they are often also hard to obtain. (Puget 2004) shows that the task of modeling constraint satisfaction problems is difficult, certainly for non-experts. This has motivated several researchers to investigate whether such models can be learned.

Learning constraints has been addressed in constraint programming (e.g. (Beldiceanu and Simonis 2012), (Bessiere et al. 2013)) and inductive logic programming (e.g. (De Raedt and Dehaspe 1997)). Recent efforts (Lallouet et al. 2010) have attempted to combine these two fields. Furthermore, (Campigotto, Passerini, and Battiti 2011)) have learned propositional MAX-SAT theories from examples together with their target value. Also, work in statistical relational learning (SRL) on learning the structure of Markov Logic networks (MLN) or other SRL models is relevant, as MAP inference can be used to query for the most likely state.

We use weighted first-order logical theories to represent constraint optimization problems. The weights will then capture the optimization function, and (first-order) weighted MAX-SAT can be used to determine the *best* solutions. Furthermore, we shall learn such models by combining inductive logic programming (ILP) principles (such as clausal discovery (De Raedt and Dehaspe 1997)) with preference learning. ILP will be used to learn the clausal theories, and preference learning to determine the weights. Therefore, users will provide examples of possible worlds and a set of preferences that specify which examples are preferred over other ones. This setting extends traditional constraint learning and ILP approaches with the ability to learn soft constraint and to integrate constraint satisfaction and optimization. It extends the approach of (Campigotto, Passerini, and Battiti 2011) as first-order theories are learned and preferences are used rather than from examples with target values. Moreover, it differs the SRL and MLN approaches in that one learns from preferences rather than from examples.

This paper is organized as follows. Section 2 will explain some background knowledge and related work. In Section 3 we shall introduce the formalism and the setting and in Section 4 the algorithm and the clausal optimization system to solve this problem. In Section 5 we shall report on some experiments, and finally, in Section 6, we conclude.

Background

In this section a brief overview of the important concepts and relevant literature will be provided.

Clauses In order to reason about clauses, the standard notions of (function free) first-order logic are described. An atom $p(t_1, t_2, ..., t_n)$ consists of a predicate symbol p that is applied to n terms t_i. Terms are either constants or variables. In this paper constants will be written in uppercase ($John$) and variables in lowercase (x). A literal can be an atom a (positive literal) or a negated atom $\neg a$ (negative literal). Clauses are disjunctions of literals $l_1 \vee l_2 \vee ... \vee l_k$ and are assumed to be universally quantified. A clause can be expressed using a body and a head $head \leftarrow body$. Hereby, positive literals are grouped in the head and negative literals

are grouped in the body. Sometimes clauses are also written as a set of literals $\{l_1, l_2, ..., l_k\}$.

Atoms are grounded if they only contain constants. A Herbrand interpretation is a set of ground atoms. All atoms in the interpretation are assumed to be true and all other possible ground atoms are assumed to be false. $\mathcal{I} \models c$ is used to denote that interpretation $\mathcal{I}$ satisfies the clause c.

Optimization The Boolean Satisfiability Problem (SAT) attempts to determine whether a propositional formula can be satisfied. Formulas can be rewritten into a special form called the Conjunctive Normal Form (CNF), which consists of a conjunction of clauses (disjunctions). An extension of this problem is the Maximum Satisfiability Problem (MAX-SAT), which is an optimization problem and attempts to maximize the amount of satisfied clauses. By assigning positive integer weights to clauses and maximizing the sum of the weights of satisfied clauses, one obtains the weighted MAX-SAT problem.

Related work (Campigotto, Passerini, and Battiti 2011) has creates a weighted MAX-SAT problem by learning propositional clauses and weights using examples and absolute scores. Inspired by their results, our research tries to learn first-order logic clauses and weights using examples and rankings. The resulting optimization problem can be seen as an extension of weighted MAX-SAT in which the clauses are in first-order logic.

Learning constraints Constraint learning is a hard problem. The search space of constraints is typically very large and, unlike in typical machine learning problems, there are usually only few examples to learn from.

Different systems attempt to learn constraints using different methods. The problem of having little examples can be alleviated by interactively generating data. Both the systems Conacq2 (Bessiere et al. 2007) en QuAcq (Bessiere et al. 2013) generate (partial) examples and query the user about their validity.

ModelSeeker (Beldiceanu and Simonis 2012) attempts to structure the problem variables in different ways. The system then tests for different global constraints whether they hold within this structure. This approach allows for constraint learning from a few examples.

As mentioned earlier, the research of (Lallouet et al. 2010) already tried to use ILP techniques to learn constraints. In their approach, the search space of constraints is explored in a bidirectional manner, primarily using negative examples.

Clausal Discovery The clausal discovery algorithm (De Raedt and Dehaspe 1997) attempts to learn first-order logic clauses from positive examples. Using a refinement operator, the search space is traversed by incrementally generalizing clauses until they cover all examples, starting from the empty clause ($\square$). The result set consists of the most specific clauses covering all examples.

Problem Statement

The research presented in this paper aims to learn constraints and optimization criteria based on positive examples and user preferences. Since it aims to simplify the process of generating formal representations it should be easy to use and require input that a user can reasonably be expected to provide.

Constraint learning

Given a set of examples and a limit t, the goal is to find maximally specific clauses that are satisfied by at least t of the given examples.

Examples are Herbrand interpretations. They consist of set of constants, the domain, and all ground atoms over these constants that are true. The learned clauses are domain-independent and contain only variables.

Learning optimization criteria

Given a set of examples $\mathcal{E}$ and a set of preferences $\mathcal{P}$ over these examples, the goal is to learn soft constraints and weights such that the order described by these optimization criteria maximally corresponds with $\mathcal{P}$.

Every preference $p \in \mathcal{P}$ describes a relative order $e_1 \succ e_2 \succ ... \succ e_n$ over a subset of examples $\{e_1, e_2, ..., e_n\} \subseteq \mathcal{E}$. A preference $e_1 \succ e_2$ indicates that out of examples e_1 and e_2 the user prefers e_1.

The soft constraints consist of clauses $\mathcal{C}$ that are satisfied by some but not all of the examples. They can be identified using a system that implements the constraint learning task. The weighted clauses $\mathcal{W}$ are tuples $\mathbb{R} \times \mathcal{C}$ that can be used to calculate a score for any example e:

$$score(e) = \sum_{(w,c) \in \mathcal{W}} w \cdot \mathbf{1}_{e \models c}$$

Example 1 (Moving)**.** "Rewrite" Consider the scenario moving to a new city. Predicates *live_in*, *work_in* and *school_in* indicate the choice of housing, job and school. There might be different areas in the city which

The user has to choose a job, a place to live and a school. There are multiple job offers, houses and schools available across different areas of the city. User provided examples include a choice of where to work, where to live and what school is chosen. By comparing and ranking some of the examples, the user can express his preferences. Weighted soft-constraints can now be learned and used to identify the best choice. In this case, a possible soft constraint could be that the school and work place are in the same area. The weights that have been learned indicate how desirable or how undesirable a constraint is. Since the constraints are domain independent they can be used for any (unseen) city.

Aside from learning optimization criteria, this paper presents an approach to use these optimization criteria to find an optimal solution in practice.

Approach

In this research, both constraint learning and learning optimization criteria were implemented. A two-step approach is used to learn optimization criteria. First, clauses learning is used to find soft constraints in the given examples. Then, user-provided preference information is used to learn weights for these constraints.

Input

The input consists of global definitions that specify the types and predicates that are used to describe examples and a set of examples $\mathcal{E}$. Typing information relates directly to the problem domain and is usually easy to provide. Including types also improves the accuracy and efficiency of the learned clauses.

Examples are interpretations, they specify the constants in their domain and then all predicates that are true. Predicates can also be generated by background knowledge, in which case they are not included in the example.

When learning optimization criteria, preference information is also provided by the user. Each preference is of the form $e_1 \succ e_2 \succ ... \succ e_n$ with $e_i \in \mathcal{E}$. Preference look at the relative position of a few examples at a time and are typically easier to provide than absolute scores for all examples.

Example 2. "Rewrite" Consider the moving problem (example 2). A model could, for example, use a type *Area* and predicate definitions low_crime(*Area*), cheap(*Area*), school_in(*Area*), work_in(*Area*) and live_in(*Area*). Examples would then specify various areas A_i and describe the conditions (e.g. cheap(A_1), low_crime(A_2)) as well as the users choices (e.g. school_in(A_2), work_in(A_2), live_in(A_1)).

Algorithm 1 The clausal discovery algorithm

```
Given: Examples E and threshold t
Q ← {□}, T ← {}
while #Q > 0 do
    c ← next(Q)
    if #{e ∈ E | e ⊨ c} ≥ t then
        if ¬(T ⊨ c) then
            T = T ∪ c
    else
        Q ← Q ∪ ρ(c)
T ← prune(T)
return T
```

Clausal Discovery

The clause learning system is based on the clausal discovery algorithm (alg. 1). Note the double use of the $\models$ operator. On the one hand, it is used in $e \models c$ to denote that clause c is true in the example (interpretation) e (c covers e). This test is used to compute if a clause covers enough examples. On the other hand, in $\mathcal{T} \models c$ it signifies that $\mathcal{T}$ entails c (i.e. for every interpretation it holds that if $\mathcal{T}$ is true, c is also true). The algorithm uses this test to remove redundant clauses. For both cases the IDP knowledge base system (De Pooter, Wittocx, and Denecker 2013; Wittocx, Mariën, and Denecker 2008) is used to compute $\models$. The algorithm can find soft constraints ($t < \#\mathcal{E}$) or hard constraints ($t = \#\mathcal{E}$).

Starting with the empty clause ($\square$), clauses are refined until they cover enough examples. If a clause covers enough examples it is added to result set, provided it is not entailed by clauses in the result set. In the refinement step, a clause $c = \{l_1, l_2, ..., l_k\}$ is extended by adding a new literal l. The new clause, $c' = c \cup \{l\}$ is more than general than c ($c' \models c$) and potentially covers more examples. By adding just one literal the refinement operator ρ generates a set of clauses that are minimally more general. In order to do this efficiently, a list of atoms is pre-calculated in advance, given a maximal amount of variables that can occur within a clause. Within the body and head of a clause, atoms may only be added in a specific order to avoid generating redundant clauses. The implementation natively supports symmetric predicates for which the order of the terms do not matter. Object Identity is used to specify that variables with different names must denote different objects. Additionally there are two syntactic restrictions. Clauses must be connected (i.e. new atoms must always contain a variable that has already occurred) and range-restricted (i.e. no new variables may be introduced in the head of the clause).

To avoid extensive coverage ($e \models c$) and entailment ($\mathcal{T} \models c$) calculations, two additional tests are used. The subset-test rejects clauses that are supersets of a clause that covers the same or all examples and has been accepted earlier. Because clauses can be formulated in multiple ways, a representative-test is used to remove clauses that are not in canonical form.

Optimization

The first step in finding weighted soft constraints is to identify soft constraints by using the constraint learning system with a (low) threshold. Examples are characterized by the soft constraints (clauses) which they satisfy. Therefore, every example e can be translated to a vector of boolean features by introducing a feature f_i for every clause c_i, with $f_i = \mathbf{1}_{c_i \text{ covers } e}$. Existing software can be used to learn a linear scoring function $\sum_i w_i \cdot f_i$ over these features based on the given rankings. For an unseen example, the feature vector is computed by calculating what clauses cover the new example. The scoring function can then be used to calculate a score for that example.

Example 3. "Rewrite" Consider examples e_1, e_2, e_3, rankings $e_1 > e_2, e_2 > e_3$ and clauses c_1, c_2, c_3. If the

Table 1: Clause coverage

Example	Covered by	Feature vector
e_1	c_1	(1, 0, 0)
e_2	c_2	(0, 1, 0)
e_3	c_1, c_2, c_3	(1, 1, 1)

clauses cover examples according to table 1, then, for example, the function $(1 \cdot c_1) + (0 \cdot c_2) + (-2 \cdot c_3)$ perfectly models the given rankings. A new example that is covered by c_1 and c_2, would be assigned a score of 1, according to this function. This score has no value in the absolute sense, it can only be used to compare it to other examples ranked by the same function. In this example clause c_1 represents a desirable property, clause c_2 is ignored because it does not influence the ranking and clause c_3 represents an undesirable property.

SVM^{rank} (Joachims 2006) was chosen to find the scoring function since it uses a linear model and offers an efficient implementation. The weights that it assigns to the features can be used directly as weights for the optimization criteria. The input format that is used by SVM^{rank} is also supported by many other learn-to-rank implementations. Therefore, other linear ranking systems such as Coordinate Ascent (Metzler and Croft 2007) could also be used.

Optimal Solution

Solvers, such as IDP, can use clauses directly as hard constraints to generate a solution. There is no solver that can use the chosen optimization criteria directly. Weighted MAX-SAT solvers use propositional clauses and only allow for positive weights.

This optimization task can be solved in IDP, using inductive definitions, aggregates and minimization. The only limitation is that the current version only supports integer values. Therefore, the weights of the clauses are divided by the smallest absolute weight, multiplied by a constant and rounded to the closest integer if necessary.

In order to model the optimization problem in IDP, every clause c_i with variables $v_1, ..., v_n$ is represented by a number i. For every clause a predicate $t(i)$ is added to capture the truth value of the clause. A function $cost(i)$ specifies the cost of not satisfying the clause, which is equal to the weight of the clause.

$$t(i) \Leftrightarrow \forall v_1, ..., v_n : c_i.$$
$$cost(i) = w_i.$$

Using t and $cost$, a function $actual(i)$ is then defined in IDP as $\mathbf{1}_{\neg t(i)} \cdot cost(i)$. This function is used in the term $\sum_i actual(i)$ to be minimized, which will allow IDP to search for an optimal solution.

Evaluation

Several experiments aim to measure the accuracy and efficiency of the learning systems for constraints and optimization criteria. In order to account for non-determinism, experiments measuring execution times or optimization scores are usually performed eight times and the results are averaged.

Constraints

Four problems have been used to evaluate constraint learning. The first problem is map coloring, where countries are assigned colors and neighboring countries may not have the same color. Two examples containing each three correctly colored countries are given. The second problem is sudoku and a single, solved 4×4 sudoku is given. For the third problem (elevator) three examples have been generated, two of which respect the underlying soft constraint. The last problem (co-housing) contains four hard constraints and five examples have been generated that respect the constraints.

Question 1 (Accuracy)**.** *Are the essential constraints discovered and what influence do different parameters have on the accuracy?*

For all problems the essential constraints are found. Often additional constraints were found that describe some structure in the problems. For example, for the map coloring problem a learned constraint states that countries are never their own neighbor. These kind of constraints may help a constraint solver to work more efficiently.

The learning process is parameterized by the maximal amount of variables and literals allowed per clause. If these limits are too large, constraints are found that over-fit the training data. These constraints are too specific and will exclude valid solutions that are not in the training set. On the other hand, if the chosen limits are too small, the necessary constraints will not be found.

Over-fitting can be addressed by removing constraints that exclude valid solutions (manually or automatically) as well as providing more (or larger) training examples. Negative examples can be used to detect under-fitting, indicating that the parameter values are too small.

Question 2 (Efficiency)**.** *How fast is the clause learning system and what is the effect of various design decisions and input on the execution time?*

Table 2 shows the execution times for several experiments. It shows that smaller problems can be solved efficiently and shows that removing the syntactical restrictions exponentially increase the search space.

All efficiency measures (i.e. symmetric predicates, subset test and representative test) have been able to improve the execution time, sometimes by more than 50%. The overhead they introduce is more than compensated by the efficiency gains they cause.

Further experiments show that increasing the number of variables or literals per clause impacts the efficiency. Especially the combination of more variables *and* literals can steeply increase the execution time. Therefore it would be useful to adapt these parameters dynamically.

Table 2: Execution times overview

Omitted	Problem	Average time (s)
Nothing (baseline)	Map coloring	1.581 (± 0.117)
	Sudoku	4.787 (± 0.062)
	Elevator	3.182 (± 0.073)
	Co-housing	25.903 (± 0.446)
Range restriction	Map coloring	4.629 (± 0.199)
	Sudoku	16.118 (± 0.154)
	Elevator	40.453 (± 0.319)
	Co-housing	207.768 (± 0.330)
Connected clauses	Map coloring	1.589 (± 0.110)
	Sudoku	7.068 (± 0.150)
	Elevator	6.157 (± 0.114)
	Co-housing	103.633 (± 0.131)

Adding additional examples only increases the execution time by a constant factor. Since adding more examples can help improve the accuracy, this trade-off is often worthwhile.

Question 3 (Compared to humans)**.** *How do learned constraints compare to human programmed constraints?*

Human programmed theories for map coloring and sudoku are available on the website of the IDP system. These theories usually focus on being compact and contain only the essential constraints. Table 3 shows the results of two experiments that measure the time to find a solution for a new problem. This time is measured for the learned theories as well as for hand made theories. The learned theories are slightly adapted to be able to solve the same problems and corrected if they contain an unfair advantage.

Table 3: CPU times human vs. learned theory

Problem	Type	Average CPU time (s)
Map coloring	Human	0.968 (±0.023)
	Learned	0.403 (±0.015)
Sudoku	Human	1.453 (±0.018)
	Learned	0.310 (±0.012)

Especially for non-experts, a learning system can be useful to assist them during the modeling process. Additionally, the learning system can function in an automatic setting. These experiments show that learned constraints can be used to solve problems efficiently and even faster than hand programmed constraints for the examined cases.

Optimization

The efficiency of the learning system for optimization criteria depends mainly on the efficiency of learning soft constraints. Therefore, the experiments in this section are focused on the accuracy of the optimization criteria and the influence of different factors.

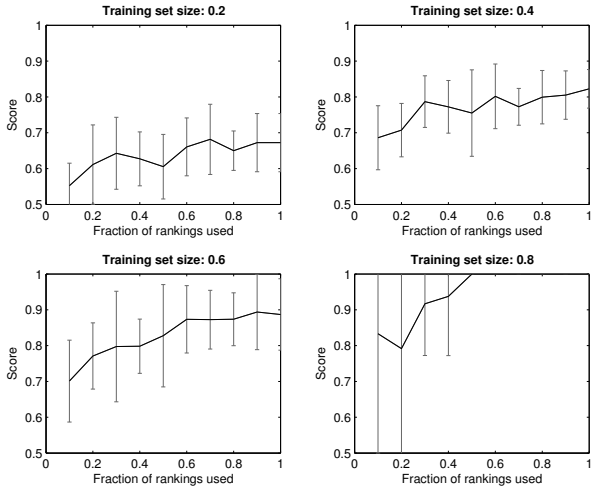

Figure 1: Influence fraction of inequalities

The moving problem with input according to example 2 is used to evaluate the learning of optimization criteria. 18 possible configurations are used as available examples. Four weighted soft constraints are used to represent the underlying model for the users preferences. Two approaches are used for evaluation. In the first approach, the examples are split into disjoint train and test sets. The second approach uses all examples as train as well as test set.

For every experiment a fraction of the training examples is selected randomly. Pairwise rankings are generated by picking two of the selected examples and predicting the better example using the underlying model. The selected examples and a random subset of the possible rankings are then used as input for learning. Noise is simulated by flipping a subset of the input rankings.

To evaluate the learned optimization criteria, all possible pairs of examples in the test set are generated. The learned model is used to predict the better example for every pair. The score is then calculated as the fraction of correctly predicted pairs. Pairs for which the underlying model ranks both example the same are omitted.

Question 4 (Accuracy)**.** *How accurately can learned optimization criteria approximate underlying models?*

Figure 1 shows how the scores improve as the amount of examples and the fraction of included preferences increases. In all cases, more than half the pairs of examples are correctly predicted and high scores can be obtained even for small datasets. Additional experiments have shown that even for lower scores the learned optimization criteria are often capable of identifying the correct optimal solution.

The standard underlying model can be directly expressed using the soft constraints that can be learned. Even though clausal theories can be very expressive, important restrictions have been placed on the learned

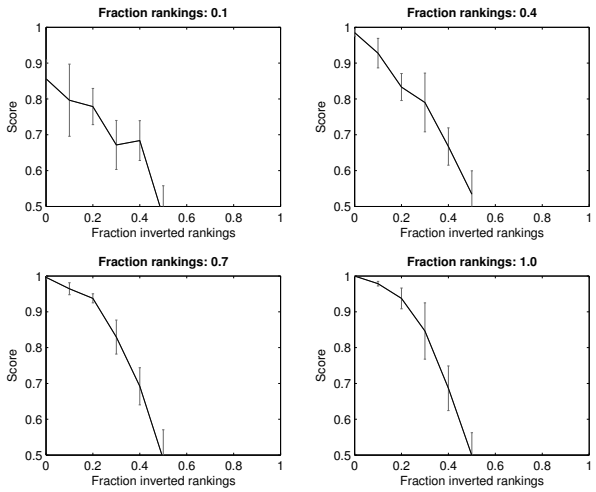

Figure 2: Influence of noise

clauses in this paper. In order to test the accuracy for models which cannot be directly expressed, a model consisting of two disconnected clauses has been tested as well. While there are limits to the expressibility, the learned optimization criteria were able to obtain similar scores and seem to be robust with respect to the exact formulation.

Question 5 (Noise)**.** *What is the influence of noise on the accuracy?*

The influence of noise is shown in figure 2. Hereby, the second approach of testing is applied, using potentially overlapping train- and test sets. High scores are obtained, even despite significant levels of noise. The figure also shows that providing more rankings improves the robustness of the algorithm, even if the relative amount of noise remains unchanged.

Table 4: Scores for different thresholds (t)

$t = 1$	$t = 2$	$t = 3$	$t = 4$
0.823	0.740	0.788	0.735
($\pm$ 0.073)	($\pm$ 0.078)	($\pm$ 0.074)	($\pm$ 0.063)

Question 6 (Threshold)**.** *What is the effect of the soft constraint threshold on the accuracy?*

Table 4 demonstrates that increasing the threshold used for finding soft constraints does not improve the score. This experiment used 40% of the examples as training set and 40% of the available rankings. However, if the size of the problem and examples is increased, a higher threshold will likely be appropriate.

Conclusion

The research in this thesis has focused on automatically acquiring constraints and optimization criteria from examples and rankings. Implementation for both clause learning and clausal optimization have been provided that accomplish these tasks using first-order logic clauses.

The constraint learning implementation has been able to learn the relevant hard and soft constraints in all experiments. For each problem, only a small number of examples was given to learn from. The system requires only a minimal amount of information from the user, however, it also allows for the use of expressive background knowledge. The learned constraints are domain independent, which facilitates the construction of positive examples.

Using the constraint learning implementation, the goal of learning optimization criteria has been accomplished. Optimization criteria can be learned that enable optimal solutions to be found. Even for small datasets and noisy rankings, constraints are found that enable most examples to be ranked correctly.

Aside from learning formal representations automatically from examples, this research shows how these representations can be used in practice. This also forms an important step to enable the learning of optimization criteria in an interactive setting.

Future work This research offers multiple opportunities for future work. It would be interesting to adapt the number of variables and literals in a clause dynamically. This could be accomplished, for example, by using negative examples. Learned clauses must be specific enough to not cover any negative examples.

Additionally, it would be interesting to add interactivity to the learning system for the generation of examples or rankings. Rankings expressing that examples are equally ranked are currently ignored. Whenever provided explicitly, however, it could be interesting to incorporate this information into the algorithm.

As mentioned earlier, the domain independence of the clauses has several advantages. In some cases, however, specific objects are inherently present in any problem instantiation. The constraint learning system could be enhanced to include such global constants.

Finally, it would desirable to improve the implementation in order to tackle larger sized problems. All the experiments were conducted with problems of limited size and the computation time increases rapidly if there are more predicates, variables and literals to be used in clauses.

Acknowledgments

The author thanks his promoters Dr. Luc De Raedt en Dr. ir. Anton Dries. Samuel Kolb is supported by the Research Foundation - Flanders (FWO).

References

Beldiceanu, N., and Simonis, H. 2012. A model seeker: Extracting global constraint models from positive examples. In *Principles and Practice of Constraint Programming*, 141–157. Springer.

Bessiere, C.; Coletta, R.; O'Sullivan, B.; Paulin, M.; et al. 2007. Query-driven constraint acquisition. In *IJCAI*, 50–55.

Bessiere, C.; Coletta, R.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.-G.; and Walsh, T. 2013. Constraint acquisition via partial queries. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 475–481. AAAI Press.

Campigotto, P.; Passerini, A.; and Battiti, R. 2011. Active learning of combinatorial features for interactive optimization. In *Learning and Intelligent Optimization*. Springer. 336–350.

De Pooter, S.; Wittocx, J.; and Denecker, M. 2013. A prototype of a knowledge-based programming environment. In *Applications of Declarative Programming and Knowledge Management*. Springer. 279–286.

De Raedt, L., and Dehaspe, L. 1997. Clausal discovery. *Mach. Learn.* 26(2-3):99–146.

Joachims, T. 2006. Training linear svms in linear time. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 217–226. ACM.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On learning constraint problems. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, 45–52. IEEE.

Metzler, D., and Croft, W. B. 2007. Linear feature-based models for information retrieval. *Information Retrieval* 10(3):257–274.

Puget, J.-F. 2004. Constraint programming next challenge: Simplicity of use. In Wallace, M., ed., *Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 5–8.

Wittocx, J.; Mariën, M.; and Denecker, M. 2008. The idp system: a model expansion system for an extension of classical logic. In *Proceedings of the 2nd Workshop on Logic and Search*, 153–165.