Consistency for Partially Defined Constraints

Student:Andreï Legtchenko; Supervisor:Arnaud Lallouet

Université d'Orléans — LIFO
BP6759, F-45067 Orléans, France
legtchen|lallouet@lifo.univ-orleans.fr

Abstract. Partially defined or *Open Constraints* can be used to model the incomplete knowledge of a concept or a relation. We propose to complete its definition by using Machine Learning techniques. Our technique is composed of two steps: first we learn a classifier for the constraint's projections and then we transform the classifier into a propagator. We show that our technique not only has good learning performances but also yields a very efficient solver for the learned constraint.
Keywords: Open Constraints; Automatic Solver Construction; Machine Learning.

1 Introduction

The success of Constraint Programming takes its roots in its unique combination of modeling facilities and solving efficiency. In this paper, we propose to use partially defined finite domain constraints called *Open Constraints*. In an Open Constraint [7], some tuples are known to be true, some other are known to be false and some are just *unknown*.

Let us take an example in which Open Constraints occur naturally: in a large company, the canteen serves a large number of meals a day. One day, the Chef is asked to prepare as first course a salad which should be good but also the cheapest possible. The Chef owns a cookbook composed of 53 recipes of salads and has various ingredients such as tomatoes, mayonnaise, shrimps... All are given with price and available quantity. A first idea would be to select from the cookbook the cheapest recipe possible given the available ingredients. But, since not all knowledge about salads is contained in the cookbook, the invention of a new salad is also an interesting option. The concept of "good salad" can be modeled as an Open Constraint whose solution tuples are *good* salads and non-solutions are *bad* ones. The cookbook is then viewed as a set of examples for the open constraint (for the sake of learning, we should also give examples of bad salads).

The idea of the technique we use for learning comes directly from the classical model of solvers computing a chaotic iteration of reduction operators [2]. We begin by learning the constraint. But instead of learning it by a classifier which takes as input all its variables and answers "yes" if the tuple belongs to the constraint and "no" otherwise, we choose to *learn the support function* of

the constraint for each value of its variables' domains. A tuple is part of the constraint if accepted by all support functions for each of its values and rejected as soon as it gets rejected by one. This method is non-standard in Machine Learning but we show in section 4 that it can achieve a low error ratio — comparable to well-established learning methods — when new tuples are submitted, which proves experimentally its validity. We also show that the learned solver yields a strong pruning along the search space.

2 Preliminaries: building consistencies

We first recall the basic notion of consistency in order to present the approximation scheme we use for learning. For a set E, we denote by $\mathcal{P}(E)$ its powerset and by $|E|$ its cardinal. Let V be a set of variables and $D = (D_X)_{X \in V}$ be their family of (finite) domains. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Projection of a tuple or a set of tuples on a set of variables is denoted by $|$. A *constraint* c is a couple (W, T) where $W \subseteq V$ are the variables of c and $T \subseteq D^W$ is the set of solutions of c. A *Constraint Satisfaction Problem* (or CSP) is a set of constraints. A solution is a tuple which satisfy all constraints. In this paper, we use the common framework combining *search* and domain reduction by a *consistency*.

A *search state* is a set of yet possible values for each variable: for $W \subseteq V$, it is a family $s = (s_X)_{X \in W}$ such that $\forall X \in W, s_X \subseteq D_X$. The corresponding *search space* is $S_W = \Pi_{X \in W} \mathcal{P}(D_X)$. Some search states we call *singletonic* represent a single tuple and play a special role as representant of possible solutions. A singletonic search state s is such that $|\Pi s| = 1$.

A consistency can be modeled as the greatest fixpoint of a set of so-called *propagators* and is computed by a chaotic iteration [2]. For a constraint $c = (W, T)$, a *propagator* is an operator f on S_W[1] having important properties like *monotony*, *contractance* and *correctness*.

We can decompose an arc-consistency operator according each value of X's domain, for all $X \in W$. We call an *Elementary Reduction Function* (or ERF) a Boolean function $f_{X=a}$ checking if a value a in X's domain has a support. In order to achieve this check, this function uses as input the domain of the other variables of the constraint. By combining ERFs for each domain value, we can reconstitute the arc-consistency operator. These functions are implemented with many optimizations in the GAC-schema [5].

If we give each domain value an ERF but if we assume that if ERF takes as input only the bounds of the other variables' domains, we get a new intermediate consistency, weaker than arc-consistency.

[1] When iterating operators for constraints on different sets of variables, a classical cylindrification on V is applied.

3 Open Constraints

A classical constraint $c = (W, T)$ is supposed to be known in totality. When dealing with incomplete information, it may happen that some parts of the constraint are unknown. We call such a partially defined constraint an *Open Constraint*:

Definition 1 (Open Constraint).
An open constraint *is a triple* $c = (W, c^+, c^-)$ *where* $c^+ \subseteq D^W, c^- \subseteq D^W$ *and* $c^+ \cap c^- = \emptyset$.

An open constraint needs to be *closed* to be usable in a constraint solving environment. The closure of an open constraint c is done by choosing a class (it belongs or not to the constraint) for all unknown tuples. This can be obtained by supervised classification: it consists in the induction of a function which associates to all tuples a class from a set of examples given with their respective class.

4 Open Constraint Acquisition

We propose to learn a classifier by using a representation which is well adapted to be turned to an efficient solver. We propose to build an independent classifier for each value a of the domain of each variable $X \in W$ in the spirit of ERFs introduced in section 2. This classifier computes a Boolean function stating if the value a should remain in the current domain (output value 1) or if it can be removed (value 0). We call it an *elementary classifier*. It takes as input the value of every other variable in $W - \{X\}$.

We propose to use as representation for learning an Artificial Neural Network (ANN) with an intermediate hidden layer. Other kinds of classifiers can also be used but we cannot describe them for lack of space. For a constraint $c = (W, c^+, c^-)$, the classifier we build for $X = a$ is a tree of neurons with one hidden layer as depicted in figure 1. Let $(n_i)_{i \in I}$ be the intermediary nodes and *out* be the output node. All neurons of the hidden layer have as input a value for each variable in $W - \{X\}$ and are connected to the output node. Let us call

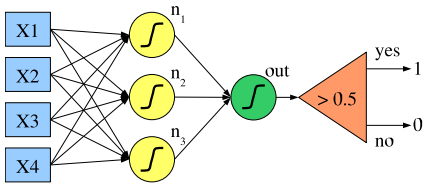

Fig. 1. Structure of the ANN

$n_{<X=a>}$ the network concerning $X = a$. Since neurons are continuous by nature,

we use an analog coding of the domains and a threshold decision unit at the last level of the network, as depicted in figure 1.

The global classifier for the open constraint is composed of all of these elementary classifiers for all values in the domain of all variables $\{n_{<X=a>} \mid X \in W, a \in D_X\}$. Following the intuition of ERFs for solving, we can use these elementary classifiers to decide if a tuple belongs to the extension of the constraint or not by checking if the tuple gets rejected or not by one of the classifiers. Let $t \in D^W$ be a candidate tuple and let $(n_{<X=t|_X>}(t|_{W-\{X\}}))_{X \in W}$ be the family of 0/1 answers of the elementary classifiers for all values. We can interpret the answers according two points of view:

- *vote with veto*: the tuple is accepted if and only if it is accepted by all elementary classifiers.
- *majority vote*: the tuple is accepted if accepted by a majority of elementary classifiers.

In order to produce the extension of the open constraint, these classifiers are trained on examples and counter-examples. For $E \subseteq D^W$, $X \in W$ and $a \in D_X$, we denote by $E_{<X=a>}$ the selection of tuples of D^W having a as value on X: $E_{<X=a>} = \{t \in E \mid t|_X = a\}$. Thus, in order to build the classifier $n_{<X=a>}$, we take the following sets of examples and counter-examples: $e^+_{<X=a>} = c^+_{<X=a>}|_{W-\{X\}}$ and $e^-_{<X=a>} = c^-_{<X=a>}|_{W-\{X\}}$.

The networks are trained by the classical backpropagation algorithm [13] which finds a value for the weights using gradient descent. This framework has been implemented in a system called Solar and tested on the *salad* example presented in introduction and on various Machine Learning databases[2] used as open constraint descriptions. We compare the generalization performance of our technique to the popular decision tree learning system *C5.0* [12] using standard 10-folds cross-validation method. The technique we propose challenges powerful techniques such as boosting [8], both in generalization performance and scattering of results as measured by standard deviation and error. Nevertheless, the vote of elementary classifiers cannot be viewed as a variant of boosting. An important difference is that we partition the set of examples.

5 From classifiers to solvers

When put in a CSP, a constraint should contribute to the domain reduction. We propose to use the learned classifiers also for solving. In order to do this, let us recall some notions on interval analysis [10]. Here are the canonical extensions to intervals of some operators we use in classifiers:

$$\begin{aligned} [a,b] + [c,d] &= [a+c, b+d] \\ [a,b] \times [c,d] &= [\min(P), \max(P)] \\ &\quad \text{where } P = \{ac, ad, bc, bd\} \\ \exp([a,b]) &= [\exp(a), \exp(b)] \end{aligned}$$

[2] The databases are taken from the UCI Machine Learning repository (`http://www.ics.uci.edu/~mlearn`).

If e is an expression using these operators and E the same expression obtained by replacing each operator by a monotonic extension, then $\forall I \in Int_{\mathbb{R}}, \forall x \in I, e(x) \in E(I)$. This property of monotonic extensions is called "The Fundamental Theorem of Interval Arithmetic" [10]. It also holds when domains are replaced by cartesian products of intervals. By taking the canonical extension of all basic operators in an expression e, we obtain an extension E which is called the *natural* extension.

An elementary classifier $n_{<X=a>}$ defines naturally a Boolean function of its input variables. Let $N_{<X=a>}$ be the natural interval extension of $n_{<X=a>}$. Then, by using as input the current domain of the variables, we can obtain a range for its output. Since we put a 0.5 threshold after the output neuron, we can reject the value a for X if the maximum of the output range is less than 0.5, which means that all tuples are rejected in the current domain intervals. Otherwise, the value remains in the domain.

Proposition 2. *$N_{<X=a>}$ is an ERF. Moreover, the ERFs $N_{<X=a>}, \forall X \in W$, $\forall a \in D_X$ define a consistency for c.*

We call *learned consistency* the consistency defined by the learned propagators. Because we use multiple occurences of the same variable and because each ERF takes as input only the bounds of the other variables' domains, we get a new intermediate consistency, weaker than arc-consistency.

The propagators for each variable are independent, thus the generalization obtained when using the solver is the one obtained with *veto* vote. This is due to the independent scheduling of the consistency operators for each variable in the fixpoint computed by chaotic iteration [2].

The Solar system takes an Open Constraint as input and outputs a set of consistency operators which can be adapted to any solver. In our experiments, we used a custom made solver. We made a number of experiments in order to evaluate the learned consistency. These experiments show that the learned consistency is weaker than more classical consistencies but still reduces notably the search space.

6 Related work and Conclusion

Open constraints were introduced in [7] in the context of distributed reasoning but with the goal of minimizing the number of requests needed to complete the definition. Solver learning has been introduced in [3] with a special rule system. This work has been extended by [1] and [9] but still in the context of closed constraints. None of these method can combine generalization and solver efficiency. The idea of learning constraints has been used in [11] in the context of soft constraints. While the learning is effective, the problem of building a solver for the constraint is not tackled in this work. In [6] and [4], a CSP composed of predefined constraints like $=$ or $\leq$ is learned. The constraints are discovered by a version-space algorithm which reduce the possible constraints during the learning process.

Summary Open Constraints allow the use of constraints partially defined by examples and counter-examples in decision and optimization problems. In this work, we propose a new technique for learning open constraints by using special classifiers. Not only the generalization we obtain has remarkable properties from a Machine Learning point of view, but it can also be turned into a very efficient solver which gives an active behavior to the learned constraint.

References

1. Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *Transaction on Computational Logic*, 5(2), 2004.
2. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
3. K.R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In Joxan Jaffar, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 58–72, Alexandria, Virginia, USA, 1999. Springer.
4. Christian Bessière, Rémi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 123–137, Toronto, Canada, 2004. Springer.
5. Christian Bessière and Jean-Charles Régin. Arc-consistency for general constraint networks: preliminary results. In *IJCAI'97*, pages 398–404, Nagoya, 1997. Morgan Kaufmann.
6. R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, pages 812–816, Kinsale, Ireland, 2003. Springer.
7. Boi Faltings and Santiago Macho-Gonzalez. Open constraint satisfaction. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 356–370, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.
8. Y. Freund and R. Shapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.
9. Arnaud Lallouet, Thi-Bich-Hanh Dao, Andreï Legtchenko, and AbdelAli Ed-Dbali. Finite domain constraint solver learning. In Georg Gottlob, editor, *International Joint Conference on Artificial Intelligence*, pages 1379–1380, Acapulco, Mexico, 2003. AAAI Press. Poster.
10. Ramon E. Moore. *Interval Analysis*. Prentice Hall, 1966.
11. F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint system. *Constraints*, 9(4), 2004.
12. RuleQuest Research. See5: An informal tutorial, 2004. http://www.rulequest.com/see5-win.html.
13. D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. *Parallel Distributed Processins*, vol 1:318–362, 1986.