Indexical-Based Solver Learning

Thi-Bich-Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, Lionel Martin

Indexical-based Solver Learning

Thi Bich Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, Lionel Martin

Université d'Orléans — LIFO
BP 6759 — F-45067 Orléans cedex 2

Abstract. The pioneering works of Apt and Monfroy, and Abdennadher and Rigotti have shown that the construction of rule-based solvers can be automated using machine learning techniques. Both works implement the solver as a set of CHRs. But many solvers use the more specialized chaotic iteration of operators as operational semantics and not CHR's rewriting semantics. In this paper, we first define a language-independent framework for operator learning and then we apply it to the learning of partial arc-consistency operators for a subset of the indexical language of Gnu-Prolog and show the effectiveness of our approach by two implementations. On tested examples, Gnu-Prolog solvers are learned from their original constraints and powerful propagators are found for user-defined constraints.

Keywords : CSP; consistency; learning; rule-based constraint solver.

1 Introduction

Building a constraint solver is a notoriously complex task [12, 18, 16] and it has been recently demonstrated that — at least some parts of — this design can be automated by systematic search [4] or by using machine learning techniques [1, 2]. An automatic tool to generate solvers may help the solver designer by giving him a first implementation and letting him concentrate on non-trivial optimization. But it is also useful to the user who may, with little experience, develop a global constraint ad-hoc to his problem and get for free a propagator for it. We provide in this paper a formal framework to define finite domain solver learning and we propose a learning method which fits in this framework.

An efficient technique for computing consistencies in Constraint Satisfaction Problems (CSPs) is to use a data representation for the CSP and a set of operators whose common fixed-point models the expected consistency. The operators are then applied via chaotic iteration [13, 3] until reaching their common fixed-point. But solvers differ in the way they represent these operators: they can be written in the implementation language [18, 16] or in a higher-level language such as indexicals [21]. An indexical operator is written "`X in r`" where `X` is the name of a variable, and `r` is an expression which limits the range of possible values for `X` and which may depend on other variables' current domains.

For example, the constraint $and(X, Y, Z)$, defined by the following table, yields three indexical operators, one for each variable:

and :

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Gnu-Prolog indexicals:

```
X in min(Z) .. max(Z)*max(Y)+1-min(Y)
Y in min(Z) .. max(Z)*max(X)+1-min(X)
Z in min(X)*min(Y) .. max(X)*max(Y)
```

It is obvious that the expression for Z depending on X and Y is simple and intuitive whereas the expression for X depending on Y and Z is not. Building manually such expressions for arbitrary constraints is undoubtedly a challenge.

In this paper, we propose a framework to define the learning of such operators in three steps. We consider that a CSP is a set of n-ary constraints.

1. We define a notion of *semantic approximation* of a CSP, which consists in replacing the original problem by an approximate one, from the point of view of the data-structure and the solutions. For a CSP C composed of several n-ary constraints, its approximation K uses different, usually simpler constraints (for example, the domain of each variable), and the notion of solution is replaced by the weaker one of consistency. Operators are introduced to transform these CSPs and their closure defines a consistent state. A similar though more general framework is the one of [7] for semiring-based CSPs.
2. We define a notion of *syntactic approximation*. The choice of a representation language, both for the data and the operators, is of crucial importance since the language may not allow to represent all possible data or operators. This defines a second level of approximation but also a concrete representation. For example, intervals are a data representation for variable domains and indexicals for operators.
3. It is then possible to define a learning space of all possible operators and check the fitness of each candidate operator towards the definition of the constraint. Operator learning consists in an exploration of this learning space to look for the best candidate.

We propose a learning technique and two implementations of indexical operator learning specialized to the context of the partial arc-consistency of Gnu-Prolog [11] which consists in contracting the bounds of the variables domain, represented as intervals. The learned operator can be intuitively thought as the summary of the best possible reductions done by the constraint on every possible interval which could be encountered during the solving process.

The paper is organized as follows. In section 2, preliminaries on CSP solving are given. In section 3, we present our approximation framework, from which the learning problem is derived. In section 4, we specialize our framework for the learning of indexicals for partial arc-consistency and we present the language biases we used. Section 5 is devoted to the implementations and section 6 gives an illustration with some examples.

2 Preliminaries

Let V be a set of variables and $D = (D_X)_{X \in V}$ their domains. The domains we consider are finite and totally ordered sets. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Therefore, we have $D^V = \Pi D$. Projection of a tuple or a set of tuples on a set of variables is denoted by $|$, natural join of two sets of tuples is denoted by $\bowtie$.

Definition 1 (Constraint). *A* constraint c *is a pair* (W, T) *where*

- $W \subseteq V$ *is the* arity *of the constraint* c *and is denoted by* $var(c)$.
- $T \subseteq D^W$ *is the* solution *of* c *and is denoted by* $sol(c)$.

Definition 2 (Approximation ordering). *A constraint* $c' = (W', T')$ approximates $c = (W, T)$, *denoted by* $c \sqsubseteq c'$, *if* $var(c') = var(c)$ *and* $sol(c) \subseteq sol(c')$.

The join of two constraints is defined as a natural extension of the join of tuples: the *join* of c and c' is the constraint $c \bowtie c' = (var(c) \cup var(c'), sol(c) \bowtie sol(c'))$.

Definition 3 (CSP). *A* CSP *is a set of constraints.*

Join is naturally extended to CSPs and a CSP C is identified to be the join of its constraints $\bowtie C$. This defines the *solutions* of the CSP. A direct computation of this join is too expensive to be tractable. This is why various methods have been considered so far to compute the solutions of a CSP. We present here a short description of some of them for a comparison with other works (the presentation order is not significant):

- *search methods*: basically, it amounts to try values for the variables and test the constraints for satisfiability. Almost all other methods are hybridized with search to get completeness.
- *symbolic transformations*: by this, we refer to a variety of syntactic transformations of the constraints in order to obtain a solved form, for example in the *CLP*($\mathcal{R}$) [15] system. A further level of abstraction comes with the CHR language [14] which allows to describe rewriting-based solvers in a simple and versatile way. For example, here is a possible rule in the CHR language to simplify a boolean store:

  $$\texttt{and(X,Y,Z)} \wedge \texttt{Z=1} \iff \texttt{X=1} \wedge \texttt{Y=1} \wedge \texttt{Z=1}$$

  In our vision, the main characteristics of this language is that it uses first-order variables, or in other terms, that a variable represents an element of the domain.
- *approximations*: to be solved, constraints are represented not only by their syntactic form but also extensionally by their solutions. A state is an over-approximating CSP and transitions from a CSP to a smaller one are represented by correct, monotonic and contracting operators which are iterated until reaching a property called *consistency*. The contracting operators are most of the time written in the solver's implementation language (C++ for

Ilog Solver [18], Claire for The Choco System [16], ...) and this is probably the reason why this formalism is often perceived as only implementation-relevant.
There is however an exception with the "indexicals" language introduced in [21] and used in the Gnu-Prolog system [9, 11] or in the Sicstus Prolog system [8]. In this framework, variables are second-order and designate constraints instead of domain elements. It allows the description of operators of the form "X in r" where X represents the domain of the variable, "in" is the set inclusion and r is an expression of some set language. We believe that this paradigm is more general than it seems since it is parametrized by the language of r. This paradigm deserves further research since there is still no formalism having the same generality and expressive power to define indexical operators as the CHR formalism for the symbolic transformation point of view. A step towards this goal may be the so-called delay clauses of B-Prolog [23]. Indeed, the choice of the "r" language is of particular importance in a learning perspective.
- *other methods*: various other methods have been tackled to solve CSPs (see for example [20]).

3 A model of approximation for CSPs

In this section, we present a high-level model for the approximation made by consistency adapted from the more general framework of [7]. This set-theoretic formulation allows to describe very precisely the approximation process in a language-independent way. The basic idea is to clearly separate the CSP to be solved from its approximation (and more generally from the sequence of approximating CSPs).

3.1 Semantic approximations

Let C be a CSP over a set of variables V. Since a CSP is somehow identified with its join constraint $\bowtie C$, it can be approximated by a CSP K over the same set of variables V and such that $\bowtie C \subseteq \bowtie K$. In this case and by extension, we write $C \subseteq K$.

Intuitively, the CSP K is intended to be physically represented, for example (but not exclusively) by sets of tuples. This is why C and K may be built on completely different constraints. When all constraints in K are unary, and thus represent the domain of variables, it yields to the well-known "domain reduction scheme" used in most solvers. We call K the *approximating* CSP. Here is an example of two different approximating CSPs:

Example 4. Let C be the CSP composed of one constraint: $c = (\{X, Y, Z\}, \{(0,0,0), (0,0,1), (0,1,0), (1,0,1)\})$. Here are two approximating CSPs $K_1 = \{x, y, z\}$ and $K_2 = \{x, yz\}$ which are represented in figure 1. Here $x = (\{X\}, \{(0), (1)\})$ is an unary constraint representing the domain of X and $yz = (\{Y, Z\},$

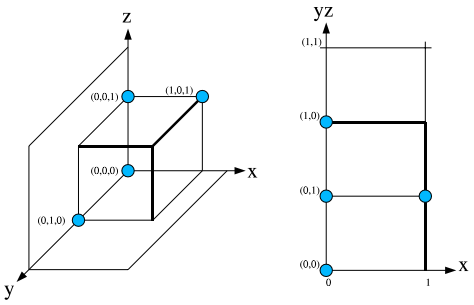

Fig. 1. Two different approximations for a CSP

$\{(0,0), (0,1), (1,0)\}$) is an arbitrary binary constraint. The constraints y and z are defined in the same straightforward way.

Most of the time, switching from C to K provides a gain in terms of memory consumption. For example, in the domain reduction approximation for n variables and a domain of size m, it boils down from m^n to $m * n$. But the trade-off is that representable approximations are less precise since they are limited to (the union of) cartesian products.

In the rest of the paper, C denotes the CSP to be solved, and K its approximating CSP. Moreover, we only use the domain reduction scheme. This means that the approximating CSP K is always composed of one unary constraint x for each variable $X \in var(C)$ (a lowercase letter designates the constraint while an uppercase designates the variable). Since our goal is to find the best approximation (in some sense), we call an approximating CSP K a *search state* and the set of such states is the *search space* $S = \Pi_{X \in V} \mathcal{P}(D_X)$. For $W \subseteq V$, we denote by $S_W = \Pi_{X \in W} \mathcal{P}(D_X)$ the search space restricted to W, hence $S = S_V$. We just indicate that this framework can be extended to approximations made with constraints of arbitrary arity but domain approximations help to keep simplicity to the notations.

3.2 Consistencies

In order to reach a certain degree of precision, approximations have to be refined: a sequence of over-approximating CSPs $(K_i)_{i \in \mathbb{N}}$ such that $\forall i \geq 0, K_i \supseteq K_{i+1}$ is built, until reaching a closure called *consistency*. Transitions between states are computed by operators. The nature of the operators determines the consistency and they are supposed to be correct, monotonic and contractant [3].

Example 5 (Projection operator). Let $c = (W, T)$ and $X \in W$. The projection $c_X : S_{W-\{X\}} \to \mathcal{P}(D_X)$ is defined by $c_X(s) = \{t_X \in D_X \mid t \in T \text{ and } t|_{W-\{X\}} \in s\}$. Suitably extended to S in order to be an operator, it is used for arc-consistency.

A set of operators is associated to the CSP and is iterated until reaching their common closure. This is often done by chaotic iteration [13, 3]. We are concerned in this paper with the learning of such operators in the particular case of partial arc-consistency used for example in Gnu-Prolog. Variable domains are represented only as intervals and thus only their bounds are contracted. Usually, to each constraint is associated an operator for each variable it contains. Let R be the set of reduction operators for a CSP C. We write $R \downarrow K$ the closure of K by the operators of R, i.e., the greatest CSP $K' \subseteq K$ such that $\forall r \in R, K' = r(K')$. A search state K is R-consistent if $K = R \downarrow K$.

3.3 Correctness and Completeness

Correctness for an operator r means that no solution is lost, or in other terms, that $C \subseteq K \Longrightarrow C \subseteq r(K)$. This is a strongly needed property since an incorrect solver may discard a solution forever. Completeness, however, is not a required property at the operator's level since this is achieved by the search mechanism.

But often solvers require the operators to be *singleton complete*, i.e., complete only for singletons. Let us define this notion:

Definition 6 (Singletonic CSP).
An approximating CSP K is singletonic if $\forall k \in K$, $|sol(k)| = 1$.

Intuitively, a singletonic CSP K on the set of variables V represents a tuple of D^V, a potential solution of the CSP C.

Definition 7 (Singleton Completeness).
Let C be a CSP and R its associated set of operators. The set R is singleton complete *if for any singletonic CSP K, we have*

$$K \not\subseteq C \Longrightarrow |sol(R \downarrow K)| = 0$$

This means that a non-solution tuple must be rejected by (at least) *one* operator. On the other side, correctness for singletonic CSPs means that a tuple is solution if accepted by *all* operators.

Singleton completeness is the basis of the compilation scheme of Gnu-Prolog since the (high-level) constraints such as $X = Y + Z$ are replaced by an "equivalent" formulation in the indexical language. However, the compilation can be effective only if the operators are able to distinguish between a solution and a non-solution, at least at the tuple level. Operationally, it means that, when dealing with a candidate solution, the consistency check can be realized by the propagation mechanism itself. But an important remark is that singleton completeness is a global property and is thus highly complex to check for an arbitrary set of operators because two non-solution tuples may be rejected by two different operators. In contrast, correctness check for an operator remains a local verification which only involves this operator.

3.4 Syntactic approximations: intervals and indexicals

The above formalism is language-independent but we still need to represent the domains and operators since learning is impossible without a representation. First, intervals are chosen for domain representation for their low space cost. This notion of approximation, different than consistency, comes from interval analysis and is presented for example in [3] where it is based on a family of subsets. This is an approximation of the domain while the consistency defines an approximation of the solutions. It defines the representable subsets of a variable's domain.

The indexical language introduced in [21] and used in Gnu-Prolog [11] and Sicstus Prolog [8] is a convenient language to define operators. An operator is expressed by a second-order constraint $x \subseteq r$ where x is a constraint of the approximating CSP K and r a constraint expression in a given language. Since we consider that all constraints in K are unary, the evaluation of r is a "range", i.e., another monadic constraint.

Example 8. For example, the high-level constraint `X #= Y+C` is compiled for partial arc-consistency by Gnu-Prolog into two indexical operators:

`X in min(Y)+C .. max(Y)+C` and `Y in min(X)-C .. max(X)-C`

For full arc-consistency, the same constraint is compiled to:

`X in dom(Y)+C` and `Y in dom(X)-C`

A subset of the Gnu-Prolog indexical language is given in figure 2.

c	::=	X in r	
r	::=	t_1 `..` t_2	(interval)
		{ t }	(singleton)
		r_1 `:` r_2	(union)
		r_1 `&` r_2	(intersection)
t	::=	`min(Y)`	(indexical term *min*)
		`max(Y)`	(indexical term *max*)
		ct	(constant term)
		t_1`+`t_2 | t_1`-`t_2 | t_1`*`t_2 | t_1`/`$^<$$t_2$ \| t_1`/`$^>$$t_2$	(integer operations)
ct	::=	n | `infinity`	(values)
		ct_1`+`ct_2 | ct_1`-`ct_2 | ct_1`*`ct_2 | ct_1`/`$^<$$ct_2$ \| ct_1`/`$^>$$ct_2$	

Fig. 2. A subset of Gnu-Prolog indexical language

Yet, since the Gnu-Prolog indexical language does not contain conditional expressions, the formulation of some operators may be operationally inefficient. This is why Gnu-Prolog implements an ad-hoc delay mechanism by the syntax `val(X)` which represents the same value as `dom(X)` but delays until X is instanciated. It is the only guard of the language.

4 Description of the learning problem

In this section, we present the example space, the learning space and the learning algorithm. Semantic approximations of CSPs are a vast framework and many choices have to be made. Here we present as restrictions the main choices we made to get a tractable learning problem.

Let $c = (W, T) \in C$ be a constraint. The exact projection function c_X of c on a variable $X \in W$ has been defined in example 5. Our goal is to learn a function L_X^c which mimics the behavior of c_X and is at least correct. This means that $\forall s \in S_{var(c)-\{X\}}, L_X^c(s) \supseteq c_X(s)$. But we also want to be the closest possible to the projection function since the constant function $s \mapsto D_X$ is correct but useless.

Definition 9 (Example set). *The function c_X, or equivalently the set of pairs $Ex(c, X) = \{(s, c_X(s)) \mid s \in S_{var(c)-\{X\}}\}$ is called the* example set *for L_X^c.*

Learning is often made from an incomplete set of examples, and a part of the learning process consists in finding a suitable generalization. In solver learning, a missing example may yield an incorrect behavior, which contradicts the most crucially expected solver property.

Restriction 1 Since we want to get correct operators, we use the whole example set in the learning process.

The size of this example set is $2^{\Sigma_{X \in var(c)}|D_X|}$, which is in general far too large to be completely traversed. In contrast, the interval space has a size in $O(\Pi_{X \in var(c)} |D_X|^2)$. In order to shrink the search space, arbitrary subsets can be "rounded" to their next including interval.

Restriction 2 We compute with intervals.

Nevertheless, the output of an operator could be any subset. In order to further reduce the learning space, we set the following language bias:

Restriction 3 The output of the learned operator is a single interval.

Because of this restriction, the operator L_X^c is expressed in the indexical language by its minimal and maximal bounds $minL_X^c$ and $maxL_X^c$, which are arithmetic expressions not involving X. Note that the power of indexicals is not fully used but also that it is sufficient to represent the partial arc-consistency used in Gnu-Prolog.

Now we make these restrictions more formal in order to define the example set really used in the learning process. First, interval lattices are a particular case of lattice approximations:

Definition 10 (Lattice approximation). *Let E be a set and P a sub-lattice of $\mathcal{P}(E)$ such that $\emptyset \in P$ and $E \in P$. For $e \in \mathcal{P}(E)$, we define:*

$$up_P(e) = \inf\{p \in P \mid e \subseteq p\}$$

Let Int_X be the interval lattice of D_X. In order to define the input of the function to be learned, we need to define the search space in which the computation takes place. It consists of a unique interval for each variable. For $W \subseteq V$, let $Int_W = \Pi_{Y \in W} Int_Y$. The interval approximation can be extended to cartesian products by $up_{Int_W} : S_W \to Int_W$ defined by:

$$s \mapsto \Pi_{Y \in W}\ up_{Int_Y}(s_Y)$$

We denote by $[s]$ this interval approximation of s. Note that the approximation domain Int_W is omitted since there is no possible confusion. By computing in the interval lattice, we preserve the correctness:

Proposition 11.

$$\forall s \in S_{var(c)-\{X\}},\ c_X(s) \subseteq c_X([s])$$

Proof. By monotony of c_X.

Definition 12 (Approximate example set).
The approximate example set *for L^c_X is given by the function*

$$Ex^{Int}(c, X) : Int_{var(c)-\{X\}} \to Int_X$$

such that

$$Ex^{Int}(c, X)(s) = [c_X(s)]$$

The bounds of L^c_X can be learned separately from a separate example set. For example, for $minL^c_X$, the example set is given by the function $Int_{var(c)-\{X\}} \to \mathbb{N}$ defined by $s \mapsto \min([c_X(s)])$. In the following, we focus on $minL^c_X$, but the other bound is obtained by the same method.

Now, we cannot be more precise without introducing a language to express operators. The remaining problem is to find a suitable expression in the subgrammar of terms and constant terms (entries t and ct in figure 2) which mimics as closely as possible the behavior of the function $\min([c_X(s)])$. In order to limit the complexity of the learning process, we use the following language bias. Since the sum is present in the language, our technique consists in fixing the general form of the function to be learned to a linear combinaison of terms, each one being expressed without the addition. Let $\mathcal{L}_1 = \{min(Y), max(Y) \mid Y \in W \text{ and } Y \neq X\}$, $\mathcal{L}_2 = \{t_1 * t_2 \mid t_1, t_2 \in \mathcal{L}_1\}$ and $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$.

Then we can express $minL^c_X$ as the following linear form:

$$minL^c_x = \Sigma_{w \in \mathcal{L}}\ \alpha_w * w + \alpha_0$$

The last part of the learning algorithm consists in finding the coefficients α_w, in order to meet the correction constraints. For each example s, the value of $minL^c_X(s)$ is correct if lesser or equal than the real value $\min([c_X(s)])$.

In contrast, completeness cannot be ensured the same way because it would sometimes prevent the system to have a solution. Singleton completeness, which

is not a local property, cannot be expressed at this level. Our approach consists in finding the most complete function in a certain sense, by minimizing a quality function for each candidate. We tried different quality functions and the results are presented in the next section. A typical quality function is to minimize the global error between the candidate and the real value. Hence we get the following linear program:

`minimize:`

$$\Sigma_{s \in Int_{var(c)-\{X\}}} \quad min([c_X(s)]) - minL^c_X(s)$$

`subject to:`

$$\forall s \in Int_{var(c)-\{X\}}\,, \quad minL^c_X(s) \;\leq\; min([c_X(s)])$$

The correctness of the algorithm is ensured by construction.

One may wonder if the restrictions we made and the biases we used do not yield a too much restricted framework. Here are some answers:

- The formalism of semantic approximations we use is very general and allows us to test many hypotheses. Indeed, with our definition of semantic approximation, all kinds of consistencies can be formalized in a language independent way. However, indexicals are high-level enough to easily allow the handling of the many syntactic transformations necessary to the learning process.
- The results we present in section 6 show that the operators learned are of good quality. In particular on regular constraints, we are able to find the classical operators of Gnu-Prolog. On arbitrary constraints, we still obtain a good pruning power.
- The choices we made present a good balance between the complexity of the learning task and the size of the explored search space. In particular, the use of a strong language bias is a common feature in machine learning [22].
- The complete construction of a solver may probably not be fully automatic. But a collection of learning tools with different techniques and algorithms may be of great help. We rather think of these tools as being part of a solver design environment.

5 Implementations

We have built two systems implementing this framework. A first system handles constraints of arbitrary arity, uses the simplex algorithm and yields X in r in the form described above (a linear combinaison of elements of $\mathcal{L}$). The second one handles only binary constraints, uses a genetic algorithm, and proposes three different (but still fixed) forms of operators.

5.1 The Simplex learner

The principle of this implementation is to solve the previous linear programming problem with the simplex algorithm: the simplex learner is a C program which calls the solver lp_solve[1].

The command-line user interface allows mainly to build minimal and maximal bounds for each variable with basic simplifications and to compute reductions obtained with these bounds.

5.2 The Genetic learner: GA-ILearner

This second implementation uses genetic algorithms to improve the quality of a population of candidate solutions. This C++ application has a graphic interface which allows the user to "draw" his constraint with the mouse. The user constraint appears on screen as red dots (see figure 3). Only binary constraints are allowed in order to be representable on screen, but variable domains range up to 50×50. The user can learn the bounds of the functions separately, or specify them manually. Three fixed forms of functions are proposed for each bound: linear ($X_{min} = A * min(Y) + B * max(Y) + C$), rational ($X_{min} = \frac{A}{min(Y)+1} + \frac{B}{max(Y)+1} + C$) and quadratic ($X_{min} = A * min(Y)^2 + B * max(Y)^2 + C * min(Y) * max(Y) + D * min(Y) + E * max(Y) + F$). An example of rational approximation is given in figure 3. The singletons which are

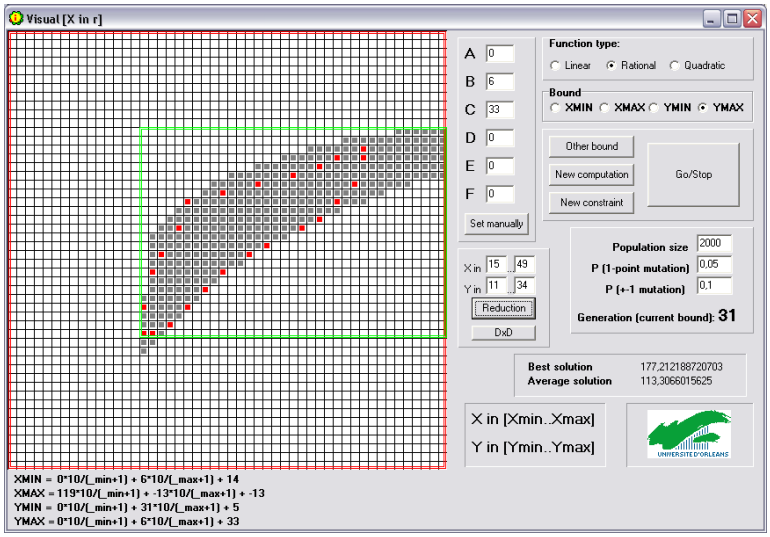

Fig. 3. Rational approximation of a cloud

accepted by the indexicals appear on screen as grey dots. The user may then

[1] ftp://ftp.es.ele.tue.nl/pub/lp_solve ©

specify a domain, which appears as a red box, and visualize the reduced domain yield by his indexicals, which appears as a green box (in figure 3, the user box is $[0..49] \times [0..49]$ and the reduced domain is given by the inner box). Since the learned indexicals may not be monotonic, the reduced interval may be smaller than the singleton solutions. Sufficient conditions for the indexicals to be monotonic can be obtained with syntactic biases. However, correctness with respect to the constraint's solutions is ensured by construction. Singleton completeness is difficult (and sometimes impossible) to ensure. We tried a greedy algorithm to generate new indexicals from the non-covered examples but this does not solve all the cases.

The construction of the indexical is left to a genetic algorithm in which individuals are a sequence of bits representing the coefficients. The quality function is inversely proportional to the square of the error between the candidate and the example set. We tried as definition of error the simple sum of the individual errors on each intervals and the sum of individual errors relative to the size of the interval. Incorrect individuals are strongly penalized. Surprisingly, these notions of error, besides the complexity of their evaluation, exhibit a relatively flat search space with very deep and narrow local minimas. This sometimes causes a slow convergence of the population, which is usually of a few seconds.

6 Examples

We present in this section some examples of learned indexicals, illustrated by reductions using them. The results we present in this section have been obtained with the simplex implementation. First, we mention that the same indexicals as Gnu-Prolog are found for ordinary boolean constraints such as *And*, *Or* or *Xor*.

Three-valued logic. Variables in three-valued logic can have one of the values 1, 0 and 0.5, which stand for true, false and unknown respectively. We present here the And_3 constraint defined in figure 4. From this constraint our algorithm

and_3 :

X_0	X_1	X_2
0	0	0
0	0.5	0
0	1	0

X_0	X_1	X_2
0.5	0	0
0.5	0.5	0.5
0.5	1	0.5

X_0	X_1	X_2
1	0	0
1	0.5	0.5
1	1	1

Fig. 4. The constraint and_3.

generates the following indexicals:

```
X0 in [ Min(X2) ..  1.00 -  Min(X1)*Min(X1) + Min(X1)*Max(X2) ]
X1 in [ Min(X2) ..  1.00 -  Min(X0)*Min(X0) + Min(X0)*Max(X2) ]
X2 in [ Min(X0)*Min(X1) ..  Max(X0) - Max(X0)*Max(X0) + Max(X0)*Max(X1) ]
```

The reduction with $X_0 = 0$, $X_1 \in [0,1]$ and $X_2 = 0.5$ gives:

```
X0 in [ 0.50 .. 0.00 ],  X1 in [ 0.50 .. 1.00 ],  X2 in [ 0.50 .. 0.00 ]
```

We notice that the first and the third indexicals represent the empty interval, which mean that all elements whose $X_0 = 0$ and $X_2 = 0.5$ are rejected by these indexicals. When $X_2 = 0.5$, these indexicals imply that X_0 and X_1 must not be 0, and when $X_2 = 1$, then X_0 and X_1 must be 1. These results are computed by the generated indexicals.

Other examples. A more complete set of examples and experimentations can be found in [6]. For example, we found indexical expressions for the "full adder" constraint `fulladder(X0,X1,X2,X3,X4)` meaning $X_0 + X_1 + X_2 = 2 * X_3 + X_4$ or for the equivalence `equiv3(X0,X1,X2)` in three-valued logic among others.

An interesting point is that the propagator built by the system makes a user-defined constraint behave as an ad-hoc global constraint. Experiments have been made on the all-different constraint for the arity 3 and 4. The pruning power lies between the naive implementation as differences and a specialized implementation [19].

Integration in Gnu-Prolog. When singleton complete indexicals are found, they can be fruitfully integrated in Gnu-Prolog, thus making a propagator for a user-defined global constraint. The main advantage is that the learned indexicals can replace the original indexicals generated by Gnu-Prolog in its compilation process. Moreover, since a global constraint replaces several atomic constraints and since Gnu-Prolog breaks long expressions into smaller ones, the result is that less indexicals have to be scheduled, yielding an increased propagation speed.

For example, for the constraint $0 \leq X - Y \leq 4$, our system generates the two following indexicals: `X in min(Y) .. max(Y)+4` and `Y in min(X)-4 .. max(X)`. In Gnu-Prolog, this constraint has to be written as two constraints $0 \leq X - Y$ and $X - Y \leq 4$, yielding 4 indexicals. Our test consists in trying all possible reductions for the intervals with bounds in [0..20] for both variables. On a laptop Pentium III-m 1GHz, 256 Mb, the time of our indexicals is $35.0s$ while Gnu-Prolog's one is $36.8s$. There is only a difference of two indexicals on this example, hence we can expect better results for more complex expressions. As a comparison, the use of Gnu-Prolog's built-in predicate `fd_relation` with the set of tuples satisfying the constraint leads to a time of $193.9s$. This shows the utility of the method in the general case.

7 Conclusion

Related work. Solver learning is an emerging technique which was pioneered by Apt and Monfroy in [4,5] and Abdennadher and Rigotti [1,2].

In [4] and [5], very simple rules of the form $x_1 = s_1, \ldots, x_n = s_n \rightarrow y \neq a$ are at first considered. They provide a notion of consistency weaker than arc-consistency. Then these rules are extended by replacing equality by membership to achieve arc-consistency. The learning algorithm generates every rule to check

its validity, and redundant rules are eliminated. This framework is language-independent, but the rules have a very restrictive form.

The work of [1] extends the language issue to the far more expressive framework of CHRs. The PROPMINER algorithm consists in an exploration of a user-restricted learning space for the rule's lefthand side and a computation of the righthand side, leading to the choice of the best covering rules. In [2], this work has been extended to handle prolog-like definition of constraints. This work is based on CHR and its rewriting semantics. We propose in contrast to define lower-lovel operators which yield in general faster solvers. CHRs subsume indexicals in term of expressivity (and indexicals can be mimicked by CHRs) but their execution mechanism is different. In addition, our learning algorithm is not related to PROPMINER, which is mostly syntactic. Instead, it is rather closer to the learning of arithmetic functions for which a difficulty is the lack of suitable generalisation ordering on the hypothese space.

Other formalisms compute upper bounds at run-time, such as Generalized Constraint Propagation [17], or Constructive Disjuction [21, 10], but are not designed to statically build a solver.

Summary. In this paper we have presented a general framework of semantic approximations for finite domains CSPs. We applied this framework to the learning of indexical operators [21, 10] which are at the core of the Gnu-Prolog system [11] and built two learning tools. The first one handles constraints of arbitrary arity and uses the simplex algorithm as optimization tool. The second is restricted to binary constraints but proposes a graphical user interface which allows the user to draw his constraint and to visualize the learned expressions. It uses internally a genetic algorithm to achieve the optimization. On regular constraints, such as arithmetic or boolean constraints, Gnu-Prolog indexicals are found. On user-defined constraints, the generated operators are of good quality in term of pruning power.

Acknowledgements. This work has benefited from discussion with Michel Bergère, AbdelAli Ed-Dbali, Gérard Ferrand and Christel Vrain.

References

1. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In Rina Dechter, editor, *Constraint Programming*, volume 1894 of *LNCS*, pages 18–34, Singapore, 2000. Springer.
2. S. Abdennadher and C. Rigotti. Towards inductive constraint solving. In Toby Walsh, editor, *Constraint Programming*, volume 2239 of *LNCS*, pages 31–45. Springer, Nov 26 - Dec 1 2001.
3. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
4. K. R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Constraint Programming CP'99*, 1999.
5. K. R. Apt and E. Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6):713 – 750, 2001.

6. Michel Bergère, Thi Bich Hanh Dao, AbdelAli Ed-Dbali, Gérard Ferrand, Arnaud Lallouet, Andrei Legtchenko, Lionel Martin, and Christel Vrain. Learning interval bounds of indexical-based solvers. Research Report RR-LIFO-2002-07, LIFO, Université d'Orléans, BP 6759, F-45067 Orléans Cedex 2, 2002.
7. Stephano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
8. Mats Carlsson, Greger Ottosson, and Björ Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206, Southampton, UK, September 3-5 1997. Springer.
9. Philippe Codognet and Daniel Diaz. A minimal extension of the wam for clp(fd). In David Scott Warren, editor, *International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, June 21-25 1993. MIT Press.
10. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3):185–226, 1996.
11. Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.
12. Mehmet Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and A. Herold. The CHIP System: Constraint Handling in Prolog. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, Argonne, May 1988. Springer.
13. François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.
14. Thom Früwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
15. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The *CLP*($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
16. François Laburthe and the OCRE project. Choco: implementing a CP kernel. In *TRICS, Techniques foR Implementing Constraint programming Systems, CP 2000 post-conference workshop*, Technical report TRA9/00, Singapore, sep 2000.
17. Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16:319–359, 1993.
18. Jean-Francois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John W. Lloyd, editor, *International Logic Programming Symposium*, pages 513–527, Portland, Oregon, 1995. MIT Press.
19. Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI, National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA,, 1994. AAAI Press.
20. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
21. P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.
22. V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.
23. Neng-Fa Zhou. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In Joxan Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 70–84, Manchester, UK, 15-19 June 1998. MIT Press.