Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2006), *20*, 297–311. Printed in the USA.

DOI: 10.1017/S0890060406060227

Partially defined constraints in constraint-based design

ARNAUD LALLOUET AND ANDREÏ LEGTCHENKO
Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, Orléans, France

(RECEIVED October 17, 2005; ACCEPTED July 6, 2006)

Abstract

In constraint-based design, components are modeled by variables describing their properties and subject to physical or mechanical constraints. However, some other constraints are difficult to represent, like comfort or user satisfaction. Partially defined constraints can be used to model the incomplete knowledge of a concept or a relation. Instead of only computing with the known part of the constraint, we propose to complete its definition by using machine-learning techniques. Because constraints are actively used during solving for pruning domains, building a classifier for instances is not enough: we need a solver able to reduce variable domains. 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: Computer-Aided Design; Constraint Programming; Machine Learning

1. INTRODUCTION

The success of constraint programming takes its roots in its unique combination of modeling facilities and solving efficiency. However, the use of constraint programming is often limited by the knowledge of the constraints that may be appropriate to represent a given problem. It can happen that a model involves a constraint that is only *partially known*, like, for example, if the constraint represents a concept we do not know, or do not want to define in extension. It can be the set of mammals in a description of animals, solar systems inside astronomical data, a preference between possibilities in a configuration problem, the notion of "good" wine, or a habit such as the usually free time slots in somebody's diary. It may also happen that the user does not know which constraint can be used to model the problem because of lack of knowledge in constraint programming, but can easily express examples or counterexamples for it. This situation appears in design (Chandrasekaran, 1999; O'Sullivan, 2002) when a requirement is difficult or is impossible to model. For example, when designing a bicycle, the angle α of the fork (see Fig. 1) has an impact on measurable concepts like the turning circle but also on less well-defined concepts like the ability to go straight when driving hands up or the feeling of comfort of the user. Some of these properties are mutually exclusive and need to be adjusted according to the use of the bicycle. For example, for a mountain bike, a greater importance will be given to maneuverability than to comfort, resulting in a small α value.

Such a constraint can be modeled by examples and counterexamples and impose requirements on the shape of the item when designing a new fork. In addition, other concepts like consumer satisfaction can be modeled alike by giving examples for already built objects.

In this paper, we propose to use partially defined finite domain constraints. In a partially defined constraint, some tuples are known to be true, some other are known to be false, and some are just *unknown*. We make use of this partial knowledge for *learning* the concept that is behind the partially defined constraint. Given positive and negative examples of the concept, the task consists in completing the definition in such a way that new examples never met by the system will be correctly classified. This framework has been extensively studied in machine learning (Mitchell, 1997) under the name of *supervised classification*. In the context studied in this paper, we assume that the full constraint is not available to the system, even by asking other agents or the environment. Hence, there is not a single way to complete a partially defined constraint, just like different people may agree on a set of examples but may

Reprint requests to: Arnaud Lallouet, Laboratoire d'Informatique Fondamentale d'Orléans, BP 6759, F-45067, Université d'Orléans, Orléans, France. E-mail: arnaud.lallouet@univ-orleans.fr

Fig. 1. The angle of a bicycle fork. [A color version of this figure can be viewed online at www.journals.cambridge.org]

have different concepts in mind. In addition, the definition may be revised when the system gets more experience or knowledge. In this paper, we are only concerned by the acquisition of a single constraint. However, its arity may be large.

Partially defined constraints can be learned whenever examples and counterexamples of the relation are available. For example, complex physical systems are often very difficult to model, even if each component is well described and understood. Interactions and uncertainty on some components may cause changes in the behavior of the system. In this case, examples can be provided by a limited number of experimentations or simulations. Then, the approximated subsystem can be used as part of bigger system. Another example comes from the field of *sensorial* analysis where a set of products with known characteristics is presented to a panel of users. The users rate the products according to different factors, and the set of observations can be used to model global preferences of potential customers. This kind of analysis is often used for designing products by the agrofood industry, but can be applied to any product. Sensorial information can be of various kind, like taste, comfort, ergonomy, and so forth. By treating this information as a constraint, it could be possible to handle it from the early stages of design of a product, thus limiting the impact of changes at later stages. Let us take an example in which partially defined constraints occur naturally:

Example 1. The design of a car dashboard is of critical importance for several reason. It must be clear and readable for safety, not tiring. But it is known that these criteria are only one aspect of the problem. Other characteristics like the position, the color, or the size of the counters are major issues in the perception the driver has of his car, and of himself as a driver. For example, it is hardly possible for a sports car to have a small speedometer and no rpm counter. Hence, producing a good design for a dashboard is a clever mixture between functionality, design, and anticipation of consumer taste. In order to integrate these features in the design procedure, we can assume that there is some degree of freedom in the characteristics of the dashboard components, like position, color, or size of the counters. A database of customer tastes has been created by presenting examples of dashboards to a panel of drivers and asking them to select the "good" ones. However, because the number of possible dashboards is very large, it would be impossible to rank each possible one individually. Hence, it is better to ask the users only on a few examples and to model the concept of "good dashboard" as a partially defined constraint.

For the example we propose to use throughout this paper, we have built a database of dashboard (using our own taste, which should not be considered as potentially representative of the common taste of the average driver). The database is composed of the following fields:

- the shape of the speedometer: circle, ellipsoid, half-circle, quarter-circle;
- size of the speedometer: small, medium, large;
- position of the speedometer: left, center, right;
- size of the rpm counter: small, medium, large;
- position of the rpm counter: left, center, right;
- instantaneous gas consumption: yes/no;
- all indicators in one group: yes/no;
- the mileage counter: mechanical, LCD;
- light color: white, blue, red;
- type of fuel gauge: linear, circular;
- the position of the fuel gauge: left, center, right;
- type of water temperature indicator: linear, circular;
- position of the water temperature indicator: left, center, right;
- type of turn signal indicator: light arrow, thick arrow;
- position of the turn signal indicator: high, middle, low;
- character font: Arial, Lucid-Sans, Helvetica, Impact;
- overall color: white, black, grey, brown;
- icon color: black, white, red, orange;
- frame color: black, white, chrome;
- counter face color: black, white, grey;
- text color: black, white, red, orange;
- color of needles: white, red, orange;
- driver's opinion: positive, negative.

The positioning of items is relative to their respective size. For example, the speedometer and the rpm counter are approximately the same size, which means that they cannot be both in the same place. The same does not hold for the fuel gauge and the temperature indicator because they are small items. However, it is possible to have both the rpm counter and the water temperature indicator on the left.

We can consider this database as a partially defined 22-ary constraint (by using the first 22 fields of the database). The

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

modeling of customer preferences as a partially defined constraint allows the system to "invent" new dashboards that are acceptable within the learned concept. One interesting point that justifies the use of constraints in design is that the design of a dashboard for a given car does not obey only to readability and customer preference constraints. There is also other requirements that come, for example, from consistency constraints. For example, the speedometer and the rpm counter cannot be both in the central position (physical constraint) and the fuel gauge and water temperature indicator should be the same color (readability constraint). Finding a dashboard meeting all requirements and following customer preferences is a constraint satisfaction problem. In addition, the dashboard has to fit into a space that is different for every car. By adding information on the surface occupied by each item, we can, for example, find the best dashboard that has the minimum item surface. This is a constraint optimization problem. ■

The idea of the technique we use for learning comes directly from the classical constraint solver model computing a chaotic iteration of reduction operators (Apt, 1999). We begin by learning the constraint. However, instead of learning it by a classifier that 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 classifiers for each of its values and rejected as soon as it gets rejected by one. This method is nonstandard 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 its validity experimentally.

As is, a classifier is only able to perform satisfiability checks for a partially defined constraint. If added to a constraint satisfied problem (CSP), this constraint would not contribute to the reduction of variables domains and it would yield a "generate and test" behavior that could quickly ruin the performances of the system. Hence, it is needed that partially defined constraints should have a *solver* and not only a satisfiability test in order to meet the standards of efficiency of constraint programming. The classifiers we learn are expressed by functions and we turn them into propagators by taking their extension to intervals. This formal transformation does not involve any more learning techniques, thus preserving the properties of the first part. Then the classifiers can be used with variable domains as input. We also show that the consistency they enforce, although weaker than arc consistency, is nevertheless interesting and 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 cardinality. 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 $\prod_{X\in W} D_X$. 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 (denoted by $\mathrm{var}(c)$) and $T \subseteq D^W$ is the set of solutions of c (denoted by $\mathrm{sol}(c)$). Given a set of variables and their respective domains, a CSP is a set of constraints. A solution is a tuple that satisfies all constraints. In this paper, we use the common framework combining *search* and domain reduction by *propagation*. This propagation computes a property called *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 $S_W = \prod_{X\in W} \mathcal{P}(D_X)$. The set S_W, ordered by pointwise inclusion $\subseteq$ is a complete lattice. 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 $|\prod s| = 1$.

A consistency can be modeled as the greatest fixpoint of a set of so-called *propagators* and is computed by a chaotic iteration (Apt, 1999). For a constraint $c = (W, T)$, a *propagator* is an operator f on S_W[1] having the following properties:

- *monotonicity:*

$$\forall s, \quad s' \in S_W, \quad s \subseteq s' \Rightarrow f(s) \subseteq f(s').$$

- *contractance:*

$$\forall s \in S_W, \quad f(s) \subseteq s.$$

- *singleton equivalence*: let $s \in S_W$ be a singletonic state, then

$$\prod s \subseteq \mathrm{sol}(c) \Leftrightarrow f(s) = s.$$

Singleton equivalence means that the operator is also a satisfiability test for a single tuple embedded in a singletonic state. Correctness is also a major property of propagators and it means that a solution tuple never gets rejected across the search space:

Proposition 1. *Let $c = (W, T)$ be a constraint and f be a propagator for c. Then f is correct for c: $\forall s \in S_W, \prod s \cap sol(c) \subseteq \prod f(s) \cap sol(c)$.* ■

Proof: Suppose that f is not correct. Then $\exists s \in S$ and $t \in \mathrm{sol}(c)$ such that $t \in \prod s$ and $t \notin \prod f(s)$. We note $st = (\{t_X\})_{X\in W} \in S_W$ the singletonic state corresponding to t. Thus, we have $st \not\subseteq f(s)$. Because $t \in \mathrm{sol}(c)$ and f is

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

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

singleton equivalent, we have $st = f(st)$. Hence, we have $st \subseteq s$ and $f(st) \not\subseteq f(s)$, which contradicts the fact that f is monotonic. ■

When search begins, the initial search state s_0 is initialized to the entire set of values: $s_0 = (D_X)_{X \in V}$.

Let us now define some consistencies associated to a constraint $c = (W, T)$. The well-known arc-consistency operator (ac_c) is defined by

$$\forall s \in S_W, \quad ac_c(s) = s' \quad \text{with} \quad \forall X \in W, \quad s'_X = \left(\prod s \cap T\right)|_X.$$

If we suppose that each variable domain D_X is equipped with a total ordering $\leq$, we denote by $[a \cdots b]$ the interval $\{e \in D_X | a \leq e \leq b\}$. For $A \subseteq D_X$, we denote by $[A]$ the set $[\min(A) \cdots \max(A)]$. By extension to Cartesian products, for $s = (s_X)_{X \in W} \in S_W$, we denote by $[s]$ the family $([s_X])_{X \in W}$ in which each variable domain is extended to its smallest enclosing interval. The bound-consistency operator (bc_c) is defined by

$$\forall s \in S_W, \quad bc_c(s) = s' \quad \text{with} \quad \forall X \in W,$$

$$s'_X = s_X \cap \left[\left(\prod s \cap T\right)|_X\right].$$

Bound consistency only enforces consistency for the bounds of the domain by shifting them to the next consistent value in the suitable direction. Consistencies are partially ordered according to their pruning power and we have $f \subseteq f'$ if $\forall s \in S_W, f(s) \subseteq f'(s)$.

Because only variables domains are reduced, a consistency operator f for a constraint $c = (W, T)$ can be split into $|W|$ projection operators $(f_X)_{X \in W}$ according to each variable of the constraint. By confluence of chaotic iterations (Apt, 1999), these operators can be scheduled independently as long as they follow the three first properties of consistency operators. In order to represent the same constraint, they have to be singleton complete collectively. It is worth noticing that there is a disymmetry between reject and acceptance and that for satisfiability, a nonsolution tuple must be rejected (at least) by *one* of these operators while correctness imposes that a solution tuple is accepted by *all* operators.

The role of a consistency operator f_X is to eliminate from the domain of its target variable X some values that are unsupported by the constraint. Arc consistency eliminates all inconsistent values. Thus, it has to find a *support* for each considered value a (a solution tuple whose projection on the target variable is a) in order to allow the value to remain in the variable's domain. This task has been proved to be NP-complete in general (Bessière, Coletta, et al., 2004) for n-ary constraints. While many useful constraints have polynomial-time arc-consistency propagators, some exist for which this task is intractable. Because we are dealing here with constraints expressed by examples, this case must be taken into account seriously and motivates an approximation scheme.

At a finer level of granularity, the way a support is computed has motivated a constant improvement in arc-consistency algorithms. The first technique consists in keeping a table that records every tuple of the constraint and sweeping this table every time a support is needed. This technique has been the most optimized, and optimal algorithms only check each value once and keep minimal data structures like the general arc-consistency (GAC) schema (instantiated by a table; Bessière & Régin, 1997). These techniques are well suited to highly irregular constraints. In Figure 2 are depicted the projections of a ternary constraint $c(X, Y, Z)$ on the three values of Xs domain for different types of constraints. On each projection black squares denote allowed tuples. Figure 2a illustrates on which type of constraint table-based techniques are well suited, but the problem is that their space requirements grow according to domain size and constraint arity.

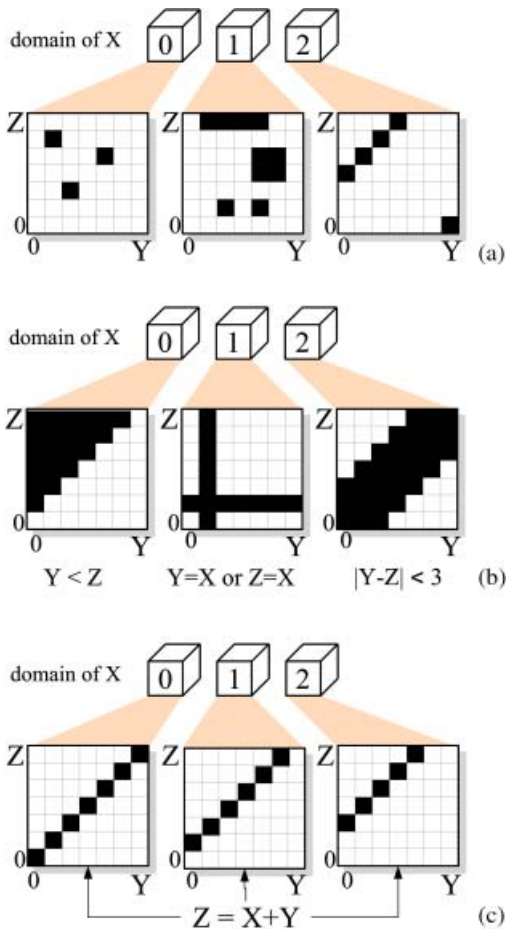

Fig. 2. A granularity analysis of consistencies. In this schema, a black dot represents a tuple of the constraint. All axes are labeled horizontally by Y and vertically by Z. Each square represents the projection of some constraint on the plane orthogonal to one of the values of X. [A color version of this figure can be viewed online at www.journals.cambridge.org]

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

Conversely, if the relation is very regular, it can happen that a single expression in a given language can capture the notion of support for every value of the domain. This opportunity is exploited by the indexical language (van Hentenryck et al., 1991). For example, in Figure 2c, all supports for the variable X of the constraint $Z = X + Y$ can be computed using the same language expression [in this case, X in $\mathrm{dom}(Z) - \mathrm{dom}(Y)$]. The "regularity" of the constraint can be measured as the length of the shortest expression for an operator in a given language. Although indexicals are powerful enough to represent any constraint operator, this can be done in some cases at the price of an expression proportional to the length of the table itself. Indexicals are thus best suited for operators that enjoy a short expression. Another advantage of the indexicals language is that it allows to express easily weaker consistencies such as bound consistency by associating an expression directly to the bounds of the variable. The issue of computing an approximation of a consistency using indexicals has been tackled in Lallouet et al. (2003).

However, there exists an intermediate situation depicted in Figure 2b where it is possible to find a compact representation for finding supports, but this representation is *different* for every value of the domain of the variable to be reduced. In other words, in order to represent an operator f_X, a Boolean function is associated to each value a of X's domain. This function evaluates to *false* if the value has no support and to *true* if one is found or if an incomplete search is not able to decide. We call this function an *elementary reduction function* (ERF) and denote it by $f_{X=a}$. By combining elementary reduction functions, we are able to build an operator for the target variable: all we have to do is to collect the answers of the functions and intersect the produced domain with the variable's current domain. Hence, for a search state s, if we assume that we have a set of ERFs $\{f_{X=a} | a \in s_X\}$, we build the operator f_X as $f_X(s) = s'$, where $s'_X = s_X \cap \{a \mid f_{X=a}(s)\}$. For bound consistency, instead of sweeping the entire domain, an upward *while* loop can be used to move the minimum bound to its next supported value (respectively downward for the max bound).

Compared to the first technique (Fig. 2a), this representation is likely to take less space because we can bound the size of each ERF. Then, the space requirement grows only according to the domain sizes. However, it puts some limits to the expressivity of the functions that can be used and arc consistency may not be representable for some constraints. Compared to the second technique (Fig. 2c), it is more versatile because different functions can be used to compute the support of different values.

By using ERFs, we are able to give each value its own support function. A function $f_{X=a}$ takes as input the current domain of the constraint's other variables. However, it may make full use of this information or just use it partially. For example, it can only use the *bounds* of the domains. From these combinations, we get four consistencies: two are classical and two are intermediate, denoted by ac^- and bc^+ (see Table 1).

Table 1. *ERFs and consistencies*

Domain Reduction of X	ERF Uses: All Values	ERF Uses: Bounds Only
All values	ac	ac^-
Bounds only	bc^+	bc

In other words, if we give each domain value an ERF but if we assume that this ERF takes as input only the bounds of the other variables' domains, we get an intermediate consistency we call ac^-:

$$\forall s \in S_W, \quad ac_c^-(s) = s' \quad \text{with}$$

$$\forall X \in W, \quad s'_X = s_X \cap \left(\prod [s] \cap T \right)|_X.$$

It does not have the full power of arc consistency because it makes use of less input information but may reduce more than bound consistency because not only the bounds can be reduced. The counterpart, bc^+, is when bounds can be reduced by a function taking as input all information available in the whole domain of the other variables:

$$\forall s \in S_W, \quad bc_c^+(s) = s' \quad \text{with} \quad \forall X \in W,$$

$$s'_X = s_X \cap \left[\left(\prod s \cap T \right)|_X \right].$$

PROPOSITION 2. *$ac \subseteq bc^+ \subseteq bc$ and $ac \subseteq ac^- \subseteq bc$.* ■

PROPOSITION 3. *ac^- and bc^+ are uncomparable.* ■

3. PARTIALLY DEFINED CONSTRAINTS

In this section, we give the definition of partially defined constraints and introduce the notion of *extension*, which provides a closure of the constraint.

A classical constraint $c = (W, T)$ is supposed to be known in totality. The underlying *closed world assumption* (CWA) states that what is not explicitly declared as true is false. Hence, the complementary $\overline{T}$ is the set of tuples that do not belong to c. In the following, we call ordinary constraints under CWA *closed* or *classical* constraints. When dealing with incomplete information, it may happen that some parts of the constraint are unknown (see Fig. 3):

DEFINITION 1 (partially defined constraint). A partially defined constraint is a triple $c = (W, c^+, c^-)$ *where* $c^+ \subseteq D^W, c^- \subseteq D^W$, and $c^+ \cap c^- = \emptyset$. ■

In a partially defined constraint $c = (W, c^+, c^-)$, c^+ represents the allowed tuples and c^- the forbidden ones. The remaining tuples, $\overline{c^+ \cup c^-}$, are simply unknown. Note that

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

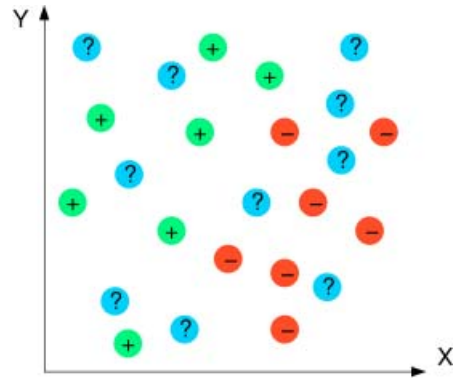

Fig. 3. A partially defined constraint. [A color version of this figure can be viewed online at www.journals.cambridge.org]

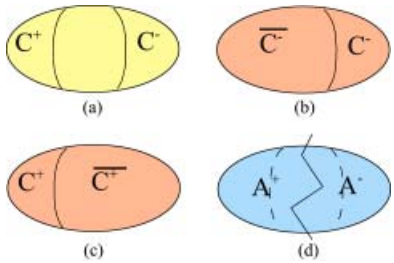

Fig. 4. (a) A partially defined constraint and its (b) cautious, (c) brave, and (d) algorithmic extensions. [A color version of this figure can be viewed online at www.journals.cambridge.org]

a classical constraint $c = (W, T)$ is a particular partially defined constraint $c = (W, T, \overline{T})$ for which the negative part is the complement of the positive part.

Partially defined constraints need a special treatment in order to be used in a CSP because little propagation can be done without knowing the integrality of the constraint. Hence, a partially defined constraint needs to be *closed* to be usable in a constraint solving environment. The closure of a partially defined constraint c is done by choosing a class (it belongs or does not belong to the constraint) for all unknown tuples. We call the resulting classical constraint an *extension* of the partially defined constraint.

Definition 2 (extension). Let $c = (W, c^+, c^-)$ be a partially defined constraint. A (classical) constraint $c' = (W, T)$ is an *extension* of c if $c^+ \subseteq T$ and $c^- \subseteq \overline{T}$. ■

In other terms, an extension is a classical constraint compatible with the known part (positive and negative) of the partially defined constraint. A partially defined constraint allows us to catch a glimpse of a hidden reality and one of its extensions corresponds to the genuine relation of the world. In most cases, the knowledge of this constraint is impossible to get and all that can be done is computing an approximation of it. In general, many extensions can be considered, and let us introduce three of them. Let $c = (W, c^+, c^-)$ be a partially defined constraint. We denote an extension of c by $[c]$.

- *Cautious* extension: $[c]_c = (W, \overline{c^-})$. All unknown tuples are assumed to be true (Fig. 4b). A solver generated according to this extension is cautious in the sense that it will not prune the search space for any unknown tuple. Thus, the consequence of a bad choice for this tuple will not compromise the correctness of a solution. The counterpart is that if the unknown part of the constraint is large, it will yield weak pruning and the user will be provided with many unwanted false solutions.
- *Brave*[2] extension: $[c]_b = (W, c^+)$. All unknown tuples are assumed to be false (Fig. 4c) as in the classical CWA. A solver generated according to this extension is brave because it will prune the search space as soon as possible. If the tuple was incorrectly classified as false, correctness is lost and a nonmonotonic restoration mechanism is needed, like in dynamic CSPs (Verfaillie & Jussien, 2003).
- *Algorithmic* extension $[c]_{\mathcal{A}}$: let $\mathcal{A}:D^W \to \{0, 1\}$ be a tuple classification algorithm such that $t \in c^+ \Rightarrow \mathcal{A}(t) = 1$ and $t \in c^- \Rightarrow \mathcal{A}(t) = 0$. Then $[c]_{\mathcal{A}} = (W, \{t \in D^W | \mathcal{A}(t) = 1\})$ (Fig. 4d).

We are mostly interested in the last extension in which the unknown part is completed by a learning algorithm.

This kind of extension is obtained by supervised classification: it consists in the induction of a function that associates to all tuples a class from a set of examples given with their respective class. Machine learning puts strong requirements on what is called a good algorithmic extension. The initial and perhaps most important step is that, the correct class for unknown tuples should be forecast with the highest possible accuracy. The ratio between the number of correctly classified tuples and the number of presented tuples defines the *correctness ratio* of the generalization. Most techniques used in machine learning provide much better performances than random classification, and more than 90% of success in prediction is not unusual. In order to achieve this, we assume that the known part of the partially defined constraint should be *representative* of the whole underlying constraint and that the underlying constraint has some *regularities* that can be approximated. Then, the representation of the classification function is searched in a space of possible functions called *hypothesis space*. A *learning algorithm* finds the best possible function in this space by optimizing some criteria, like accuracy, simplicity, or generalization, and so forth.

[2]Note that we choose the terminology of *cautious* and *brave* with respect to the solver and not to the constraint. Indeed, making all unknown tuples true can be considered as *brave* with respect to the constraint.

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

4. CONSTRAINT ACQUISITION

At first, we address the problem of constructing a good extension for a partially defined constraint. In order to represent a relation, the first idea is to build a classifier taking as input an instantiation of all variables of the relation and returning a Boolean stating if the tuple belongs or not to the relation. However, unfortunately, although learning is effective with this technique (see Rossi & Sperduti, 2004), it would be difficult to extract a solver from this representation. Motivated by the equivalence between a constraint and a correct and singleton complete solver, we propose to acquire a partially defined constraint $c = (W, c^+, c^-)$ by learning the support function for all values of domain variables. More precisely, 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 an artificial neural network (ANN) with an intermediate hidden layer. This representation has been chosen for its good properties in learning and the possibility of a further transformation into a solver. Other kinds of classifiers can be used but we cannot describe them for lack of space. For $W \subseteq V$, a *neuron* is a function $n(W)$:$\mathbb{R}^{|W|} \to \mathbb{R}$ computing the weighted sum of its inputs followed by a threshold unit. A dummy input is added to tune the threshold. This one is assigned to 1. The sigmoid function is often chosen for the threshold unit because derivability is an important property for the learning algorithm. Let $(w_X)_{X \in W}$ be the weights associated to each input variable and w_0 be the adjustment weight for the dummy input. Hence, the function computed by a neuron taking as input $a = (a_X)_{X \in W}$ is

$$n(a) = \frac{1}{1 + \exp\left(w_0 - \sum_{X \in W} w_X \cdot a_X\right)}.$$

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 5. 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 $n_{\langle X=a \rangle}$ the network concerning $X = a$. Because neurons are continuous by nature, we use an analog coding of the domains. Let D be a finite domain and $<$ a total order on D (natural or arbitrary), then we can write D as $\{a_0, \ldots, a_n\}$ with $\forall i \in [1 \cdots n], a_{i-1} < a_i$. According to this order, we can map D onto $[0 \cdots 1]$ by coding a_i by i/n. In a similar way, the output is in the interval $[0 \cdots 1]$

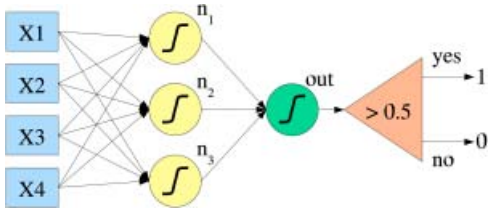

Fig. 5. The structure of the artificial neural network. [A color version of this figure can be viewed online at www.journals.cambridge.org]

and we choose as convention that the value a should be removed from the domain of X if *out* ≤ 0.5. This threshold is the last level of the network depicted in Figure 5.

The global classifier for the partially defined constraint is composed of all of these elementary classifiers for all values in the domain of all variables $\{n_{\langle X=a \rangle} | X \in W, a \in D_X\}$. 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 $(n_{\langle X=t|_X \rangle}(t|_{W-\{X\}}))_{X \in W}$ 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 partially defined constraint, these classifiers are trained on examples and counterexamples selected from the projection of the known part of the constraint on the subspace orthogonal to a variable's value. For $E \subseteq D^W$, $X \in W$ and $a \in D_X$, we denote by $E_{\langle X=a \rangle}$ the selection of tuples of D^W having a as value on X: $E_{\langle X=a \rangle} = \{t \in E \mid t|_X = a\}$. Thus, in order to build the classifier $n_{\langle X=a \rangle}$, we take the following sets of examples and counterexamples:

$$e^+_{\langle X=a \rangle} = c^+_{\langle X=a \rangle}|_{W-\{X\}},$$

$$e^-_{\langle X=a \rangle} = c^-_{\langle X=a \rangle}|_{W-\{X\}}.$$

For example, for a partially defined constraint defined by $W = \{X, Y, Z\}$, $c^+ = \{(1, 1, 0), (1, 0, 1)\}$ and $c^- = \{(1, 1, 1)\}$, we get

$$e^+_{\langle X=1 \rangle} = \{(1, 0), (0, 1)\}.$$

$$e^-_{\langle X=1 \rangle} = \{(1, 1)\}.$$

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

Table 2. *Learning results*

Database	Dashboard	Mushroom	Cancer	Votes-84
Arity	22	22	9	16
Size of				
DB	334	8124	699	435
Domain	2–4	2–12	10	3
Number of				
Neurons in HL	3	3	5	5
Classifiers	64	116	90	48
Learning time	55 min	2.5 h	8.5 h	4.5 h
General ratio				
SOLAR veto (SD)	88.08 (3.66)	93.06 (1.91)	95.36 (1.80)	74.07 (3.82)
SOLAR major (SD)	96.36 (3.51)	99.29 (0.99)	96.52 (2.15)	96.23 (3.26)
C5.0 (SD)	90.14 (6.87)	99.19 (1.21)	94.54 (2.44)	96.29 (3.39)
C5.0 boost (SD)	95.17 (3.42)	99.80 (0.61)	96.33 (2.26)	95.63 (3.63)

The networks are trained by the classical backpropagation algorithm (Rumelhart et al., 1986), which finds a value for the weights using gradient descent. The algorithm is stopped when all examples and counterexamples are correctly classified. This requirement comes from the need of correctness of constraint programming but it may be adjusted according to the application and to how noisy the training set is. In general, the choice of the architecture of a neural network is a complex problem that does not have a theoretical solution. In a similar way, the number of examples required to learn a relation with sufficient accuracy depends on the learning technique and the difficulty of the concept. It is nevertheless known that a multilayer perceptron with only one hidden layer is a universal approximator (Hornik et al., 1989). But the theory does not make it possible to determine how many neurons in the hidden layer are necessary to approximate a given function. Thus, this number is found in an empirical way starting from a relatively small value. This is done in order to minimize the complexity of the classifier and to improve the quality of generalization. In many cases, it is better to keep a small network size in order to preserve its generalization capabilities.

Because the technique we propose for learning relations is not classical in machine learning, we present validation results to show that the concept lying behind the partially defined constraint is actually learned. This framework has been implemented in a system called SOLAR and tested on the *dashboard* example presented in the introduction and on various machine learning databases[3] used as constraint descriptions. This is in contrast with classical constraint programming experiments, for example on random CSPs, because we need to be sure that there is an actual concept behind the partially defined constraint for the learning to make sense. With random constraints, no learning is possible. The results are summarized in Table 2. The database *mushroom* gives attributes to recognize edible candidates, *breast-cancer-wisconsin* to recognize benign and malignant forms of disease, and *house-votes-84* to classify democrats and republicans. For the dashboard and mushroom constraints, we have three neurons in the hidden layer whereas we have five for breast-cancer-wisconsin and house-votes-84.

We compare the generalization performance of our technique to the popular decision tree learning system *C5.0* (RuleQuest Research, 2004). For each base, we have performed a cross-validation by using the following method: we cut the base in 10 parts, we use 9 of them for learning and the 10th for validation. This process is repeated 10 times, each part of the base being used in turn for validation. The cutoff is identical for the test with all methods. Then, the whole test is repeated on five sessions with different cutoffs, yielding 50 tests for each technique. The generalization ratio is the percentage of correctly classified tuples.

Table 2 contains three parts. The first one contains a description of the data: database name, arity, size, and size of the variables' domains. Then follows some information about the learned classifiers: the number of neurons in the hidden layer, the number of individual classifiers learned, and the learning time. In comparison, the learning time for C5.0 is very low, typically a few seconds, but the interest of our technique is not only for classification, as described in the next section. The last part presents the generalization results: the generalization ratio and standard deviation (SD) for SOLAR with veto vote, for SOLAR with majority vote, and for C5.0 and for C5.0 with boosting (with number of trials equal to the arity of the constraint in order to balance our number of classifiers). The mushroom database is very easy to learn. Hence, we only used 500 examples out of 8124 for all techniques; otherwise, they all would have reached 100%. Let us note that the techniques of machine learning such as decision trees and ANNs require a rela-

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

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

tively large set of examples to be effective. More generally, it is the case for any statistical approach of learning. Unfortunately, in certain real cases it can be difficult (expensive or just impossible) to obtain a large database of examples. In these cases the use of our technique could be ineffective, in terms of the quality of learning and generalization. Nevertheless, the approach is well adapted for the situations where enough data are provided by simulation or when sufficient experimental data are available.

The technique we propose challenges powerful techniques such as boosting (Freund & Shapire, 1999), both in generalization performance and scattering of results as measured by SD 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. In veto mode, the learned concept is more focused on the core of the real concept as we impose more elementary classifiers to agree on a tuple. Thus, it is not surprising that veto mode performs less satisfactorily than majority mode. The tuples that are accepted at unanimity are in some sense the most "pure" ones with respect to the real concept and the error depicted in Table 2 corresponds to rejected positive tuples and never to accepted negative ones. For optimization purposes, this could even be an advantage because the solution space is smaller and the correctness of the answer is better preserved.

5. FROM CLASSIFIERS TO SOLVERS

When added in a CSP, a constraint will have an active behavior; it should contribute to the domain reduction. This is why simply keeping the classifier as a definition of the constraint is not an interesting option. The induced "generate and test" behavior would not prune the search space at all. Another idea could be to first generate offline the solutions of the extension of the constraint and use them for solving with a standard but efficient arc-consistency propagation algorithm like the GAC schema (Bessière & Régin, 1997). However, unfortunately, this method suffers from two major drawbacks. First, the generation time is prohibitive because some constraints have a huge number of solutions. For example, 3 h of "generate and test" computation on the mushroom database could hardly produce 76,835 solutions out of 1.5×10^7 tries. Second, a problem comes from the representation size of the relation. The extension of the 22-ary constraint mushroom contains more than 4E6 solutions and would thus need more than 88 MB of memory to be stored. In contrast, the representation we have is rather economic. For a constraint of arity n, if we assume that the hidden layer contains m neurons and the size of the domains is d, it requires $n(n+1)dm + 1$ floats (10 bytes) to store the weights. For the dashboard constraint ($n = 22$, $m = 3, d = 4$), we attain a size of 60 kb, for mushroom ($n = 22$, $m = 3, d = 12$), 180 kb.

We propose to use the learned classifiers also for solving. In order to do this, let us recall some notions on interval analysis (Moore, 1966). We call $\mathrm{Int}_{\mathbb{R}}$ the interval lattice built on the set $\mathbb{R}$ of real numbers. All functions have extensions to intervals.

Definition 3 (extension to intervals). Let $f: \mathbb{R} \to \mathbb{R}$ be a function. A function $F: \mathrm{Int}_{\mathbb{R}} \to \mathrm{Int}_{\mathbb{R}}$ is an extension to intervals of f if $\forall I \in \mathrm{Int}_{\mathbb{R}}, \forall x \in I, f(x) \in F(I)$. ■

An extension F is monotonic if $A \subseteq B \Rightarrow F(A) \subseteq F(B)$. Between all extensions to intervals of f, there is a smallest one, called *canonical extension to intervals*: $\hat{f}(I) = [\{f(x) | x \in I\}]$. The canonical extension is monotonic. Here are the canonical extensions to intervals of the operators we use in classifiers:

$$[a, b] + [c, d] = [a + c, b + d],$$

$$[a, b] - [c, d] = [a - d, b - c],$$

$$[a, b] \times [c, d] = [\min(P), \max(P)],$$

where $P = \{ac, ad, bc, bd\}$

$$[a, b]/[c, d] = [a, b] \cdot [1/d, 1/c] \quad \text{if } 0 \notin [c, d],$$

$$\exp([a, b]) = [\exp(a), \exp(b)].$$

Division is not a problem in our setting because no interval contains 0 (see the sigmoid denominator). If e is an expression using these operators and E the same expression obtained by replacing each operator by a monotonic extension, then $\forall I \in \mathrm{Int}_{\mathbb{R}}$, $\forall x \in I, e(x) \in E(I)$. This property of monotonic extensions is called "the fundamental theorem of interval arithmetic" (Moore, 1966). 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 do not always obtain an extension E that is canonical. We instead call it the *natural* extension. Multiplication is only subdistributive in interval arithmetic (Moore, 1966), that is, $A \times (B + C) \subseteq (A \times B) + (A \times C)$. Hence, the natural extension is canonical only for expressions with single variable occurrences (single occurrence theorem, Moore, 1966).

An elementary classifier $n_{\langle X=a \rangle}$ defines naturally a Boolean function of its input variables. Let $N_{\langle X=a \rangle}$ be its natural interval extension, defined by taking the canonical extension of each basic operator $+, -, \times, /$, exp. Then, by using as input the current domain of the variables, we can obtain a range for its output. In order to do this, we compute the interval range of every neuron of the hidden layer and we use these results to feed the output neuron and compute its domain. Because 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.

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

PROPOSITION 4. *$N_{\langle X=a\rangle}$ is an ERF.* ■

Proof: It is only needed to check the correctness of the function applied to a search state s with respect to the partially defined constraint's accepted tuples. If a tuple t such that $t|_X = a$ belongs to the solutions of the learned constraint, then $n_{\langle X=a\rangle}((t|_Y)_{Y\in W-\{X\}}) = 1$. Hence, if $t \in \prod s$, we have $\max(N_{\langle X=a\rangle}(s|_{W-\{X\}})) = 1$ because N is a monotonic extension. ■

By doing this for every value of every variable's domain, we are able to define a consistency operator f_X that gathers the results of the ERFs. For $s \in S_W$, $f_X(s) = s'$ where $s'_X = s_X \cap \{a \in D_X | \max(N_{\langle X=a\rangle}(s|_{W-\{X\}})) = 1\}$ and $s'_Y = s_Y$ for $Y \neq X$.

PROPOSITION 5. *The operators $(f_X)_{X\in W}$ define a consistency for c.* ■

Proof: Each operator f_X is monotonic, contractant and correct (by the fundamental theorem of interval arithmetic). They are together singleton complete (because the extension of the partially defined constraint is defined by them). ■

We call *learned consistency* (lc) the consistency defined by the learned propagators. Because we use multiple occurrences of the same variable, the lc computes an approximation of ac^-.

PROPOSITION 6. *$ac^- \subseteq lc$.* ■

Proof: Because multiple occurrences of variables yield a less precise interval, it follows that the maximum of the output interval of the last neuron of an ERF $N_{\langle X=a\rangle}$ may exceed 0.5 even if there is no support for $X = a$. Thus, the value is not suppressed as it would be by ac^-. ■

Note that if we had used single-layer perceptrons, the extension would have been exact and we would have been closer to ac^-. However, single-layer perceptrons have severe limitations in learning (Mitchell, 1997). 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 (Apt, 1999). The exact ac^- consistency could be reached only if all elementary classifiers agree on each value, which is not the case in general. On the one hand, this allows us to strengthen the learning power as examplified in the preceding section; but on the other hand, it weakens the propagation a little bit.

The SOLAR system takes a partially defined constraint as input and outputs a set of consistency operators that can be adapted to any solver. In our experiments, we used a custom-made solver. We made two sets of experiments in order to evaluate the learned consistency.

The first one is summarized in Table 3 and describes the search for all solutions using the lc. It is done by taking a CSP containing the partially defined constraint alone. The results are averages on 10 runs for dashboard, breast-cancer-wisconsin, and house-votes-84 databases. Only one run was done for mushroom database. Each run consists of learning a constraint followed by the resolution. Indeed, two runs of backpropagation algorithm with a different random initialization of weights provide slightly different results. For every partially defined constraint, we use our system to count the number of solutions (#Sol). Because we do not have arc consistency, we record the number of failures lc makes while finding these solutions (#Fail lc). Then we compute the ratio lc = #Fail lc/#Sol. If we had arc consistency, there would not be any backtracks. The purpose of this experiment is to compare lc to what ac could have done if ac was available for partially defined constraints. In terms of failure, the average ratio on all experiments is of 1.6 failures per solution. This result should be put into balance with the huge number of failures "generate and test" would have done. We also report the time lc needs to find these solutions (only the time of the resolution; see Table 2 for the learning time). The mushroom constraint has a very large space and a medium tightness and we could not obtain its full extension. However, for the other constraints, this is the only method to get all solutions since the Cartesian products of the domains are so large that this prevents the use of generate and test with the classifier.

Table 3. *Results for solutions and failure*

Database	Dashboard	Mushroom	Cancer	Votes-84
Avg. number of				
Solutions	7.4E5	≥4.1E6	1.27E5	1.27E5
Failures lc	1.34E6	≥3.1E6	1.28E5	3.47E5
Average				
Ratio lc	1.8	0.75	0.99	2.86
Time lc	5.25 h	≥2 h	3 h	7.5 h

Our second set of experiment is a random sampling of the reductions made by the different consistencies on random search states (the domain of each variable are arbitrary sets, not intervals). These results are depicted in Tables 4–7.

For each constraint, we give the number of tuples of the initial search state ($|\prod s|$) and the number of tuples after an application of each of the operators lc, bc, bc^+, ac^-, and ac. The data are the average of 1000 experiments with the same average size of search state. In order to compute exactly the consistencies bc, bc^+, ac^-, and ac, we have first computed all solutions included in s in a table with the help of lc. In a second step, we have computed all needed projections from the solution table. For example, the last column of Table 4 for the dashboard example shows that, starting from a search space containing 166,563 tuples (in its Cartesian product), the learned consistency reduces it to a search space of 9037 tuple whereas arc consistency reduces it to 1326 tuples (all these values are average). This shows that the learned consistency is weaker than more classical consistencies but still reduces notably the search space. Fig-

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

Table 4. *Consistency tests for dashboard*

$\|s\|$	29.881	127.406	561.79	2381.654	6721.499	24929.11	64928.5	166563.9
lc	0.003	0.106	0.31	4.094	104.220	1138.65	2440.9	9037.4
bc	0.001	0.005	0.04	0.256	31.889	473.99	280.5	2315.9
bc^+	0.001	0.005	0.04	0.260	31.888	473.15	275.8	1998.2
ac^-	0.001	0.005	0.03	0.237	22.210	257.37	186.3	1533.7
ac	0.001	0.005	0.03	0.233	22.209	256.63	185.0	1326.5

Table 5. *Consistency tests for mushroom*

$\|s\|$	24.53	500.2	3711.2	23249.4
lc	1.57	181.6	1067.2	10226.1
bc	0.74	194.2	451.6	4078.4
bc^+	0.69	147.8	430.1	3933.2
ac^-	0.55	126.4	275.6	2148.4
ac	0.52	96.0	261.4	2084.1

Table 6. *Consistency tests for breast cancer*

$\|s\|$	11.663	54.70	366.6	997.5	3224.7	8003.6	32014.0
lc	1.166	8.31	104.6	351.5	1483.4	4350.8	22142.7
bc	0.111	1.17	50.5	127.8	1012.6	1907.2	9423.7
bc^+	0.108	0.93	37.1	101.3	955.5	1550.0	7727.8
ac^-	0.085	0.78	33.6	84.9	838.0	1528.7	6500.2
ac	0.083	0.72	22.0	69.4	686.4	1260.7	4998.0

ures 6–9 depict a graphical view of *lc* compared to the other consistencies for all examples (using data of Tables 4–7).

In addition, the 22-ary partially defined constraint dashboard has been tested in different optimization problems as described in Example 1. As expected, the best solution found is a disposal of the different items *invented* by the system and that was not in the database. Every problem comprises the set of 22 variables describing a dashboard, the dashboard constraint, and some constraints specific to each problem. The optimization criterion consists in minimizing the overall surface, subject to physical and preference constraints. The computed surface corresponds to the global surface of the actual items and not of the dashboard they define, which is larger. For this purpose, an elementary surface is associated to each item, as well as to some particular combinations that take into account a reasonable separating space. Hence, the

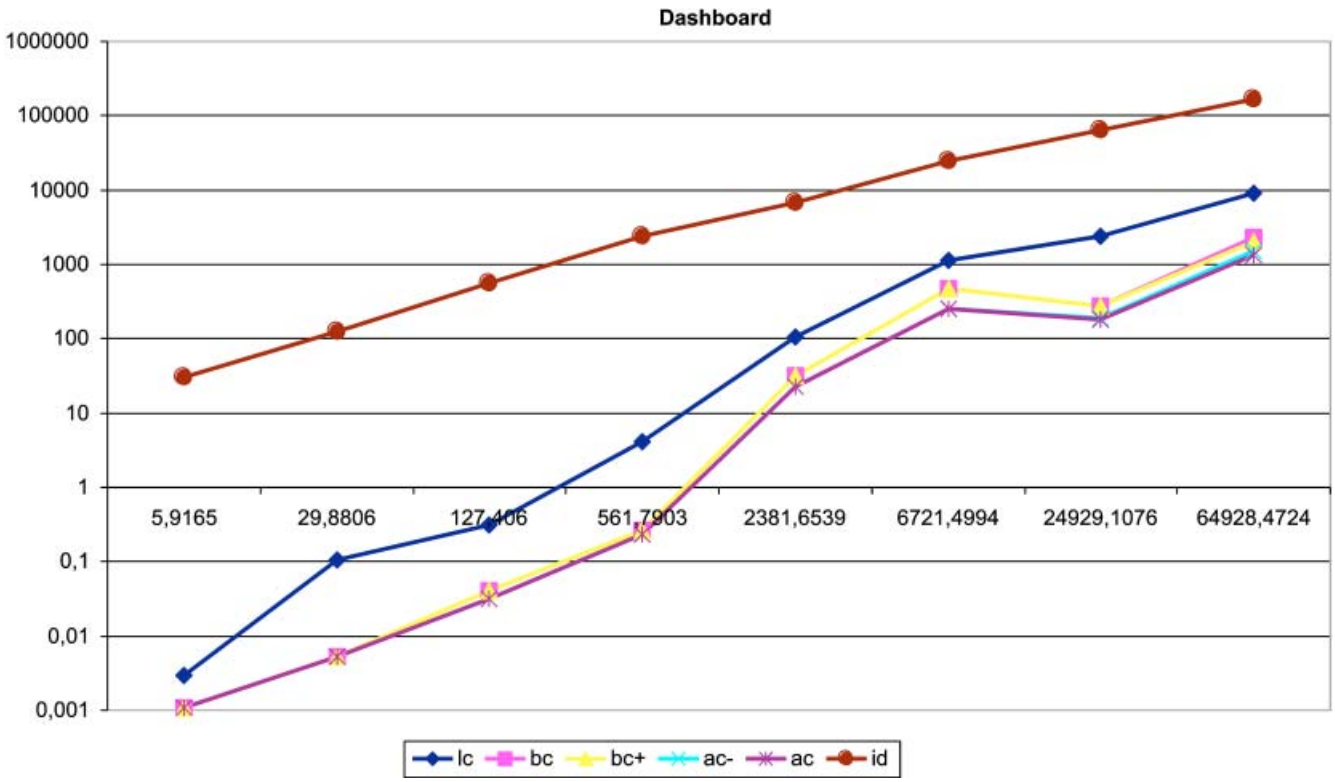

Fig. 6. Tests on the *dashboard* database. [A color version of this figure can be viewed online at www.journals.cambridge.org]

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

Table 7. *Consistency tests for votes-84*

$\|s\|$	4.5207	18.408	68.62	229.5	766.9	2096.6	4159.5	13752.9	25147.4
lc	0.1256	1.352	11.98	53.6	231.4	858.9	1670.5	7787.2	12597.4
bc	0.0334	0.225	3.45	14.0	82.0	456.0	850.6	3990.5	6792.2
bc^+	0.0330	0.217	3.28	12.7	77.4	441.1	796.9	3450.0	6038.1
ac^-	0.0326	0.219	3.43	13.9	81.9	453.3	850.1	3984.5	6777.4
ac	0.0322	0.211	3.26	12.6	77.3	438.5	796.4	3444.0	6027.4

objective function is more complex than just a sum of the individual surfaces. For every problem, a classical branch and bound search has been used. We have compared the time needed to solve the optimization problem in three contexts.

1. *Learned:* The learned constraint and its propagators are used as described in this paper.
2. *GAC:* All solutions of the constraint have been produced using our technique and have been put in a table. The table is processed during search by using an hyperarc-consistency algorithm.
3. *Classifier:* Only the classifier has been used for the partially defined constraint during search. This means that most of the time, a dashboard is generated and refused at the very last time by the classifier (generate and test behavior).

The results follow.

Problem 1. The CSP is composed of four constraints: the learned dashboard constraint, position of speedometer ≠ position of rpm counter, text color ≠ face color, and icons color ≠ face color.

- *learned:* 2.23 s.
- *GAC:* 31.34 s.
- *classifier:* 2 h, 55 min, 46 s.

Minimal surface = 158 cm^2.

Problem 2. The CSP is composed of three constraints: the learned dashboard constraint, rpm counter size = large, and rpm counter position = center.

- *learned:* 2.94 s.
- *GAC:* 25.67 s.
- *classifier:* >4 h (time out).

Minimal surface = 217 cm^2. An artistic view of the solution of this example is depicted in Figure 10.

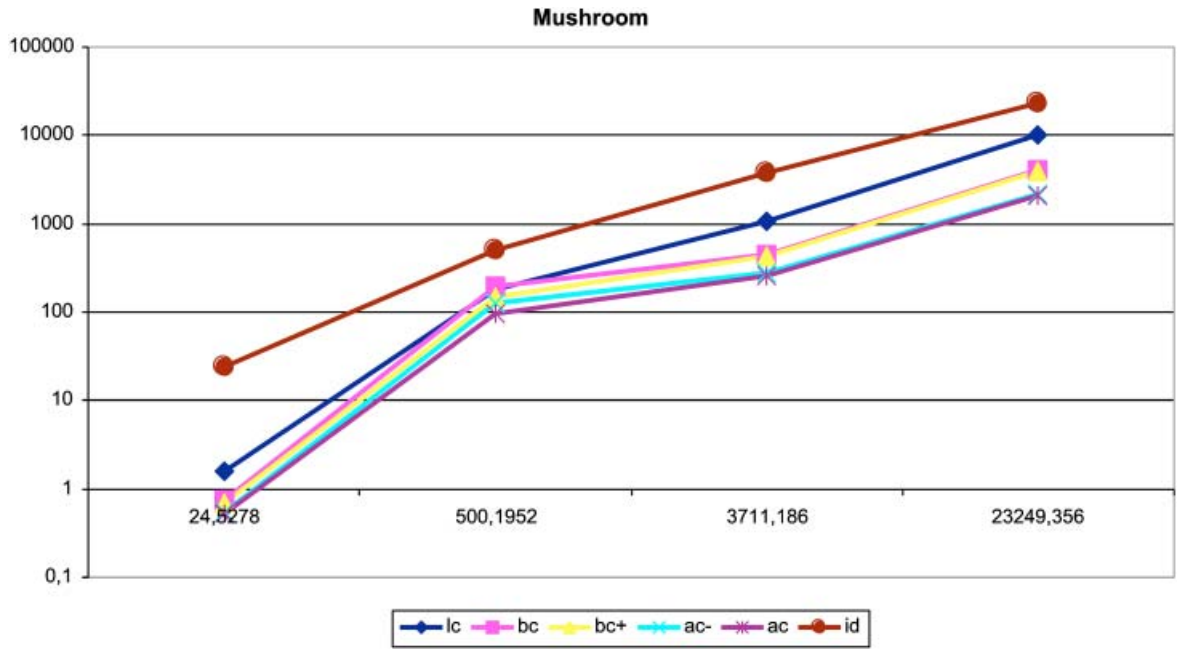

Fig. 7. Tests on the *mushroom* database. [A color version of this figure can be viewed online at www.journals.cambridge.org]

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

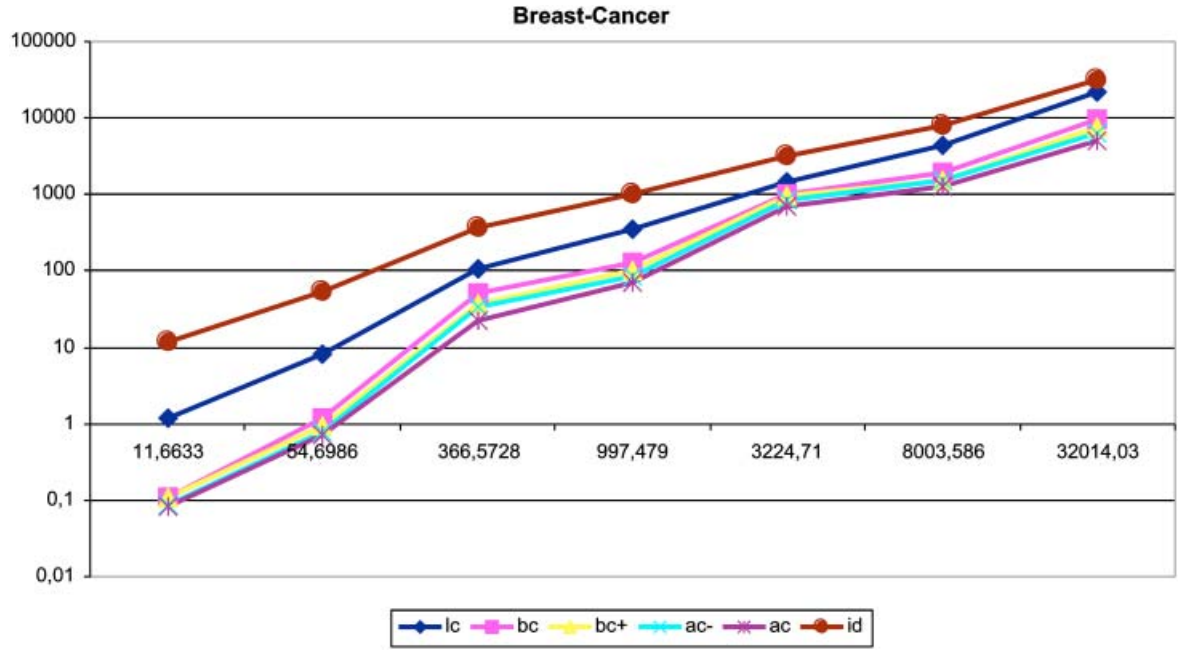

Fig. 8. Tests on the *breast cancer* database. [A color version of this figure can be viewed online at www.journals.cambridge.org]

Problem 3. The CSP is composed of six constraints: the learned dashboard constraint, rpm counter size = large, rpm counter position = center, fuel indicator = circular, temperature indicator = circular, and frame color = chrome.

- *learned:* 0.21 s.
- *GAC:* 5.61 s.
- *classifier:* 1 h, 53 min, 23 s.

Minimal surface = 233 cm^2.

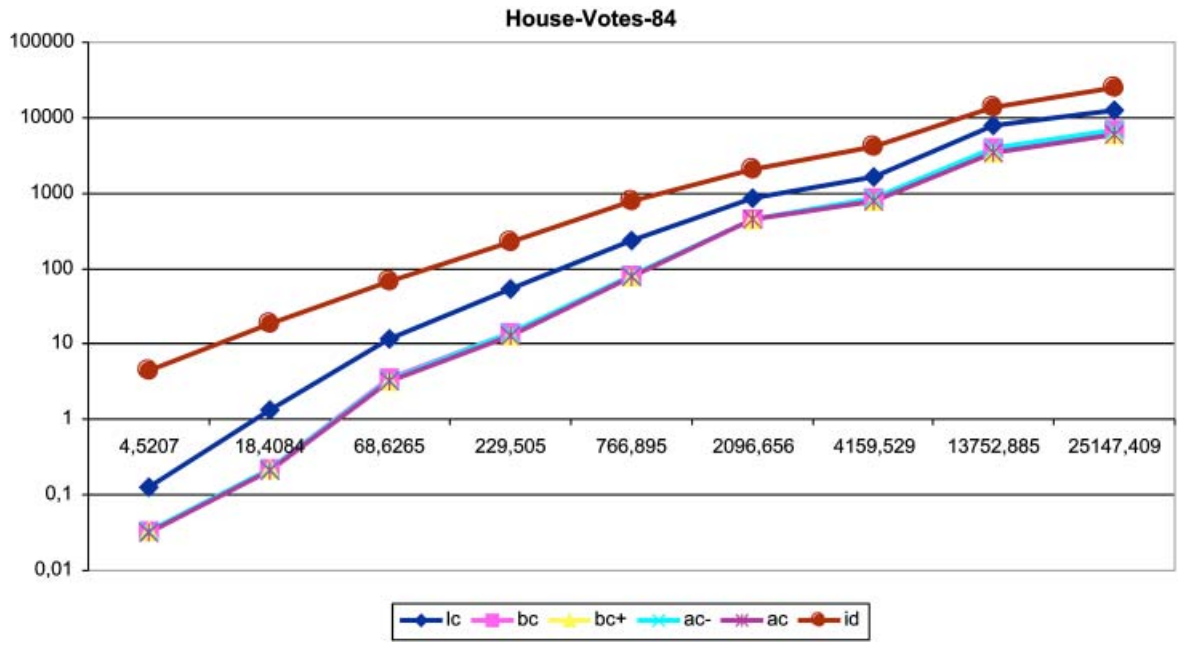

Fig. 9. Tests on the *votes-84* database. [A color version of this figure can be viewed online at www.journals.cambridge.org]

https://doi.org/10.1017/S0890060406060227 Published online by Cambridge University Press

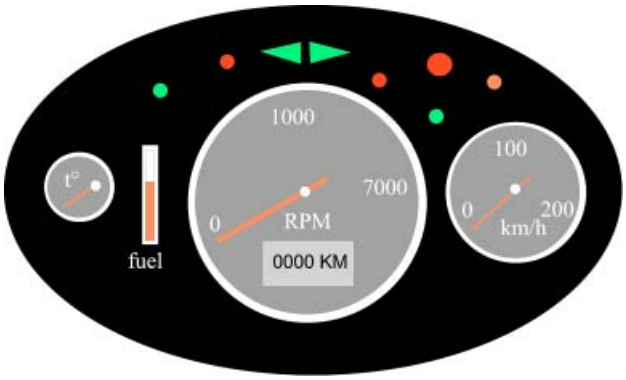

Fig. 10. An artist's view of an invented dashboard. [A color version of this figure can be viewed online at www.journals.cambridge.org]

As expected, the use of a generated propagator improves the resolution time. The observed improvement is of an order of magnitude over arc consistency computed on a table and more over generate and test. However, it should be noted that the generation of the table itself requires a propagator in order to be obtained in reasonable time (by avoiding sweeping the whole Cartesian product). In addition, some constraints are too large to be stored in extension in memory, which is a point in favor of our approach.

6. RELATED WORK AND CONCLUSION

Partially defined constraints were introduced in Lallouet et al. (2004). In Faltings and Macho-Gonzalez (2002), the comparable concept of open constraint was proposed in the context of distributed reasoning but with the goal of minimizing the number of requests needed to complete the definition. They were similarly used in the framework of interactive constraint satisfaction (Alberti et al., in press). Solver learning was introduced in Apt and Monfroy (1999) with a special rule system, but the generation algorithm was a bottleneck to handle large constraints. This work was extended by Abdennadher and Rigotti (2004) and Lallouet et al. (2003), but still in the context of closed constraints. None of these methods can combine generalization and solver efficiency. Partially defined constraints are also related to uncertainty because an uncertain constraint (Yorke-Smith & Gervet, 2003) can be viewed as a limited form of partially defined constraint for which it is assumed that only a few tuples are missing. The idea of learning constraints, extended to the learning of a preference instead of just a Boolean for a tuple, was used in Rossi and Sperduti (2004) in the context of soft constraints. They use an ad hoc neural network to represent the constraint. Although the learning is effective, the problem of building a solver for the constraint was not tackled in this work. In Coletta et al. (2003) and Bessière, Hebrard, et al. (2004), a CSP composed of predefined constraints like $=$ or $\leq$ was learned. The constraints were discovered by a version-space algorithm that reduces the possible constraints during the learning process. ANNs were considered for solving CSPs in the GENET system (Davenport et al., 1994) but with a completely different approach.

SUMMARY

Partially defined constraints allow the use of constraints partially defined by examples and counterexamples in decision and optimization problems. In this work, we propose a new technique for learning partially defined constraints by using classifiers. Not only does the generalization we obtain have remarkable properties from a machine learning point of view, but it can also be turned into a very efficient solver that gives an active behavior to the learned constraint. In a design perspective, partially defined constraints can be used to represent complex requirements for which a precise definition is either too complex or impossible to get.

ACKNOWLEDGMENTS

The authors thank Patrick Sebastian for suggesting the link to sensorial analysis.

REFERENCES

Abdennadher, S. & Rigotti, C. (2004). Automatic generation of rule-based constraint solvers over finite domains. *Transactions on Computational Logic 5(2)*.

Alberti, M., Gavanelli, M., Lamma, E., Mello, P., & Milano, M. (in press). A chr-based implementation of known arc-consistency. *Theory and Practice of Logic Programming*.

Apt, K.R. (1999). The essence of constraint propagation. *Theoretical Computer Science 221(1–2)*, 179–210.

Apt, K.R. & Monfroy, E. (1999). Automatic generation of constraint propagation algorithms for small finite domains. In *Int. Conf. Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* (Joxan, J., Ed.), Vol. 1713, pp. 58–72, Alexandria, VA. New York: Springer.

Bessière, C., Coletta, R., Freuder, E.C., & O'Sullivan, B. (2004). Leveraging the learning power of examples in automated constraint acquisition. In *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* (Wallace, M., Ed.), Vol. 3258, pp. 123–137, Toronto. New York: Springer.

Bessière, C., Hebrard, E., Hnich, B., & Walsh, T. (2004). The complexity of global constraints. In *National Conf. Artificial Intelligence* (McGuiness, D.L., & Ferguson, G., Eds.), pp. 112–117. San Jose, CA, July 25–29, 2004. Menlo Park, CA: AAAI Press/MIT Press.

Bessière, C. & Régin, J.-C. (1997). Arc-consistency for general constraint networks: preliminary results. In *IJCAI*, pp. 398–404, Nagoya, San Francisco, CA: Morgan Kaufmann.

Chandrasekaran, B. (1999). Design problem solving: a task analysis. *AI Magazine 11(4)*, 59–71.

Coletta, R., Bessière, C., O'Sullivan, B., Freuder, E.C., O'Connell, S., & Quinqueton, J. (2003). Semi-automatic modeling by constraint acquisition. In *Int. Conf. Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* (Rossi, F., Ed.), Vol. 2833, pp. 812–816, Kinsale, Ireland. New York: Springer.

Davenport, A., Tsang, E., Wang, C., & Zhu, K. (1994). GENET: a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *National Conf. Artificial Intelligence*, pp. 325–330, Seattle, WA. Menlo Park, CA: AAAI Press.

Faltings, B. & Macho-Gonzalez, S. (2002). Open constraint satisfaction. In *Int. Conf. Principles and Practice of Constraint Programming, Lec-*

ture Notes in Computer Science (van Hentenryck, P., Ed.), Vol. 2470, pp. 356–370, Ithaca, NY, September 7–13, 2002. New York: Springer.
Freund, Y. & Shapire, R. (1999). A short introduction to boosting. *Journal of the Japanese Society for Artificial Intelligence 14(5)*, 771–780.
Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks 2(5)*, 359–366.
Lallouet, A., Dao, T.B.H., Legtchenko, A., & Ed-Dbali, A. (2003). Finite domain constraint solver learning. In *Int. Joint Conf. Artificial Intelligence* (Gottlob, G., Ed.), pp. 1379–1380, Acapulco, Mexico. Menlo Park, CA: AAAI Press.
Lallouet, A., Legtchenko, A., Monfroy, E., & Ed-Dbali, A. (2004). Solver learning for predicting changes in dynamic constraint satisfaction problems. In *Changes'04, Int. Workshop on Constraint Solving Under Change and Uncertainty* (Brown, K., Beck, C., & Verfaillie, G., Eds.), Toronto.
Mitchell, T.M. (1997). *Machine Learning*. New York: McGraw–Hill.
Moore, R.E. (1966). *Interval Analysis*. Englewood Cliffs, NJ: Prentice Hall.
O'Sullivan, B. (2002). *Constraint-Aided Conceptual Design*. London: Professional Engineering Publishing.
Rossi, F. & Sperduti, A. (2004). Acquiring both constraint and solution preferences in interactive constraint system. *Constraints 9(4)*.
RuleQuest Research. (2004). *See5: An informal tutorial*. Available on-line at http://www.rulequest.com/see5-win.html
Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representations by error propagation. *Parallel Distributed Processing 1*, 318–362.
van Hentenryck, P., Saraswat, V., & Deville, Y. (1991). *Constraint processing in cc(fd)*. Unpublished manuscript.
Verfaillie, G. & Jussien, N. (2003). *Dynamic constraint solving. CP'2003 Tutorial*. Unpublished manuscript.
Yorke-Smith, N. & Gervet, C. (2003). Certainty closure: A framework for reliable constraint reasoning with uncertainty. In *9th Int. Conf. Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* (Rossi, F., Ed.), Vol. 2833, pp. 769–783, Cork, Ireland. New York: Springer.

Arnaud Lallouet is an Assistant Professor at the Laboratoire d'Informatique Fondamentale d'Orléans (LIFO), University of Orléans. Dr. Lallouet's main interest concerns artificial intelligence and more specifically the relationship between constraint programming and machine learning.

Andreï Legtchenko is a temporary Assistant at LIFO, University of Orléans, where he is a member of the Constraints and Machine Learning Team. Dr. Legtchenko is working on machine learning techniques for building propagators for constraint solvers.