Acquisition de Contraintes Ouvertes par Apprentissage de Solveur

Andreï Legtchenko, Arnaud Lallouet

Acquisition de Contraintes Ouvertes et Construction de Solveur

Andreï Legtchenko **Arnaud Lallouet**

Université d'Orléans — LIFO
BP6759, F-45067 Orléans
{Andrei.Legtchenko|Arnaud.Lallouet}@lifo.univ-orleans.fr

Résumé

Les contraintes partiellement définies, encore appelées *Contraintes Ouvertes* peuvent être utilisées pour modéliser les connaissances incomplètes d'un agent ou bien ses préférences. Au lieu de simplement calculer avec la partie connue de la contrainte, nous proposons de compléter sa définition en utilisant des techniques d'Apprentissage Artificiel. Mais la Programmation par Contraintes nécessite que les contraintes aient un comportement actif pendant la résolution. Nous montrons que les techniques que nous proposons pour l'acquisition d'une contrainte ont non-seulement un faible taux d'erreur en généralisation, mais qu'elle peuvent aussi être converties en un solveur très efficace pour la contrainte fraîchement apprise.

Abstract

Partially defined or *Open Constraints* can be used to model the incomplete knowledge of an agent or its preferences. Instead of only computing with the known part of the constraint, we propose to complete its definition by using Machine Learning techniques. But Constraint Programming requires that constraints also have an active behavior during solving and we show that our technique for learning constraint not only has a low error ratio in generalization but can also be turned into a very efficient solver for the newly learned constraint.

1 Introduction

Le succès de la Programmation par Contraintes doit beaucoup à sa combinaison équilibrée entre la facilité de description du modèle et l'efficacité de sa résolution. Toutefois, son usage est parfois limité par la connaissance des contraintes qui seraient appropriées pour décrire tel ou tel problème. Il peut arriver qu'un modèle incorpore une contrainte qui n'est que *partiellement connue* comme par exemple un concept que nous ne voulons ou ne pouvons pas représenter en extension : l'ensemble des mammifères dans une description d'animaux, les systèmes solaires au sein de données astronomiques, des préférences entre diverses possibilités dans un problème de configuration, un concept ayant une définition vague comme celui de "bon vin" ou encore une habitude comme les créneaux habituellement libres dans l'agenda d'une personne. Il peut aussi arriver que l'utilisateur ne sache quelle contrainte s'applique à son problème par manque de connaissance en Programmation par Contraintes, mais en revanche sache facilement exprimer des exemples et contre-exemples pour son problème.

Dans cet article, nous proposons d'utiliser à cette fin des contraintes partiellement définies sur les domaines finis appelées *Contraintes Ouvertes*. Dans une contrainte ouverte, certains n-uplets sont connus comme étant vrais, d'autres comme étant faux et d'autres encore ont une valeur de vérité *inconnue*. Nous complétons cette connaissance partielle par *apprentissage* afin d'acquérir le concept sous-jacent. Etant donnés des exemples positifs et négatifs, la tâche consiste à compléter la définition de telle sorte que de nouveaux n-uplets encore non rencontrés par le système seront classifiés correctement. Ce cadre a été activement étudié dans le champ de l'Apprentissage Artificiel [13] sous le nom de *classification supervisée*.

Prenons un exemple dans lequel les contraintes ouvertes apparaissent de façon naturelle. La cantine d'une entreprise sert un grand nombre de repas par jour. Un jour, on demande au chef cuisinier de préparer en entrée une salade qui soit bonne (pour respecter le standing de l'entreprise) mais aussi qui soit la moins chère possible (car les bénéfices de la compagnie n'ont pas été élevés l'an dernier). Le chef possède un livre de cuisine composé de 53 recettes de salade et diffé-

rents ingrédients comme des tomates, des crevettes ou bien de la mayonnaise ... Chaque ingédient possède un prix et une quantité disponible. La première idée serait de sélectionner dans le livre la recette la moins chère étant donnés les ingrédients disponibles. Mais il peut aussi être intéressant d'inventer une nouvelle salade, étant donné que la totalité de la connaissance concernant les salades n'est pas contenue dans le livre de recettes. Le concept de "bonne salade" peut être modélisé comme une contrainte ouverte dont les solutions sont les *bonnes* salades et les non-solutions les *mauvaises*. Le livre de cuisine est ainsi vu comme l'ensemble d'exemples pour la contrainte ouverte (pour les besoins de l'apprentissage, nous lui ajouterons des exemples de mauvaises salades)

Les contraintes ouvertes peuvent être apprises dès lors que des exemples et contre-exemples de la relation sont disponibles. Par exemple, dans le contexte d'un système distribué de rendez-vous utilisant des agendas sur PDAs, chaque agent peut apprendre une représentation de l'emploi du temps des autres afin de minimiser les futurs conflits lors de la recherche d'un rendez-vous commun. Cette représentation peut être utilisée comme heuristique afin de proposer en premier les créneaux qui ont le plus de chance d'être libres. Dans ce cas les exemples viennent de l'historique des interactions entre les agents.

L'idée de la technique que nous utilisons pour apprendre les contraintes ouvertes vient directement du modèle de solveur classique opérant par itération chaotique d'un ensemble d'opérateurs [3].

Nous commençons par apprendre la contrainte. Mais au lieu d'apprendre un simple classificateur qui répond oui ou non selon que le n-uplet fait partie ou non de la contrainte, nous apprenons les projections de la contrainte sur chacunes de ses variables. Un n-uplet fait ainsi partie de la contrainte s'il est accepté par tous les classificateurs et il est rejeté dès qu'il est rejeté par un classificateur. Nous montrons que cette méthode est capable d'atteindre un bon niveau de généralisation (comparable aux méthodes d'apprentissage classiques), ce qui justifie expérimentalement l'intérêt de la technique.

Pourtant, en tant que tel, un classificateur est simplement capable d'effectuer le test de satisfaction pour une contrainte ouverte. Si on le plaçait en tant que propagateur pour la contrainte ouverte dans un CSP, il ne contribuerait pas à la réduction du domaine des variables. Il en résulterait un comportement du type "générer/tester" qui est capable de rapidement ruiner les performances du système. Il est donc nécessaire d'être capable de construire un véritable *solveur* pour la contrainte ouverte, et non un simple test de satisfiabilité afin de ne pas dégrader les capacités de résolutions de la Programmation par Contraintes. Les classificateurs que nous apprenons sont exprimés par des fonctions et nous les transformons en solveurs en prenant leur extension aux intervalles. Cette transformation est purement formelle et ne nécessite plus de technique d'apprentissage, ce qui préserve les propriétés acquise lors de la première phase. Dès lors, les classifieurs peuvent être utilisés avec les domaines des variables en entrée. Nous montrons alors que le niveau de consistance qu'ils assurent n'est pas trop éloigné de l'arc-consistance et assure un élagage efficace de l'espace de recherche.

2 Construction de consistances

Nous rappelons tout d'abord les notions de base sur les consistances afin de présenter le schéma d'approximation utilisé pour l'apprentissage. Pour un ensemble E, nous représentons par 2^E son ensemble des parties et par $|E|$ son cardinal. Soit V un ensemble de variables et soit $D = (D_X)_{X \in V}$ la famille de leurs domaines (finis). Pour $W \subseteq V$, D^W représente l'ensemble des n-uplets sur W, soit $\Pi_{X \in W} D_X$. La projection d'un n-uplet ou d'un ensemble de n-uplet sur une variable ou un ensemble de variables est notée $|$. Une *contrainte* c est un couple (W, T) où $W \subseteq V$ (ou $var(c)$) désigne les variables de c et $T \subseteq D^W$ (ou $sol(c)$) l'ensemble des solutions de c. Un *Problème de Satisfaction de Contraintes* ou CSP est un ensemble de contraintes. Une solution est un n-uplet satisfaisant toutes les contraintes. Nous nous plaçons dans le cadre de la résolution par combinaison de *recherche* et *consistance* par réduction de domaine.

Un *état de recherche* est un ensemble de valeurs encore possibles pour chaque variable : pour $W \subseteq V$, cela correspond à une famille $s = (s_X)_{X \in W}$ telle que $\forall X \in W, s_X \subseteq D_X$. L'*espace ed recherche* correspondant est $S_W = \Pi_{X \in W} 2^{D_X}$. L'ensemble S_W ordonné par l'inclusion point à point $\subseteq$ est un treillis complet. Certains états de recherche appelés *singletoniques* représentent un n-uplet unique et jouent un rôle particulier en tant que représentants d'une possible solution. Un état de recherche singletonique s est tel que $|\Pi s| = 1$.

Une consistance peut être modélisée par le plus grand point-fixe d'un ensemble d'*opérateurs de consistance*, calculé par une itération chaotique [3]. Pour une contrainte $c = (W, T)$, un opérateur de consistance est un opérateur f sur S_W[1] possédant les propriétés suivantes :

- *monotonie* : $\forall s, s' \in S_W, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.
- *contractance* : $\forall s \in S_W, f(s) \subseteq s$.

[1] Si les opérateurs portent sur des variables différentes, on peut utiliser une cylindrification sur V.

– *correction* : $\forall s \in S_W, \Pi s \cap sol(c) \subseteq \Pi f(s) \cap sol(c)$.
– *complétude singletonique* : soit s un état de recherche singletonique, alors

$$\Pi s \in sol(c) \Leftrightarrow f(s) = s$$

La correction signifie qu'un n-uplet solution ne sera jamais rejeté dans l'espace de recherche tandis que la complétude singletonique signifie que l'opérateur est aussi un test de satisfaction pour la contrainte.

Voici quelques consistances associées à une contrainte $c = (W, T)$. L'arc-consistance ac_c est définie par :

$$\forall s \in S_W, ac_c(s) = s' \text{ with } \forall X \in W, s'_X = (\Pi s \cap T)|_X$$

Si nous supposons que chaque domaine de variable D_X est équippé d'un ordre total $\leq$, on représente par $[a..b]$ l'intervalle $\{e \in D_X \mid a \leq e \leq b\}$. Pour $A \subseteq D_X$, on note $[A]$ l'ensemble $[\min(A)..\max(A)]$. L'opérateur de consistance de bornes bc_c est défini par :

$$\forall s \in S_W, bc_c(s) = s' \text{ with } \forall X \in W, s'_X = s_X \cap [(\Pi s \cap T)|_X]$$

La consistance de bornes ne réduit que les bornes du domaine de la variable en les repoussant à la prochaine valeur consistante. Les consistances peuvent être partiellement ordonnées selon leur pouvoir de réduction et nous avons $f \subseteq f'$ if $\forall s \in S_W, f(s) \subseteq f'(s)$.

Puisque seuls les domaines des variables sont réduits, un opérateur de consistance f pour une contrainte $c = (W, T)$ peut être découpé en $|W|$ opérateurs de projection $(f_X)_{X \in W}$ selon chaque variable de la contrainte. Par le théorème de confluence des itérations chaotiques, [3], chaque opérateur peut être exécuté indépendamment s'il remplit les trois premières conditions des opérateurs de consistance. Afin de représenter la même contrainte, ils doivent être complets singletoniquement collectivement. Il est utile de noter qu'il y a ici une dissymétrie entre rejet et acceptation puisque pour respecter la satisfaction, un tuple non-solution doit être rejeté par (au moins) *un* opérateur tandis que la correction nous impose qu'un tuple solution soit accepté par *tous* les opérateurs.

Le rôle d'un opérateur de consistance f_X est d'éliminer du domaine de sa variable cible X certaines valeurs non-supportées par la contrainte. L'arc-consistance élimine toutes ces valeurs. Pour ce faire, elle doit trouver un *support* pour chaque valeur a considérée, c'est à dire un tuple solution dont la projection sur la variable-cible est a. Si aucun support n'est trouvé, alors la valeur est éliminée. Cette tâche a été montrée NP-complète en général [6] pour les contraintes n-aires. Bien que de nombreuses contraintes très utiles aient des propagateurs de complexité polynomiale, il en existe aussi pour lesquelle cette tâche est déraisonable. Puisque nous prenons en compte des contraintes définies par des exemples, ce cas doit être traité sérieusement.

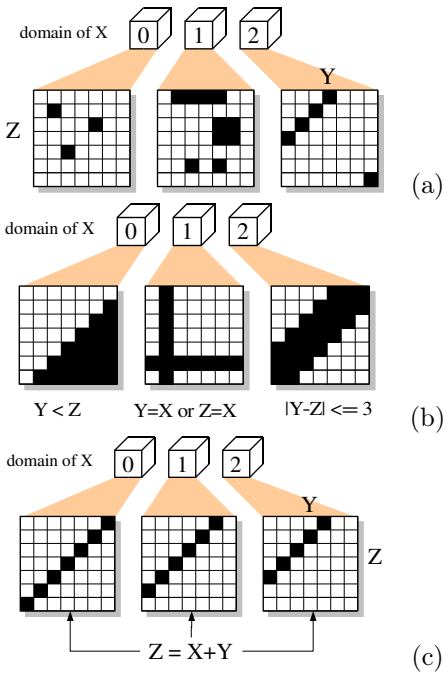


FIG. 1 – Analyse des consistances à plusieurs échelles

A un niveau de granularité plus fin, la technique de calcul d'un support a été l'objet de constantes améliorations dans les algorithmes d'arc-consistance. La première technique consiste à conserver la table avec toutes les solutions de la contrainte et parcourir cette table à chaque fois que le calcul de support est nécessaire. Cette technique est très optimisée et les algorithmes optimaux testent chaque valeur une seule fois et gardent une structure minimale, comme le schéma GAC (instancié par une table) [7]. Ces techniques sont très bien adaptées aux contraintes très irrégulières. Sur la figure 1 sont représentées les projection d'une contrainte $c(X, Y, Z)$ d'arité 3 sur les 3 valeurs du domaine de la variable X, avec 3 différents types de contraintes. La figure 1(a) illustre une contrainte peu régulière, pour laquelle les techniques basées sur la table de solutions sont bien adaptées, mais les exigences en espace mémoire croissent avec la taille de domaines et l'arité.

Au contraire, si la relation est très régulière, il est possible qu'une seule simple relation dans un langage donné décrive la fonction de support pour chaque valeur du domaine. Cette situation est exploitée par le langage des indexicaux [18]. Par exemple, sur la figure 1(c), tous les supports pour la variable X de la contrainte $Z = X + Y$ sont calculés par une seule expression, ici `X in dom(Z)-dom(Y)`. La "régularité" de la contrainte peut être mesurée comme la longueur

de l'expression minimale (dans un langage donné) de l'opérateur correspondant. Bien que le langage des indexicaux soit suffisamment puissant pour décrire les opérateurs pour n'importe quelle contrainte, dans certains cas la taille de l'expression nécessaire est proportionnelle à la taille de la table de solutions. Un des avantages du langage des indexicaux est la possibilité d'exprimer les consistances plus faibles que l'arc-consistance, comme la consistance de bornes. La construction automatique d'une approximation de consistances en utilisant le langage des indexicaux est décrite dans [12].

Mais il existe une situation intermédiaire, illustré sur la figure 1(b). Dans ce cas il est possible de trouver une représentation simple et compacte pour le calcul de supports, mais cette représentation est *différente* pour chaque valeur du domaine de variable. Autrement dit, pour représenter l'opérateur f_X, une fonction booléenne est associée à chaque valeur a du domaine de X. Cette fonction retourne *false* si la valeur n'a pas de support, et *true* si le support existe ou si le calcul incomplet ne permet pas de décider. Nous appelons cette fonction *une Fonction de Réduction Élémentaire* (ou FRE) notée $f_{X=a}$. En combinant les FREs, nous pouvons construire l'opérateur pour la variable donnée : il suffit de collecter les réponses de toutes les fonctions associées à X et de faire l'intersection avec le domaine courant de X. Soit s un état de calcul. On suppose que nous avons un ensemble de FREs $\{f_{X=a} | a \in s_X\}$. Alors nous définissons l'opérateur f_X comme $f_X(s) = s'$ où $s'_X = s_X \cap \{a \mid f_{X=a}(s)\}$.

Comparée à la technique basée sur la table de solutions, cette représentation est beaucoup plus compacte, si on arrive à borner la taille de chaque FRE. Si c'est le cas, la taille de la représentation croît seulement proportionnellement avec la taille des domaines et non pas avec celle de leur produit cartésien. Mais dans le cas général et pour une contrainte irrégulière, la taille de cette représentation à base de FREs est proportionnelle à la taille de la table de solutions.

En utilisant les FREs, nous donnons à chaque valeur sa propre fonction de support. Une fonction $f_{X=a}$ prend en entrée les domaines courants de toutes les variables de la contrainte (sauf celui de X). On peut considérer ces domaines directement comme des ensembles, ou seulement représentés par leurs bornes. De même, l'action des FREs, pour une variable X donnée, peut être répercutée sur le domaine de X de deux façons : soit seulement par le changement des bornes du domaine, soit par la suppression de valeurs partout dans le domaine. En combinant ces choix, nous obtenons quatre consistances dont deux largement utilisées (l'arc-consistance ac et la consistance de bornes bc) :

Modification des domaines en sortie	Entrée des FREs	
	valeurs	bornes
valeurs	ac	ac^-
bornes	bc^+	bc

Par exemple, la consistance de bornes change les bornes du domaine de X en fonction des bornes des domaines des autres variables de la contrainte. Si maintenant on utilise toute l'information sur les domaines pour modifier les bornes du domaine de X, on obtient une consistance de bornes renforcée, que nous appelons bc^+. Mais la consistance qui semble être la plus intéressante est la consistance ac^- : le support de chaque valeur du domaine de X est calculé, mais seules les bornes des autres domaines sont utilisées afin d'accélérer le calcul.

Proposition 1 *$bc \subseteq bc^+ \subseteq ac$ et $bc \subseteq ac^- \subseteq ac$.*

3 Contraintes ouvertes

Dans cette section, on introduit le concept de contrainte partiellement définie appelée *Contrainte Ouverte*. Une contrainte classique $c = (W, T)$ est supposée être connue en entier. L'hypothèse du monde clos établit que tout ce qui n'est pas vrai est faux. Ainsi, l'ensemble complémentaire $\overline{T}$ représente toutes les non-solutions de la contrainte c. Nous pouvons également définir une contrainte, sans changer le concept, par l'ensemble de ses non-solutions. Dans la suite, nous appelons *contraintes fermées* les contraintes classique sous l'hypothèse du monde clos. A l'opposé, dans certaines applications, il est possible que seulement une partie de la contrainte soit connue. Nous appelons une telle contrainte partiellement connue *une contrainte ouverte* :

Definition 2 (Contrainte ouverte) *Une* contrainte ouverte *est un triplet $c = (W, c^+, c^-)$ où $c^+ \subseteq D^W, c^- \subseteq D^W$ et $c^+ \cap c^- = \emptyset$.*

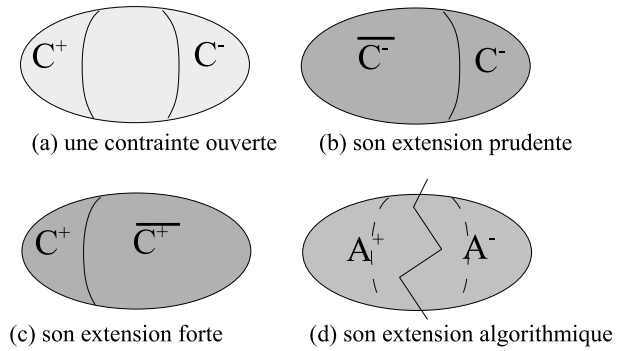
Dans une contrainte ouverte $c = (W, c^+, c^-)$, l'ensemble c^+ représente l'ensemble des solutions *connues*, et l'ensemble c^- représente l'ensemble des non-solutions *connues*. Les autres n-uplets, c'est-à-dire $\overline{c^+ \cup c^-}$ sont tout simplement inconnus (on ne sait pas s'ils sont solution ou non). Une contrainte ouverte est représentée sur la figure 2(a). Notons qu'une contrainte classique $c = (W, T)$ est une contrainte ouverte particulière $c = (W, T, \overline{T})$, pour laquelle l'ensemble de non-solutions est le complémentaire de T.

Les contraintes ouvertes demandent un traitement spécial pour pouvoir être inclues dans un CSP, car souvent très peu de propagation est possible sans connaître l'intégralité de la contrainte (pouvoir classer

tout n-uplet comme solution ou non-solution). Ainsi, nous devons *fermer* la contrainte ouverte pour l'utiliser dans un environnement de résolution classique. La fermeture d'une contrainte ouverte est obtenue en choisissant une classe (solution ou non-solution) pour tous les n-uplets inconnus. On appelle la contrainte classique résultant *une extension* de la contrainte ouverte :

Definition 3 (Extension) *Soit* $c = (W, c^+, c^-)$ *une contrainte ouverte. Une contrainte (classique)* $c' = (W, T)$ *est une* extension *de* c *si* $c^+ \subseteq T$ *et* $c^- \subseteq \overline{T}$.

Autrement dit, une extension est une contrainte fermée, classique compatible avec la partie connue de la contrainte ouverte. Une contrainte ouverte est un aperçu de la réalité cachée, et une de ses extensions correspond à la relation authentique. Dans la plupart



(a) une contrainte ouverte (b) son extension prudente

(c) son extension forte (d) son extension algorithmique

FIG. 2 – Une contrainte ouverte et quelques unes de ses extensions.

des cas, il est impossible de connaître exactement la véritable contrainte extension, ainsi on ne peut qu'à l'approximer. Généralement, beaucoup d'extensions sont possibles, introduisons en trois :

- Extension *prudente* : $c_{prudente} = (W, \overline{c^-})$. Dans cette extension, tous les n-uplets inconnus sont classés comme solutions (figure 2b). Un solveur généré pour cette extension est prudent, car il ne va pas supprimer de l'espace de recherche aucun de n-uplets inconnus. Ainsi, une erreur sur la classe d'un n-uplet n'introduit pas d'incorrection dans les solutions de CSP. Par contre, dans le cas où la partie inconnue de la contrainte est trop grande, l'extension prudente implique un élagage faible de l'espace de recherche et conduit à un très grand nombre de fausses solutions du CSP.
- Extension *audacieuse*[2] : $c_{forte} = (W, c^+)$. Tous les n-uplets inconnus sont classés dans les non-solutions, comme sous l'hypothèse du monde clos. Un solveur généré selon cette extension élague fortement l'espace de recherche, car il supprime dès que possible de l'état de recherche, les n-uplets inconnus. Si un n-uplet a été par erreur classifié comme non-solution, la correction est perdue et un mécanisme de restauration non-monotone est nécessaire, comme dans le cas des CSPs dynamiques.
- Extension *algorithmique* $c_{\mathcal{A}}$: soit $\mathcal{A} : D^W \rightarrow \{0, 1\}$ un algorithme de classification de n-uplets tel que $t \in c^+ \Rightarrow \mathcal{A}(t) = 1$ et $t \in c^- \Rightarrow \mathcal{A}(t) = 0$. Alors $c_{\mathcal{A}} = (W, \{t \in D^W \mid \mathcal{A}(t) = 1\})$ (figure 2d).

Dans ce contexte, une extension algorithmique est obtenue par une classification supervisée, qui consiste à induire, à partir des exemples étiquetés, une fonction associant une classe à chaque n-uplet. L'Apprentissage Artificiel propose des propriétés indispensables à une bonne extension algorithmique. La première propriété, et probablement la plus importante, est la correction de la prédiction de la classe par la technique de classification, pour tout n-uplet inconnu. Le rapport entre le nombre de n-uplets correctement classifiés et le nombre de n-uplets présentés définit la qualité de la généralisation de la technique. L'espace des n-uplets inconnus étant souvent très grand, la qualité de la généralisation est plutôt estimée que calculée exactement [19]. Il existe un grand nombre de techniques de classification, très performantes, et une qualité de prédiction supérieure à 90% n'est pas rare. Pour obtenir cette qualité, l'ensemble d'exemples (la partie connue de la contrainte ouverte) doit être représentatif du concept caché. Une représentation de la fonction de classification est alors recherchée dans un espace de fonctions possibles, appelé *l'espace d'hypothèse*. Un algorithme d'apprentissage cherche dans cet espace d'hypothèses celle qui satisfait au mieux les critères voulus, comme la correction, la précision ou la simplicité ...

4 L'acquisition de contrainte ouverte

Pour commencer, nous posons le problème de construire une bonne extension pour une contrainte ouverte donnée. Pour représenter une relation, la première idée est de construire un classificateur, qui prendrait en entrée toutes les variables de la relation et qui retournerait un booléen indiquant l'appartenance du n-uplet à la relation. Malheureusement, malgré la possibilité d'apprendre ce type de classificateur (voir [16]), il est difficile d'extraire un solveur à partir d'une telle représentation de la contrainte. Motivés par l'équivalence entre une contrainte et un opérateur de réduc-

[2]Notons que la terminologie de *prudente* et *audacieuse* a été choisie en accord avec le solveur et son niveau d'élagage et non pas selon la contrainte car classer tous les n-uplets comme solution pourrait aussi être considéré comme *audacieux* par rapport à la contrainte.

tion correct et singletoniquement complet, nous proposons d'acquérir une contrainte ouverte $c = (W, c^+, c^-)$ sous une forme particulière, inspirée directement par la forme des opérateurs de réduction. Plus exactement, nous proposons d'apprendre un classificateur indépendant pour chaque valeur a du domaine de chaque variable $X \in W$, à l'image des Fonctions de Réduction Élémentaires introduites dans la section 2. Le classificateur détermine si la valeur a doit rester dans le domaine courant de X (en produisant la valeur 1 en sortie du classificateur) ou si elle doit être supprimée (valeur 0). Le classificateur prend en paramètres toutes les autres variables de $W - \{X\}$ (le classificateur est donc une fonction de type $D^{W-\{X\}} \to \mathbb{B}$).

Nous proposons d'utiliser les perceptrons avec une couche cachée pour représenter les classificateurs. Cette représentation a été choisi pour ses bonnes performances en apprentissage et pour sa capacité à être ensuite transformée en Fonction de Réduction Élémentaire. Pour $W \subseteq V$, un *neurone* est une fonction $n(W) : \mathbb{R}^{|W|} \to \mathbb{R}$ qui calcule la somme pondérée de ses entrées suivie par une fonction seuil. Une entrée constante de valeur 1 est classiquement ajoutée à tous les neurones pour augmenter l'expressivité du réseau. La fonction sigmoïde ($F(x) = \frac{1}{1+e^{-x}}$) est la fonction de sortie fréquemment utilisée, car elle est une approximation dérivable de la fonction seuil. La dérivabilité est une propriété importante pour l'algorithme d'apprentissage. Soit $(w_X)_{X \in W}$ les poids associés à chaque variable d'entrée et w_0 le poids associé à l'entrée constante. Alors la fonction calculée par un neurone prenant comme entrée $a = (a_X)_{X \in W}$ est

$$n(a) = \frac{1}{1 + e^{w_0 - \sum_{X \in W} w_X . a_X}}$$

Pour une contrainte $c = (W, c^+, c^-)$, le classificateur que nous construisons pour $X = a$, avec, par exemple 3 neurones dans la couche cachée, est représenté sur la figure 3. Soit $(n_i)_{i \in I}$ les neurones de la couche cachée et *out* le neurone de sortie. Tous les neurones de la couche cachée sont connectés à tous les neurones d'entrée, et le neurone de sortie est relié à tous les neurones de la couche cachée. Nous appelons $n_{<X=a>}$ le perceptron (multicouches) affecté à $X = a$. Comme les neurones sont des fonctions continues, nous utilisons un codage analogique des domaines. Soit D un domaine fini et $<$ un ordre total sur D, alors on peut écrire D comme $\{a_0, \ldots, a_n\}$ avec $\forall i \in [1..n], a_{i-1} < a_i$. En accord avec cet ordre, on transpose le domaine D dans l'intervalle $[0..1]$ en codant chaque a_i par i/n. La valeur de sortie du réseau est également dans l'intervalle $[0..1]$, et nous avons choisi comme convention que la valeur a va être retirée du domaine de X si $out \leq 0.5$. Ce seuil est la dernière unité du réseau (voir figure 3).

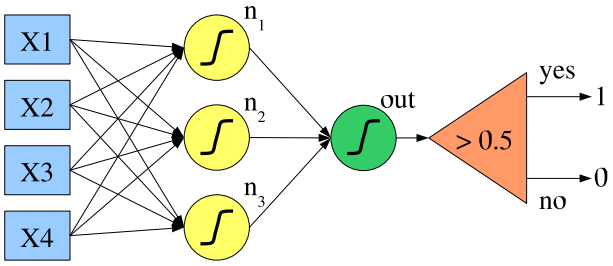


FIG. 3 – Structure du réseau

Le classificateur global correspondant à la contrainte ouverte est composé de tous les classificateurs élémentaires, pour toutes les valeurs des domaines de toutes les variables $\{n_{<X=a>} \mid X \in W, a \in D_X\}$. En suivant l'intuition des FREs, on utilise les classificateurs élémentaires pour décider si un n-uplet est une solution de l'extension ou non, en collectant les réponses de tous les classificateurs. Soit $t \in D^W$ un n-uplet candidat, et soit $(n_{<X=t|_X>}(t|_{W-\{X\}}))_{X \in W}$ la famille de réponses (0 ou 1) de tous les classificateurs élémentaires concernés. Nous pouvons combiner ces réponses selon deux schémas :

- *le vote avec veto* : un n-uplet est accepté s'il est accepté par tous les classificateurs.
- *le vote à la majorité* : un n-uplet est accepté s'il est accepté par la majorité des classificateurs.

Pour obtenir l'extension de la contrainte ouverte, ces classificateurs sont appris sur les exemples et les contre-exemples obtenus à partir de la partie connue de la contrainte. Pour $E \subseteq D^W$, $X \in W$ et $a \in D_X$, nous notons $E_{<X=a>}$ la sélection de n-uplets de D^W ayant a comme projection sur X : $E_{<X=a>} = \{t \in E \mid t|_X = a\}$. Alors, pour construire le classificateur $n_{<X=a>}$, nous utiliserons :

- l'ensemble d'exemples $e^+_{<X=a>} = c^+_{[X=a]}|_{W-\{X\}}$
- l'ensemble de contre-exemples $e^-_{<X=a>} = c^-_{[X=a]}|_{W-\{X\}}$

Par exemple, pour la contrainte ouverte définie par $(\{X, Y, Z\}, \{(1,1,0),(1,0,1)\}, \{(1,1,1)\})$, on obtient :

- $e^+_{<X=1>} = \{(1,0),(0,1)\}$.
- $e^-_{<X=1>} = \{(1,1)\}$.

Les réseaux sont appris par l'algorithme classique de rétropropagation de gradient qui trouve des poids minimisant l'erreur quadratique moyenne sur l'échantillon d'apprentissage. Nous arrêtons l'apprentissage quand tous les exemples et les contre-exemples sont correctement classés. Ainsi on garantit la correction du solveur sur la partie connue de la contrainte. Mais comme il est parfois impossible de garantir la correction sur la partie connue pour les données trop complexes ou bruitées, cette contrainte de correction sur la partie connue peut être exceptionnellement re-

lâchée. L'expressivité du réseau est (empiriquement) adaptée par le choix du nombre de neurones dans la couche cachée afin d'obtenir la meilleure qualité de généralisation. Nous avons implanté cette technique

BD	**salades**	**mush**	**cancer**	**votes84**
Ar	22	22	9	16
/dom/	2-4	2-12	10	3
#Cls	64	116	90	48
/BD/	334	8124	699	435
CC	3	3	5	5
T app.	55″	2′30″	8′30″	4′30″
Err Sveto	11.9%	6.9%	4.6%	25.9%
Err Smaj	**3.6**%	0.7%	**3.5**%	3.8%
Err C5.0	9.9%	0.8%	5.5%	**3.7**%
Err C5.0 b	4.8%	**0.2%**	3.7%	4.4%

TAB. 1 – Résultats d'apprentissage.

au sein de notre système appelé *Solar*. Comme notre méthode d'acquisition n'est pas standard, nous avons testé sa qualité en tant que méthode de classification à 2 classes. Nous avons pris 4 jeux de données différents : la base "recettes de salades" de l'exemple **??** et 3 bases trouvées dans *UCI Machine Learning Repository*[3] . Pour avoir une idée de la performance relative de notre technique nous avons testé sur les mêmes bases l'algorithme d'apprentissage d'arbres de décision C5.0 (qui est une version amélioré du très populaire C4.5 [15]) et C5.0 avec boosting [17] avec $|W|$ votants pour contrebalancer le nombre de nos classificateurs. Notons au passage que notre méthode de construction de classificateurs est différente de celle du boosting [11] par la façon de construire les ensembles d'exemples pour les classificateurs. Nous avons utilisé la validation croisée classique avec 10 blocs. Le jeu de données est divisé de manière aléatoire en 10 blocs de même taille. Un bloc est utilisé pour la validation, et les autres pour l'apprentissage. Après avoir fait 10 tests, à chaque fois avec un bloc de validation différent, on calcule la moyenne. Cinq sessions différentes de validation croisée ont été effectuées pour augmenter la fiabilité de résultats (il y a donc 50 tests au total). Les résultats des tests ainsi que la description des bases sont regroupés dans le tableau 1 où :

- *Ar* est l'arité de la contrainte,
- */Dom/* est la taille des domaines de variables,
- *# cls* est le nombre de classificateurs appris,
- */BD/* est le nombre de n-uplets dans la base de données,
- *Cc* est le nombre de neurones dans la couche cachée,
- *T app* est le temps d'apprentissage de l'ensemble des classificateurs ;
- *Err Sveto* l'erreur sur un échantillon de validation de Solar en mode "vote avec veto" ;
- *Err Smaj* l'erreur sur un échantillon de validation de Solar en mode "vote avec la majorité" ;
- *Err C5.0* l'erreur du système C5.0 ;
- *Err C5.0b* l'erreur du système C5.0 avec boosting.

Le nombre optimal de neurones dans les couches cachées a été à chaque fois déterminé empiriquement. Le temps d'apprentissage (T app) est le temps nécessaire pour apprendre tous les classificateurs. Le temps d'apprentissage pour le système C5.0 avec ou sans boosting est moins d'une seconde. Mais le temps d'apprentissage n'a absolument aucune importance dans l'application aux contraintes, car le gain apporté par l'utilisation des propagateurs obtenus compense très largement les minutes nécessaires pour apprendre les classificateurs élémentaires. Ainsi, la comparaison de notre technique avec les algorithmes connus, comme les arbres de décision, sert seulement à signaler la qualité d'acquisition de concepts, sans se mettre en compétition avec les autres technique d'apprentissage, car les techniques d'apprentissage connues ne fournissent pas de solveur pour le concept appris.

Comme on le constate, la qualité de classification défie celle des algorithmes réputés comme très efficaces. On constate que la méthode de vote avec veto est visiblement moins performante que la méthode avec vote majoritaire. Mais il faut comprendre que lors de la classification par le vote avec veto, l'erreur est principalement faite (en proportion écrasante) sur la classe "solution". En fait, cette méthode accepte seulement les n-uplets qui sont des solutions "irréprochables". Ainsi cette méthode tend à "purifier" la classe de solutions, ce qui peut même être recherché par l'utilisateur (dans les cas où il vaut mieux avoir une solution sûre quitte à en avoir moins au total). Par exemple, dans le cas de la base "salades", on passe par apprentissage de 53 recettes à 7.4E4 recettes (le nombre de n-uplets acceptés unanimement). Probablement, la vraie classe "bonne salade" est encore plus grande, mais 7.4E4 recettes sont déjà largement suffisantes. En plus, les 7.4E4 recettes sont des recettes "sûres" car acceptées unanimement (un expert confirme qu'il y a moins de 3% de mauvaises recettes dans un échantillon de 100 recettes prélevées aux hasard dans les 7.4E4 solutions).

5 Un solveur à partir de classificateurs

Une contrainte mise dans un CSP doit avoir un comportement actif, elle doit participer à la réduction de domaines. Or le comportement "générer et tester" produit avec les classificateurs n'est pas suffisam-

[3] `http ://www.ics.uci.edu/~mlearn`

ment puissant et utile. Une autre approche serait de générer, avec le classificateur de n-uplets, toutes les solutions de la contrainte généralisée et ensuite utiliser un algorithme efficace de calcul d'arc-consistance, comme par exemple le schéma GAC [7]. Mais cette approche a deux désavantages majeurs. D'abord, le temps de génération de solutions peut être extrêmement long. Par exemple, pour la contrainte *Mushroom*, en 3h on arrive à générer difficilement 76835 solutions en faisant plus de 1.5 10^7 essais. Or, on sait que cette contrainte a plus de 4.1E6 solutions ! Et ceci constitue le second problème : il faudrait donc plus de 88Mo de mémoire pour stocker une telle table, alors que notre représentation est beaucoup plus économique. Pour une contrainte d'arité n, on suppose que la couche cachée comporte m neurones et que la taille de domaines est d. Alors il faut $n(n+1)dm+1$ flottants (de 10 octets) pour stocker tous les poids de tous les classificateurs. Ainsi, pour la contrainte *salades* ($n = 22, m = 3, d = 4$) il faut 60ko, et pour la contrainte *mushroom* ($n = 22, m = 3, d = 12$), 180 Ko...

Nous proposons d'utiliser les classificateurs élémentaires appris pour fabriquer des propagateurs. Avant d'expliquer le principe, rappelons quelques notions d'analyse par intervalles [14]. On appelle $Int_{\mathbb{R}}$ le treillis d'intervalles construit sur l'ensemble $\mathbb{R}$ de nombres réels. Toutes les fonctions réelles ont une extension aux intervalles :

Definition 4 (Extension aux intervalles) *Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction. La fonction $F : Int_{\mathbb{R}} \to Int_{\mathbb{R}}$ est une extension aux intervalles de f si $\forall I \in Int_{\mathbb{R}}, \forall x \in I, f(x) \in F(I)$.*

Une extension F est croissante si $A \subseteq B \Rightarrow F(A) \subseteq F(B)$. Parmi toutes les extensions aux intervalles de f, il y a la plus petite extension, appelée *l'extension canonique aux intervalles* : $\widehat{f}(I) = [\{f(x) \mid x \in I\}]$. L'extension canonique est croissante. Voici les extensions canoniques aux intervalles des opérateurs que nous utilisons dans nos classificateurs :

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

La division ne pose ici pas de problème car jamais aucun intervalle (dans nos calculs) ne contient la valeur 0. Si e est une expression construite en utilisant ces opérateurs, et si E est l'expression obtenue en remplaçant dans e les opérateurs par leurs extensions aux intervalles croissantes, alors $\forall I \in Int_{\mathbb{R}}, \forall x \in I, e(x) \in E(I)$. Cette propriété est appelée "Le théorème fondamental de l'arithmétique des Intervalles" [14]. La propriété est conservée par le passage aux fonctions sur le produit cartésien de $\mathbb{R}$.

Un classificateur élémentaire $n_{<X=a>}$ définit naturellement une fonction à valeurs booléennes et aux paramètres réels. On appelle $N_{<X=a>}$ son extension naturelle, définie en prenant l'extension canonique de tous les opérateurs composant $n_{<X=a>}$ comme $+, -, \times, \exp$. Ensuite, en appliquant la fonction $N_{<X=a>}$ sur les domaines courants des variables, on obtient un intervalle de sortie. L'intuition derrière l'extension aux intervalles est simple : en supposant que les paramètres d'une fonction ne sont pas connus exactement, mais comme des domaines de valeurs possibles, quel sera l'ensemble des sorties possibles pour cette fonction ? L'extension aux intervalles permet d'approximer l'ensemble des valeurs possibles par un intervalle englobant ces valeurs. En pratique, pour calculer l'extension d'un classificateur représenté par un perceptron à 3 couches, nous calculons d'abord les intervalles de sortie pour les neurones de la couche cachée, puis, en utilisant ces intervalles, on calcule l'intervalle de sortie du classificateur. Si la borne supérieure de cet intervalle est inférieure à 0.5, cela veut dire que quelque soient les affectations des variables d'entrée, et compte tenu des domaines courants, la réponse du classificateur sera forcement inférieure à 0.5. Dans ce cas, la valeur a peut être supprimée du domaine de X. Sinon, la valeur reste dans le domaine.

Proposition 5 *$N_{<X=a>}$ est une FRE.*

En faisant ce calcul pour toute valeur du domaine courant de X, et pour toute variable de la contrainte $c = (W, c^+, c^-)$, nous définissons un opérateur de consistance f_X qui rassemble les résultats de toutes les FRE pour X. Pour $s \in S_W$, $f_X(s) = s'$ où $s'_X = s_X \cap \{a \in D_X \mid N_{<X=a>}(s|_{W-\{X\}}) = 1\}$ et $s'_Y = s_Y$ pour $Y \neq X$.

Proposition 6 *Les opérateurs $(f_X)_{X \in W}$ définissent une consistance pour c.*

Proof Chaque opérateur f_X est croissant, contractant et correct (par Le théorème fondamental de l'arithmétique des Intervalles). Ils sont singletoniquement complets, car l'extension de la contrainte ouverte est définie par les classificateurs).

D'autre part, la multiplication est seulement sous-distributive en Arithmétique des Intervalles [14], c'est-à-dire que $A \times (B+C) \subseteq (A \times B) + (A \times C)$. A cause des occurrences multiples de mêmes variables, nous avons le résultat suivant :

Proposition 7 *Les opérateurs $(f_X)_{X \in W}$ calculent une approximation de ac^-, c'est à dire une consistance plus faible que ac^- selon l'ordre $\subseteq$ des consistances.*

Notons que les opérateurs de chaque variable sont indépendants, et donc que la généralisation de la contrainte correspondant aux opérateurs est celle obtenue en mode vote avec veto. On ne peut pas obtenir simplement la généralisation correspondant au vote majoritaire avec les opérateurs $(f_X)_{X \in W}$. Ceci est dû à l'enchaînement indépendant des opérateurs de réduction lors une itération chaotique [3]. Dans la suite nous appelons aussi ac^- l'approximation de cette consistance que nous calculons. Le système SOLAR prend en

BD	**salad**	**mush**	**cancer**	**votes84**
$\#Sol$	7.4E5	$\geq$ 4.1E6	1.27E5	1.27E5
$\#E\ ac^-$	1.34E5	$\geq$ 3.1E6	1.28E5	3.47E5
$e/s\ ac^-$	1.8	0.75	0.99	2.86
$T\ ac^-$	$5'15''$	$\geq$ 2h	$3'00''$	$7'30''$

TAB. 2 – Tests de consistances

entrée une contrainte ouverte et retourne directement un ensemble d'opérateurs de réduction utilisables dans n'importe quel solveur. Dans nos tests, nous avons utilisé notre propre solveur. Les tests sont résumés dans la table 2. Dans cette table, $\#Sol$ signifie le nombre de solutions de la contrainte généralisée, $\#E\ ac^-$ le nombre d'échecs, $T\ ac^-$ le temps de calcul et $e/s\ ac^-$ le rapport le nombre d'échecs par le nombre de solutions trouvés. Le test consiste à générer toutes les solutions pour une contrainte isolée. Le nombre de solutions dans la contrainte *Mushroom* est si grand que l'on n'a pas pu toutes les calculer. Les tests suggèrent que la consistance obtenue est plutôt intéressante, car elle permet de trouver les solutions en faisant peu d'échecs (1.6 échec par solution en moyenne), mais des tests complémentaires sont actuellement en cours pour situer plus exactement les consistances obtenues pour une contrainte ouverte. La contrainte ouvertes *salades* a été utilisée dans le problème d'optimisation décrit en introduction, et la meilleure solution ne se trouve pas parmi le livre de recettes. Les recettes trouvées ont été jugées comme intéressantes par notre chef culinaire.

6 L'État de l'Art et Conclusion

Les contraintes ouvertes ont été introduites pour la première fois dans [10] dans un contexte de raisonnement distribué, mais dans le but de minimiser le nombre de requêtes nécessaires pour compléter la définition (la généralisation n'est pas effectuée). Les contraintes ouvertes ont été utilisées dans le cadre de la Satisfaction Interactive de Contraintes [2], mais également sans apprentissage. L'apprentissage de solveurs a été introduit dans [4] avec un système de génération de règles de propagation, mais l'approche était limitée aux contraintes classiques de petite taille. Ce travail a été étendu dans [1] et [12], mais toujours dans le cadre de contraintes classiques. Aucune de ces méthodes ne combine la possibilité de généralisation avec la construction de solveur.

Les extensions prudentes et audacieuses ont été indirectement introduites dans [20] sous le nom de la fermeture de certitude (certainty closure). Une contrainte incertaine peut être considérée comme une forme limitée de contrainte ouverte, dans laquelle seulement un petit nombre de n-uplets est inconnu. Si ces n-uplets inconnus sont considérés comme non-solutions, seulement les solutions *robustes* du CSP sont trouvées. Si les n-uplets inconnus sont considérés comme solutions de la contrainte, alors un ensemble plus important de solutions du CSP sera trouvé (avec un certain nombre de fausses solutions). Mais ce contexte est inutilisable dans notre cadre, car pour une contrainte ouverte le nombre de n-uplets inconnus est souvent très grand. Une extension audacieuse conduirait à un solveur trop restrictif, et le solveur pour l'extension prudente laisserait trop de fausses solutions dans les états de calcul et n'élaguerait pas assez. *L'hypothèse du monde ouvert* est sous-jacente dans le cadre de la Satisfaction Interactive de Contraintes [2] dans lequel seuls les domaines sont partiellement connus et sont éventuellement acquis au cours de la résolution. Seule une propagation prudente est effectuée, appelée l'arc-consistance *connue*. Cette approche est inapplicable aux contraintes ouvertes en général, car le cadre est limitée aux contraintes unaires et suppose que les domaines doivent être connus petit à petit pour compléter la résolution.

L'idée d'apprendre des contraintes, étendue à l'apprentissage de préférences pour les n-uplets (au lieu d'un booléen) apparaît dans [16] dans le contexte de contraintes molles. Les auteurs utilisent un réseau de neurones ad-hoc pour représenter la contrainte. Même si l'apprentissage est efficace, l'apprioche semble limitée aux contraintes de petite taille et la construction de solveur pour la contrainte acquise n'est pas proposée.

Dans [8] and [5], on propose d'apprendre une représentation de la contrainte ouverte en tant que CSP composé de contraintes prédéfinies comme $=$ ou $\leq$. La topologie du réseau est fixe, mais pas les contraintes reliant les nIJuds. Les contraintes sont trouvées par un algorithme basé sur l'espace des versions qui réduit au cours de l'apprentissage le nombre de contraintes possibles pour chaque arc du réseau. Les difficultés de cette approche sont : (a) la difficulté de traitement des contre-exemples, (b) la définition d'une bonne topologie afin d'éviter les redondances et (c) la sensibilité au bruit. Mais d'autre part, la méthode permet de décou-

vrir des relations fortes entre variables, elle est transparente et intelligible et elle peut être particulièrement intéressante dans le cas de contraintes de très grande arité.

Les réseaux de neurones ont déjà été utilisés dans le résolution de CSP, par exemple dans le système GENET [9], mais l'approche est complètement différente.

Sommaire

Les contraintes ouvertes autorisent l'utilisation de contraintes partiellement définies dans les problèmes de décision ou d'optimisation. Dans ce travail nous proposons une nouvelle technique d'apprentissage de solveurs pour les contraintes ouvertes. Les résultats de tests démontrent la qualité de classification obtenue par l'apprentissage de solveurs, mais aussi l'efficacité de la réduction de domaines par les opérateurs appris. Les contraintes ouvertes ont le potentiel d'ouvrir de nouveaux champs d'application à la modélisation par contraintes, notament dans des usages où elle n'a jamais pu être utilisée auparavant.

Références

[1] Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *Transaction on Computational Logic*, 5(2), 2004.

[2] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and M. Milano. A chr-based implementation of known arc-consistency. *Theory and Practice of Logic Programming*, to appear.

[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 Joxan Jaffar, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 58–72, Alexandria, Virginia, USA, 1999. Springer.

[5] Christian Bessière, Rémi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 123–137, Toronto, Canada, 2004. Springer.

[6] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The complexity of global constraints. In Deborah L. McGuinness and Georges Ferguson, editors, *National Conference on Artificial Intelligence*, pages 112–117, San Jose, CA, USA, July, 25-29 2004. AAAI Press / MIT Press.

[7] Christian Bessière and Jean-Charles Régin. Arc-consistency for general constraint networks : preliminary results. In *International Joint Conference on Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997. Morgan Kaufmann.

[8] R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, pages 812–816, Kinsale, Ireland, 2003. Springer.

[9] A. Davenport, E. Tsang, C. Wang, and K. Zhu. GENET : A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *National Conference on Artificial Intelligence*, pages 325–330, Seattle, WA, USA, 1994. AAAI Press.

[10] Boi Faltings and Santiago Macho-Gonzalez. Open constraint satisfaction. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 356–370, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.

[11] Y. Freund and R. Shapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5) :771–780, 1999.

[12] Arnaud Lallouet, Thi-Bich-Hanh Dao, Andreï Legtchenko, and AbdelAli Ed-Dbali. Finite domain constraint solver learning. In Georg Gottlob, editor, *International Joint Conference on Artificial Intelligence*, pages 1379–1380, Acapulco, Mexico, 2003. AAAI Press. Poster.

[13] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[14] Ramon E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[15] J. Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann, 1993.

[16] F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint system. *Constraints*, 9(4), 2004.

[17] RuleQuest Research. See5 : An informal tutorial, 2004. http ://www.rulequest.com/see5-win.html.

[18] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.

[19] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[20] Neil Yorke-Smith and Carmen Gervet. Certainty closure : A framework for reliable constraint reasoning with uncertainty. In Francesca Rossi, editor, *9th International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, pages 769–783, Cork, Ireland, 2003. Springer.