Branch and Learn pour l'acquisition de CSP

Christian Bessiere[1] **Rémi Coletta**[1] **Frédéric Koriche**[1]
Arnaud Lallouet[2] **Matthieu Lopez**[3]
[1] LIRMM, Université de Montpellier, 34095 Montpellier Cedex 5
[2] GREYC, Université de Caen, 14032 Caen CEDEX
[3] LIFO, Université d'Orléans, F-45067 ORLEANS Cedex 2
{bessiere, coletta, koriche}@lirmm.fr arnaud.lallouet@unicaen.fr
matthieu.lopez@univ-orleans.fr

Résumé

L'utilisation de la programmation par contraintes, notamment la modélisation des problèmes, est devenue limitée à des utilisateurs possédant une bonne expérience dans le domaine. Ce papier s'inscrit dans un cadre visant à automatiser la modélisation. Les techniques existantes ont montré des résultats encourageants mais certaines exigences rendent leur utilisation encore problématique. Nous nous intéressons à la possibilité pour l'utilisateur de ne pas donner de solutions et de non-solution de son problème. En partant d'un CSP sans aucune contrainte, notre méthode consiste à résoudre le problème de l'utilisateur de manière classique en développant un arbre de recherche. Quand notre outil ne peut décider si l'affectation partielle courante est correcte ou non, nous demandons à l'utilisateur de guider la recherche sous forme de requêtes. Ces requêtes permettent de guider la recherche en répondant à des requêtes et de trouver des contraintes à ajouter aux modèles du CSP et ainsi améliorer la recherche.

Abstract

Constraint programming (CP) allows to model problem with efficient solving methods. However, its use is limited to CP expert users notably for the modeling step. This paper aims to partially automate this modeling part. Existing methods have shown encouraging results, but their uses are still complicated. We are interesting in simplifying this allowing the user to not give solutions and non-solutions of her problem. Beginning with a CSP without constraint, our method consists in solving the problem in a classical way. Developing the search tree, we progressively assign values to variables. When our tool cannot decide for itself if a partial assignment is correct or not, we ask, with a query, to the user to guide the search. These queries allow to learn constraint and then improve the solving.

1 Introduction

Il est classique de dire que la Programmation par Contraintes permet de résoudre efficacement un grand nombre de problèmes combinatoires avec une relative facilité de modélisation. Toutefois, cette facilité n'est qu'apparente car elle cache en fait un certain nombre de difficultés, la première desquelles [13] étant la modélisation. La modélisation, première étape de la résolution d'un problème, possède trois niveaux de difficulté. Le premier est de déterminer quelles sont les variables et leur domaine, aussi appelé "point de vue" [13]. Le second consiste à déterminer les contraintes du problème. Le troisième enfin consiste à reformuler l'ensemble pour adapter le modèle aux particularités du solveur qui servira à le résoudre. Il peut s'agir d'introduire de nouvelles variables, des contraintes redondantes [9], voire des modèles redondants [7] qui ne serviront qu'à accélérer le processus de résolution.

Traditionnellement, cette phase de modélisation est faite par l'utilisateur. Si obtenir un modèle performant d'un problème difficile nécessite une expertise considérable [12], on peut noter que de nombreux utilisateurs sont tout autant déroutés par l'écriture d'un modèle simple. Nous proposons d'assister l'utilisateur en lui proposant de l'aider à construire son modèle en répondant à une série de questions permettant la découverte des contraintes. On pourra noter qu'il existe des approches visant à assister l'utilisateur dans la construction du point de vue [6] ou la reformulation de son modèle [5].

La construction d'un CSP par apprentissage a été introduite avec le système CONACQ [8] qui construit un espace de version contenant tous les CSP possibles et le fait converger avec le traitement de solutions et de

non-solutions du problème. Toutefois, cette approche est limitée à la découverte du modèle car l'utilisateur se doit de posséder des solutions de son problème. Dans un cadre général, il est probable que la découverte d'une solution soit suffisante à satisfaire l'utilisateur. Afin de remédier à ce problème, une autre approche utilise des solutions et non-solutions de problèmes proches et réalise l'apprentissage grâce à des techniques de Programmation Logique Inductive [10].

Nous présentons ici une nouvelle technique basée sur Conacq mais ne nécessitant plus de connaitre à l'avance des solutions du problème à résoudre. Pour cela, en partant d'un CSP initial vide, nous parcourons un arbre de recherche destiné à résoudre le problème (encore inconnu) de l'utilisateur. A chaque fois que la validité d'une branche est inconnue, nous posons la question de sa validité, sous forme de requêtes d'appartenance avec instances partielles des variables, à l'utilisateur et ajoutons au CSP en cours de construction les contraintes que l'on peut déduire de sa réponse, tout comme le branch-and-bound ajoute des contraintes pour changer la validité de solutions sous-optimales. Nous appelons cette approche *branch-and-learn*. Bien entendu, le système utilise la connaissance acquise jusqu'ici afin de ne pas poser de question qui soit déductible de la propagation des contraintes déjà connues.

Nous présentons en détail le système Conacq dans la section 2, puis en section 3 l'extension de Conacq aux requêtes partielles nécessaires pour pouvoir poser des questions au cours de la création de l'arbre de recherche. Dans la section 4, l'algorithme de génération de requêtes parcourant l'arbre de recherche est présenté comme un générateur d'exemples pour l'apprentissage, et nous terminons par une évaluation des résultats sur notre implantation.

2 CONACQ et la génération de requêtes

Dans un premier temps, nous commençons par décrire CONACQ, un système visant à acquérir grâce à des exemples étiquetés, les contraintes d'un problème. Nous débuterons en donnant une intuition du fonctionnement du système qui est basé sur l'espace des versions. Nous décrirons comment l'approche est encodée dans un problème SAT et la manière dont les exemples sont traités. Enfin, nous verrons comment étendre CONACQ à l'utilisation de requêtes partielles.

2.1 Acquisition de contraintes, espace des versions et formulation par un problème SAT

CONACQ est donc un système visant à répondre au problème d'acquisition de réseaux de contraintes. Pour ce faire, il propose de construire le réseau en se basant sur des solutions et non-solutions du problème à résoudre. Un exemple positif pourra éliminer certaines contraintes car cet exemple ne peut être rejeté par le réseau, alors qu'un exemple négatif informe qu'il existe au moins une contrainte qui le rejette. Cela revient à maintenir un espace des versions, indiquant d'une part les contraintes ne pouvant pas être dans le réseau, raffiné grâce aux exemples positifs, et d'autre part en représentant les contraintes encore possibles, construites à partir des exemples négatifs.

Mais un des atouts principaux de CONACQ est la possibilité d'encoder le problème grâce à une formulation SAT à la fois efficace et intuitive. Ainsi dans [3, 2], les auteurs montrent qu'en représentant les deux bornes par une seule théorie clausale, on peut à la fois représenter le réseau de contraintes et rendre le traitement des exemples simple à exprimer. Nous allons détailler cette formalisation dans la suite et nous nous en servirons dans les sections suivantes pour décrire notre approche.

Nous utilisons les notations suivantes. Nous représentons un CSP par un triplet $\langle X, D, C\rangle$, où X est l'ensemble des variables, D les domaines et enfin C les contraintes. Une contrainte est représentée par $\langle S, R\rangle$ où S est la portée de la contrainte et R sa relation de satisfaction. Cet ensemble C est dans le cas de CONACQ inconnu et doit être construit grâce aux exemples fournis par l'utilisateur et une bibliothèque de contraintes, que nous notons C_X, représentant toutes les contraintes autorisées sur les variables X. Il s'agit alors de trouver un sous-ensemble C de C_X, tel que les exemples positifs soient solutions de C et les exemples négatifs non-solutions.

Dans CONACQ, la présence d'une contrainte de C_X est représentée par une variable binaire qui, si elle est vraie, indique que la contrainte appartient au réseau et, si elle est fausse, qu'elle n'y est pas. Il faut être vigilant sur le fait qu'un booléen à faux représente l'absence de la contrainte et non sa négation. Ainsi, pour le CSP où $C = \emptyset$, toute affectation des variables est solution et les booléens associés aux contraintes sont à faux. Afin de simplifier la notation, nous ne considérons que des contraintes binaires, cependant la formalisation peut être étendue aux contraintes n-aires. Pour chaque contrainte $c(X_i, X_j)$ de C_X, nous associons donc une variable binaire que nous noterons b^c_{ij}. Nous utiliserons également une notation simplifiée pour $c(X_i, X_j)$ dans les exemples qui est c_{ij} comme utilisées dans [3, 2].

Exemple

Considérons l'ensemble des variables $\{X_0, X_1, X_2, X_3\}$ *ayant toutes comme domaine*

$\{0,1,2,3\}$. *Nous utiliserons comme types de contraintes* $\{\leq, \neq, \geq\}$. *Nous supposons que le problème cible est* $X_0 \leq X_3$, $X_1 \neq X_2$ *et* $X_2 \geq X_3$. *Les variables binaires associées à ces contraintes,* $\leq_{03}$, $\neq_{12}$ *et* $\geq_{23}$, *devront être mise à vrai à la fin de l'acquisition du réseau alors que les autres seront à faux.*

Les exemples nécessaires à CONACQ sont des solutions et non-solutions du problème cible. Autrement dit, ce sont des affectations pour chacune des variables du problème d'une valeur de son domaine. Une substitution peut se représenter comme une fonction qui pour chaque variable retourne la valeur qui lui est attribuée, $\sigma = (X_0 = v_0, X_1 = v_1, \ldots, X_n = v_n)$ où $X = \{X_i \mid 0 \leq i \leq n\}$ et $v_i \in D_i$ pour tout i de 0 à n. Dans la suite, nous simplifierons cette notation en indiquant uniquement les valeurs : $(v_0, v_1, \ldots, v_n)$.

Exemple (suite)

Dans l'exemple précédent, $e_1 = (0,0,0,0)$ *représente un exemple négatif ne satisfaisant pas la contrainte* $X_1 \neq X_2$ *alors que* $e_2 = (0,1,0,0)$ *est un exemple positif.*

Maintenant que les exemples sont représentés, il nous reste à présenter comment un exemple apporte une connaissance sur les valeurs des booléens associés aux contraintes. Selon que l'exemple soit positif ou négatif, nous avons dit que le traitement serait différent. Avant de présenter ces traitements, il est important de définir un ensemble utilisé également dans la suite, il s'agit de l'ensemble des contraintes non satisfaites. Nous rappelons qu'étant donnée une substitution σ des variables d'un CSP, une contrainte $c(X_i, X_j)$ est satisfaite si $(\sigma(X_i), \sigma(X_j))$ appartient à la relation de satisfaction de la contrainte.

Définition 2.1 ($\mathcal{K}(e)$)**.** *Étant donné un exemple* e, *on définit l'ensemble* $\mathcal{K}(e)$ *comme l'ensemble des variables booléennes* b^c_{ij} *telles que la contrainte* $c(X_i, X_j)$ *ne soit pas satisfaite dans* e.

Plus formellement,

$$\mathcal{K}(e) = \{b^c_{ij} \mid c(X_i, X_j) \in C_X \wedge c(X_i, X_j) = \langle S, R\rangle \wedge (e(X_i), e(X_j)) \notin R\}$$

Pour chaque exemple, CONACQ va exploiter cet ensemble pour apporter une information complémentaire. L'idée est que CONACQ part d'une théorie clausale K vide au départ de l'algorithme. Il ajoutera dans cette théorie des clauses sur les variables b^c_{ij}. L'apprentissage sera terminé quand il n'y aura plus qu'une seule solution au problème SAT correspondant à K. Plus précisément, CONACQ exploitera l'ensemble $\mathcal{K}(e)_{[K]}$ restreint aux variables booléennes non encore fixées dans K.

Exemple (suite)

Pour les exemples e_1 *et* e_2, *on a :*

$\mathcal{K}(e_1)_{[K]} = \{\neq_{01}, \neq_{02}, \neq_{03}, \neq_{12}, \neq_{13}, \neq_{23}\}$

et

$\mathcal{K}(e_2)_{[K]} = \{\geq_{01}, \neq_{02}, \neq_{03}, \leq_{12}, \leq_{13}, \neq_{23}\}$

CONACQ traite les exemples les uns après les autres en ajoutant des clauses selon que l'exemple est une solution ou une non-solution.

Dans le cas où l'exemple est une solution, on peut déduire que les contraintes représentées dans $\mathcal{K}(e)_{[K]}$ ne peuvent être présentes dans le réseau. En effet, si une de ces contraintes étaient présentes dans le CSP, alors par définition, cet exemple ne pourrait être une solution. On ajoutera donc pour chaque b^c_{ij} de $\mathcal{K}(e)_{[K]}$, $\neg b^c_{ij}$ à K.

Quand l'exemple est une non-solution, alors il existe une contrainte représentée dans $\mathcal{K}(e)_{[K]}$ qui appartient au réseau. En effet, si l'ensemble C_X est suffisant pour décrire le problème, au moins une des contraintes non satisfaites doit être présente pour rejeter l'exemple. On ajoutera donc la clause $\bigvee_{b^c_{ij} \in \mathcal{K}(e)_{[K]}} b^c_{ij}$ à la théorie clausale K.

Exemple (suite)

Ainsi, si CONACQ reçoit les exemples e_1 *et* e_2 *dans cet ordre, on obtient progressivement les théorie* K *ci-dessous.*

e	$\mathcal{K}(e)_{[K]}$	l	K
$e_1 = (0,0,0,0)$	$\{\neq_{01}, \neq_{02}, \neq_{03}, \neq_{12}, \neq_{13}, \neq_{23}\}$	$-$	$(\neq_{01} \vee \neq_{02} \vee \neq_{03} \vee \neq_{12} \vee \neq_{13} \vee \neq_{23})$
$e_2 = (0,1,0,0)$	$\{\geq_{01}, \neq_{02}, \neq_{03}, \leq_{12}, \leq_{13}, \neq_{23}\}$	$+$	$(\neq_{01} \vee \neq_{12} \vee \neq_{13}) \wedge \neg \geq_{01} \wedge \neg \neq_{02} \wedge \neg \neq_{03} \wedge \neg \leq_{12} \wedge \neg \leq_{13} \wedge \neg \neq_{23}$

À remarquer que e_2 *permet de fixer des valeurs pour plusieurs variables booléennes et par conséquent simplifie la première clause apprise avec* e_1.

CONACQ va ainsi traiter tous les exemples fournis par l'utilisateur. Il pourra cependant s'arrêter si toutes les variables booléennes sont fixées. En effet, dans ce cas il n'y a plus rien à apprendre et l'espace des versions a convergé sur une unique hypothèse : le réseau cible.

Ceci est un point très intéressant de CONACQ puisqu'il permet de garantir à l'utilisateur que l'algorithme d'acquisition est complètement terminé et que le réseau retourné est exactement celui ciblé par l'utilisateur. CONACQ propose différents mécanismes, non

détaillés dans ce papier, permettant d'améliorer son efficacité comme les règles de redondance, les ensembles de conflit ou encore la mise à jour d'un *backbone*.

2.2 Générateurs de requêtes

CONACQ a évolué pour répondre aux limites qui lui était reprochées vers la résolution d'un problème d'acquisition de réseaux de contraintes avec requêtes (voir [4] pour la version dirigée par les requêtes). Au lieu de demander à l'utilisateur de fournir les exemples, le système doit poser des questions à l'utilisateur : des requêtes. Ce cadre d'apprentissage est assez large[1] et nous ne nous intéressons qu'aux requêtes d'appartenance, consistant, dans le cas de l'acquisition de réseaux de contraintes, en des affectations des variables du CSP.

Définition 2.2 (Requête complète)**.** *Dans le cas de la résolution d'un problème d'acquisition d'un CSP $\langle X, D, C\rangle$, où C est inconnu, on appelle* requête complète *une affectation de toutes les variables de X à une valeur de leur domaine.*

Nous disons que la requête est positive, si l'utilisateur (l'oracle) répond que l'affectation est une solution du problème. Nous parlons sinon de requête négative.

Il faut remarquer que nous définissons ici une requête *complète* alors que dans [4] il est question simplement de requêtes. Nous avons ajouté cette qualification de « complète » pour faire la différence dans la suite avec les *requêtes partielles* que nous introduirons.

Les requêtes complètes sont donc des exemples générés par le système et étiquetés par l'utilisateur. L'intérêt est que l'utilisateur n'a plus à chercher par lui même ces exemples ce qui rend ainsi son travail plus simple. Les exemples précédemment décrits sont remplacés par ces requêtes dans le processus d'apprentissage de CONACQ. Le traitement des exemples reste le même et seule la génération de requêtes est à ajouter au système.

Pour générer des requêtes, les auteurs proposent différentes approches : affectation aléatoire des variables, génération d'une requête avec un « petit » $\mathcal{K}(e)_{[K]}$ ou encore une requête avec un « gros » $\mathcal{K}(e)_{[K]}$. Ils font également remarquer qu'en générant les requêtes de cette manière, il se peut que certaines requêtes ne soient pas intéressantes. En effet, il peut arriver qu'une requête ne contienne pas de nouvelle information car déjà rejetée de l'espace des versions. On parlera alors de requête redondante.

Définition 2.3 (Requête redondante)**.** *Une requête redondante est une requête qui ne permet pas d'apprendre de nouvelles informations. Étant donnée une requête e, il y a deux cas de requêtes redondantes :*

- *soit $\mathcal{K}(e)_{[K]} = \emptyset$ et dans ce cas on peut automatiquement détecter que e est une solution ;*
- *soit $\mathcal{K}(e)_{[K]}$ est un sur-ensemble des littéraux d'une clause présente dans K et donc e est une non-solution.*

Ainsi ces requêtes ne présentent aucun intérêt et il n'est pas nécessaire de demander à l'utilisateur de les étiqueter.

Le dernier point à évoquer dans cette section concerne un biais utilisé dans l'apprentissage par requêtes consistant à commencer l'apprentissage en fournissant au système d'apprentissage, un exemple positif (voir [1]). D'un point de vue cognitif, cela s'explique par le fait que l'enseignant doit déjà montrer un exemple positif de ce qu'il veut faire apprendre : « je veux que tu généralises les choses ressemblant à cela ». Le système d'apprentissage commence alors à poser des questions en essayant d'apprendre le concept se cachant derrière ce premier exemple. C'est également le cas dans CONACQ où les expériences menées dans [4] commencent avec une solution du problème cible. Cela pose à nouveau des difficultés car cette solution doit être fournie par l'utilisateur. Or dans notre approche, c'est justement cette solution que l'utilisateur cherche à trouver. Nous verrons cependant que ce cas n'est pas nécessaire pour la *recherche interactive* décrite dans la prochaine section.

3 Extension de CONACQ aux requêtes partielles

Dans le cadre de la recherche interactive, les requêtes peuvent prendre la forme d'affectation partielle des variables du CSP cible. Cela présente plusieurs intérêts : les requêtes étant plus petites, elles sont plus faciles à étiqueter pour l'utilisateur, il est plus facile de produire des requêtes qui correspondent à des solutions partielles pour les problèmes ayant peu de solutions, ou encore elles peuvent être plus simples à générer dans le cas par exemple de la simplification de clauses. De plus dans le cas de notre approche basée sur l'arbre de résolution d'un CSP, ces requêtes sont très naturelles à générer puisqu'il s'agit pour l'utilisateur d'étiqueter les nœuds de l'arbre de recherche que le système ne sera pas capable d'étiqueter seul (dans le cas des requêtes redondantes, le système peut décider seul).

Étant donné un CSP $\langle X, D, C\rangle$, on parle d'affectation partielle des variables X pour une substitution incomplète des variables de X à une valeur de leur domaine. Nous définissons la sémantique des requêtes

ainsi : une solution partielle est une affectation partielle où les contraintes concernées uniquement par les variables affectées de X sont satisfaites.

Une requête partielle est donc une question posée à l'utilisateur sous la forme d'une affectation partielle. Si c'est une solution partielle, elle est étiquetée positivement, sinon négativement. Pour les variables non-substituées dans une requête partielle nous utiliserons la notation Ω signifiant que la valeur pour cette variable est inconnue. Par exemple, en considérant un CSP avec trois variables X_1, X_2 et X_3 partageant le même domaine $\{1, 2\}$, une requête complète pourrait être $(1, 2, 1)$ et une requête partielle où X_2 n'est pas substituée, pourrait être $(1, \Omega, 1)$.

Pour finir avec les notations, nous introduisons une notion intéressante appelée la partie pertinente d'une requête.

Définition 3.1 (Partie pertinente d'une requête)**.** *Étant donnée une requête $e = (a_1, a_2, \ldots, a_n)$, où les a_i sont des constantes ou le symbole Ω, nous disons que $a_i \in e$ est pertinente si $a_i \neq \Omega$ et qu'il existe $b^c_{ij} \in \mathcal{K}(e)_{[K]}$ ou $b^c_{ji} \in \mathcal{K}(e)_{[K]}$.*

La partie pertinente $r(e)$ d'une requête e est le sous-ensemble minimal de e contenant les constantes pertinentes.

L'intérêt d'une telle notion est que la partie pertinente est la seule chose utile à montrer à l'utilisateur et donc nous pouvons réduire la taille de la requête de cette manière. Par la suite, nous ne considérons que les parties pertinentes quand nous parlerons de la taille d'une requête.

Pour éviter une énumération de toutes les affectations possibles, notre approche consiste à apprendre progressivement les contraintes du problème cible. La recherche se fera alors d'une manière habituelle en programmation par contraintes, alternant les phases de branchement, affectant une valeur à une variable de X, et les phases de propagation, supprimant des domaines les valeurs ne pouvant faire partie d'une solution. Après chaque étape de propagation, notre algorithme, décrit dans la section 4, évalue si l'affectation courante est une solution (partielle). S'il n'est pas capable de le déterminer seul, il soumet une requête à l'utilisateur et, dépendant de la réponse, ajoute des contraintes au CSP. La redondance d'une requête partielle est identique à celle d'une requête complète. On peut donc déterminer automatiquement si une requête redondante est une solution partielle ($\mathcal{K}(e)_{[K]}$ vide) ou s'il s'agit d'une non-solution ($\mathcal{K}(e)_{[K]}$ est un sur-ensemble d'une clause de K).

Les traitements de CONACQ pour les requêtes complètes dépendent uniquement de l'ensemble $\mathcal{K}(e)_{[K]}$. Pour étendre ces traitements aux requêtes partielles, il suffit de redéfinir cet ensemble afin qu'il prenne en compte le fait que certaines variables ne sont pas substituées. Plus précisément, il faut redéfinir $\mathcal{K}(e)$ de la manière suivante :

$$\begin{aligned}\mathcal{K}(e) = \{b^c_{ij} \mid \quad & c(X_i, X_j) \in C_X \\ & \wedge\ c(X_i, X_j) = \langle S, R\rangle \\ & \wedge\ e(X_i) \neq \Omega\ \wedge\ e(X_j) \neq \Omega \\ & \wedge\ (e(X_i), e(X_j)) \notin R\}\end{aligned}$$

Alors qu'une requête partielle est plus facile à étiqueter pour l'utilisateur, elle contient en contre-partie potentiellement moins d'information. Or, les auteurs de CONACQ évaluent leur approche en regardant le nombre de requêtes nécessaires pour faire converger leur système. Cette manière d'évaluer ne prenant pas en compte la taille des requêtes, nous proposons une mesure correspondant à $\#q \times |q|$, un genre d'aire, de surface, qui sera noté $m(q)$ dans la suite. Elle permet de relativiser le nombre de requêtes avec leur taille.

4 Génération de requêtes biaisée par la résolution du CSP

Notre algorithme est une recherche usuelle alternant la propagation des contraintes et les branchements. Pour éviter une exploration complète, nous apprenons les contraintes du problèmes grâce à des requêtes. Pour chaque nœud de l'arbre de recherche, il existe plusieurs cas. Nous rappelons qu'un nœud représente une affectation partielle des variables du CSP. Si l'affectation (partielle) représentée par le nœud est une requête redondante, nous pouvons décider sans l'utilisateur si l'exploration peut continuer dans cette branche dans le cas où nous avons une solution partielle ou bien s'arrêter et continuer l'exploration dans une nouvelle branche. Dans le cas où la requête n'est pas redondante, il faut demander à l'utilisateur de l'étiqueter pour continuer la recherche. Nous commençons donc par résoudre un problème sans ses contraintes et ajoutons les contraintes apprises lors de l'apprentissage. Cette démarche est similaire à celle que l'on peut trouver dans les algorithmes de recherche *branch-and-bound*. Dans ces derniers, à chaque solution trouvée dans l'arbre de recherche une contrainte est ajouté pour forcer la prochaine solution à être meilleure. Dans notre cas, à chaque requête nous ajoutons une contrainte pour éviter de parcourir des états de recherche similaires (requêtes redondantes). Pour mettre en avant cette analogie, nous appelons notre approche *branch-and-learn*. Ce n'est évidemment pas la seule façon d'exploiter les requêtes partielles mais celle-ci a

l'avantage de fournir un cadre clair à l'utilisateur. La machine cherche à résoudre le problème dès le départ et interagit avec l'utilisateur pour cerner le modèle jusqu'à trouver une solution.

Pour généraliser, nous voyons notre processus comme un générateur de requête pour un système comme CONACQ qui serait capable de gérer les requêtes partielles. Dans [4], les auteurs proposent de générer les requêtes à la volée en fonction de l'état du processus d'apprentissage. Leur générateur ne produit que des requêtes non redondantes, utiles pour améliorer la théorie clausale courante. Les différentes étapes d'un tel système sont :

- Génération d'une requête q ;
- Étiquetage de q par l'utilisateur ;
- Mise à jour de K avec q en fonction de la réponse.

Nous présentons notre algorithme de résolution de ce point de vue. Notre approche consiste donc en deux modules, un premier recherchant une solution dans l'arbre de résolution et qui retourne une requête quand cela est nécessaire, et un autre mettant à jour le modèle de contraintes du problème à la manière de CONACQ mais gérant les requêtes partielles.

En utilisant comme biais de génération de requêtes l'arbre de résolution du CSP, nous commençons par la racine de l'arbre décrivant une affectation partielle vide (aucune variable n'a de substitution). Ensuite, un nœud n_2 est obtenu à la suite un nœud n_1 si l'affectation partielle représentée par n_2 est un sur-ensemble de celle de n_1. La structure de l'arbre dépend en fait de la manière dont sont produits les nouveaux nœuds à partir d'un existant. Par exemple, une stratégie usuelle consiste à fixer une valeur pour une variable. Il y a alors deux nouveaux nœuds : celui où la variable est affectée la valeur et celui où cette valeur est retirée du domaine de la variable. Mais d'autres stratégies existent comme par exemple fixer plus d'une variable. Ces stratégies forment les stratégies de branchement et pour généraliser nous dirons qu'une stratégie ajoute seulement des contraintes aux nouveaux nœuds. Ainsi, pour fixer une variable X_i à la valeur a, on ajoutera la contrainte $X_i = a$ pour créer le nœud. L'autre nœud où l'on retire a du domaine de X_i aura la contrainte $X_i \neq a$.

Comme le CSP est progressivement appris par le module de type CONACQ, il est important, avant de traiter un nœud, de mettre à jour ses contraintes. En effet, nous partons d'un nœud où il n'y a aucune contrainte et il faut les ajouter dès que celles-ci sont connues. Pour se faire nous proposons un encodage proche de celui proposé par CONACQ. Le CSP de départ de notre arbre est composé des variables X associées à leur domaine D. Nous ajoutons également les variables booléennes b^c_{ij} comme défini dans CONACQ. Pour résumer, les variables du CSP sont les suivantes :

$$\forall X_i \in X, X_i \in D_{X_i}$$
$$\forall c(X_i, X_j) \in C_X, b^c_{ij} \in \{0, 1\}$$

Les contraintes de départ de ce CSP sont alors les suivantes :

$$\forall c(X_i, X_j) \in C_X, b^c_{ij} \Rightarrow c(X_i, X_j)$$

Ainsi, une affectation d'un booléen décidera si une contrainte sera « activée » ou pas. Ces variables booléennes ayant la même sémantique que pour CONACQ, si l'une d'elles est mise à faux alors la contrainte n'est pas présente (ce qui ne signifie pas que sa négation le soit). Cela explique pourquoi nous avons une implication plutôt qu'une équivalence.

La mise à jour d'un nœud revient à fixer à vrai ou faux les variables booléennes dont la valeur est connue dans le module d'apprentissage des contraintes. On ajoutera également les clauses apprises afin de limiter les requêtes redondantes.

Maintenant que les nœuds sont définis, le dernier point concerne la manière dont le générateur de requêtes va les explorer. Plusieurs stratégies peuvent être utilisées comme la recherche en profondeur (DFS), le *restart*, le *branch-and-bound*... Nous nommons ces stratégies les stratégies d'exploration.

L'algorithme de la Figure 1 résume le processus de génération de requêtes biaisé par l'arbre de résolution. Premièrement, introduisons quelques notations. Soit *nodes* l'ensemble ordonné des nœuds qui attendent d'être traités. Son implémentation dépend de la stratégie d'exploration. L'opération *next* de *nodes* retourne le premier nœud de l'ensemble et *erase* supprime le nœud en argument de l'ensemble. Pour un nœud n, l'opération *update* met à jour le modèle de contraintes du nœud en fonction de la théorie clausale courante K, *propagate* réalise la propagation des contraintes et *query* extrait la requête représentée par le nœud. Finalement, l'opération *branching*, dépendant de la stratégie de branchement, produit tous les fils possibles du nœud donné en paramètre.

L'algorithme commence par récupérer le prochain nœud. Après la mise à jour des contraintes et leur propagation, la requête q est extraite du nœud. Si la requête n'est pas redondante, il faut la soumettre à l'utilisateur et l'algorithme s'arrête en la retournant. Sinon, nous pouvons déterminer s'il s'agit d'une solution (partielle) ou non. L'algorithme produit alors de nouveaux nœuds si la requête n'est pas complète et procède à un appel récursif pour explorer les nœuds

Algorithm : GENERATE

```
1.  cur ← nodes.next()
2.  cur.update(K)
3.  cur.propagate()
4.  q ← cur.query()
5.  if q is non-redundant then
6.    return q
7.  else
8.    nodes.erase(cur)
9.    if q is a (partial) solution then
10.     if q is a complete query then
11.       cur is a solution
12.     else
13.       nodes ← nodes ∪ branching(cur)
14.     end if
15.   end if
16.   GENERATE
17. end if
```

FIGURE 1 – Génération de requêtes biaisées par l'arbre de résolution

suivants. Reste le cas particulier où il y a une solution (requête complète redondante positive). Dans le cas où nous cherchons justement une solution le processus s'arrêtera en fournissant cette solution à l'utilisateur.

Exemple

Pour illustrer notre algorithme, nous l'appliquons au problème suivant. Le problème a quatre variables $\{X_0, X_1, X_2, X_3\}$ *avec comme domaine* $\{0, 1, 2, 3\}$ *et les types de contraintes sont* $\{\leq, \neq, \geq\}$*. Les contraintes du problème cible sont* $X_0 \leq X_3$*,* $X_1 \neq X_2$ *et* $X_2 \geq X_3$*.*

Nous utilisons une recherche en profondeur d'abord, en branchant sur les variables dans l'ordre suivant X_0*,* X_1*,* X_2 *et* X_3 *avec la plus petite valeur possible. La Figure 2 représente l'arbre de recherche exploré par notre algorithme. Les nœuds bleus (*e_1 *et* e_2*) représentent des requêtes redondantes et ne sont pas soumis à l'utilisateur. Les nœuds verts (*e_3*,* e_5 *et* e_6*) et rouges (*e_4*) représentent respectivement des requêtes positives et négatives étiquetées par l'utilisateur.*

Le tableau de la Figure 3 montre les contraintes apprises avec les requêtes successives. La première colonne donne les requêtes soumises à l'utilisateur. La seconde montre les $\mathcal{K}(e)_{[K]}$ *correspondant aux requêtes et la troisième la réponse de l'utilisateur,* $+$ *signifiant positif et* $-$ *négatif. La dernière montre les contraintes ajoutées à* K *qui seront également ajoutées au CSP en construction lors de la mise à jour.*

La recherche commence avec deux requêtes redondantes où le $\mathcal{K}(e)_{[K]}$ *est vide. Comme ces requêtes correspondent à des solutions (partielles) la recherche continue. La première requête soumise est* e_3 *qui est une solution partielle permettant d'éliminer la contrainte* $X_0 \neq X_1$*. Pour la seconde, l'utilisateur répond négativement et notre système apprend la clause* $\neq_{02} \vee \neq_{12}$*. La recherche retourne alors en arrière et branche sur* $X_2 = 1$*. Ensuite, nous avons deux requêtes positives pour obtenir une solution au problème.*

Il est clair que sur un exemple jouet comme celui-ci, il est difficile de voir l'intérêt apporté par l'apprentissage de contraintes, qui est censé éviter de parcourir exhaustivement tout l'espace de recherche. Nous verrons cependant dans la section suivante qu'en continuant la recherche sur ce problème les contraintes apprises permettent d'élaguer une bonne partie de l'arbre de recherche. Sur des problèmes vraiment complexes, où peu de solutions existent, les contraintes apprises permettent également de converger plus efficacement vers une solution.

L'utilisateur peut souhaiter plusieurs solutions, si la première ne lui convenait pas tout à fait (même si elle respecte les contraintes de son problème). Dans ce cas, nous pouvons continuer la recherche là où elle avait été arrêtée et rechercher la prochaine solution, voire toutes les solutions. Pour cela, nous poursuivons l'exploration de l'arbre de recherche de la même manière que précédemment.

En faisant ainsi, nous pouvons nous demander s'il est possible d'apprendre le modèle exact du problème. Comme nous le verrons dans la partie concernant les expériences que nous avons menées, notre méthode ne permet pas de garantir une convergence de l'espace des versions comme c'était le cas dans CONACQ. Cela est dû à la manière dont nous générons nos requêtes et à l'absence de règles de redondances dans notre système. Cependant les contraintes apprises auxquelles on ajoute les clauses sur les variables non fixées sont suffisantes pour reproduire la résolution du CSP sans aucune requête pour l'utilisateur. Nous finissons cette section en montrant que les ensembles de solutions du CSP cible et celles du CSP construit à partir des contraintes et clauses apprises pendant la résolution sont les mêmes. Avant cela nous introduisons quelques notations nécessaires. Dans la suite, nous noterons le CSP que cible l'utilisateur CSP_c. L'ensemble des solutions d'un CSP nommé P sera noté $sol(P)$. Enfin, étant donnés le $\text{CSP}_c = \langle X, D, C \rangle$, la bibliothèque de contraintes C_X et une théorie clausale K, nous définissons le CSP CSP_K de la manière suivante :

- $\text{CSP}_K = \langle X', D', C' \rangle$
- $X' = X \cup \{b^c_{ij} \mid c(X_i, X_j) \in C_X\}$
- $D' = D \cup \{D_{b^c_{ij}} = \{0, 1\} \mid c(X_i, X_j) \in C_X\}$
- $\forall c(X_i, X_j) \in C_X, (b^c_{ij} \Rightarrow c(X_i, X_j)) \in C'$
- Pour toute clause m de K, $m \in C'$

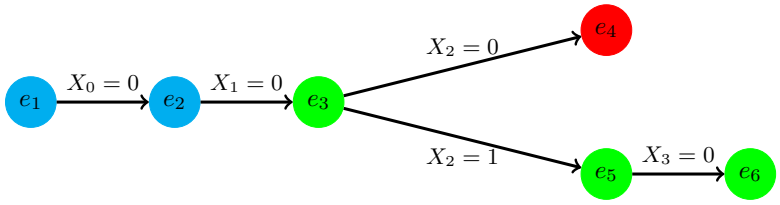

FIGURE 2 – Recherche d'une solution

e	$\mathcal{K}(e)_{[K]}$	Label	Added constraints
$e_3 = (0, 0, \Omega, \Omega)$	$\{\neq_{01}\}$	+	$\neg \neq_{01}$
$e_4 = (0, 0, 0, \Omega)$	$\{\neq_{02}, \neq_{12}\}$	−	$\neq_{02} \vee \neq_{12}$
$e_5 = (0, 0, 1, \Omega)$	$\{\geq_{02}, \geq_{12}\}$	+	$\neg \geq_{02} \wedge \neg \geq_{12}$
$e_6 = (0, 0, 1, 0)$	$\{\neq_{03}, \geq_{23}, \neq_{13}\}$	+	$\neg \neq_{03} \wedge \neg \geq_{23} \wedge \neg \neq_{13}$

FIGURE 3 – Contraintes apprises pendant la recherche interactive d'une solution

Ce CSP possède les mêmes variables et domaines que le CSP cible, les mêmes contraintes réifiées permettant de faire le lien entre les variables booléennes et les contraintes de C_X ainsi que les clauses de la théorie clausale sous forme de contraintes réifiées quand les clauses contiennent plus d'un littéral.

Théorème 4.1. *Étant donnés un CSP cible* CSP_c, C_X *permettant de le représenter et la théorie clausale* K *obtenue en cherchant toutes les solutions avec un parcours complet de l'arbre de recherche, nous avons* $sol(\text{CSP}_c) = sol(\text{CSP}_K)$.

Démonstration Pour obtenir K, notre méthode consiste de partir d'une théorie vide K_0 et d'ajouter à chaque requête, soit une clause m si la requête est négative, soit un ensemble de clauses unitaires de la forme $\neg b^c_{ij}$ si elle est positive.

Nous avons donc une séquence de raffinements $K_0, K_1, \ldots, K_n = K$ vérifiant l'implication logique $K_{i+1} \rightarrow K_i$.

Montrons que $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_K)$. Supposons qu'il existe une requête complète e telle que $e \in sol(\text{CSP}_c)$ et $e \notin sol(\text{CSP}_K)$. Il existe donc un K_i rejettant e ce qui est une contradiction avec le parcours complet de l'arbre de recherche. Montrons que $sol(\text{CSP}_K) \subseteq sol(\text{CSP}_c)$. Trivialement, nous avons $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_{K_0})$. Supposons maintenant qu'il existe i compris entre 0 et n tel que $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_i})$. Nous montrons maintenant que $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_{i+1}})$. Dans le cas d'une requête positive en passant de K_i à K_{i+1}, les solutions de $\text{CSP}_{K_{i+1}}$ restent les mêmes que celles de CSP_{K_i}, la différence entre les deux théories résidant uniquement dans l'élimination de contraintes (ajout de clauses unitaires du type $\neg b^c_{ij}$). On a donc $\forall s \in sol(\text{CSP}_c), s \in sol(\text{CSP}_{K_{i+1}})$. Dans le cas d'une requête négative, nous avons $K_{i+1} = K_i \cup \{m\}$ où m est une clause. Alors, $sol(\text{CSP}_{K_{i+1}}) = sol(\text{CSP}_{K_i}) \cap (\bigcup_{b^c_{ij} \in m} sol(c(X_i, X_j))$, où $sol(c(X_i, X_j))$ est l'ensemble des affectations de X satisfaisant la contrainte $c(X_i, X_j)$. Par hypothèse $sol(\text{CSP}_c) \subseteq sol(\text{CSP}_{K_i})$, il nous reste donc à montrer $sol(\text{CSP}_c) \subseteq (\bigcup_{b^c_{ij} \in m} sol(c(X_i, X_j))$. Supposons qu'il existe une solution s de $sol(\text{CSP}_c)$ telle que pour tout b^c_{ij} de m, s n'est pas dans $sol(c(X_i, X_j))$. Or d'après la définition des traitements des requêtes, les b^c_{ij} de m devraient être à 0 comme $s \in sol(\text{CSP}_c)$. La requête aurait donc dû être positive ce qui mène à une contradiction.

Nous obtenons donc bien l'égalité entre ces deux ensembles de solutions.

5 Évaluation de la recherche interactive

Nous proposons deux heuristiques directement inspirées de la version dirigée par les requêtes de CONACQ. Nommées respectivement `MaxK` et `MinK`, ces heuristiques choisissent de compléter une instance partielle des variables en affectant une valeur à une variable telle que la nouvelle instance e créée possède respectivement le plus grand $\mathcal{K}(e)_{[K]}$ possible ou le plus petit. Afin de pouvoir comparer ces heuristiques, nous utiliserons également `Dom` qui affecte à la variable ayant le plus petit domaine la plus petite valeur de son domaine.

Pour étudier le comportement de notre approche, nous avons utilisé des problèmes ayant peu de solutions comme le problème du zèbre. Les spécifications de ces problèmes sont données en annexes A de [11]. Notre motivation initiale étant d'aider l'utilisateur à trouver des solutions à des problèmes qu'il ne peut résoudre facilement nous avons donc préféré ce genre de problème.

La plupart n'ayant qu'une solution, une comparaison entre notre approche et CONACQ n'a pas réellement de sens. De plus, CONACQ ne gére pas les requêtes partielles et son objectif diffère (trouver une ou toutes les solutions pour nous, trouver les contraintes pour CONACQ). On peut cependant faire remarquer que sur les jeux de données testés dans [4] notre approche nécessite plus de requêtes. Cela est à relativiser car la taille de nos requêtes est naturellement plus petite que celles de CONACQ qui n'utilise pas de requêtes partielles. Nous proposons de concentrer notre analyse sur notre approche.

Nous avons effectué deux séries d'expériences, celles-ci diffèrent par leur utilisation de la simplification de clauses, mécanisme créé dans CONACQ visant à simplifier une clause apprise en générant des requêtes avec un $\mathcal{K}(e)_{[K]}$ de taille 1. À noter que cette technique a été étendue à l'utilisation de requêtes partielles permettant de générer plus facilement les requêtes. Les Figures 4 et 5 résument les résultats obtenus. Le sens des colonnes est le suivant :

- # variables : le nombre de variables du CSP
- # contraintes cibles : le nombre de contraintes du CSP quand il est écrit « à la main » ;
- stratégie : stratégie d'exploration de l'arbre de recherche ;
- #q : le nombre de requêtes soumis à l'utilisateur ;
- |q| : la taille moyenne des requêtes ;
- m(q) : $\#q \times |q|$;
- temps : le temps mis par le système (avec étiquetage automatique) ;
- nœuds explorés : le nombre de nœuds explorés dans l'arbre de recherche ;
- contraintes apprises : le nombre de contraintes dont le littéral est fixé (à vrai ou à faux).

Comme nous pouvons le voir sur la Figure 4, l'utilisation d'heuristiques telles que `MaxK` et `MinK` ne permet pas de faire baisser le nombre de requêtes par rapport à une heuristique comme `Dom`. Cela peut s'expliquer par le fait que ces heuristiques ont été pensées pour chercher des requêtes visant à apprendre le réseau de contraintes plutôt qu'à arriver rapidement vers une solution. On peut s'interroger sur le fait que `MaxK` produit des requêtes plus petites que `MinK` alors que l'intuition aurait été l'inverse. En fait, cela s'explique par le fait qu'en cherchant à maximiser la taille du $\mathcal{K}(e)_{[K]}$, on augmente le nombre de contraintes violées par la requête et donc la probabilité que l'une d'elles appartienne au réseau cible. Par conséquent, on retournera plus rapidement en arrière dans l'arbre de recherche et on gardera donc des requêtes assez petites.

Dans la Figure 5, nous nous sommes intéressés à l'utilisation de la simplification de clauses. Comme dit précédemment, `MaxK` et `MinK` ont été conçus dans l'esprit de trouver les contraintes du réseau cible et si l'on se contente des clauses (rarement unaires) apprises quand on a une requête négative, ces clauses n'ont que peu d'incidence sur l'exploration de l'arbre de recherche. Par contre, en recherchant dans ces clauses, une contrainte à ajouter à notre réseau, nous pouvons constater que chercher à simplifier les clauses produira plus de requêtes mais également de plus petites requêtes. Si, nous nous intéressons à la colonne m(q), nous pouvons voir qu'une heuristique comme $MaxK$ devient très intéressante.

6 Conclusion

Dans ce papier, nous avons voulu fournir un nouveau regard sur les problématiques d'acquisition de réseaux de contraintes. Plutôt que de se fixer comme objectif d'apprendre le réseau, nous partons de l'hypothèse que seule une solution au problème intéresse l'utilisateur. Nous avons répondu à cela en proposant la recherche interactive. En partant d'un CSP sans aucune contrainte, nous résolvons ce problème en posant à certains moments des requêtes à l'utilisateur permettant ainsi de connaître certaines contraintes. Nous avons étendu le système CONACQ aux requêtes partielles. Ces dernières offrent des perspectives plus larges d'utilisation de ce système.

Les requêtes partielles n'ont d'ailleurs pas encore été complètement exploitées. Premièrement, leur utilisation pose des problèmes avec les règles de redondance utilisées dans CONACQ. Il serait intéressant de voir comment permettre leur utilisation afin de bénéficier de plus d'information sémantique sur les contraintes. Deuxièmement, les requêtes partielles rendent plus facile la simplification des clauses et pourraient permettre d'autres méthodes que celles qui avaient été proposées dans CONACQ.

Références

[1] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2 :319–342, April 1988.

[2] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *ECML*, pages 23–34, 2005.

[3] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. Acquiring constraint networks using a sat-based version space algorithm. In *AAAI*. AAAI Press, 2006.

[4] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint

Jeux de données	# variables	# contraintes cibles	stratégie	#q	|q|	m(q)	temps	nœuds explorés	contraintes apprises
Prudey's general store	12	45	Dom	109	5,45	594	0,61	711	121/198
			MaxK	142	4,18	594	2,06	632	171/198
			MinK	139	5,26	731	2,64	728	164/198
Allergic reactions	12	29	Dom	60	6,55	393	0,14	175	99/198
			MaxK	69	5,77	398	0,29	181	119/198
			MinK	99	6,67	660	0,20	318	143/198
Miffed millionaires	16	43	Dom	132	7,67	1012	1,11	618	237/360
			MaxK	169	5,81	982	6,07	1086	287/360
			MinK	185	6,85	1267	6,28	1178	292/360
Problème du zèbre	25	71	Dom	511	9,34	4773	92,47	3727	1741/2100
			MaxK	508	6,85	3292	1379,18	8728	2021/2100
			MinK	-	-	-	>10h	-	-

FIGURE 4 – Résultats de recherche d'une solution avec les différentes heuristiques

Jeux de données	# variables	# contraintes cibles	stratégie	#q	|q|	m(q)	temps	nœuds explorés	contraintes apprises
Prudey's general store	12	45	Dom	141	3,08	434	0,22	249	161/198
			MaxK	142	2,99	425	0,76	289	168/198
			MinK	132	3,14	414	0,49	238	162/198
Allergic reactions	12	29	Dom	101	3,08	311	0,21	155	122/198
			MaxK	98	3,04	298	0,32	163	124/198
			MinK	92	3,13	288	0,19	143	118/198
Miffed millionaires	16	43	Dom	224	3,47	777	0,47	337	288/360
			MaxK	196	3,36	659	0,76	306	277/360
			MinK	224	3,40	762	0,67	319	275/360
Problème du zèbre	25	71	Dom	589	7,38	4347	212,71	2737	1860/2100
			MaxK	384	4,68	1797	393,6	3518	2000/2100
			MinK	-	-	-	>10h	-	-

FIGURE 5 – Résultats de recherche d'une solution avec les heuristiques et de la simplification de clauses

acquisition. In Manuela M. Veloso, editor, *IJCAI*, pages 50–55, 2007.

[5] Christian Bessière, Remi Coletta, and Thierry Petit. Acquiring parameters of implied global constraints. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 747–751. Springer, 2005.

[6] Christian Bessiere, Joël Quinqueton, and Gilles Raymond. Mining historical data to build constraint viewpoint. In Ian Miguel and Steve Prestwich, editors, *CP'06 Workshop on modelling and Reformulation*, 2006.

[7] B. M. W. Cheng, Jimmy Ho-Man Lee, and J. C. K. Wu. Speeding up constraint propagation by redundant modeling. In Eugene C. Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 1996.

[8] Remi Coletta, Christian Bessière, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joël Quinqueton. Semi-automatic modeling by constraint acquisition. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 812–816. Springer, 2003.

[9] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *ECAI*, pages 290–295, 1988.

[10] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *ICTAI (1)*, pages 45–52. IEEE Computer Society, 2010.

[11] Matthieu Lopez. *Apprentissage de problèmes de contraintes.* PhD thesis, Université d'Orléans, 2011.

[12] Jean-Francois Puget. Constraint programming next challenge : Simplicity of use. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8. Springer, 2004.

[13] Barbara M. Smith. *Handbook of Constraint Programming*, chapter Modelling, pages 377–406. Elsevier, 2006.