Inférence de supports pour les algorithmes de filtrage générique

Frédéric Boussemart   Fred Hemery   Christophe Lecoutre   Lakhdar Sais

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France

{boussemart,hemery,lecoutre,sais}@cril.univ-artois.fr

Résumé

Dans cet article, nous proposons une analyse statique des différentes contraintes d'un réseau afin d'identifier certaines propriétés (ou caractéristiques) générales. L'utilisation de ces propriétés rend possible une inférence de supports qui permet de réduire le nombre de tests de consistance. En effet, l'exploitation de certaines propriétés identifiées lors d'une phase de pré-traitement peut apporter une amélioration substantielle des algorithmes de recherche qui maintiennent une forme de consistance locale telle que la consistance d'arc. Les résultats d'expérimentations menées sur de nombreuses classes d'instances démontrent l'intérêt de cette approche.

Abstract

In this paper, we show that a static analysis of constraints can deliver interesting structural knowledge. Such knowledge, that can be viewed as some general constraint properties, is very useful in achieving support inference, thereby reducing the cost of consistency checking. Indeed, exploiting during search some properties established in a preprocessing step can then lead to substantial improvements of backtracking algorithms that maintain a form of local consistency such as arc consistency. Extensive experiments on many classes of constraint satisfaction instances demonstrate the effectiveness of our approach.

1 Introduction

Le développement d'approches génériques efficaces permettant la résolution d'instances du Problème de Satisfaction de Contraintes est considéré comme l'un des enjeux majeurs de la communauté Programmation par Contraintes (CP). La propagation de contraintes étant considérée comme particulièrement importante lors de la mise au point de procédures de recherche efficaces, de nombreuses techniques de filtrage ont été proposées. On peut néanmoins opérer une distinction entre les techniques de filtrage spécifique et les techniques de filtrage générique. Contrairement au filtrage générique, le filtrage spécifique ne s'applique qu'à certaines classes de contraintes. Elles nécessitent un effort de développement particulier mais, s'avèrent en contrepartie plus efficaces. Dans ce papier, nous nous intéressons à l'exploitation de certaines propriétés générales des contraintes dans le but d'améliorer l'efficacité des techniques de filtrage générique, et par conséquent, de réduire l'écart existant entre les deux approches.

La consistance d'arc est une propriété générale des réseaux de contraintes. Cette propriété peut être établie avant la recherche d'une solution et/ou maintenue pendant la recherche. De nombreuses améliorations ont été apportées (voir [14, 4, 5, 6, 18, 17, 12]) pour réduire la complexité théorique et pratique de l'algorithme élémentaire de consistance d'arc ([13]). Un intérêt immédiat de ces travaux est de permettre la réduction de l'effort produit par l'algorithme qui maintient la consistance d'arc au cours de la recherche, algorithme appelé MAC (Maintaining Arc Consistency) [15]. Par ailleurs, il a été montré que la manipulation de structures de données légères dans le contexte d'un algorithme de consistance d'arc à gros grain [6, 18, 17, 12] était une approche efficace.

Lorsque la consistance d'arc est maintenue au cours de la recherche, un processus d'initialisation est réalisé lors d'une étape de pré-traitement, et un processus de propagation est ensuite itéré à chaque étape de la recherche d'une solution. Nous proposons ici de nouvelles améliorations pour réduire le coût du processus de propagation. En fait, nous proposons d'utiliser les informations obtenues au cours de l'étape de pré-traitement pour éviter des tests de consistance inutiles au cours de la recherche.

De manière plus précise, nous proposons d'analyser la structure de chaque contrainte dans le but d'en extraire certaines caractéristiques intéressantes telles que :

- les ensembles de conflits et, plus précisément, leur cardinalité,
- les sous-ensembles de valeurs, appelées couvertures, garantissant la consistance d'arc au niveau d'une contrainte,
- les valeurs liées par la relation de substituabilité.

Nous montrons ensuite comment exploiter ces caractéristiques au niveau d'un algorithme de maintien de la consistance d'arc. Il est important de noter que l'analyse effectuée au cours du pré-traitement peut l'être également au cours de la recherche car les propriétés évoluent (par exemple, certaines valeurs peuvent devenir substituables alors qu'elles ne l'étaient pas) lorsque le réseau est modifié. Le coût engendré par une telle analyse nous amène à renoncer à effectuer ce traitement à chaque étape de la recherche.

Ce papier est organisé comme suit : après quelques définitions préliminaires, nous présentons quelques propriétés générales permettant une inférence de support. Nous montrons ensuite comment exploiter ces propriétés au niveau des algorithmes de consistance d'arc à gros grain. Nous présentons les résultats expérimentaux obtenus à partir de différentes classes d'instances CSP. Enfin, avant de conclure, nous présentons les différents travaux connexes.

2 Définitions préliminaires

Définition 1 *Un réseau de contraintes P est un couple $(\mathscr{X}, \mathscr{C})$ où :*

- *$\mathscr{X} = \{X_1, \ldots, X_n\}$ est un ensemble fini de n variables tel que chaque variable X_i possède un domaine $dom(X_i)$ représentant l'ensemble des valeurs pouvant être affectées à X_i,*
- *$\mathscr{C} = \{C_1, \ldots, C_m\}$ est un ensemble fini de m contraintes tel que chaque contrainte C_j correspond à une relation $rel(C_j)$ représentant l'ensemble des tuples autorisés pour les variables $vars(C_j) \subseteq \mathscr{X}$ liées par la contrainte C_j.*

On dit qu'une contrainte C lie une variable X si et seulement si X appartient à *vars*(C). On appelle arité d'une contrainte C le nombre de variables liées par C. Une solution est une assignation de valeurs à l'ensemble des variables telle que toutes les contraintes soient satisfaites. Un réseau de contraintes est satisfiable lorsqu'il admet au moins une solution. Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem), qui consiste à déterminer si un réseau de contraintes donné est satisfiable, est NP-complet. Un réseau de contraintes est également appelé instance CSP. Dans cet article, résoudre une instance revient soit à trouver une solution, soit à démontrer qu'elle est insatisfiable.

Nous introduisons quelques notations fréquemment utilisées dans la suite de ce papier :

- $|S|$ représente la cardinalité d'un ensemble S,
- $\prod_{i=1}^{k} S_i$ représente le produit cartésien des k ensembles $S_1, \ldots, S_k$, i.e. l'ensemble $\{(a_1, \ldots, a_k) \mid a_i \in S_i, 1 \leq i \leq k\}$,
- dom(C) représente le domaine de C, i.e. le produit cartésien $\prod_{j=1}^{k} dom(X_{i_j})$, C étant une contrainte d'arité k telle que vars(C) = $\{X_{i_1}, \ldots, X_{i_k}\}$,
- $dom(C \backslash X_{i_p})$ correspond à $\prod_{j=1}^{k} dom(X_{i_j}) \mid j \neq p$,
- pour tout élément $t = (a_1, \ldots, a_k)$, appelé tuple ou plus précisément k-uplet, de $dom(C)$:
  - $t[X_{i_p}]$ représente a_p,
  - $t[\overline{X}_{i_p}]$ représente $(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_k)$.

Les méthodes de recherche complètes utilisent, en règle générale, un algorithme de recherche en profondeur d'abord avec gestion de retours-arrières, où à chaque étape de la recherche, une assignation de variable est effectuée suivie par un processus de filtrage appelé propagation de contraintes. Les algorithmes de propagation de contraintes sont basés sur certaines propriétés des réseaux telles que la consistance d'arc [13] afin d'éliminer, le plus souvent, des valeurs qui ne peuvent apparaître dans aucune solution.

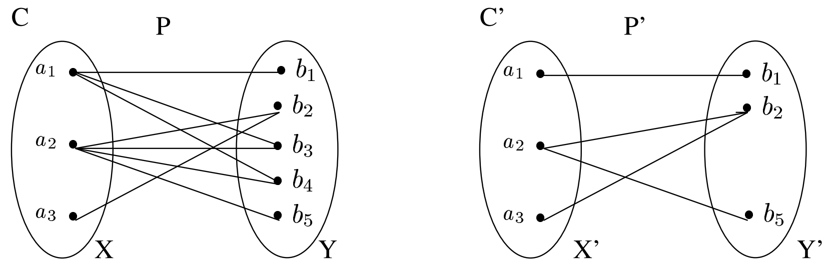
Définition 2 *Soient C une contrainte, $X \in vars(C)$ et $a \in dom(X)$. Un tuple t est un support pour C ssi $t \in rel(C)$. Un tuple t est un support de (X,a) pour C ssi t est un support pour C tel que $t[X] = a$.*

Définition 3 *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes, $C \in \mathscr{C}$, $X \in vars(C)$ et $a \in dom(X)$. (X, a) est dit arc consistant pour C ssi il existe un support de (X, a) pour C. X est dit arc consistant pour C ssi $\forall a \in dom(X)$, (X, a) est arc consistant pour C. P est dit arc consistant ssi $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ et $\forall C \in \mathscr{C}$, $\forall X \in vars(C)$, X est arc consistant pour C.*

Définition 4 *Soient C une contrainte, $X \in vars(C)$ et $a \in dom(X)$.*

- *l'ensemble des supports de C restreint à $X = a$, noté $supps(C)_{X=a}$, est l'ensemble $\{u \in dom(C \backslash X) \mid \exists t \in rel(C)$ avec $t[\overline{X}] = u \wedge t[X] = a\}$.*
- *l'ensemble des conflits de C restreint à $X = a$, noté $confs(C)_{X=a}$, est l'ensemble $\{u \in dom(C \backslash X) \mid \exists t \in dom(C) - rel(C)$ avec $t[\overline{X}] = u \wedge t[X] = a\}$.*

Notons que pour une contrainte d'arité k, les ensembles restreints de supports et de conflits contiennent des $(k-1)$-uplets. Par exemple, si (a, b, c) et (a, d, e) sont



$|confs(C)_{X=a1}| = 2$
$|confs(C)_{X=a2}| = 1$
$|confs(C)_{X=a3}| = 4$

$|dom(Y')| > 2 \Rightarrow |supps(C')_{X'=a_1}| > 0$
$|dom(Y')| > 1 \Rightarrow |supps(C')_{X'=a_2}| > 0$
$|dom(Y')| \leq 4 \Rightarrow |supps(C')_{X'=a_3}|$?

FIG. 1 – Inférence de supports par utilisation de la cardinalité des ensembles de conflits

deux supports pour une contrainte ternaire C telle que $vars(C) = \{X, Y, Z\}$, alors (b, c) et (d, e) appartiennent à $supps(C)_{X=a}$.

3 Cadre formel

Dans cette section, nous présentons quelques propriétés des réseaux de contraintes pouvant être exploitées par tout algorithme de propagation qui permet de réduire le domaine des variables en se basant sur une *domain filtering consistency* [9]. De manière plus précise, nous présentons des propriétés qui peuvent être utilisées avantageusement par tout algorithme de recherche avec retour-arrières qui réalise alternativement assignation de variable et propagation de contraintes en vérifiant à chaque étape le maintien d'une propriété telle que la consistance d'arc, la consistance de chemin restreinte, la singleton consistance d'arc, etc.

Ces propriétés permettent de réduire le coût de la recherche de supports réalisée par les algorithmes de filtrage. Le nombre de tests de consistance réalisés au cours de la propagation de contrainte s'en trouve significativement réduit. Nous décrivons ces propriétés dans le cadre de contraintes d'arité quelconque même si, par soucis de simplicité, nous en donnons des illustrations dans le cadre binaire.

On notera $P \rightarrow P'$ lorsqu'une instance CSP P' peut être construite à partir d'une instance CSP P par réduction des domaines associés aux variables de P et en adaptant en conséquence les relations associées aux contraintes de P. Plus concrètement, on peut considérer P comme l'instance que l'on cherche à résoudre, et P' l'instance obtenue à partir de P après avoir assigné un nombre quelconque (éventuellement nul) de variables de P et après avoir exécuté un algorithme de propagation de contraintes.

Définition 5 *Soient $P = (\mathscr{X}, \mathscr{C})$ et $P' = (\mathscr{X}', \mathscr{C}')$ deux réseaux de contraintes, $P \rightarrow P'$ ssi il existe deux bijections f_x et f_c telles que :*

- *f_x est défini de $\mathscr{X}$ vers $\mathscr{X}'$ tel que $\forall X \in \mathscr{X}$, $f_x(X) = X'$ avec $dom(X') \subseteq dom(X)$, i.e X' est la variable obtenue à partir de X par réduction de son domaine,*
- *f_c est défini de $\mathscr{C}$ vers $\mathscr{C}'$ tel que $\forall C \in \mathscr{C}$, $f_c(C) = C'$ avec $vars(C') = \{f_x(X) \mid X \in vars(C)\} \wedge rel(C') = rel(C) \cap dom(C')$, i.e. C' est la contrainte obtenue à partir de C en éliminant de $rel(C)$ les tuples n'appartenant pas à $dom(C')$.*

Dans la suite de cette section, nous considérons fixés deux réseaux de contraintes $P = (\mathscr{X}, \mathscr{C})$ et $P' = (\mathscr{X}', \mathscr{C}')$ tels que $P \rightarrow P'$. Les deux bijections f_x et f_c sont également considérées comme fixées. Par souci de simplicité, et sans perte de généralité, $f_x(X) = X'$ et $f_c(C) = C'$ seront respectivement notées $X \rightarrow X'$ et $C \rightarrow C'$.

3.1 Ensembles de conflits

Nous présentons un premier mécanisme d'inférence de supports basé sur la cardinalité des ensembles de conflits.

Proposition 1 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $a \in dom(X)$ tels que $X \rightarrow X'$ et $C \rightarrow C'$.*
Si $a \in dom(X')$ et $|confs(C)_{X=a}| < |dom(C' \backslash X')|$ alors $supps(C')_{X'=a} \neq \emptyset$.

Pour illustrer cette proposition, considérons une contrainte binaire C telle que $vars(C) = \{X, Y\}$. Soit $C \rightarrow C'$ avec $vars(C') = \{X', Y'\}$. Pour toute valeur $a \in dom(X')$ telle que le nombre de conflits de C restreint à $X = a$ est strictement inférieur au cardinal de $dom(Y')$, il est garanti qu'il existe un support de (X', a) pour C'. L'exploitation de cette propriété peut permettre d'éviter un nombre important de tests de consistance. Nous en donnons une illustration sur la figure 1.

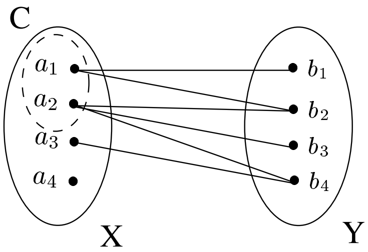


FIG. 2 – Exemple de couverture

3.2 Couvertures

Nous présentons un second mécanisme d'inférence de supports basé sur la notion de couverture, c'est à dire de sous-ensemble de valeurs du domaine d'une variable garantissant la consistance d'arc des autres variables impliquées dans une contrainte donnée. Considérons à nouveau une contrainte binaire C telle que $vars(C) = \{X, Y\}$ et un sous-ensemble $S \subseteq dom(X)$. Si, pour toute valeur b de $dom(Y)$, il existe une valeur a de S telle que (a, b) est un support pour C, alors S est appelé couverture de X pour C. Nous illustrons notre propos sur la figure 2 où $\{a_1, a_2\}$ représente une couverture de X pour C. Notons que sur cette figure, il n'existe pas de couverture de Y pour C.

Définition 6 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $S \subseteq dom(X)$. S est appelé couverture de X pour C ssi $\forall u \in dom(C \backslash X)$, $\exists t \in rel(C) \mid t[\overline{X}] = u \wedge t[X] \in S$.*

Le lemme 1 montre que l'existence d'une couverture de X pour C garantit l'existence d'un support pour C de chaque couple (Y,a) avec $Y \neq X$ pourvu que $dom(C)$ soit non vide.

Lemme 1 *Soient $C \in \mathscr{C} \mid dom(C) \neq \emptyset$ et $X \in vars(C)$. Si il existe une couverture de X pour C alors $\forall Y \in vars(C) | Y \neq X$, $\forall a \in dom(Y)$, $supps(C)_{Y=a} \neq \emptyset$.*

Le lemme 2 montre que l'opérateur $\rightarrow$ préserve les couvertures.

Lemme 2 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $S \subseteq dom(X)$ tels que $X \rightarrow X'$ et $C \rightarrow C'$. Si $S \subseteq dom(X')$ est une couverture de X pour C, alors S est une couverture de X' pour C'.*

La proposition 2 est une conséquence directe des lemmes 1 et 2. Elle indique qu'une couverture identifiée dans le contexte d'un réseau de contraintes P est valide pour tout réseau P' tel que $P \rightarrow P'$ tant qu'aucun élément de la couverture n'est supprimée. Cette propriété peut être exploitée afin d'éviter certains tests de consistance lorsqu'on applique un algorithme de propagation à P'. Toutefois, en pratique, il est préférable de se limiter à des couvertures de taille réduite car le coût de leur exploitation est fonction de leur taille.

Proposition 2 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $S \subseteq dom(X)$ tels que S est une couverture de X pour C, $X \rightarrow X'$ et $C \rightarrow C'$. Si $S \subseteq dom(X')$ et $dom(C') \neq \emptyset$ alors $\forall Y' \in vars(C') | Y' \neq X'$, $\forall a \in dom(Y')$, $supps(C')_{Y'=a} \neq \emptyset$.*

3.3 Substituabilité

La substituabilité, ainsi que la substituabilité au voisinage, ont été introduites par Freuder [10] et sont définies de manière globale, i.e. par rapport à l'ensemble des contraintes d'un réseau. Dans ce papier, nous nous intéressons à une forme restreinte de substituabilité définie, de manière locale, au niveau de chaque contrainte d'un réseau. Cette restriction permet de déterminer en pratique un nombre de valeurs substituables beaucoup plus grand que celui envisageable au niveau global. Nous en donnons ci-dessous une définition formelle, ainsi qu'une illustration dans le contexte binaire (voir figure 3).

Définition 7 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $\{a, b\} \subseteq dom(X)$. a est substituable à b par rapport à X pour C, noté $a \succeq_{C,X} b$, ssi $supps(C)_{X=a} \supseteq supps(C)_{X=b}$.*

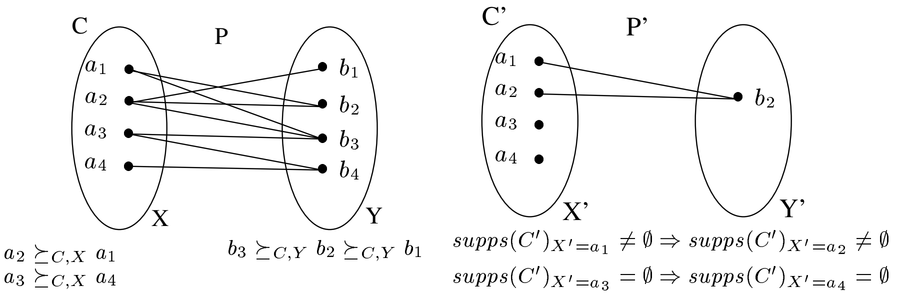
Notons que $\succeq_{C,X}$ est un pré-ordre sur $dom(X)$, i.e. $\succeq_{C,X}$ est une relation réflexive et transitive. Les propositions qui suivent établissent que si $a \succeq_{C,X} b$, alors la présence d'un support de (X,b) pour C garantit la présence d'un support de (X,a) pour C, tandis que l'absence de support de (X,a) pour C garantit l'absence de support de (X,b) pour C.

Proposition 3 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $\{a, b\} \subseteq dom(X)$ tels que $a \succeq_{C,X} b$.*
- *Si $supps(C)_{X=b} \neq \emptyset$ alors $supps(C)_{X=a} \neq \emptyset$.*
- *Si $supps(C)_{X=a} = \emptyset$ alors $supps(C)_{X=b} = \emptyset$.*

Pour finir, nous pouvons observer que si deux valeurs sont liées par la relation $\succeq_{C,X}$ alors elles restent liées par $\succeq_{C',X'}$ lorsque $X \rightarrow X'$ et $C \rightarrow C'$. Autrement dit, les opérations de filtrage par réduction de domaine préservent cette relation.

Proposition 4 *Soient $C \in \mathscr{C}$, $X \in vars(C)$ et $\{a, b\} \subseteq dom(X)$ tels que $X \rightarrow X'$ et $C \rightarrow C'$. Si $\{a, b\} \subseteq dom(X')$ et $a \succeq_{C,X} b$ alors $a \succeq_{C',X'} b$.*

Pour illustrer la manière dont les propositions 3 et 4 peuvent être utilisées afin de réduire le nombre de tests de consistance, considérons à nouveau l'illustration donnée par la figure 3. Supposons que P' ait été obtenu à partir de P en assignant Y à la valeur b_2. Une révision du domaine de X' doit alors être effectuée afin d'éliminer les valeurs inconsistantes. En premier lieu, un support de la valeur a_1 est recherché. La valeur b_2 est trouvée. On peut alors inférer que a_2 possède également un support (puisque



(a) avant assignation

(b) après assignation

FIG. 3 – Inférence de supports par utilisation de valeurs substituables

$a_2 \succeq_{C,X} a_1$). En second lieu, un support de la valeur a_3 est recherché. On constate que a_3 n'a plus de support pour C, ce qui permet d'inférer qu'il en est de même pour a_4 (puisque $a_3 \succeq_{C,X} a_4$).

4 Inférence de supports pour les algorithmes de consistance d'arc

Dans cette section, nous montrons comment il est possible d'exploiter les différentes propriétés identifiées précédemment lorsqu'on utilise un algorithme qui établit la consistance d'arc. Plus précisément, nous avons choisi d'illustrer notre approche avec l'algorithme AC3 [13]. AC3 est en effet assez simple à présenter, et l'ensemble des extensions proposées ici peut y être directement intégré, tout comme aux différents algorithmes à gros grain qui en sont dérivés (tels que AC2001/3.1, AC3.2 et AC3.3).

4.1 Procédure principale des algorithmes de type AC3^{RX}

Nous présentons en premier lieu la procédure principale appelée AC3^{RX}. Nous pouvons obtenir différentes variantes en fonction de l'implantation des deux procédures auxiliaires $init^{RX}$ et $revise^{RX}$. Pour des raisons de simplicité, nous présentons ici une version orientée arc de l'algorithme de propagation, bien que nos expérimentations aient été réalisées avec une version orientée variable.

Décrivons brièvement la procédure principale de l'algorithme 1. Notons tout d'abord qu'un ensemble de propagation, noté Q, est introduit pour mémoriser l'ensemble des arcs qui doivent être révisés. Le but de la révision d'un arc (C, X) est de supprimer du domaine de X les valeurs qui sont inconsistantes par rapport à C. Initialement, tous les arcs (C, X) sont placés dans l'ensemble Q, et un appel à $init^{RX}$ permet d'initialiser les structures de données spécifiques à chaque variante d'AC3. Les arcs sont alors révisés un à un par la procédure $revise^{RX}$, et lorsqu'une révision est effective, i.e. une valeur au moins a été enlevée du domaine de X, l'ensemble Q est mis à jour. Une révision consiste à retirer du domaine de X l'ensemble des valeurs qui sont devenues inconsistantes par rapport à C.

Algorithm 1 AC3$^{RX}(P = (\mathscr{X}, \mathscr{C})$: CSP)

init^{RX}()
$Q \leftarrow \{(C, X) \mid C \in \mathscr{C} \wedge X \in vars(C)\}$
while $Q \neq \emptyset$ **do**
  choisir et éliminer (C, X) de Q
  if revise$^{RX}(C,X)$ **then**
    if $dom(X) = \emptyset$ **then**
      retourner FAILURE
    else
      $Q \leftarrow Q \cup \{(C', X') \mid X \in vars(C') \wedge$
        $X' \in vars(C') \wedge X \neq X' \wedge C \neq C'\}$
    end if
  end if
end while
retourner SUCCESS

4.2 AC3^{R0} (AC3)

Au niveau de notre notation, AC3 correspond à AC3^{R0}. Pour l'algorithme de base AC3, aucune structure de données spécifique n'étant nécessaire, $init^{R0}$ ne comporte aucune instruction. La révision (c.f. algorithme 2) nécessite de rechercher un support pour chaque valeur du domaine de X. Une valeur est éliminée de ce domaine lorsqu'aucun support ne peut être trouvé.

4.3 AC3^{R1}

AC3^{R1} est la première variante d'AC3. Elle met en oeuvre l'inférence de supports en utilisant la cardinalité des ensembles de conflits, ainsi que les couvertures (voir

Algorithm 2 revise^{R0}(C, X) : boolean

nbValues $\leftarrow \mid dom(X) \mid$
for chaque valeur $a \in dom(X)$ **do**
  if $seekSupport(C, X, a) = false$ **then**
    éliminer a de $dom(X)$
  end if
end for
retourner nbValues $\neq \mid dom(X) \mid$

propositions 1 et 2). Pour cela, on comptabilise le nombre de conflits de chaque triplet (C,X,a) au cours de la phase d'initialisation, ce qui nécessite le parcours du domaine de la contrainte C de manière à en identifier les conflits. Trois structures de contrôle spécifiques sont donc introduites :

- $nbConfs$ est un tableau à trois dimensions qui permet de comptabiliser le nombre d'éléments appartenant à $confs(C)_{X=a}$ pour chaque triplet (C,X,a),
- $nbMaxConfs$ est un tableau à deux dimensions qui indique, pour chaque couple (C,X), le nombre maximum de conflits identifiés par rapport à la variable X,
- $coverings$ est un tableau à deux dimensions qui identifie pour chaque couple (C,X) un ensemble de couvertures de X pour C.

Dans la procédure $init^{R1}$ (c.f. algorithme 3), on initialise les compteurs de conflits et on mémorise les couvertures de taille 1 (i.e. dont la cardinalité est égale à 1). Remarquons à ce propos qu'il est assez facile d'identifier les couvertures de taille 1, dans la mesure où elles correspondent aux valeurs sans conflit.

Dans la procédure $revise^{R1}$ (c.f. algorithme 5), on évite un traitement inutile en vérifiant, pour chaque valeur, si il est possible d'inférer la présence d'un support soit en utilisant le nombre maximum de conflits, soit en vérifiant la présence d'une couverture (lignes 1 à 5). Pour chaque valeur, on peut également éviter les tests de consistance inutiles en testant le nombre de conflits de la valeur (ligne 8). Notons que la présence d'une couverture est vérifiée par la procédure $isCovered$ (c.f. algorithme 4), et qu'en pratique, nous avons choisi de nous limiter aux couvertures de cardinalité 1.

Algorithm 3 init^{R1}()

$\forall C \in \mathscr{C}, \forall X \in vars(C), \forall a \in dom(X)$
  $nbConfs[C, X, a] \leftarrow 0$
$\forall C \in \mathscr{C}, \forall t \in dom(C) - rel(C), \forall X \in vars(C)$
  $nbConfs[C, X, t[X]]$++
$\forall C \in \mathscr{C}, \forall X \in vars(C)$
  $nbMaxConfs[C, X] \leftarrow max\{nbConfs[C, X, a] \mid a \in dom(X)\}$
  $coverings[C, X] \leftarrow \{\{a\} \mid a \in dom(X) \wedge nbConfs[C, X, a] = 0\}$

Algorithm 4 isCovered(C, X) : boolean

for chaque variable $Y \in vars(C)$ telle que $Y \neq X$ **do**
  for chaque ensemble $S \in coverings[C, Y]$ **do**
    if $S \subset dom(Y)$ **then**
      retourner true
    end if
  end for
end for
retourner false

Algorithm 5 revise^{R1}(C, X) : boolean

1: **if** nbMaxConfs[C,X] $< |dom(C \backslash X)|$ **then**
2:   retourner false
3: **else if** $isCovered(C, X)$ **then**
4:   retourner false
5: **end if**
6: nbValues $\leftarrow \mid dom(X) \mid$
7: **for** chaque valeur $a \in dom(X)$ **do**
8:   **if** nbConfs[C,X,a] $\geq |dom(C \backslash X)|$ **then**
9:     **if** $seekSupport(C, X, a) = false$ **then**
10:       éliminer a de $dom(X)$
11:     **end if**
12:   **end if**
13: **end for**
14: retourner nbValues $\neq \mid dom(X) \mid$

Pour une contrainte d'arité k, la procédure $init^{R1}$ est de complexité temporelle $O(md^k)$ où m représente le nombre de contraintes, et d la taille du plus grand domaine. En pratique, comme nous limitons arbitrairement le nombre de couvertures à identifier pour chaque couple (C,X), la complexité temporelle de la procédure $isCovered$ est en $O(kd)$. Par conséquent, pour les réseaux de contraintes binaires, la complexité dans le pire des cas des algorithmes de consistance d'arc qui utilisent la variante $R1$ reste inchangée : $O(md^3)$ pour AC3, et $O(md^2)$ pour AC2001/3.1, AC3.2 et AC3.3. La complexité spatiale de AC3^{R1} est en $O(md)$ lorsqu'on borne arbitrairement l'arité des contraintes.

4.4 AC3^{R2}

Nous proposons une deuxième variante de AC3 que nous appelons AC3^{R2}. Cette variante consiste à exploiter non seulement les ensembles de conflits et les couvertures, mais également la propriété de substituabilité. Pour tous les triplets (C,X,a), il est nécessaire de déterminer l'ensemble des valeurs b de dom(X) telles que a soit substituable à b par rapport à X pour C.

Trois nouvelles structures de données sont introduites :

- $lastInference$ est un tableau à trois dimensions qui permet d'indiquer, pour chaque triplet (C,X,a), le nu-

méro de la dernière révision qui a permis de réaliser une inférence,
- $next$ est un tableau à trois dimensions qui permet d'indiquer, pour chaque triplet (C,X,a), la liste des valeurs b telles que a soit substituable à b par rapport à X pour C,
- $prev$ est un tableau à trois dimensions qui permet d'indiquer, pour chaque triplet (C,X,a), la liste des valeurs b telles que b soit substituable à a par rapport à X pour C.

Algorithm 6 init^{R2}()

```
init^{R1}()
∀C ∈ 𝒞, ∀X ∈ vars(C), ∀a ∈ dom(X)
   lastInference[C, X, a] ← 0
   next[C, X, a] ← ∅
   prev[C, X, a] ← ∅
∀C ∈ 𝒞, ∀X ∈ vars(C), ∀a, b ∈ dom(X)
   a ⪰_{C,X} b ⇒ next[C, X, a] ← next[C, X, a] ∪ {b}
   b ⪰_{C,X} a ⇒ prev[C, X, a] ← prev[C, X, a] ∪ {b}
```

Algorithm 7 revise^{R2}(C, X) : boolean

```
1: if nbMaxConfs[C,X] < |dom(C\X)| then
2:       retourner false
3: else if isCovered(C, X) then
4:       retourner false
5: end if
6: nbRevisions ++
7: nbValues ← | dom(X) |
8: for chaque valeur a ∈ dom(X) do
9:    if nbConfs[C,X,a] ≥ |dom(C\X)| then
10:      if lastInference[C,X,a] = nbRevisions then
11:         continue
12:      else if lastInference[C,X,a] = -nbRevisions then
13:         éliminer a de dom(X)
14:      else if seekSupport(C, X, a) = false then
15:         éliminer a de dom(X)
16:         ∀b ∈ prev[C,X,a],
17:            lastInference[C,X,b] ← - nbRevisions
18:      else
19:         ∀b ∈ next[C,X,a],
20:            lastInference[C,X,b] ← nbRevisions
21:      end if
22:   end if
23: end for
24: retourner nbValues ≠ |dom(X)|
```

Dans la procédure $init^{R2}$ (c.f. algorithme 6), après avoir exécuté la procédure $init^{R1}$, les structures de données sont initialisées. Il est important de remarquer que lorsqu'on initialise $next[C, X, a]$, pour toute valeur b de dom(X) telle que $nbConfs[C, X, a] > nbConfs[C, X, b]$, on peut garantir que a n'est pas substituable à b par rapport à X pour C. Un raisonnement similaire peut être tenu vis à vis du tableau $prev$. Pour des raisons de simplicité, cette remarque n'est pas prise en compte dans la description de l'algorithme qui est donnée.

On peut également observer que la propriété de substituabilité est exploitée par la procédure $revise^{R2}$ (c.f. algorithme 7). En effet, à chaque fois qu'un support est trouvé pour une valeur a, une mise à jour de $lastInference[C, X, b]$ pour l'ensemble des valeurs b appartenant à $next[C, X, a]$ est effectuée (lignes 19 et 20). Cette information peut être utilisée afin d'inférer qu'aucune recherche n'est nécessaire (lignes 10 et 11). Par ailleurs, lorsqu'une valeur a ne possède aucun support, une mise à jour de $lastInference[C, X, b]$ pour l'ensemble des valeurs b appartenant à $prev[C, X, a]$ est effectuée (lignes 16 et 17). Cette information peut également être utilisée pour inférer qu'aucune recherche n'est nécessaire (lignes 12 et 13). Remarquons qu'une valeur négative indique l'absence de support alors qu'une valeur positive en indique la présence.

Pour les contraintes binaires, la procédure $init^{R2}$ admet une complexité temporelle en $O(md^3)$ dans le pire des cas dans la mesure où il est nécessaire de déterminer les couples de valeurs substituables par rapport à chaque variable et pour chaque contrainte. La complexité temporelle pour les algorithmes de consistance d'arc qui utilisent la variante $R2$ est en $O(md^3)$. La complexité spatiale d'AC3^{R2} est en $O(md)$ si on limite arbitrairement l'arité des contraintes.

4.5 MAC3^{RX}

Les deux variantes d'AC3 que nous avons introduites ici ne se révèlent pas particulièrement intéressantes lorsqu'on se limite simplement à établir la consistance d'arc. En effet, la phase d'initialisation est lourde, ce qui pénalise fortement l'algorithme si aucune recherche de solution n'est réalisée. Clairement, les deux variantes que nous proposons ne présentent d'intérêt que si elles sont utilisées par un algorithme tel que MAC. L'initialisation des nouvelles structures de données avant ou après la phase de pré-traitement, et l'exploitation de celles-ci au cours de la recherche nous paraît être un bon compromis entre le coût élevé de la récolte des informations, et le bénéfice que l'on peut en retirer. Il est important de noter que les structures de données que nous utilisons dans le cadre des variantes $R1$ et $R2$ ne sont jamais mises à jour au cours de la recherche.

5 Expérimentations

Pour montrer l'intérêt pratique des propriétés que nous avons présentées dans ce papier, nous avons implanté les différents algorithmes décrits précédemment. Nous avons

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
$scen11$-$f10$	$R0$	3.39	4,588K	3.37	2,305K	3.13	1,310K
$(unsat)$	$R1$	3.22	3,685K	3.27	2,901K	3.26	2,745K
	$R2$	4.40	3,505K	4.12	2,789K	4.19	2,663K
$scen11$-$f8$	$R0$	55.32	89,774K	53.21	38,482K	49.25	20,766K
$(unsat)$	$R1$	44.65	29,828K	44.66	14,823K	44.77	13,574K
	$R2$	46.21	24,323K	46.67	11,409K	46.37	10,775K
$scen11$-$f5$	$R0$	4,592.87	5,738,844K	4,507.22	2,223,098K	4,162.32	1,152,517K
$(unsat)$	$R1$	4,033.44	2,285,528K	4,051.57	972,496K	3,991.33	751,010K
	$R2$	4,062.98	1,764,900K	4,119.59	657,428K	4,086.84	539,834K

TAB. 1 – Instances RLFAP

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
$<RG = 0.1, BK = 1>$	$R0$	251.64	1,449,387K	192.22	611,045K	98.37	242,679K
(sat)	$R1$	123.26	600,751K	91.38	266,557K	80.81	215,979K
	$R2$	100.55	375,148K	80.59	173,698K	75.69	152,896K
$<RG = 0.1, BK = 2>$	$R0$	374.49	2,281,435K	169.28	522,253K	116.64	291,574K
(sat)	$R1$	262.59	1,448,883K	145.67	427,742K	113.24	278,557K
	$R2$	189.27	778,953K	147.63	257,601K	116.98	172,229K
$<RG = 0.0, BK = 1>$	$R0$	497.57	2,883,037K	350.90	1,158,383K	175.21	425,669K
(sat)	$R1$	228.72	1,042,013K	176.96	493,222K	146.39	383,073K
	$R2$	179.69	695,099K	147.10	335,314K	131.83	277,671K
$<RG = 0.0, BK = 2>$	$R0$	418.46	2,454,923K	222.26	727,814K	154.72	429,273K
(sat)	$R1$	298.89	1,563,852K	194.98	588,483K	152.37	398,453K
	$R2$	226.73	880,159K	188.48	341,490K	159.91	241,383K

TAB. 2 – Instances job-shop

mené différentes expérimentations par rapport à de nombreuses classes d'instances issues de problèmes réels, académiques et aléatoires sur un PC Pentium IV 2,4GHz 512Mo sous Linux. Les performances ont été mesurées en termes de nombre de tests de consistance (#ccks) et de temps CPU en secondes (cpu). Pour les expérimentations menées sur 100 instances d'un même problème, les critères de performance sont donnés en moyenne.

Nous avons utilisé un algorithme qui maintient la consistance d'arc[1] en utilisant plus particulièrement les algorithmes à gros grain AC3 [13], AC2001/3.1 [6, 18] et AC3.2 [12] dans la mesure où ils présentent un bon compromis entre simplicité et performance. Nous avons d'autre part utilisé l'heuristique de choix de variable $dom/wdeg$ [7] qui peut être actuellement considérée comme la plus efficace.

Nous avons généré des instances difficiles à partir des instances binaires des problèmes réels issus de l'archive fullRLFAP (Radio Link Frequency Assignment Problem) en supprimant certaines fréquences. Par exemple, $scen11$-$f5$ correspond à l'instance réelle $scen11$ pour laquelle on a retiré les cinq plus hautes fréquences. La table 1 fournit les résultats obtenus sur quelques instances représentatives. On peut remarquer que sur une instance facile telle que $scen11$-$f10$, les variantes $R1$ et $R2$ n'améliorent pas les performances de l'algorithme MAC. Par contre l'intérêt de ces variantes apparaît clairement dès que l'on aborde des instances plus difficiles. Pour les instances $scen11$-$f8$ et $scen11$-$f5$, les variantes $R1$ et $R2$ permettent de diviser par deux, voire trois, le nombre de tests de consistance par rapport à la variante $R0$, ce qui engendre un gain de temps de l'ordre de 10%.

Nous avons ensuite mené une série d'expérimentations sur différentes instances de problèmes d'ordonnancement. Nous en avons sélectionné quatre, représentatives des 60 instances *job-shop* proposées par [16]. Pour chaque instance, un premier paramètre RG permet de contrôler la date de début au plus tôt et la date de fin au plus tard de chaque opération tandis qu'un second paramètre BK détermine le nombre de goulots d'étranglement. On observe sur la table 2 un gain significatif par rapport

- à AC3 lorsqu'on exploite les propriétés d'unidirectionnalité (AC3.1/AC2001) et de bidirectionnalité partielle (AC3.2),
- à $R0$ lorsqu'on exploite la cardinalité les ensembles de conflits et les couvertures ($R1$) ainsi que la substituabilité ($R2$).

On peut constater que sur cette classe de problèmes, plus l'algorithme exploite d'information, plus il est efficace.

Nous présentons ensuite les expérimentations réalisées sur différentes instances de problèmes académiques :

- deux instances du problème de coloration d'échiquier [3], notées *cc*-7-2 et *cc*-7-3 (arité 4),
- deux instances du *Golomb ruler*[2], notée *gr*-44-9 et *gr*-44-10, (arités 2 et 3),

[1]Dans le cadre des instances non binaires, l'algorithme qui maintient la consistance d'arc, parfois dénommé MGAC, sera également noté MAC.

[2]voir problem006 at `http://4c.ucc.ie/~tw/csplib/`

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
cc-7-2	$R0$	1.32	$1,929$K	1.71	$1,685$K	1.74	712K
$(unsat)$	$R1$	1.14	204K	1.18	204K	1.18	126K
cc-7-3	$R0$	20.14	$36,710$K	27.86	$29,791$K	27.77	$15,521$K
(sat)	$R1$	14.92	$8,070$K	19.87	$8,070$K	21.27	$5,621$K
gr-44-9	$R0$	49.58	$216,362$K	52.81	$98,226$K	49.11	$64,140$K
(sat)	$R1$	40.65	$148,520$K	32.54	$57,421$K	33.42	$49,143$K
gr-44-10	$R0$	200.45	$864,112$K	185.79	$366,066$K	143.83	$185,511$K
$(unsat)$	$R1$	157.16	$594,685$K	97.93	$204,866$K	86.81	$141,990$K
$K_5 \otimes Q_{20}$	$R0$	658.43	$2,046,863$K	61.29	$134,613$K	52.94	$104,268$K
$(unsat)$	$R1$	638.97	$1,980,609$K	48.38	$105,710$K	43.56	$89,940$K
$K_5 \otimes Q_{25}$	$R0$	$2,197.92$	$6,706,112$K	132.21	$267,668$K	124.47	$213,684$K
$(unsat)$	$R1$	$2,138.12$	$6,568,260$K	104.96	$209,136$K	108.89	$188,086$K

TAB. 3 – Instances académiques

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
$< 30, 10, 1330, 5 >$	$R0$	39.52	$58,253$K	46.47	$32,047$K	42.98	$16,611$K
$(32/100\ sat)$	$R1$	26.40	$2,723$K	28.93	$2,636$K	27.99	$1,738$K
	$R2$	27.42	$2,409$K	29.52	$2,331$K	28.54	$1,512$K
$< 30, 14, 725, 20 >$	$R0$	93.20	$150,315$K	105.16	$80,536$K	92.25	$42,699$K
$(49/100\ sat)$	$R1$	57.17	$24,380$K	71.56	$20,916$K	68.14	$15,228$K
	$R2$	66.10	$22,357$K	79.20	$18,951$K	77.69	$13,597$K
$< 50, 10, 494, 20 >$	$R0$	208.39	$348,812$K	240.57	$196,167$K	198.85	$97,260$K
$(70/100\ sat)$	$R1$	157.22	$109,300$K	197.82	$85,754$K	180.89	$55,579$K
	$R2$	186.94	$97,946$K	222.95	$75,997$K	209.97	$48,575$K
$< 40, 20, 200, 175 >$	$R0$	14.49	$25,037$K	12.32	$12,827$K	9.10	$6,848$K
$(51/100\ sat)$	$R1$	12.28	$16,918$K	11.63	$9,977$K	9.60	$6,243$K
	$R2$	13.73	$16,359$K	12.88	$9,705$K	11.12	$6,123$K
$< 150, 50, 230, 2200 >$	$R0$	26.78	$60,057$K	14.70	$25,687$K	9.95	$13,089$K
$(31/100\ sat)$	$R1$	24.34	$51,818$K	14.62	$24,348$K	9.79	$13,569$K
	$R2$	24.11	$50,183$K	14.84	$23,726$K	10.39	$13,389$K

TAB. 4 – Instances aléatoires

– deux instances du problème des reines et des cavaliers [7], notées $K_5 \otimes Q_{20}$ et $K_5 \otimes Q_{25}$ (arité 2).

Pour les instances non binaires, nous n'avons pas exploité la substituabilité (variante $R2$) car le calcul des valeurs substituables entraînerait un sur-coût trop important. La table 3 montre clairement que, quelque soit l'algorithme de consistance d'arc utilisé, la variante $R1$ reste toujours meilleure que $R0$.

Pour terminer, nous avons étudié le comportement des différentes variantes sur différentes instances de problèmes aléatoires. Nous avons ici considéré cinq classes $<n, d, m, t>$ de 100 instances aléatoires situées à la transition de phase où n représente le nombre de variables, d la taille uniforme des domaines, m le nombre de contraintes binaires et t leur dureté (le nombre de couples de valeurs interdits). La dureté des contraintes des instances étudiées varie de 5% à 88%. On peut constater que plus la dureté des instances est faible, et plus les variantes $R1$ et $R2$ sont efficaces. Par exemple, dans le cadre des instances $<30, 10, 1330, 5>$ (dureté de 5%), on remarque que AC3^{R0} réalise 20 fois plus de tests de consistance que AC3^{R1}. Cela peut s'expliquer par le fait que la cardinalité des ensembles de conflits est très faible, ce qui permet d'éviter des révisions inutiles (c.f. ligne 1 de l'algorithme 5). Ce rapport tombe à 1.2 lorsqu'on considère les instances $<150, 50, 230, 2200>$ (dureté de 88%).

6 Travaux connexes

L'utilisation de méta-connaissances pour réaliser l'inférence de supports a été proposée dans le cadre d'un schéma d'algorithme de consistance d'arc, appelé AC-inférence [5]. Celui-ci permet d'exploiter les propriétés génériques et spécifiques des contraintes de manière à éviter les tests de consistance inutiles. AC7 est un algorithme dérivé d'AC-inférence qui utilise uniquement la bidirectionnalité, une propriété générique appelée multidirectionnalité dans le cas non binaire. Nous avons montré dans ce papier que la substituabilité est également une propriété générique qui permet l'inférence de supports. De ce fait, il paraîtt possible de concevoir, à partir d'AC-inférence, un cadre général d'algorithmes à grain fin pour exploiter tant la bidirectionnalité que la substituabilité. Notons cependant que l'inférence de supports à partir des couvertures et de la cardinalité des ensembles de conflits ne paraît pas adaptée au schéma d'AC-inférence pour les algorithmes à grain fin.

L'interchangeabilité et ses extensions ont été introduites dans [10]. Dans [2] il est montré expérimentalement que, sous certaines conditions, le simple fait d'éliminer les valeurs interchangeables au cours de la phase de pré-traitement s'avère efficace. L'utilisation de la propriété d'interchangeabilité de manière indépendante pour chaque contrainte a été proposée dans [11] dans le but de diminuer le nombre de tests de consistance. L'exploitation de la propriété de substituabilité que nous proposons ici peut être considérée comme un prolongement de ces travaux dans le cadre de la propagation de contraintes. La propriété de substituabilité au voisinage a également été utilisée dans [1] pour réduire la taille d'un problème sans en modifier la satisfiabilité. Elle peut être exploitée en tant qu'opérateur de réduction impliquant une séquence convergente de substitutions au voisinage [8].

7 Conclusion

Dans ce papier, nous avons présenté une exploitation de quelques propriétés générales de contraintes. Nous avons montré comment il est possible d'utiliser ces propriétés pour effectuer une inférence de supports dans le cadre des algorithmes qui établissent la consistance d'arc. Nous avons montré qu'un algorithme MAC peut largement bénéficier de l'utilisation de ces propriétés puisque :

- les deux variantes proposées sont simples à implanter,
- l'espace mémoire requis reste limité ($O(md)$ contre, par exemple, $O(md^2)$ pour AC-inférence),
- les variantes proposées permettent un accroissement réel des performances sur certaines instances.

Nos résultats expérimentaux montrent que, si la consistance d'arc doit être maintenue au cours de la recherche, MAC3^{R1} est un bon choix lorsque la dureté des contraintes est faible, tandis que MAC3.2^{R1} s'avère plus efficace lorsque la dureté des contraintes devient importante. L'intérêt de l'utilisation de la substituabilité (variante $R2$) semble se limiter à quelques problèmes particuliers, tels que les problèmes d'ordonnancement, pour lesquels, étant donnée la nature des contraintes, le nombre de valeurs substituables est important. Enfin, il ressort de cette étude que l'exploitation du cardinal des ensembles de conflits et des couvertures (variante $R1$) améliore systématiquement les performances des algorithmes MAC pour la résolution des instances CSP difficiles.

Remerciements

Ce papier a été soutenu par le programme Cocoa de la Région Nord/Pas-de-Calais et par l'IUT de Lens.

Références

[1] A. Bellicha, C. Capelle, M. Habib, T. Kokény, and M.C. Vilarem. CSP techniques using partial orders on domain values. *Actes de ECAI'94 Workshop on constraint satisfaction issues raised by practical applications*, 1994.

[2] B.W. Benson and E.C. Freuder. Interchangeability preprocessing can improve forward checking search. *Actes de ECAI'92*, pages 28–30, 1992.

[3] M. Beresin, E. Levin, and J. Winn. A chessboard coloring problem. *The College Mathematics Journal*, 20(2) :106–114, 1989.

[4] C. Bessière. Arc consistency and arc consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[5] C. Bessière, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148, 1999.

[6] C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. *Actes de IJCAI'01*, pages 309–315, 2001.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. *Actes de ECAI'04*, pages 146–150, 2004.

[8] M.C. Cooper. Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90 :1–24, 1997.

[9] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[10] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. *Actes de AAAI'91*, pages 227–233, 1991.

[11] A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. *Actes de IJCAI'93*, pages 282–287, 1993.

[12] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. *Actes de CP'03*, pages 480–494, 2003.

[13] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[14] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[15] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. *Actes de PPCPA'94*, 1994.

[16] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86 :1–41, 1996.

[17] M.R.C. van Dongen. AC3$_d$ an efficient arc consistency algorithm with a low space complexity. *Actes de CP'02*, pages 755–760, 2002.

[18] Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. *Actes de IJCAI'01*, pages 316–321, 2001.