Maintien asynchrone de la consistance d'arc dans la recherche distribuée synchrone

Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, El Houssine Bouyakhf

Maintien asynchrone de la consistance d'arc dans la recherche distribuée synchrone

Mohamed Wahbi[1] **Redouane Ezzahir**[2] **Christian Bessiere**[3] **El Houssine Bouyakhf**[4]

[1] TASC INRIA, LINA CNRS, École des Mines de Nantes, France
[2] ENSA Agadir, Université Ibn Zohr, Maroc
[3] LIRMM, Université Montpellier 2, France
[4] LIMIARF, Université Mohammed V–Agdal, Maroc

mohamed.wahbi@emn.fr red.ezzahir@gmail.com bessiere@lirmm.fr bouyakhf@fsr.ac.ma

Abstract

Nous avons proposé récemment Nogood-Based Asynchronous Forward Checking (AFC-ng), un algorithme efficace et robuste pour résoudre des Problèmes de Satisfaction de Contraintes Distribués (DisCSPs). AFC-ng exécute de manière asynchrone une phase de vérification en avant pendant la recherche synchrone. Dans ce papier, nous proposons deux nouveaux algorithmes basés sur le même mécanisme qu'AFC-ng. Cependant, au lieu d'utiliser le *forward checking* comme propriété de filtrage, nous proposons de maintenir d'une manière asynchrone la consistance d'arc (MACA). Le premier algorithme que nous proposons, MACA-del, maintient la consistance d'arc grâce à un type de messages supplémentaire, des messages de suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Nous fournissons une analyse théorique et une évaluation expérimentale de l'approche proposée. Nos expérimentations montrent la bonne performance des algorithmes MACA et plus particulièrement ceux de MACA-not.

1 Introduction

Les problèmes de satisfaction de contraintes (CSP) permettent de formaliser de nombreux problèmes réels de forte combinatoire tels que l'allocation de ressources, le car sequencing, etc. Un problème de satisfaction de contraintes consiste en une recherche de solutions pour un réseau de contraintes, c'est à dire, trouver un ensemble d'affectations de valeurs aux variables qui satisfont les contraintes du problème. Ces contraintes spécifient des combinaisons de valeurs admissibles.

Beaucoup d'algorithmes de recherche basés sur le backtrack ont été développés pour résoudre les problèmes de satisfaction de contraintes. Certains parmi ces algorithmes de recherche essayent de construire une solution d'un CSP en alternant l'affectation de variables avec la propagation de contraintes. Forward Checking (FC) [9] et le maintien de la consistance d'arc (Maintaining Arc Consistency (MAC)) [17] sont des exemples de tels algorithmes. Dans les années 80, FC était considéré comme l'algorithme de recherche le plus efficace. Dans le milieu des années 90, plusieurs travaux ont montré de manière empirique que MAC est plus efficace que FC sur les problèmes durs et de grande taille [3, 8].

Les réseaux de capteurs [11, 1] ou les problèmes distribués d'allocation de ressources [15, 16] comme la planification de réunion distribuée [22, 13] sont des applications réelles de nature distribuée. La connaissance est distribuée parmi plusieurs entités. Ces applications peuvent être naturellement formalisées et résolues par un processus de CSP une fois que la connaissance et la description du problème sont livrées à un solveur centralisé. Cependant, dans de telles applications, assembler toute la connaissance dans un solveur centralisé n'est pas indiqué. En général, cette restriction est principalement due à des raisons de confidentialité et/ou de sécurité. Le coût ou l'incapacité de traduire toutes les informations en un format unique peuvent en être une autre raison. Ainsi, un modèle distribué permettant un processus de résolution décentralisé est plus indiqué. Les *problèmes de satisfaction de contraintes*

distribués (DisCSP) ont justement de telles propriétés.

Un DisCSP est composé d'un groupe d'agents autonomes, où chaque agent a le contrôle de quelques éléments d'information sur le problème. Chaque agent possède son réseau de contraintes local. Les variables dans les différents agents sont connectées par des contraintes. Les agents doivent attribuer des valeurs à leurs variables pour que toutes les contraintes soient satisfaites. Ainsi, les agents essayent de générer des affectations localement consistantes (affectation de valeurs à leurs variables), qui sont aussi consistantes avec les contraintes entre agents [24, 23].

Durant les deux dernières décennies, de nombreux algorithmes distribués de résolution de DisCSPs ont été proposés. Synchronous Backtrack (SBT) est l'algorithme de recherche de DisCSP le plus simple. SBT effectue des affectations séquentiellement et de manière synchrone. Dans SBT, seul l'agent disposant de l'affectation partielle courante (CPA pour Current Partial Assignment) effectue une affectation ou un backtrack [25]. Meisels et Zivan [14] ont étendu SBT à l'algorithme Asynchronous Forward Checking (AFC), un algorithme dans lequel l'algorithme FC [9] est exécuté d'une manière asynchrone [14]. Dans AFC, afin d'exécuter FC d'une manière asynchrone, à chaque fois qu'un agent réussit l'extension de l'affectation partielle courante, il envoie la CPA à son successeur et des copies de cette CPA aux autres agents non assignés.

Dans un travail récent [21], nous avons proposé l'algorithme Nogood-Based Asynchronous Forward Checking (AFC-ng), qui est une amélioration d'AFC. Contrairement à AFC, AFC-ng utilise les nogoods comme justification des valeurs supprimées et permet plusieurs backtrack simultanés provenant de différents agents et allant vers différentes destinations. On a montré que AFC-ng est plus performant que AFC.

Bien que de nombreuses études aient incorporé FC avec succès dans les CSPs distribués [5, 14, 7, 21], l'algorithme MAC n'a pas encore été bien examiné. Les seules tentatives pour inclure le maintien de consistance d'arc dans des algorithmes distribués ont été faites sur l'algorithme Asynchronous Backtracking (ABT). Silaghi *et al.* [19] ont introduit Distributed Maintaining Asynchronously Consistency for ABT (*DMAC-ABT*), le premier algorithme capable de maintenir la consistance d'arc dans les CSP distribué [19]. DMAC-ABT considère le maintien de la consistance comme une inférence hiérarchique basée sur les nogoods. Brito et Meseguer [6] ont proposé ABT-uac et ABT-dac, deux algorithmes qui connectent ABT avec la consistance d'arc [6]. ABT-uac propage inconditionnellement des valeurs supprimées pour faire respecter un certain nombre de consistances d'arc complètes. ABT-dac propage conditionnellement et inconditionnellement des valeurs supprimées en utilisant la consistance d'arc directionnelle. ABT-uac montre une amélioration mineure en termes d'envoi de messages. Quant à ABT-dac, il dégrade les performances dans bien des cas.

Dans ce travail, nous proposons deux nouveaux algorithmes de recherche synchrone basés sur le même mécanisme que AFC-ng. Cependant, au lieu de maintenir "forward checking" d'une manière asynchrone sur des agents non encore instanciés, nous proposons de maintenir la consistance d'arc d'une manière asynchrone sur ces agents futurs. Nous appelons ce nouveau schéma MACA, pour maintien de consistance d'arc de manière asynchrone (Maintaining Arc Consistency Asynhronously). Comme dans AFC-ng, seul l'agent disposant de l'affectation partielle courante (CPA) peut accomplir une affectation. Cependant, contrairement à AFC-ng, MACA essaie de maintenir la consistance d'arc au lieu de se contenter uniquement d'exécuter FC. Le premier algorithme que nous proposons, MACA-del, fait respecter la consistance d'arc grâce à un type supplémentaire de messages, des messages de suppression (***del***). Ainsi, à chaque fois que des valeurs sont supprimées pendant l'étape de propagation de contraintes, les agents de MACA-del notifient les autres agents qui peuvent être affectés par ces suppressions en leurs envoyant un message ***del***. Les messages ***del*** contiennent toutes les valeurs supprimées ainsi que le nogood justifiant leur suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Nous réalisons ceci en stockant toutes les suppressions exécutées par un agent sur les domaines de ses agents voisins et en envoyant ces informations à ces voisins dans le message CPA.

Ce papier est organisé comme suit. La Section 2 donne le contexte nécessaire. La Sections 3 décrit les algorithmes MACA-del et MACA-not. L'analyse théorique et les preuves de correction sont développées dans la Section 4. La Section 5 présente une évaluation expérimentale de nos algorithmes comparés à d'autres algorithmes. Finalement, nous concluons le papier dans la Section 6.

2 Préliminaires

2.1 Définitions et notations basiques

Le Problème de Satisfaction de Contraintes Distribué (DisCSP pour distributed constraint satisfaction problem) a été formalisé dans [24] comme étant un quadruplet $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{A}$ est un ensemble de a agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ est un ensemble de n variables

$\{x_1, \ldots, x_n\}$, tel que chaque variable x_i est contrôlée par un seul agent dans $\mathcal{A}$. $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$ est un ensemble de domaines, où $D^0(x_i)$ est l'ensemble initial de valeurs qui peuvent être affectées à la variable x_i. Pendant la recherche, les valeurs peuvent être supprimées du domaine. À chaque nœud donné, l'ensemble des valeurs possibles pour la variable x_i est désigné par $D(x_i)$. Ce dernier est appelé le domaine courant de x_i. $\mathcal{C}$ est un ensemble de contraintes qui spécifient les combinaisons de valeurs autorisées pour les variables qu'elles impliquent. Dans ce papier, nous supposons un réseau de contraintes distribué binaire où toutes les contraintes sont binaires (elles impliquent deux variables). Une contrainte $c_{ij} \in \mathcal{C}$ entre deux variables x_i et x_j est un sous-ensemble du produit cartésien de leurs domaines, c.-à-d. $c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$.

Pour des raisons de simplicité, nous considérons une version restreinte de DisCSP où chaque agent contrôle exactement une seule variable. Nous utilisons les termes agent et variable d'une manière interchangeable et nous identifions l'index d'un agent par l'indice de sa variable. En outre, tous les agents mémorisent un ordre unique $\prec$ sur les agents. Les agents qui apparaissent avant A_i dans cet ordre sont les agents les plus prioritaires (prédécesseurs) et inversement, les agents moins prioritaires (successeurs) sont ceux qui apparaissent après A_i. Par souci de clarté, nous supposons que l'ordre total sur les agents $\prec$ est l'ordre lexicographique $[A_1, A_2, \ldots, A_n]$.

Chaque agent maintient son propre compteur et l'incrémente chaque fois qu'il change sa valeur. La valeur courante du compteur sera associée à chaque nouvelle affectation.

Définition 1 *Une* ***affectation*** *pour un agent $A_i \in \mathcal{A}$ est un uplet (x_i, v_i, t_i), où v_i est une valeur du domaine de x_i et t_i la valeur du compteur de A_i. Quand on compare deux affectations, la plus récente est celle associée avec le plus grand compteur t_i.*

Définition 2 ***L'affectation partielle courante*** *CPA (pour Current Partial Assignment) est un ensemble ordonné d'affectations $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. Deux CPA sont* compatibles *si à chaque variable commune à ces deux sous ensembles est affectée la même valeur.*

Définition 3 ***L'horodatage****, associé à un CPA, est une liste ordonnée des compteurs $[t_1, t_2, \ldots, t_i]$ où t_j est le compteur de la variable x_j. Quand on compare deux CPA, la* ***plus forte*** *est celle associée à l'horodatage lexicographiquement supérieur. Autrement dit, celle qui a la plus grande valeur du premier compteur où les deux horodatages diffèrent, le cas échéant, sinon la plus longue.*

Définition 4 *La vue* ***AgentView*** *d'un agent $A_i \in \mathcal{A}$ contient les affectations les plus récentes reçues des agents plus prioritaires. Elle a une forme similaire à celle de la CPA. Elle est initialisée par l'ensemble des affectations vides $\{(x_j, empty, 0) \mid x_j \prec x_i\}$.*

En se basant sur les contraintes du problème, les agents peuvent inférer des ensembles d'affectations inconsistants appelés ***nogoods***.

Définition 5 *Un* ***nogood*** *écartant la valeur v_k du domaine initial de la variable x_k est une clause sous forme $x_i = v_i \wedge \ldots \wedge x_j = v_j \rightarrow x_k \neq v_k$, signifiant que l'affectation $x_k = v_k$ est inconsistante avec les affectation $x_i = v_i, \ldots, x_j = v_j$. La partie gauche (lhs) et la partie droite (rhs) sont définies à partir de la position de $\rightarrow$.*

Le domaine courant $D(x_i)$ de la variable x_i contient toutes les valeurs du domaine initial qui ne sont pas écartées par un nogood. Lorsque toutes les valeurs d'un agent x_k sont supprimées par des nogoods (c.-à-d. $D(x_i) = \emptyset$), cet agent procède à la résolution de ces derniers. Cette résolution produit un nouveau nogood ng . Soit x_j la variable ayant la plus basse priorité dans les parties gauches de tous les nogoods enregistrés par x_k et $x_j = v_j$. $lhs(ng)$ est la conjonction de toutes les parties gauches de tous ces nogoods excepté $x_j = v_j$ et $rhs(ng)$ est $x_j \neq v_j$.

2.2 Le maintien de la consistance d'arc

Il est bien connu que la propagation de contraintes joue un rôle central pour la résolution des problèmes de satisfaction de contraintes [2]. La technique de propagation de contraintes la plus ancienne et la plus utilisée est la consistance d'arc.

Définition 6 *Une contrainte c_{ij} est* ***arc consistante*** *si et seulement si $\forall\ v_i \in D(x_i), \exists\ v_j \in D(x_j)$ tel que (v_i, v_j) est autorisé par c_{ij} et $\forall\ v_j \in D(x_j), \exists\ v_i \in D(x_i)$ tel que (v_i, v_j) est autorisé par c_{ij}. Un réseau de contraintes est arc consistant ssi toutes ses contraintes sont arc consistantes.*

L'algorithme MAC (Maintaining Arc Consistency) [17] alterne des étapes d'exploration avec d'autres de propagation. Autrement dit, à chaque étape de la recherche, une affectation d'une variable est suivie par un processus de filtrage qui correspond à l'application de la consistance d'arc.

Pour la mise en œuvre de MAC dans un CSP distribué, chaque agent A_i est censé connaître toutes les contraintes dans lesquelles il est impliqué et les agents avec qui il partage ces contraintes (ses voisins). Ces derniers et les contraintes qui les relient à A_i forment le réseau de contraintes local de A_i, désigné par $CSP(i)$.

Définition 7 *Un* ***réseau de contraintes local*** $CSP(i)$ *d'un agent* $A_i \in \mathcal{A}$ *est constitué de toutes les contraintes impliquant* x_i *et toutes les autres variables de ces contraintes.*

Pour permettre aux agents de maintenir la consistance d'arc dans les CSP distribués, notre approche consiste à appliquer la consistance d'arc dans le réseau de contraintes local de chaque agent. Chaque agent A_i enregistre localement des copies de toutes les variables de $CSP(i)$. Nous supposons également que chaque agent connaît le voisinage qu'il a en commun avec ses voisins, sans connaître les contraintes qui les relient. Autrement dit, pour chacun de ses voisins A_k, un agent A_i connaît la liste des agents A_j tel qu'il existe une contrainte entre x_i et x_j et une autre liant x_k à x_j.

3 Le maintien de la consistance d'arc de manière asynchrone

Dans l'algorithme nogood-based Asynchronous Forward Checking (AFC-ng), la phase de forward-checking (FC) vise à anticiper le backtrack. Néanmoins, nous ne tirons pas profit des suppressions produites par FC si le domaine de la variable n'est pas entièrement vidé. Ces suppressions peuvent être utilisées en appliquant la consistance d'arc. Ceci est justifié par le fait que la propagation de la suppression d'une valeur, pour un agent A_i, peut générer un domaine vide pour une variable dans son réseau de contraintes local. Ainsi, nous pouvons détecter promptement l'échec et par conséquence anticiper aussitôt que possible l'opération de backtrack.

Dans les algorithmes de recherche synchrones pour résoudre les DisCSPs, les agents affectent séquentiellement leurs variables. Ainsi, les agents assignent des valeurs à leurs variables seulement lorsqu'ils détiennent l'affectation partielle courante, CPA. Nous proposons un algorithme dans lequel les agents affectent leur variables un par un suivant l'ordre total sur les agents. Par conséquent, quand un agent réussit à étendre la CPA par l'affectation de sa variable, il envoie la CPA à son successeur pour qu'il l'étende à son tour. Des copies de cette CPA sont également envoyées aux autres agents qui ne sont pas encore affectés dans la CPA pour maintenir la consistance d'arc de manière asynchrone (MACA pour *maintain arc consistency asynchronously*). Par conséquent, quand un agent reçoit une copie de la CPA, il maintient la consistance d'arc dans son réseau de contraintes local. Pour appliquer la consistance d'arc sur toutes les variables du problème, les agents communiquent aux autres agents des informations sur les suppressions de valeurs produites localement. Nous proposons deux méthodes pour réaliser ceci. La première, à savoir MACA-del, utilise un nouveau type de messages (des messages ***del***) pour propager ces informations. La deuxième méthode, nommée MACA-not, inclut les informations concernant les suppressions générées localement dans les messages ***cpa***.

3.1 Maintenir l'AC à l'aide des messages del

Dans MACA-del, tout agent A_i maintient la consistance d'arc dans son réseau de contraintes local, $CSP(i)$, chaque fois qu'un domaine d'une variable dans $CSP(i)$ est modifié. Les changements peuvent se produire sur le domaine de A_i soit sur un autre domaine dans $CSP(i)$. Dans l'algorithme MACA-del, l'agent A_i partage avec les autres agents seulement les suppressions produites dans son domaine $D(x_i)$. Elles sont propagées à travers la communication aux autres agents des nogoods qui les justifient. Ces suppressions et ces nogoods sont donc envoyés aux voisins par l'intermédiaire de messages ***del***.

L'agent A_i enregistre tous les nogoods associés à ses valeurs. Ces nogoods sont enregistrés dans $NogoodStore[x_i]$. En plus des nogoods stockés pour ses propres valeurs, A_i a besoin de mémoriser les nogoods des valeurs retirées des domaines des autres variables x_j appartenant à $CSP(i)$. Les nogoods justifiant les suppressions des valeurs de $D(x_j)$ sont stockés dans $NogoodStore[x_j]$. Par conséquent, le NogoodStore d'un agent A_i est un vecteur de plusieurs NogoodStores, un pour chaque variable dans $CSP(i)$.

Le pseudo-code de MACA-del exécuté par chaque agent A_i est illustré en Fig. 1. L'agent A_i démarre la recherche en appelant la procédure `MACA-del()`. Dans cette dernière procédure, A_i appelle `Propagate()` pour maintenir la consistance d'arc (ligne 1) dans son réseau de contraintes local, c.-à-d. $CSP(i)$. Ensuite, si A_i est l'agent d'initialisation IA (le premier agent dans l'ordre), il initialise la recherche en appelant la procédure `Assign()` (ligne 2). Puis, une boucle considère la réception et le traitement des différents types de messages.

Lors de l'appel à la procédure `Assign()`, A_i tente de trouver une affectation consistante avec l'AgentView. Si A_i ne parvient pas à trouver une telle affectation, il appelle la procédure `Backtrack()` (ligne 14). Si A_i a réussi à trouver une affectation, il incrémente son compteur t_i et génère une nouvelle CPA de son AgentView augmentée par sa propre affectation (lignes 11 et 12). Ensuite, A_i appelle la procédure `SendCPA(CPA)` (ligne 13). Si le CPA comprend les affectations de tous les agents (A_i est le dernier agent dans l'ordre ligne 15), A_i rapporte la CPA comme solution du problème et marque le drapeau *end* vrai pour arrêter la boucle principale (ligne 16). Sinon, A_i envoie en avant

```
procedure MACA-del()
01.  end ← false;  Propagate() ;
02.  if ( A_i = IA ) then Assign() ;
03.  while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.      cpa  : ProcessCPA(msg);
07.      ngd  : ProcessNogood(msg);
08.      del  : ProcessDel(msg);
09.      stp  : end ← true;

procedure Assign()
10.  if ( D(x_i) ≠ ∅ ) then
11.    v_i ← ChooseValue() ; t_i ← t_i+1 ;
12.    CPA ← {AgentView ∪ (x_i, v_i, t_i)} ;
13.    SendCPA(CPA) ;
14.  else Backtrack() ;

procedure SendCPA(CPA)
15.  if ( size(CPA) = n ) then
16.    broadcastMsg: stp⟨CPA⟩ ;  end ← true ;
17.  else
18.    foreach ( x_k ≻ x_i ) do  sendMsg: cpa⟨CPA⟩ to A_k ;

procedure ProcessCPA(msg)
19.  if ( msg.CPA is stronger than the AgentView ) then
20.    AgentView ← msg.CPA ;
21.    Remove all nogoods incompatible with AgentView ;
22.    if ( ¬Propagate() ) then Backtrack() ;
23.    else if ( msg.sender = A_i−1 ) then Assign() ;

function Propagate()
24.  if ( ¬AC(CSP(i)) ) then return false ;
25.  else if ( D(x_i) was changed ) then
26.    foreach ( x_j ∈ CSP(i) ) do
27.      nogoods ← get nogoods from NogoodStore[x_i]
         that are relevant to x_j ;
28.      sendMsg: del⟨nogoods⟩ to A_j ;
29.  return true ;

procedure ProcessDel(msg)
30.  Let A_k be the agent that sent msg;
31.  foreach ( ng ∈ msg.nogoods ) do
32.    if ( Compatible(ng, AgentView) ) then
33.      add(ng, NogoodStore[x_k]) ;
34.  if ( D(x_k) = ∅ ∧ x_i ∈ NogoodStore[x_k] ) then
35.    newNogood ← solve(NogoodStore[x_k]) ;
36.    add(newNogood, NogoodStore[x_i]) ;
37.    Assign() ;
38.  else if ( D(x_k) = ∅ ∨ ¬Propagate() ) then
39.    Backtrack();

procedure Backtrack()
40.  Let x_k be the variable within the empty domain;
41.  ng ← solve(NogoodStore[x_k]) ;        /* D(x_k) = ∅ */
42.  if ( ng = empty ) then
43.    broadcastMsg: stp⟨∅⟩ ;
44.    end ← true ;
45.  else        /* Let x_j be the variable on the rhs(ng) */
46.    sendMsg: ngd⟨ng⟩ to A_j ;
47.    from ( l = j to i-1 ) do
48.      AgentView[l].value ← empty ;
49.    Remove all nogoods incompatible with AgentView ;

procedure ProcessNogood(msg)
50.  if ( Compatible(lhs(msg.nogood), AgentView) ) then
51.    add(msg.nogood, NogoodStore[x_i]) ;
52.    if ( rhs(msg.nogood).value = v_i ) then Assign() ;
53.    else if ( ¬Propagate() ) then Backtrack() ;
```

FIGURE 1 – L'algorithme MACA-del exécuté par un agent A_i.

la CPA à tous les agents non encore instanciés dans la CPA (ligne 18). Ainsi, le prochain agent dans l'ordre (successeur) tentera d'élargir cette CPA en affectant sa variable dessus pendant que les autres agents maintiennent la consistance d'arc de manière asynchrone.

Quand un agent A_i reçoit un message de type ***cpa***, la procédure `ProcessCPA()` est appelée (ligne 6). Le message reçu sera traité seulement quand il détient une CPA plus forte que l'AgentView de l'agent A_i. Si c'est le cas, A_i met à jour son AgentView (ligne 20). Puis, il met à jour le NogoodStore de chaque variable dans $CSP(i)$ pour être compatible avec sa nouvelle AgentView (ligne 21). Ensuite, A_i appelle la fonction `Propagate()` pour maintenir la consistance d'arc sur $CSP(i)$ (ligne 22). Si l'AC vide un domaine dans $CSP(i)$, A_i appelle la procédure `Backtrack()` (ligne 22). Sinon, A_i vérifie s'il doit affecter sa variable (ligne 23). A_i tente d'affecter sa variable en appelant la procédure `Assign()` seulement s'il a reçu le message ***cpa*** de son prédécesseur.

Lorsque la fonction `Propagate()`, appelant A_i maintient l'AC sur son réseau de contraintes local en considérant les affectations dans son AgentView (ligne 24). Dans notre implémentation, nous avons utilisé *AC-2001* [4] pour maintenir la consistance d'arc, mais tout algorithme générique qui applique l'`AC` peut être utilisé dans MACA. MACA-del a seulement besoin que l'algorithme maintenant l'`AC` mémorise un nogood pour chaque valeur supprimée. Quand deux nogoods sont possibles pour une même valeur, nous choisissons le meilleur en utilisant l'heuristique HPLV (*Highest Possible Lowest Variable*) [10]. Si le maintien de la consistance d'arc sur $CSP(i)$ a échoué, c.-à-d. un domaine a été vidé, la fonction retourne false (ligne 24). Sinon, si le domaine de x_i a été changé (c.-à-d. il existe des suppressions à propager), A_i informe ses voisins en leurs envoyant des messages ***del*** qui contiennent les nogoods justifiant ces suppressions (lignes 27-28). Enfin, la fonction retourne true (ligne 29). Lors de l'envoi d'un message ***del*** à un voisin A_j, les nogoods qui seront communiqués à A_j sont ceux dont toutes les variables dans leurs parties gauches ont une priorité plus élevée que A_j. En outre, tous les nogoods ayant la même partie gauche sont factorisés dans un seul nogood dont la partie droite est l'ensemble de toutes les valeurs supprimées par cette partie gauche.

Quand l'agent A_i reçoit un message de type ***del***, il ajoute au NogoodStore $NogoodStore[x_k]$ de l'expéditeur A_k tous les nogoods compatibles avec l'AgentView de A_i (lignes 31-33). Puis, A_i vérifie si le domaine de x_k est vidé (c.-à-d. les valeurs restantes dans $D(x_k)$ ont été supprimées par les nogoods qui viennent d'être reçus de l'agent A_k) et x_i appartient au Nogood-

Store de x_k (c.-à-d. x_i est déjà affectée et son affectation est incluse dans au moins un nogood qui justifie la suppression d'une valeur de $D(x_k)$) (ligne 34). Si c'est le cas, A_i supprime sa valeur actuelle en mémorisant comme justification de cette suppression le nogood $newNogood$, généré à partir de la résolution du NogoodStore de x_k (ligne 35-36). Ensuite, il appelle la procédure `Assign()` pour tenter d'assigner une autre valeur (ligne 37). Sinon, si $D(x_k)$ est vidé (x_i n'est pas affectée) ou si un échec se produit lors du maintien de la consistance d'arc, A_i doit faire un retour-arrière (`Backtrack()`, ligne 39).

Chaque fois qu'un échec se produit sur le domaine d'une variable x_k dans $CSP(i)$ (comprenant x_i), la procédure `Backtrack()` est appelée. L'agent A_i procède à la résolution de ces nogoods qui ont produit l'échec, et génère un nouveau nogood ng (ligne 41). ng est la conjonction des parties gauches de tous ces nogoods mémorisés par A_i dans $NogoodStore[x_k]$. Si ng est vide, le problème n'a pas de solution. A_i termine son exécution après l'envoi d'un message de type ***stp*** à tous les agents dans le système (lignes 42-44). Sinon, A_i fait un retour-arrière par l'envoi d'un message ***ngd*** à l'agent A_j qui détient la variable dans la partie droite de ng (ligne 46). Ensuite, A_i met à jour son AgentView afin de ne garder que les affectations des agents qui sont placés avant A_j dans l'ordre (lignes 47-48). A_i met également à jour le NogoodStore de toutes les variables dans $CSP(i)$ en enlevant les nogoods incompatibles avec sa nouvelle AgentView (ligne 49).

À la réception d'un message ***ngd***, A_i vérifie la validité du nogood reçu (ligne 50). S'il est compatible avec son AgentView, A_i l'ajoute à son NogoodStore (c.-à-d. $NogoodStore[x_i]$, ligne 51). Puis, A_i vérifie si la valeur dans la partie droite du nogood reçu correspond à sa valeur actuelle. Si c'est le cas, A_i appelle la procédure `Assign()` pour essayer de trouver une autre affectation à sa variable qui soit consistante avec les choix des agents les plus prioritaires (ligne 52). Sinon, A_i appelle la fonction `Propagate()` pour maintenir l'AC. Si un domaine vide apparaît dans son réseau de contraintes local, A_i appelle la procédure `Backtrack()` (ligne 53).

3.2 Maintenir l'AC sans type supplémentaire de message

Dans la suite, nous montrons comment maintenir la consistance d'arc sans utiliser un type supplémentaire de messages. Dans MACA-del, la maintenance de la consistance globale est réalisée en communiquant aux agents voisins (agents dans $CSP(i)$) toutes les valeurs élaguées de $D^0(x_i)$. Cela peut générer de nombreux messages de type ***del*** dans le réseau et donc créer un goulot d'étranglement au niveau de la communication.

```
procedure MACA-not()
01.  end ← false;  Propagate() ;
02.  if ( A_i = IA ) then Assign() ;
03.  while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.      cpa  : ProcessCPA(msg);
07.      ngd  : ProcessNogood(msg);
08.      stp  : end ← true;

procedure SendCPA(CPA)
09.  if ( size(CPA) = n ) then
10.    broadcastMsg : stp⟨CPA⟩ ;
11.    end ← true ;
12.  else
13.    foreach ( x_k ≻ x_i ) do
14.      nogoods ← ∅;
15.      foreach ( x_j ∈ {CSP(i) ∩ CSP(k)} | x_j ≻ x_i ) do
16.        nogoods ← nogoods ∪ getNogoods(x_j) ;
17.      sendMsg : cpa⟨CPA, nogoods⟩ to A_k ;

procedure ProcessCPA(msg)
18.  if ( msg.CPA is stronger than the AgentView ) then
19.    AgentView ← CPA ;
20.    Remove all nogoods incompatible with AgentView ;
21.    foreach ( ng ∈ msg.nogoods ) do
22.      let x_k be the variable in the rhs(ng);
23.      add(ng, NogoodStore[x_k]) ;
24.    if ( ¬Propagate() ) then Backtrack() ;
25.    else if ( msg.sender = A_{i-1} ) then Assign();

function Propagate()
26.  return AC(CSP(i)) ;
```

FIGURE 2 – Nouvelles lignes/procédures de MACA-not par rapport à MACA-del.

En outre, les agents doivent effectuer plus d'efforts pour traiter ces nombreux messages ***del***.

Dans MACA-not, la communication des suppressions produites dans $CSP(i)$ est retardée jusqu'à ce que l'agent A_i veuille envoyer un message ***cpa***. Lors de l'envoi d'un message ***cpa*** à un agent moins prioritaire A_k, A_i attache les nogoods justifiant la suppression des valeurs dans $CSP(i)$ au message ***cpa***. Mais il n'attache pas tous ces nogoods car certaines variables ne sont pas pertinentes pour l'agent A_k (non connectées à x_k par une contrainte). MACA-not partage avec A_k tous les nogoods justifiant les suppressions concernant les variables non encore instanciées qui partagent une contrainte avec à la fois A_i et A_k (c.-à-d. les variables dans $\{CSP(i) \cap CSP(k)\} \setminus vars(CPA)$). Ainsi, quand A_k reçoit la CPA dans un message ***cpa***, il reçoit avec toutes les suppressions effectuées dans $CSP(i)$ qui peuvent l'amener à plus de propagation par la consistance d'arc.

Nous présentons dans Fig. 2 le pseudo-code de l'algorithme MACA-not. Seules les procédures qui sont nouvelles ou différentes de celles de MACA-del (Fig. 1) sont présentées. La fonction `Propagate()` n'envoie

plus des messages ***del***, elle ne fait que le maintien de la consistance d'arc dans $CSP(i)$ et renvoie true ssi aucun domaine n'a été vidé.

Dans la procédure `SendCPA(CPA)`, lors de l'envoi d'un message ***cpa*** à un agent A_k, A_i attache au CPA les nogoods qui justifient les suppressions des domaines des variables dans $CSP(i)$ qui sont contraintes avec A_k (lignes 13-17, Fig. 2).

Chaque fois que l'agent A_i reçoit un message ***cpa***, la procédure `ProcessCPA()` est appelée (ligne 6). Le message reçu sera traité seulement quand il contient une CPA plus forte que l'AgentView de A_i. Dans ce cas, A_i met à jour son AgentView (ligne 19). Puis, il met à jour les NogoodStores pour être compatibles avec la CPA qui vient d'être reçue (ligne 20). Ensuite, tous les nogoods contenus dans le message reçu sont ajoutés au NogoodStore (lignes 21-23). Les nogoods sont évidemment ajoutés au NogoodStore des variables dans leurs parties droites (c.-à-d. ng est ajouté au $NogoodStore[x_k]$ si x_k est la variable dans $rhs(ng)$). Puis, A_i appelle la fonction `Propagate()` pour maintenir la consistance d'arc dans $CSP(i)$ (ligne 24). Si le domaine d'une variable dans $CSP(i)$ se vide, A_i appelle la procédure `Backtrack()` (ligne 24). Sinon, A_i vérifie s'il doit affecter sa variable (ligne 25). A_i essaie d'assigner sa variable en appelant la procédure `Assign()` seulement si l'expéditeur du message ***cpa*** est son prédécesseur A_{i-1}.

4 L'analyse théorique

Nous démontrons que MACA termine, qu'il est correct et complet. Nous démontrons également qu'il a une complexité spatiale polynomiale.

Lemme 1 *MACA termine.*

Schéma de preuve. La preuve est proche de celle donnée dans [21]. On peut facilement obtenir par induction sur l'ordre des agents qu'il y aura un nombre limité de nouvelles CPA générées (au plus d^n, où n est le nombre de variables et d est la taille maximale des domaines), et que les agents ne peuvent jamais tomber dans une boucle infinie pour une CPA donnée.

Lemme 2 *MACA ne peut déduire une incohérence si une solution existe.*

Preuve. Chaque fois qu'une forte CPA ou un message ***ngd*** sont reçus, les agents dans MACA mettent à jour leurs NogoodStore. Dans MACA-del, les nogoods contenus dans les messages ***del*** sont acceptés seulement s'ils sont compatibles avec l'AgentView (lignes 31-33, Fig. 1). Dans MACA-not, les nogoods contenus dans les messages ***cpa*** sont compatibles avec la CPA reçue et ils sont acceptés seulement quand la CPA est plus forte que l'AgentView (ligne 18, Fig. 2). Ainsi, pour chaque CPA qui peut potentiellement conduire à une solution, les agents enregistrent seulement des nogoods valides. Puisque tous les nogoods sont générés par des inférences logiques à partir des contraintes existantes, un nogood vide ne peut pas être déduit si une solution existe.

Théoreme 1 *MACA est correct.*

Preuve. Pour montrer que MACA est correct, on suit le même schéma que [21]. Le fait que les agents envoient en avant seulement des solutions partielles consistantes dans des messages ***cpa*** à un seul endroit `Assign()` (ligne 13, Fig. 1), implique que les agents reçoivent seulement des affectations consistantes. La solution est rapportée uniquement par le dernier agent dans la procédure `SendCPA(CPA)` (ligne 16, Fig. 1 et ligne 10, Fig. 2). À ce stade, tous les agents ont affecté leurs variables et leurs affectations sont consistantes. Ainsi, MACA est correct. La complétude provient du fait que MACA termine et ne rapporte pas d'incohérence si une solution existe (Lemmes 1 and 2).

Théoreme 2 *MACA nécessite un espace polynomial par agent.*

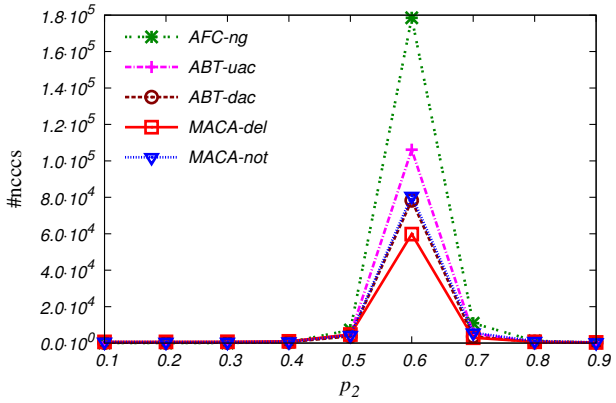
Preuve. Dans chaque agent, MACA mémorise un nogood de taille maximale n par valeur supprimée dans son réseau de contraintes local. Le réseau de contraintes local contient au plus n variables. Ainsi, la complexité spatiale de MACA est en $O(n^2d)$ dans chaque agent où d est la taille maximale des domaines initiaux.

Théoreme 3 *Les messages dans MACA sont polynomialement bornés.*

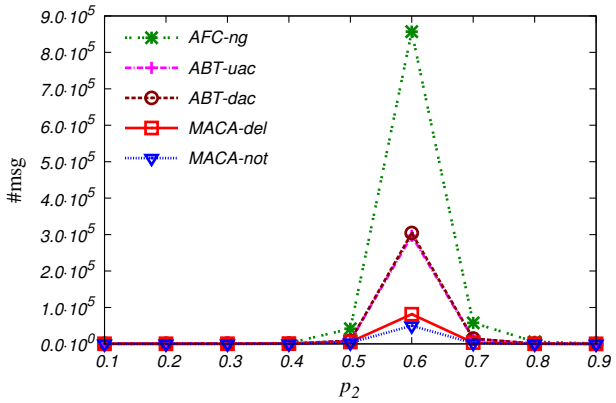
Preuve. Les plus grands messages dans MACA-del sont les messages ***del***. Dans le pire des cas, un message ***del*** contient un nogood pour chaque valeur. Donc, la taille des messages ***del*** est en $O(nd)$. Dans MACA-not, les plus grands messages sont des messages de type ***cpa***. Le pire des cas est un message ***cpa*** qui contient une CPA et un nogood pour chaque valeur dans le réseau de contraintes local. Ainsi, la taille d'un message ***cpa*** est en $O(n + n^2d) = O(n^2d)$.

5 Résultats expérimentaux

Dans cette section, nous comparons expérimentalement les algorithmes MACA avec ABT-uac, ABT-dac [6] et AFC-ng [7]. Ces algorithmes ont été évalués sur des DisCSPs binaires uniformes aléatoires. Toutes

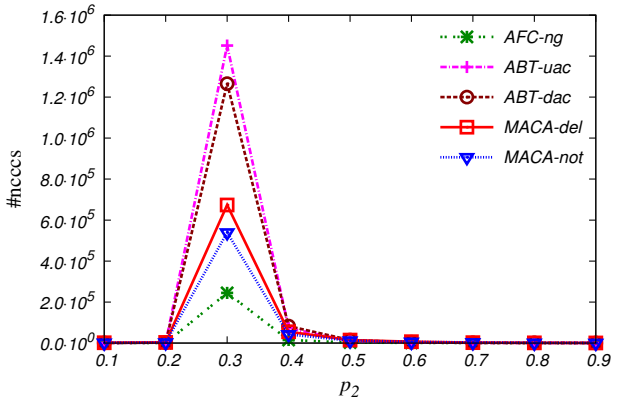


(a) *Le #ncccs effectués*

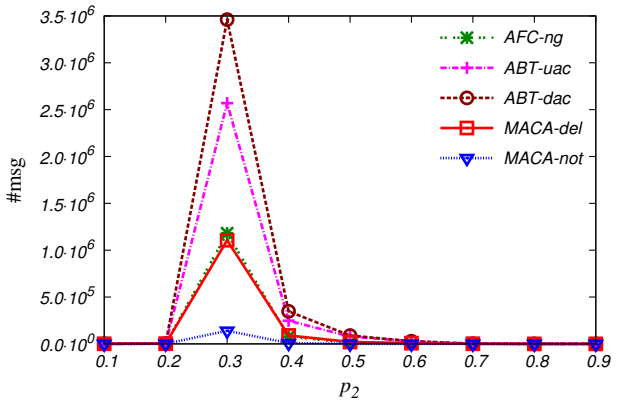


(b) *Le #msg échangés*

FIGURE 3 – Le *#ncccs* effectués et le *#msg* échangés pour résoudre les problèmes binaires uniformes aléatoires avec une faible densité où $p_1 = 0.25$.



(a) *Le #ncccs effectués*



(b) *Le #msg échangés*

FIGURE 4 – Le *#ncccs* effectués et le *#msg* échangés pour résoudre les problèmes binaires uniformes aléatoires avec une densité élevée où $p_1 = 0.70$.

nos expériences ont été réalisées sur la plateforme DisChoco 2.0 platform [1] [20] où les agents sont simulés par des threads java qui communiquent par échange de messages. Tous les algorithmes ont été testés en utilisant le même ordre d'agents (ordre lexicographique) et la même heuristique de sélection de nogood (*HPLV*) [10]. Pour ABT-dac et ABT-uac nous avons implémenté une version améliorée du mécanisme de détection de terminaison de Silaghi [18].

Nous évaluons la performance des algorithmes par le coût de communication [12] et l'effort de calcul. Le coût de communication est mesuré par le nombre total des messages échangés entre les agents durant l'exécution de l'algorithme (*#msg*). Ce nombre total inclut les messages échangés pour détecter la terminaison (messages système). L'effort de calcul est mesuré par le nombre de tests non-concurrents de contraintes (*#nccc*) (non-concurrent constraint checks) [26]. *#ncccs* est la métrique utilisée dans la résolution de contraintes distribuée pour simuler le temps de calcul.

Les algorithmes sont testés sur des DisCSPs uniformes binaires $\langle n, d, p_1, p_2\rangle$, avec n agents/variables, d valeurs par variable, une connectivité définie comme étant la proportion p_1 des contraintes binaires existantes, et une dureté de contraintes définie comme étant la proportion p_2 des paires de valeurs interdites. Nous présentons les résultats pour deux classes de problèmes : des problèmes avec une faible densité $\langle 20,\ 10,\ 0.25, p_2\rangle$ et des problèmes avec une densité élevée $\langle 20,\ 10,\ 0.7, p_2\rangle$. Pour chaque paire fixée (p_1, p_2), nous avons généré 100 instances. Nous reportons les moyennes de 100 exécutions.

La Fig. 3 présente les résultats sur les problèmes avec une faible densité ($p_1 = 0.25$). En terme d'effort de calcul (*#nccc*) (Fig. 3(a)), les algorithmes de maintien de la consistance d'arc sont largement meilleurs que AFC-ng qui maintient seulement le forward checking. MACA-del domine tous les autres al-

1. http ://dischoco.sourceforge.net/

gorithmes. MACA-not a une performance similaire à celle de ABT-dac mais il est plus performant que ABT-uac. En ce qui concerne le coût de communication ($\#msg$) (Fig. 3(b)), l'amélioration des algorithmes qui maintiennent l'AC (par rapport à AFC-ng) est plus nette par rapport à celle d'effort de calcul. ABT-uac et ABT-dac nécessitent presque le même nombre de messages échangés. En comparant les algorithmes qui maintiennent la consistance d'arc entre eux, la figure montre que ceux qui ont un comportement synchrone (les algorithmes MACA) améliorent ceux qui ont un comportement asynchrone (ABT-dac et ABT-uac) d'un facteur 6. Dans les problèmes avec une faible densité, le maintien de la consistance d'arc dans une recherche synchrone semble donner des performances meilleures par rapport à une recherche asynchrone. Dans le pic de complexité, MACA-not échange moins de messages que MACA-del.

Nous présentons les résultats sur les problèmes avec une densité élevée ($p_1 = 0.7$) dans la Fig. 4. En ce qui concerne les $\#msg$ (Fig. 4(a)), la première remarque est que les algorithmes asynchrone sont moins performants que ceux qui affectent leurs variables d'une manière séquentielle. Contrairement aux problèmes avec une faible densité, AFC-ng domine tous les autres algorithmes. Ceci est cohérent avec les résultats dans les CSP centralisés où FC a un meilleur comportement sur les problèmes denses que sur ceux avec une faible densité [3, 8]. De nouveau, ABT-dac est plus performant que ABT-uac. Au contraire des problèmes avec une faible densité, MACA-not est maintenant plus performant que MACA-del. En terme de charge de communication (Fig. 4(b)) dans les problèmes denses, les algorithmes asynchrones (ABT-uac et ABT-dac) nécessitent un grand nombre de messages. MACA-del n'améliore pas AFC-ng en raison du grand nombre de messages ***del*** échangés par MACA-del. Dans ces problèmes, MACA-not est l'algorithme qui nécessite le moins de messages. MACA-not améliore les deux autres algorithmes synchrones (AFC-ng et MACA-del) d'un facteur 11 et les deux autres asynchrones (ABT-uac et ABT-dac) d'un facteur 40.

À partir de ces expérimentations, nous pouvons conclure que dans les algorithmes synchrones, le maintien de la consistance d'arc est mieux que le maintien du forward checking en termes de temps de calcul sur les réseaux à faible densité, et est toujours mieux en termes de charge de communication. Nous pouvons également conclure que le maintien de la consistance d'arc dans les algorithmes synchrones produit plus de bénéfices que le maintien de la consistance d'arc dans les algorithmes asynchrones comme ABT.

6 Conclusion

Nous avons proposé deux nouveaux algorithmes de recherche synchrone pour résoudre les DisCSP. Il s'agit des premières tentatives de maintien de la consistance d'arc dans une recherche synchrone pour les DisCSP. Le premier algorithme, MACA-del, maintient la consistance d'arc grâce à un type de messages supplémentaire, des messages de suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Malgré le caractère synchrone de la recherche dans ces deux algorithmes, ils exécutent la phase de consistance d'arc d'une manière asynchrone. Nos expérimentations montrent que le maintien de la consistance d'arc dans une recherche synchrone produit des bénéfices bien supérieures à ceux offerts par le maintien de la consistance d'arc dans les algorithmes asynchrones comme ABT. La charge de communication de MACA-del peut être nettement inférieure à celle de AFC-ng, le meilleur algorithme synchrone à ce jour. Grâce à son utilisation plus parcimonieuse des messages, MACA-not montre encore plus de gain.

Références

[1] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman et Magda Valls : Sensor networks and distributed CSP : communication, computation and complexity. *Artificial Intelligence*, 161:117–147, 2005.

[2] Christian Bessiere : Constraint Propagation. *In Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapitre 3. Elsevier, New York, NY, USA, 2006.

[3] Christian Bessiere et Jean-Charles Régin : MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. *In Proceedings of CP'96*, pages 61–75, 1996.

[4] Christian Bessiere et Jean-Charles Régin : Refining the basic constraint propagation algorithm. *In Proceedings of IJCAI'01*, pages 309–315, 2001.

[5] Ismel Brito et Pedro Meseguer : Distributed Forward Checking. *In Proceeding of CP'03*, pages 801–806, Ireland, 2003.

[6] Ismel Brito et Pedro Meseguer : Connecting ABT with Arc Consistency. *In CP*, pages 387–401, 2008.

[7] Redouane Ezzahir, Christian Bessiere, Mohamed Wahbi, Imade Benelallam et El-Houssine

Bouyakhf : Asynchronous inter-level Forward-Checking for DisCSPs. *In Proceedings of CP'09*, pages 304–318, 2009.

[8] Stuart A. Grant et Barbara M. Smith : The phase transition behaviour of maintaining arc consistency. *In Proceedings of the 12th European conference on Artificial intelligence*, ECAI'96, pages 175–179, 1996.

[9] Robert M. Haralick et Gordon L. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 (3):263–313, 1980.

[10] Katsutoshi Hirayama et Makoto Yokoo : The Effect of Nogood Learning in Distributed Constraint Satisfaction. *In Proceedings of ICDCS'00*, pages 169–177, 2000.

[11] Hyuckchul Jung, Milind Tambe et Shriniwas Kulkarni : Argumentation as Distributed Constraint Satisfaction : Applications and Results. *In Proceedings of AGENTS'01*, pages 324–331, 2001.

[12] Nancy A. Lynch : *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[13] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce et Pradeep Varakantham : Taking DCOP to the real world : Efficient complete solutions for distributed multi-event scheduling. *In Proceedings of AAMAS'04*, 2004.

[14] Amnon Meisels et Roie Zivan : Asynchronous Forward-Checking for DisCSPs. *Constraints*, 12 (1):131–150, 2007.

[15] Adrian Petcu et Boi Faltings : A value ordering heuristic for distributed resource allocation. *In Proceeding of CSCLP'04*, pages 86–97, Feb 2004.

[16] Patrick Prosser, Chris Conway et Claude Muller : A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83, oct 1992.

[17] Daniel Sabin et Eugene Freuder : Contradicting conventional wisdom in constraint satisfaction. *In Proceedings of CP'94*, volume 874, pages 10–20, 1994.

[18] Marius-Calin Silaghi : Generalized dynamic ordering for asynchronous backtracking on DisCSPs. *In DCR workshop, AAMAS-06*, 2006.

[19] Marius-Calin Silaghi, Djamila Sam-Haroud et Boi Faltings : Consistency maintenance for ABT. *In Proceedings of CP'01*, pages 271–285, 2001.

[20] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere et El-Houssine Bouyakhf : DisChoco 2 : A Platform for Distributed Constraint Reasoning. *In Proceedings of DCR'11*, pages 112–121, 2011. URL http://www.lirmm.fr/coconut/dischoco/.

[21] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere et El Houssine Bouyakhf : Nogood-Based Asynchronous Forward-Checking Algorithms. *Constraints*, 18(3):404–433, 2013.

[22] Richard J. Wallace et Eugene C. Freuder : Constraint-based multi-agent meeting scheduling : effects of agent heterogeneity on performance and privacy loss. *In Proceeding of the workshop on DCR'02*, pages 176–182, 2002.

[23] Makoto Yokoo : Algorithms for distributed constraint satisfaction problems : A review. *Autonomous Agents and Multi-Agent Systems*, 3(2): 185–207, 2000.

[24] Makoto Yokoo, Edmund H. Durfee, Toru Ishida et Kazuhiro Kuwabara : The distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[25] Roie Zivan et Amnon Meisels : Synchronous vs asynchronous search on DisCSPs. *In Proceedings of EUMA'03*, 2003.

[26] Roie Zivan et Amnon Meisels : Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006.