Backtracking asynchrone agile pour les problèmes de satisfaction de contraintes distribués

Christian Bessiere[1] **El Houssine Bouyakhf**[2] **Younes Mechqrane**[2] **Mohamed Wahbi**[1,2]

[1]LIRMM/CNRS, University Montpellier 2, France
[2]LIMIARF/FSR, University Mohammed-V Agdal, Morroco
{bessiere,wahbi}@lirmm.fr ymechqrane@spsm.ma bouyakhf@fsr.ac.ma

Résumé

Le Backtracking Asynchrone est la procédure de recherche standard pour le raisonnement par contraintes distribué. Cette procédure nécessite un ordre total sur les agents. Tous les algorithmes avec une complexité spatiale polynomiale proposés jusqu'à présent pour améliorer le Backtracking Asynchrone ne permettent de réordonner les agents que de manière restreinte. Dans ce papier, nous proposons Agile-ABT, une procédure de recherche qui permet de réordonner les agents avec plus de liberté que les approches précédentes. Cette flexibilité est due à la notion de *valeurs de terminaison*, qui sont des vecteurs d'entiers positifs associés aux ordres échangés par les agents au cours de la recherche. Dans Agile-ABT, les agents peuvent proposer de nouveaux ordres autant qu'ils veulent à condition que les valeurs de terminaison décroissent au fur et à mesure que la recherche progresse. Nos expériences montrent les bonnes performances de Agile-ABT en comparaison avec les autres algorithmes de changement d'ordre.

1 Introduction

Des problèmes divers dans l'intelligence artificielle distribuée peuvent être formalisés sous forme de *Problèmes de Satisfaction de Contraintes Distribués* (DisCSP). Ces problèmes consistent à trouver une combinaison cohérente des actions de plusieurs agents (ex, allocation distribuée de ressources [6], réseaux de capteurs [1]). Les DisCSP sont composés d'agents, chacun ayant localement son propre réseau de contraintes. Les variables des différents agents sont liées par des contraintes. Les agents doivent affecter des valeurs à leurs variables de telle sorte que toutes les contraintes soient satisfaites.

Plusieurs algorithmes distribués ont été développés pour résoudre les DisCSP. Le backtracking asynchrone (ABT) est un des principaux.

ABT est un algorithme asynchrone exécuté de manière autonome par chaque agent où les agents n'ont pas à attendre les décisions des autres. Néanmoins, les agents sont soumis à un ordre total (priorité). Chaque agent essaie de trouver une affectation qui soit consistante avec les choix des agents les plus prioritaires. Quand un agent affecte une valeur à sa variable, la valeur sélectionnée est envoyée aux agents moins prioritaires. Lorsqu'un agent ne trouve aucune valeur consistante, l'incohérence est signalée aux agents les plus prioritaires par le biais d'un nogood (une combinaison de valeurs infructueuses). ABT calcule une solution (ou détecte qu'il n'y en a pas) en un temps fini. L'ordre total sur les agents est statique.

Il est connu des CSP centralisés que le fait de modifier l'ordre des variables d'une manière dynamique durant la recherche peut accélérer d'une manière drastique la recherche. Dans ce cadre, plusieurs algorithmes ont été proposés.

Asynchronous Weak Commitment (AWC) réordonne les agents d'une manière dynamique au cours de la recherche en plaçant l'émetteur d'un nogood plus haut dans l'ordre que les autres agents impliqués dans le nogood [9]. Cependant AWC a une complexité spatiale exponentielle.

Silaghi et al. (2001) ont présenté une tentative d'hybridation entre ABT et AWC où les changements d'ordre sont effectués par des agents abstraits. Dans, [7] l'heuristique du backtracking dynamique centralisé a été appliquée à ABT. Toutefois, dans les deux études précitées, les améliorations obtenues par rapport à ABT étaient mineures.

Zivan et Meisels (2006) ont proposé un algorithme de changement d'ordre pour le backtracking asynchrone

(ABTDO). Dans ABTDO, quand un agent affecte une valeur à sa variable, il peut modifier l'ordre des agents moins prioritaires.

Un nouveau type d'heuristiques pour le changement d'ordre, appelées heuristiques rétroactives, est présenté dans [12]. Dans la meilleure de ces heuristiques, l'agent qui a généré le nogood doit être placé entre le dernier et l'avant dernier agent dans le nogood. Cependant, la position du générateur du nogood ne sera effectivement modifiée que si la taille de son domaine est plus petite que les tailles des domaines des agents situés entre sa position actuelle et sa position cible.

Dans ce papier, nous proposons Agile-ABT, un algorithme de changement d'ordre asynchrone qui s'affranchit des restrictions habituelles des algorithmes de backtracking asynchrone. L'ordre des agents plus prioritaires que l'agent qui reçoit le message backtrack peut être changé avec plus de liberté tout en garantissant une complexité spatiale polynomiale. En plus, l'agent qui reçoit le message backtrack, appelé la *cible* du backtracking, n'est pas nécessairement l'agent ayant la priorité la plus faible parmi les agents conflictuels.

Le principe de Agile-ABT est construit sur les valeurs de terminaison échangées par les agents au cours de la recherche. Une valeur de terminaison est un tuple d'entiers positifs attaché à un ordre. Chaque entier positif dans ce tuple représente la taille présumée du domaine d'un agent ; la position de cet agent dans l'ordre étant la même que la position de l'entier dans le tuple. De nouveaux ordres sont proposés par les agents, sans aucun contrôle global, de manière à ce que les valeurs de terminaison décroissent lexicographiquement au fur et à mesure que la recherche progresse. Puisque une taille de domaine ne peut pas être négative, les valeurs de terminaison ne peuvent pas décroitre indéfiniment.

Quand un agent propose un nouvel ordre, il informe les autres en leur envoyant son nouvel ordre et sa nouvelle valeur de terminaison. Quand un agent compare deux ordres contradictoires, il va garder l'ordre qui est associé à la plus petite valeur de terminaison.

Le reste de ce papier est organisé comme suit. La section 2 rappelle quelques définitions de base. La section 3 décrit les concepts de base nécessaires pour sélectionner des nouveaux ordres qui font décroitre la valeur de terminaison. Nous présentons en détail notre algorithme dans la section 4. Nous apportons la preuve que cet algorithme est correct dans la section 5. Une étude expérimentale approfondie est donnée en section 6. La section 7 conclut le papier.

2 Préliminaires

Le Problème de Satisfaction de Contraintes Distribué a été formalisé dans [10] comme étant un tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{A}$ est un ensemble d'agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ est un ensemble de variables $\{x_1, \ldots, x_n\}$, tel que chaque variable x_i est contrôlée par un seul agent dans $\mathcal{A}$. $\mathcal{D} = \{D_1, \ldots, D_n\}$ est un ensemble de domaines, où D_i est un ensemble fini de valeurs qui peuvent être affectées à la variable x_i. La taille du domaine initial de la variable x_i est notée d_i^0. $\mathcal{C}$ est un ensemble de contraintes binaires qui spécifient les combinaisons de valeurs autorisées pour les couples de variables qu'elles impliquent. Une contrainte $c_{ik} \in \mathcal{C}$ entre deux variables x_i et x_k est un sous-ensemble du produit cartésien $D_i \times D_k$.

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.

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 tuple (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. Deux ensembles d'affectations $\{(x_{i_1}, v_{i_1}, t_{i_1}), \ldots, (x_{i_k}, v_{i_k}, t_{i_k})\}$ et $\{(x_{j_1}, v_{j_1}, t_{j_1}), \ldots, (x_{j_q}, v_{j_q}, t_{j_q})\}$ sont **cohérents** si chaque variable commune à ces deux sous ensembles est affectée à la même valeur.*

A_i est autorisé à mémoriser un ordre unique noté o_i. Les agents qui apparaissent avant A_i dans o_i sont les agents les plus prioritaires (prédécesseurs) notés par $Pred(A_i)$ et inversement les agents moins prioritaires (successeurs) $Succ(A_i)$ sont les agents qui apparaissent après A_i.

Définition 2 *La **Vue** (AgentView) d'un agent A_i est un tableau qui contient les affectations les plus récentes de $Pred(A_i)$.*

Les agents peuvent inférer des ensembles d'affectations inconsistants appelés ***nogoods*** . Un nogood peut être représenté sous forme d'une implication. Évidemment, il y a plusieurs possibilités pour représenter un nogood donné sous forme d'une implication. Par exemple, $\neg[(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k)]$ est logiquement équivalent à $[(x_2 = v_2) \wedge \cdots \wedge (x_k = v_k)] \rightarrow (x_1 \neq v_1)$. Quand un nogood est représenté sous forme d'une implication, la partie gauche (lhs) et la partie droite (rhs) sont définies à partir de la position de $\rightarrow$. La partie droite d'un nogood est également appelée *conclusion* de ce nogood. Un nogood (ng) est ***compatible*** avec un ordre o_i si tous les agents dans $lhs(ng)$ apparaissent avant $rhs(ng)$ dans o_i.

Le domaine courant de x_i est l'ensemble de valeurs $v \in D_i$ telles que $x_i \neq v$ n'apparait dans aucune conclusion

de nogood mémorisé par A_i. Un agent maintient un seul nogood par valeur supprimée. La taille du domaine courant de A_i est notée par d_i.

3 Quelques notions et fonctions clés

Avant de présenter Agile-ABT, nous avons besoin d'introduire quelques nouvelles notions et de présenter quelques fonctions clés.

3.1 Établir une priorité entre les ordres

Quand les agents sont autorisés à proposer de nouveaux ordres d'une manière asynchrone, il faut que ces agents soient en mesure de décider d'une manière cohérente quel ordre sélectionner.

Nous proposons que la priorité entre les ordres soit basée sur les *valeurs de terminaison*. D'une manière informelle, si $o_i = [A_1, \ldots, A_n]$ est l'ordre courant de A_i, le tuple des tailles des domaines $[d_1, \ldots, d_n]$ est la valeur de terminaison associée à o_i. Pour construire les valeurs de terminaison, les agents échangent des *explications*.

Définition 3 *Une* ***explication*** e_j *est une assertion sous forme* $lhs(e_j) \rightarrow d_j$*, où* $lhs(e_j)$ *est la conjonction des parties gauches de tous les nogoods mémorisés par* A_j*, et* d_j *est le nombre de valeurs dans le domaine de* A_j *non supprimées par les nogoods.* d_j *est notée par* $rhs(e_j)$.

Chaque fois qu'un agent communique son affectation aux autres agents (en leur envoyant un message **ok?**), il insère son explication dans le message **ok?** afin de permettre aux autres agents de construire leurs valeurs de terminaison.

Les variables dans la partie gauche d'une explication e_j doivent précéder la variable x_j dans l'ordre puisque les affectations de ces variables ont été utilisées pour éliminer des valeurs du domaine de x_j. Une explication e_j induit certaines contraintes d'ordre appelées *safety conditions* dans [4].

Définition 4 ***Une contrainte d'ordre*** *est une assertion* $x_k \prec x_j$*. Étant donnée une explication* e_j*,* $S(e_j)$ *est l'ensemble des contraintes d'ordre d'ordre induites par* e_j*, avec* $S(e_j) = \{(x_k \prec x_j) \mid x_k \in lhs(e_j)\}$.

Une explication e_j est ***compatible*** avec un ordre o si toutes les variables dans $lhs(e_j)$ apparaissent avant x_j dans o. Chaque agent A_i mémorise un ensemble E_i d'explications envoyées par les autres agents. Durant la recherche, E_i est mis à jour afin de supprimer les explications qui ne sont plus valides.

Définition 5 *Pour un agent* A_i*, une explication* e_j *dans* E_i *est* ***valide*** *si elle est* compatible *avec l'ordre courant* o_i *et* $lhs(e_j)$ *est cohérent avec la Vue de* A_i.

Quand E_i contient une explication e_j (associée à A_j), A_i utilise cette explication pour déduire la taille du domaine courant de A_j. Sinon, A_i suppose que la taille du domaine courant de A_j est égale à d_j^0. La valeur de terminaison dépend de l'ordre et de l'ensemble des explications.

Définition 6 *Soit* E_i *un ensemble d'explications mémorisées par* A_i*,* o *un ordre sur les agents tel que chaque explication dans* E_i *est compatible avec* o*, et* $o(k)$ *telle que* $A_{o(k)}$ *est le* $k^{ème}$ *agent dans* o*. La* ***valeur de terminaison*** $TV(E_i, o)$ *est un tuple* $[tv^1, \ldots, tv^n]$*, où* $tv^k = rhs(e_{o(k)})$ *si* $e_{o(k)} \in E_i$*, sinon,* $tv^k = d^0_{o(k)}$.

Dans Agile-ABT, un ordre est toujours associé à une valeur de terminaison. Quand on compare deux ordres, le *plus fort* entre les deux est celui associé à la valeur de terminaison lexicographiquement inférieure. En cas d'égalité, on utilise l'ordre lexicographique sur les indexes des agents, le plus petit étant le plus fort.

3.2 La cible du backtracking

Lorsque toutes les valeurs d'un agent A_i sont supprimées par des nogoods, cet agent procède à la résolution de ses nogoods. Cette résolution produit un nouveau nogood ng qui est la conjonction des parties gauches de tous les nogoods mémorisés par A_i. Si ng est vide, alors l'inconsistance est prouvée. Sinon, un des agents impliqués dans le conflit doit changer sa valeur. Pour les algorithmes de backtracking asynchrone standards, l'agent ayant la plus basse priorité doit changer sa valeur. Agile-ABT s'affranchit de cette restriction en autorisant A_i à sélectionner avec plus de liberté la cible du backtracking. La seule restriction pour placer une variable x_k dans la conclusion de ng est de trouver un ordre o' tel que $TV(up_E, o')$ est lexicographiquement inférieur à la valeur de terminaison associée à l'ordre courant de A_i. up_E est obtenu en mettant à jour E_i après avoir mis x_k dans la conclusion de ng.

La fonction `updateExplanations` prend en argument l'ensemble E_i, le nogood ng et la variable x_k à placer dans la conclusion de ng. `updateExplanations` supprime toutes les explications qui ne sont plus cohérentes une fois qu'on a placé x_k dans la conclusion de ng. Elle met à jour l'explication de l'agent A_k mémorisée par A_i et retourne un ensemble d'explications up_E.

Cette fonction ne crée pas de cycles dans l'ensemble des contraintes d'ordre $S(up_E)$ si $S(E_i)$ est acyclique. En effet, toutes les explications ajoutées ou supprimées de $S(E_i)$ pour obtenir $S(up_E)$ contiennent x_k. Donc, si $S(up_E)$ contient des cycles, tous ces cycles doivent contenir x_k. Toutefois, il n'y aucune contrainte d'ordre sous forme $x_k \prec x_j$ dans $S(up_E)$ puisque toutes ces contraintes d'ordre ont été supprimées dans la ligne 2. Par conséquent, $S(up_E)$ ne peut pas être cyclique.

```
function updateExplanations(E_i, ng, x_k)
1.  up_E ← E_i ; setRhs(ng, x_k);
2.  remove each e_j ∈ up_E such that x_k ∈ lhs(e_j);
3.  if (e_k ∉ up_E ) then
4.      setLhs(e_k,∅);
5.      setRhs(e_k,d_k^0);
6.      add e_k to up_E ;
7.  setLhs(e'_k, lhs(e_k) ∪ lhs(ng));
8.  setRhs(e'_k, rhs(e_k)−1);
9.  replace e_k by e'_k ;
10. return(up_E) ;
```

Comme nous allons le montrer dans la section 4, les mises à jour effectuées par A_i garantissent que $S(E_i)$ reste toujours acyclique. Par conséquent, $S(up_E)$ est aussi acyclique et peut être représenté par un graphe acyclique dirigé $G = (N, U)$ tel que $N = \{x_1, \ldots, x_n\}$ est l'ensemble des noeuds et U est l'ensemble des arcs dirigés. Un arc $(j, l) \in U$ si et seulement si $(x_j \prec x_l) \in S(up_E)$.

3.3 Comment faire décroitre les valeurs de terminaison ?

La terminaison de Agile-ABT est basée sur le fait que les valeurs de terminaison associées aux ordres sélectionnés par les agents décroissent au fur et mesure que la recherche progresse. Pour accélérer la recherche, Agile-ABT est écrit de façon à ce que les agents fassent décroitre les valeurs de terminaison autant qu'ils le peuvent. Quand un agent procède à la résolution de ses nogoods, il vérifie s'il peut trouver un nouvel ordre sur les agents de manière à ce que la valeur de terminaison associée à ce nouvel ordre soit meilleure que celle associée à l'ordre courant. Si c'est le cas, l'agent en question remplace son ordre courant et sa valeur de terminaison par ceux qu'il vient de calculer, et informe tous les autres agents.

Supposons maintenant qu'après avoir procédé à la résolution de ses nogoods, un agent A_i décide de placer x_k dans la conclusion du nogood ng produit par la résolution et soit up_E =`updateExplanations` (E_i, ng, x_k). La fonction `computeOrder` prend comme paramètre l'ensemble up_E et retourne un ordre up_o compatible avec l'ordre partiel induit par up_E.

Soit G le graphe acyclique dirigé associé à up_E. La fonction `computeOrder` détermine, à chaque itération p, l'ensemble $Roots$ des noeuds qui n'ont plus de prédécesseurs. Étant donné qu'on cherche à minimiser la valeur de terminaison, la fonction `computeOrder` sélectionne dans $Roots$ le noeud x_j ayant le plus petit domaine. Ce noeud est placé dans la $p^{ème}$ position et supprimé de G. Enfin, p est incrémenté et tous les arcs sortants qui partent de x_j sont supprimés de G.

Ayant proposé un algorithme qui détermine un ordre avec une petite valeur de terminaison pour une cible de backtracking x_k donnée, on a besoin de savoir comment

```
function computeOrder(up_E)
11. G = (N, U) is the acyclic graph associated to up_E ;
12. p ← 1 ; o is an array of length n;
13. while G ≠ ∅ do
14.     Roots ← {x_j ∈ N | x_j has no incoming edges} ;
15.     o[p] ← x_j such that d_j = min{d_k | x_k ∈ Roots} ;
16.     remove x_j from G ; p ← p + 1 ;
17. return(o) ;
```

choisir x_k pour obtenir un ordre qui fera décroitre davantage la valeur de terminaison.

La fonction `chooseVariableOrder` teste toutes les variables x_k contenues dans le nogood, calcule le nouvel ordre et la valeur de terminaison pour chaque variable testée x_k (lignes 20 et 22), et renvoie l'information correspondant à l'ordre le plus fort.

```
function chooseVariableOrder(E_i, ng)
18. o' ← o_i ;  TV' ← TV_i ; E' ← nil ; x' ← nil;
19. for each x_k ∈ ng do
20.     up_E ← updateExplanations(E_i, ng, x_k);
21.     up_o ←computeOrder(up_E);
22.     up_TV ← TV(up_E, up_o);
23.     if (up_TV is smaller than TV') then
24.         x' ←x_k ;
25.         o' ← up_o;
26.         TV' ← up_TV;
27.         E' ← up_E;
28. return(⟨x', o', TV', E'⟩) ;
```

4 L'algorithme

Chaque agent A_i mémorise localement une certaine quantité d'informations sur l'état global de la recherche, à savoir une Vue, un NogoodStore qui contient l'ensemble des nogoods mémorisés par A_i, un ensemble d'explications (E_i), un ordre courant (o_i) et la valeur de terminaison (TV_i). Agile-ABT autorise les types de messages suivants (A_i étant l'émetteur) :

- Un message **ok?** est envoyé par A_i aux agents moins prioritaires pour demander si la valeur qu'il a choisie est acceptable. En plus de la valeur choisie, le message **ok?** contient une explication e_i qui indique la taille du domaine courant de A_i. Le message **ok?** contient également l'ordre courant o_i et la valeur de terminaison courante TV_i mémorisés par A_i.
- Un message **ngd** est envoyé par A_i lorsque toutes ses valeurs sont supprimées par les nogoods contenus dans NogoodStore. Ce message contient le nogood proprement dit, ainsi que o_i et TV_i.
- Un message **order** sert à proposer un nouvel ordre. Ce message contient l'ordre o_i proposé par A_i et la valeur de terminaison TV_i.

Agile-ABT (Figures 1 et 2) est exécuté par chaque agent. Après l'initialisation, chaque agent affecte une valeur à sa

```
procedure Agile-ABT ( )
29.  t_i ← 0 ; TV_i ← [∞, ∞, ..., ∞] ; end ← false ; v_i ← empty;
30.  CheckAgentView( ) ;
31.  while (¬end) do
32.     msg ← getMsg( );
33.     switch (msg.type) do
34.        ok?    : ProcessInfo(msg);
35.        order  : ProcessOrder(msg);
36.        ngd    : ResolveConflict(msg);
37.        stp    : end ← true ;

procedure ProcessInfo(msg)
38.  CheckOrder(msg.Order, msg.TV) ;
39.  UpdateAgentView(msg.Assig ∪ lhs(msg.Exp)) ;
40.  if msg.Exp is valid then add(msg.Exp, E);
41.  CheckAgentView( ) ;

procedure ProcessOrder(msg)
42.  CheckOrder(msg.Order,msg.TV) ;
43.  CheckAgentView( ) ;

procedure ResolveConflict(msg)
44.  CheckOrder(msg.Order,msg.TV) ;
45.  UpdateAgentView(msg.Assig ∪ lhs(msg.Nogood)) ;
46.  if Coherent(msg.Nogood, AgentView ∪ x_i = v_i) and
     Compatible(msg.Nogood, o_i) then
47.     add(msg.Nogood,NogoodStore) ; v_i ← empty;
48.     CheckAgentView( );
49.  else if rhs(msg.Nogood) = v_i then
50.     sendMsg:ok?(v_i, e_i, o_i, TV_i) to msg.Sender ;

procedure CheckOrder(o, TV)
51.  if o is stronger than o_i then o_i ← o ; TV_i ← TV;
52.  remove nogoods and explanations incompatible with o_i;

procedure CheckAgentView( )
53.  if ¬Consistent(v_i,AgentView) then
54.     v_i ← ChooseValue( ) ;
55.     if (v_i) then sendMsg:ok?(v_i, e_i, o_i, TV_i) to Succ(A_i);
56.     else Backtrack( );
57.  else if (o_i was modified) then
58.     sendMsg:ok?(v_i, e_i, o_i, TV_i) to Succ(A_i) ;

procedure Backtrack( )
59.  ng ← solve(NogoodStore) ;
60.  if (ng = empty) then
61.     end ← true ; sendMsg:stp(system) ;
62.  ⟨x_k, o', TV', E'⟩ ←chooseVariableOrder(E_i, ng) ;
63.  if TV' is smaller than TV_i then
64.     TV_i ← TV' ; o_i ← o' ; E_i ← E' ;
65.     setRhs(ng, x_k);
66.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
67.     remove e_k from E_i ;
68.     broadcastMsg:order(o_i, TV_i) ;
69.  else
70.     setRhs(ng, x_k);
71.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
72.  UpdateAgentView (x_k ← unknown) ;
73.  CheckAgentView( ) ;
```

FIGURE 1 – L'algorithme Agile-ABT (1ère Partie)

```
procedure UpdateAgentView(Assignments)
74.  for each (var ∈ Assignments) do
75.     if (Assignments[var].t > AgentView[var].t ) then
76.        AgentView[var] ← Assignments[var];
77.  remove nogoods and explanations incoherent with AgentView;

function ChooseValue( )
78.  for each v ∈ D_i not eliminated by NogoodStore do
79.     if ∃x_j ∈ AgentView such that ¬Consistent(v,x_j) then
80.        add(x_j = v_j ⇒ x_i ≠ v, NogoodStore) ;
81.  if D_i = ∅ then return((empty));
82.  else t_i ← t_i+1 return(v);                    /* v ∈ D_i */
```

FIGURE 2 – L'algorithme Agile-ABT (2ème Partie)

variable et informe les agents moins prioritaires de sa décision (`CheckAgentView`, ligne 30) en envoyant des messages **ok?**.

Puis, une boucle considère la réception des différents types de messages. Si aucun message ne circule dans le réseau, l'état de quiescence est détecté par un algorithme spécialisé [5] et une solution globale est annoncée. La solution est l'ensemble des affectations courantes des variables.

Quand un agent A_i reçoit un message (de n'importe quel type), il vérifie si l'ordre inclus dans le message reçu est plus fort que son ordre courant o_i (`CheckOrder`, lignes 38, 42 et 44). Si c'est le cas, A_i remplace o_i et TV_i par ceux nouvellement reçus (ligne 51). Les nogoods et les explications qui ne sont plus compatibles avec o_i sont supprimés afin de garantir que $S(E_i)$ reste acyclique (ligne 52).

Si le message reçu est un message **ok?**, la Vue de A_i est mise à jour pour inclure les nouvelles affectations (`UpdateAgentView`, ligne 39). En plus de l'affectation de l'agent émetteur, A_i prend également les affectations contenues dans la partie gauche de l'explication transportée dans le message reçu afin de mettre à jour sa Vue. Ensuite, les nogoods et les explications qui ne sont plus cohérents avec la Vue de A_i sont supprimés (`UpdateAgentView` ligne 77). Après, si l'explication contenue dans le message reçu est valide, A_i met à jour l'ensemble des explications qu'il mémorise en ajoutant l'explication qu'il vient de recevoir. Enfin, la procédure `CheckAgentView` est appelée. (ligne 41).

A la réception d'un message **order**, A_i traite le nouvel ordre (`CheckOrder`) et la procédure `CheckAgentView` est appelée (ligne 43).

Quand A_i reçoit un message **ngd**, les procédures `CheckOrder` et `UpdateAgentView` (lignes 44 et 45) sont appelées. Ensuite, le nogood contenu dans le message est accepté s'il est cohérent avec la Vue de A_i et l'affectation de x_i et si ce nogood est compatible avec l'ordre courant de A_i. Sinon, le nogood est écarté et un message **ok?** est envoyé à l'émetteur comme dans ABT (lignes 49 et 50). Quand le nogood est accepté, il est mémorisé. Ce nogood justifiera la suppression de la valeur courante de A_i (ligne

47). Une nouvelle valeur qui soit consistante avec la Vue de A_i est recherchée (appel de `CheckAgentView`, ligne 48).

La procédure `CheckAgentView` vérifie si la valeur courante v_i est consistante avec la Vue de A_i. Si v_i est consistante, A_i vérifie si o_i a été modifié (ligne 57). Si c'est le cas, A_i doit envoyer son affectation aux agents moins prioritaires par le biais de messages **ok?**. Si v_i n'est pas consistante avec la Vue de A_i, ce dernier tente de trouver une valeur consistante (`ChooseValue`, ligne 54). Dans ce processus, quelques valeurs de A_i vont s'avérer inconsistantes. Dans ce cas, les nogoods justifiant la suppression de ces valeurs sont ajoutés au NogoodStore (`ChooseValue`, ligne 80). Si une nouvelle valeur consistante est trouvée, une explication e_i est construite et la nouvelle affectation est notifiée aux agents moins prioritaires que A_i par le biais de messages **ok?** (ligne 55). Sinon, chaque valeur de A_i est interdite par un nogood appartenant au NogoodStore et A_i doit faire un retour-arrière (`Backtrack`, ligne 56).

Dans la procédure `Backtrack`, A_i procède à la résolution de ses nogoods, et génère un nouveau nogood (ng). Si ng est vide, le problème n'a pas de solution. A_i termine l'exécution après l'envoi d'un message **stp** (ligne 61). Sinon, un des agents inclus dans ng doit changer sa valeur.

La fonction `chooseVariableOrder` sélectionne la variable x_k qui sera modifiée et un nouvel ordre o' tel que la nouvelle valeur de terminaison TV' est aussi petite que possible. Si TV' est inférieure à celle mémorisée par A_i, l'ordre courant et la valeur de terminaison sont remplacés par o' et TV' et A_i met à jour l'ensemble de ses explications par celui retourné par `chooseVariableOrder` (ligne 64). Ensuite, un message **ngd** est envoyé à l'agent A_k propriétaire de x_k (ligne 66). Puis, e_k est supprimée de E_i puisque A_k va probablement changer son explication après avoir reçu le nogood (ligne 67). Ensuite, A_i envoie un message **order** à tous les autres agents (ligne 68). Quand TV' n'est pas inférieure à la valeur de terminaison courante, A_i ne peut pas modifier l'ordre et la variable x_k qui sera modifiée est celle qui est la moins prioritaire selon l'ordre courant de A_i (lignes 70 et 71). Ensuite, l'affectation de x_k (la cible du backtracking) est supprimée de la Vue de A_i (ligne 72). Finalement, la recherche est continuée en faisant appel à la procédure `CheckAgentView` (ligne 73).

5 Preuves

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

Théoreme 1 *Agile-ABT nécessite un espace $O(nd + n^2)$ par agent.*

Théoreme 2 *Agile-ABT est correct.*

Schéma de preuve. Quand l'état de quiescence est atteint, tous les agents connaissent nécessairement l'ordre o qui est l'ordre le plus fort calculé jusqu'à présent. En plus, tout agent A_i connait les affectations les plus récentes de tous ses prédécesseurs selon o. Par conséquent, toutes les contraintes entre A_i et ses prédécesseurs ont été testées avec succès par A_i. Sinon, A_i aurait tenté de changer son affectation et aurait envoyé un message ok? ou un message ngd, ce qui va casser l'état de quiescence. □

Théoreme 3 *Agile-ABT est complet.*

Preuve. Tous les nogoods sont générés par des inférences logiques à partir des contraintes existantes. Par conséquent, un nogood vide ne peut pas être déduit si une solution existe. □

La preuve de terminaison est construite en se basant sur les lemmes 1 et 2.

Lemme 1 *Pour chaque agent A_i, tant qu'une solution n'est pas trouvée et l'inconsistance du problème n'est pas prouvée, la valeur de terminaison mémorisée par A_i décroit après un temps fini.*

Schéma de preuve. Quand un agent est coincé suffisamment longtemps avec la même valeur de terminaison TV_i, tous les autres agents vont finir par avoir la même valeur de terminaison et le même ordre o auquel cette valeur de terminaison est associée. A partir de ce point de temps, Agile-ABT va fonctionner exactement comme ABT (ABT est un algorithme correct, il termine et il est complet [2]). Donc, soit une solution sera trouvée soit le premier agent dans l'ordre, $A_{o(1)}$, va recevoir un nogood avec une partie gauche lhs vide. Ce nogood va supprimer une des valeurs restantes du domaine de $A_{o(1)}$. Aussitôt que $A_{o(1)}$ envoie la nouvelle taille de son domaine courant (inférieure à tv^1) aux agents moins prioritaires dans o, tout agent qui fera un backtracking va générer une valeur de terminaison inférieure à TV_i. □

Lemme 2 *Soit $TV = [tv^1, \ldots, tv^n]$ une valeur de terminaison générée par un agent quelconque. Nous avons $tv^j \geq 0, \forall j \in 1..n$*

Schéma de preuve. Toues les explications e_k mémorisées par un agent A_i ont $rhs(e_k) \geq 1$ puisque $rhs(e_k)$ représente la taille du domaine courant de A_k. Les valeurs de terminaison sont construites avec les rhs des explications. L'unique cas où tv^j peut être égal à 0 (ligne 8) est lorsque $A_{o(j)}$ est sélectionné par A_i comme cible du backtracking, et dans un tel cas l'explication $e_{o(j)}$ est supprimée juste après l'envoi du message nogood à $A_{o(j)}$ (ligne 67). Donc, A_i ne va, à aucun moment, mémoriser une explication e_k avec $rhs(e_k) = 0$ et donc il ne pourra jamais produire une valeur de terminaison avec des valeurs négatives. □

Théoreme 4 *Agile-ABT termine.*

Preuve. Preuve directe à partir des lemmes 1 et 2. □

6 Évaluation expérimentale

Nous avons comparé Agile-ABT à ABT, ABTDO et ABTDO avec les heuristiques rétroactives. Toutes nos expériences ont été réalisées sur la plateforme DisChoco [3] où les agents sont simulés par des threads java qui communiquent en se passant des messages. Nous évaluons la performance des algorithmes par le coût de communication et l'effort de calcul. Le coût de communication est mesuré par le nombre total de 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 une adaptation du nombre de tests non-concurrents de contraintes ($\#nccc$) (non-concurrent constraint checks) qui prend en considération les tests de nogoods.

Pour ABT, nous avons implémenté la version standard. Pour ABTDO [11], nous avons implémenté la meilleure version basée sur l'heuristique *nogood-triggered*. Dans cette version le récepteur d'un nogood modifie l'ordre de telle sorte que l'émetteur du nogood soit placé immédiatement après le récepteur.

Pour ABTDO avec heuristiques rétroactives [12], nous avons implémenté la meilleure version où un agent qui génère un nogood après avoir vidé son domaine, tente de modifier l'ordre en essayant de se placer entre le dernier et l'avant dernier agent dans le nogood. Cependant, le générateur du nogood ne peut se placer avant un autre agent A_k que si la taille du domaine courant du générateur du nogood est inférieure à la taille du domaine courant de A_k (noté par ABTDO-retro).

Les DisCSP binaires uniformes aléatoires

Ces algorithmes sont testés sur des DisCSP 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 $\langle 20, 10, 0.20, 0.66\rangle$ et des problèmes avec une densité élevée $\langle 20, 10, 0.70, 0.30\rangle$. Les deux classes sont dans le pic de complexité. Pour chaque classe, nous avons généré 25 instances, chaque instance a été résolue 4 fois. Nous reportons les moyennes de 100 exécutions.

La table 1 présente les résultats sur les problèmes aléatoires. En terme d'effort de calcul ($\#nccc$), ABT est l'algorithme le moins performant. ABTDO-ng est largement meilleur que ABT et ABTDO-retro est plus performant que ABTDO-ng. Ces constats sont similaires à ceux qui ont été formulés dans [12]. Agile-ABT domine tous ces algorithmes. Ces résultats suggèrent que dans le pic de complexité, plus l'algorithme est sophistiqué, plus il est performant. En ce qui concerne le coût de communication ($\#msg$), les messages supplémentaires envoyés par les algorithmes de changement d'ordre n'impliquent pas nécessairement une résolution plus rapide des problèmes. ABT nécessite moins de messages que ABTDO-ng pour les deux classes de problèmes, et moins de messages que ABTDO-retro sur les problèmes denses. Par contre, Agile-ABT est l'algorithme qui nécessite le moins de messages, même quand il est comparé à ABT. Ce n'est pas uniquement parce que Agile-ABT termine plus rapidement que les autres algorithmes, c'est aussi parce que Agile-ABT est plus parcimonieux dans le choix des nouveaux ordres que les autres algorithmes. Les valeurs de terminaison semblent limiter les changements d'ordres à ceux qui seront plus rentables.

TABLE 1 – Problèmes aléatoires

Algorithme	faible densité		densité élevée	
	$\#nccc$	$\#msg$	$\#nccc$	$\#msg$
ABT	710,911	213,702	14,015,644	3,947,616
ABTDO-ng	246,357	336,693	20,545,772	18,546,859
ABTDO-retro	80,908	136,146	8,844,655	8,727,913
Agile-ABT	46,287	55,121	2,432,920	1,321,120

Les problèmes distribués de capteur-mobile

Le problème distribué de capteur-mobile [1] est un benchmark basé sur un problème distribué réel. Il consiste en m capteurs qui traquent n mobiles. Chaque mobile doit être traqué par 3 capteurs. Chaque capteur peut traquer au plus un mobile. Une solution doit satisfaire les contraintes de visibilité et de compatibilité. Les contraintes de visibilité définissent l'ensemble des capteurs qui sont visibles pour chaque mobile. Les contraintes de compatibilité définissent les capteurs qui sont compatibles entre eux. Les problèmes sont caractérisés par $\langle m,\, n,\, p_c,\, p_v\rangle$ avec p_c est la probabilité que deux capteurs soient compatibles et p_v est la probabilité qu'un capteur soit visible à un mobile. Nous présentons les résultats pour deux classes de problèmes : des problèmes sous-contraints $\langle 25, 5, 0.6, 0.4\rangle$ et des problèmes dans le pic de complexité $\langle 25, 5, 0.6, 0.6\rangle$. Nous avons généré 25 instances et chaque instance a été résolue 4 fois. La moyenne est calculée pour 100 exécutions.

Le tableau 2 présente les résultats. La classe $\langle 25, 5, 0.6, 0.4\rangle$ est intéressante parce que dans cette région, les procédures de recherche font beaucoup d'aller-retours lorsque de mauvaises décisions sont prises au début de la recherche. C'est vraisemblablement la raison pour laquelle ABT est l'algorithme le moins performant en terme de $\#nccc$. ABTDO-retro est le meilleur algorithme sur ces problèmes. Au pic, tous les algorithmes ont une

TABLE 2 – Capteur-mobile

Algorithme	$p_c = 0.6, p_v = 0.4$		$p_c = 0.6, p_v = 0.6$	
	$\#nccc$	$\#msg$	$\#nccc$	$\#msg$
ABT	1,725,258	84,099	742,998	51,477
ABTDO-ng	312,400	111,511	845,475	350,930
ABTDO-retro	63,784	30,429	832,141	417,526
Agile-ABT	154,734	7,103	331,994	39,172

performance similaire, à l'exception de Agile-ABT qui est deux fois plus rapide. En ce qui concerne les $\#msg$, ABTDO-retro domine ABT sur $\langle 25, 5, 0.6, 0.4 \rangle$ parce qu'il termine plus rapidement que ABT. En dehors de cela, les choses sont semblables aux problèmes aléatoires, où les algorithmes ABTDO et ABTDO-retro sont pénalisées par les messages de changements d'ordre alors que le coût de communication de Agile-ABT reste faible.

7 Conclusion

Nous avons proposé Agile-ABT, un algorithme qui est capable de modifier l'ordre d'une manière plus agile que les approches précédentes. Grâce au concept des valeurs de terminaison, Agile-ABT est capable de choisir une cible du backtracking qui n'est pas nécessairement l'agent ayant la plus faible priorité parmi les agents conflictuels. En plus, l'ordre des agents qui apparaissent avant la cible du backtracking peut être modifié. Ces caractéristiques intéressantes sont inhabituelles pour un algorithme de complexité spatiale polynomiale. Nos expériences confirment l'importance de ces caractéristiques.

Références

[1] R. Bejar, C. Domshlak, C. Fernandez, K. Gomes, B. Krishnamachari, B.Selman et M.Valls. Sensor networks and distributed CSP : communication, computation and complexity. *Artificial Intelligence*, 161 :1-2:117–148, 2005.

[2] C. Bessiere, A. Maestre, I. Brito et P. Meseguer. Asynchronous backtracking without adding links : a new member in the ABT family. *Artificial Intelligence*, 161 :1-2:7–24, 2005.

[3] R. Ezzahir, C. Bessiere, M. Belaissaoui et E. H. Bouyakhf. Dischoco : a platform for distributed constraint programming. Dans *Proceeding of Workshop on DCR of IJCAI-07*, pages 16–21, 2007.

[4] Matthew L. Ginsberg et David A. McAllester. Gsat and dynamic backtracking. Dans *KR*, pages 226–237, 1994.

[5] K. Chandy Mani et L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. ISSN 0734-2071.

[6] A. Petcu et B. Faltings. A value ordering heuristic for distributed resource allocation. Dans *Proceeding of CSCLP04*, Lausanne, Switzerland, 2004.

[7] M. C. Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. Dans *DCR workshop, AAMAS-06*, Hakodate, Japan, 2006.

[8] M.-C. Silaghi, D. Sam-Haroud et B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Rapport technique, 2001.

[9] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. Dans *Proceeding of CP*, pages 88–102, Cassis, France, 1995.

[10] M. Yokoo, E. H. Durfee, T. Ishida et K. Kuwabara. Distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.

[11] R. Zivan et A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11 (2-3):179–197, 2006. ISSN 1383-7133.

[12] R. Zivan, M. Zazone et A.Meisels. Min-domain retroactive ordering for asynchronous backtracking. *Constraints*, 14(2):177–198, 2009. ISSN 1383-7133.