Le BackTracking intelligent distribué optimal

Dr. Belaïssaoui Mustapha
ENCG, Université Hassan I Maroc
Email : m.belaissaoui@encg-settat.ma
Dr. Bouyakhf El Houssine
Faculté de Sciences, Université Mohamed V Maroc
E-mail : Bouyakhf@fsr.ac.ma

1. Introduction

Les Problèmes de Satisfaction de Contraintes Distribués (DCSP) consistent à repérés entre les systèmes à contraintes en agents, entités indépendantes et communicantes capables d'agir sur elles-mêmes et sur l'environnement [12]. Parmi les nombreuses méthodes étudiées afin de contourner cette difficulté on retient la méthode Backtracking distribuée qu'est une technique de résolution des DCSP. Dans ce cadre, nous avons proposé une généralisation (plusieurs variables par agent) optimale en envois de message [1], et [3] de l'algorithme DIBT (Distributed Intelligent BackTracking) [8].

Cette généralisation OGDIBT (Optimal Generalization DIBT) est basée sur l'ordonnancement selon une heuristique des agents d'un DCSP. Pour établir cet ordonnancement, nous avons proposé une Extension EDisAO [1] de la méthode DisAO (Distributed Agents Ordering) [8]. EDisAO est définie pour accroître les performances des algorithmes de résolution des DCSP. Malheureusement, DIBT n'est pas complet [5]. Ce papier s'intéresse à étudier la complétude de ce dernier et il propose une généralisation complète et optimale en envoi de message OGDIBT (Optimal Generalization DIBT).

2. Exemple d'illustration

Nous considérons un exemple de Gestion de Ressources Distribuées pour illustrer les différentes méthodes de résolution présentées. La Firme Hewlett Packard (HP) va sortir un nouveau modèle d'ordinateur fabriqué dans trois pays d'Afrique, le Maroc, la Tunisie et le Sénégal :

- L'Unité Centrale (UC), le lecteur CD-ROM (D) et le lecteur Disquette (A) sont faits à Casablanca où le constructeur ne dispose que de la couleur blanche, grise, bleue et noire.
- La Souris (S) et le Clavier (CL) sont faits à Tunis où on ne dispose que de la couleur grise et bleue.
- L'Ecran (EC) est fait à Dakar où l'on a la couleur blanche, grise et noire.

Le concepteur de l'ordinateur impose quelques-uns de ses désirs quant à l'agencement des couleurs pour cet ordinateur :

- La Souris et le Clavier, qui sont de la même couleur, doivent être de la même couleur que le lecteur CD-ROM, qui doit être de la même couleur que le lecteur Disquette, lui-même de la même couleur que la Souris et le clavier.
- L'Ecran, le lecteur CD-ROM et le lecteur Disquette doivent être plus clairs que l'Unité Centrale.
- La Souris et le Clavier doivent être de la même couleur.

Modélisation en DCSP :

Afin qu'une solution globale puisse être trouvée, nous choisissons de modéliser le problème ci-dessus en un DCSP *P* à représentation explicite [4] de la façon suivante :

- Chaque site de la Firme HP est un agent. Donc, le système est composé de 3 agents : Casa, Tunis et Dakar.
- Les variables représentent les différents composants de l'ordinateur. L'agent Casa possède les variables UC (Unité Centrale), D (lecteur CD) et A (Lecteur Disquette). L'agent Tunis a les variables S (Souris) et CL (Clavier). L'agent Dakar possède la variable EC (Ecran).
- Les domaines de ces variables sont les couleurs autorisées pour chaque composant de l'ordinateur. Les lettres a, b, c et d correspondent respectivement aux couleurs blanche, grise, bleue et noire.
- Les deux agents Casablanca et Tunis possèdent des contraintes binaires locales entre ces variables. La contrainte «de la même couleur que» se traduit par «=». La contrainte «plus clair que» se traduit par «<» dans l'ordre lexicographique. Le système possède des contraintes binaires entre les agents. 4 contraintes entre l'agent Casablanca et l'agent Tunis (les lecteurs doivent être de la même

couleur que la souris et le clavier). Une contrainte entre l'agent Casablanca et l'agent Dakar (l'écran doit être plus clair que l'unité centrale).

Nous avons donc, P = (X, D, C, A) avec :

$A = \{Agent_{Casa}, Agent_{Tunis}, Agent_{Dakar}\}$

X = {UC, D, A, CL, S, EC};

$D = \{D_C, D_T, D_D\}$ tels que :

D_C = {a, b, c, d}, le domaine des variables d'$Agent_{Casa}$

D_T = {b, d}, le domaine des variables d'$Agent_{Tunis}$

D_D = {a, b, d}, le domaine des variables d'$Agent_{Dakar}$.

C'est l'ensemble des ensembles suivants :

{{UC < D}, {UC < A}, {D = A}} : ensemble des contraintes locales d'$Agent_{Casa}$;

{{CL = S}} : ensemble des contraintes locales d'$Agent_{Tunis}$;

{{D = S}, {A = S}, {D = CL},{A = CL}} : entre $Agent_{Casa}$ et $Agent_{Tunis}$;

{{EC < UC }} : ensemble des contraintes entre $Agent_{Casa}$ et $Agent_{Dakar}$.

3. Backtracking Distribué

Dans [12], les auteurs définissent informellement les DCSP comme des CSP binaires où les variables sont réparties parmi différents agents. Ces agents communiquent par envoi de message et possèdent l'ensemble du savoir relatif à leurs variables i.e. domaines/contraintes. Leur approche utilise une variable par agent ainsi qu'un système de communication suffisamment réaliste (sur chaque canal l'ordre de réception correspond à l'ordre d'émission, transmissions en temps fini). Nous pouvons résumer leur approche en trois points : la structuration des agents, la gestion de l'asynchronisme et la gestion des échecs [8].

3.1. hiérarchie

La recherche arborescente d'une solution nécessite la mise en place d'un ordre total entre les agents destinés au calcul. Le calcul de cet ordre revient à affecter des priorités entre agents. L'objectif commun des approches distribuées de types arborescents est d'offrir des méthodes distribuées calculant un ordonnancement des agents. Ces méthodes sont exécutées avant la méthode de résolution proprement dite. Dans ce sens, les auteurs [7] ont proposé un algorithme d'ordonnancement distribué DisAO (Distributed Agents Ordering) dont voici le principe :

3.1.1. Algorithme DisAO

Au départ, une phase d'échanges, entre les agents, permet aux agents de connaître la valeur de l'heuristique d'ordonnancement de chaque agent connecté. Un agent est capable de déterminer localement parmi ses accointances celles qui seront précédentes et celles qui seront suivantes durant la résolution. Chaque agent utilise les structures de données suivantes :

Γ : ensemble des agents du système avec lesquels Self (agent générique) partage au moins une contrainte inter-variable.
Level : table d'entiers destinée au tri des accointances précédentes.
f et **op** sont respectivement la fonction heuristique et l'opérateur de comparaison.
getMsg() : instruction bloquante, retourne le premier message disponible.
broadcast(P, m) : le message m est envoyé à l'ensemble d'agents P.
DisAO prend en entrée Γ, f et op, il retourne en sortie Γ^- et Γ^+ qui sont respectivement les ensembles des agents précédents et des agents suivants. Après la construction des deux ensembles Γ^- et Γ^+ (lignes 1-4), le calcul du niveau de l'agent dans la hiérarchie peut débuter. Ce niveau est représenté par l'entier **max** (ligne 5). Chaque agent conserve le niveau maximum transmis par Γ^+ et calcule son propre niveau en rajoutant 1 au maximum (lignes 6-10). Ensuite, chaque agent envoie sa propre position dans la hiérarchie vers Γ^- (message de type value : **(Γ^-, value : max)**) (ligne 11) et vers Γ^+ (message de type position : **(Γ^+, position: (max, Self)**)) (ligne 12). Enfin, l'ordonnancement consiste dans la récupération et le stockage de chaque position dans la table **Level** (lignes 13-15). Γ^- est ensuite trié en fonction de ces positions stockées dans la table **Level** (ligne 16).

Algorithm : Distributed Agent Ordering

DisAO(in : f, Γ, op ; out: Γ^- et Γ^+)
% partition de Γ
1. $\Gamma^- \leftarrow \varnothing$; $\Gamma^+ \leftarrow \varnothing$
2. pour $Agent_j \in \Gamma$ faire
3. si (f($Agent_j$) op f(self) alors $\Gamma^+ \leftarrow \Gamma^+ \cup \{Agent_j\}$
4. sinon $\Gamma^- \leftarrow \Gamma^- \cup \{Agent_j\}$
% ordonnancement de Γ^-
5. max←0
6. pour (i=0; i<$|\Gamma^+|$; i++) faire
7. m←getMsg()
8. si (m = value : v) alors
9. si (max < v) alors max←v
10. max++
11. broadcast(Γ^-, value : max)
12. broadcast(Γ^+, position : (max, self))
13. pour (i=0; i<$|\Gamma^-|$; i++) faire
14. m←getMsg()
15. si (m= position : (p, j)) alors $Level_j \leftarrow p$
16. tri de Γ^- en fonction de Level
17. Fin

3.2. Algorithme DIBT

Principe de DIBT :

Chaque agent instancie sa variable en respectant les contraintes qu'il partage avec ses Γ^-. Si une telle instanciation est possible l'agent informe ses agents suivants Γ^+. Sinon, un contexte d'échec est construit et adressé au plus proche des agents précédents. C'est à dire au premier référencé dans Γ^-. Après avoir testé la pertinence de la demande, le récepteur propose s'il le peut une valeur inédite, sinon, s'occupe de prolonger le retour arrière. Ce prolongement s'effectue en opérant une union ordonnée entre son propre contexte de retour arrière et le reste du contexte envoyé (i.e le contexte envoyé privé de son premier élément).

Problème :

L'information des agents Γ^+ est de modifier l'instanciation courante des agents informés. Cependant, Hamadi [8] a montré que la considération unique du lien unissant un agent à ses suivants ne suffit pas. Il a ensuite étendu les deux ensembles Γ^- et Γ^+, afin d'assurer la complétude de l'exploration de l'arborescence [8]. Chaque couple d'agent nouvellement connecté partage une contrainte tautologique qui a pour effet de remettre à jour le domaine de l'agent récepteur. L'extension des deux ensembles est comme suit :

$\Gamma_i^+ = \Gamma_i^+ \cup \{x / (\exists j \in \Gamma_i^+) \wedge (x \in \Gamma_j^-) \wedge (Level(x) < Level(i))\}$

$\Gamma_i^- = \Gamma_i^- \cup \{x / (\exists j \in \Gamma_i^+) \wedge (x \in \Gamma_j^-) \wedge (Level(x) > Level(i))\}$

3.2.1. Extension de DisAO

a- Extension [8]

Après le partitionnement de Γ en Γ^- et Γ^+ par un appel à DisAO, ces deux ensembles sont ensuite étendus comme ci-dessus. L'extension décrite dans [8] est comme suit :

- DisAO .
- Self (l'agent générique) envoie à tout les agents de Γ^- son ensemble Γ^- .
- A la réception de chaque Γ_i^-, Self rajoute dans Γ^+ l'ensemble des agents j tels que Level(j)<Level(Self) et envoie à ces agents j une demande d'insertion dans Γ_j^- .
- A la réception de chaque demande d'insertion, Self insère l'envoyeur dans Γ^-.

b- Analyse comportementale de l'extension

Après que Self (l'agent générique) envoie à ses accointances précédentes son ensemble ordonné Γ^- (voir figure 1), il reçoit $\Gamma_i^- = \{Agent_j, Self, Agent_k\}$. Self rajoute dans son ensemble Γ^+ l'$Agent_j$. Ensuite, il envoie à l'$Agent_j$ une demande d'insertion dans Γ_j^-.

L'$Agent_k$ reçoit $\Gamma_i^- = \{Agent_j, Self, Agent_k\}$ et affecte à son ensemble Γ_k^+ l'$Agent_j$ et Self. Ensuite, il envoie à l'$Agent_j$ et Self une demande d'insertion dans Γ_j^- et Γ^-.

L'$Agent_k$ a envoyé deux messages d'insertion et l'agent Self en a envoyé un. Donc, au total nous avons 3 demandes d'insertion.

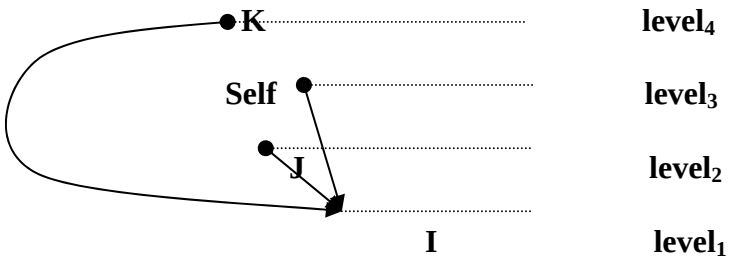

Figure1 : *Un réseau de contraintes à 4 agents après l'application de Dis AO*

c- Communication optimale (minimisation du nombre de messages)

L'$Agent_i$ envoie à Self, $Agent_j$ et $Agent_k$ son ensemble $\Gamma_i^- = \{Agent_j, Self, Agent_k\}$ qui est ordonné suivant les niveaux: $(Level(Agent_j) < Level(Self) < Level(Agent_k))$. Supposons que Self rajoute dans son ensemble Γ^+ $Agent_j$ et dans Γ^- $Agent_k$ sans demande d'insertion. De même, $Agent_k$ affecte à son ensemble Γ_k^+ l'$Agent_j$ et Self. $Agent_j$ affecte à son ensemble Γ_j^- $Agent_k$ et Self. Les messages de demande d'insertion sont inutiles.

En utilisant les insertions directes, suivant les niveaux, nous réduisons le nombre de messages de demande d'insertion.

d- Algorithme EDisAO (Extension Distributed Agent Ordering)

Dans chaque système (algorithme ci-dessous), Self (l'agent générique) utilise les mêmes structures de données que DisAO ainsi que, le message de type extension: **(Extension : Γ^-, $Level_{self}$)** (ligne 2). **$Level_{self}$** est utilisé par les précédents de Self pour ordonner leur ensemble d'accointances précédentes (lignes 4-11).

Algorithm : Extension Distributed Agent Ordering

EDisAO(in : f, Γ, op ; out: Γ^- et Γ^+)
Début
1. DisAO(f, Γ,op)
2. broadcast(Γ^-, Extension: (Γ^-, $Level_{self}$))
% Extension de Γ^- et Γ^+
3. pour (i=0; i <$|\Gamma^+|$; i++) faire
4. m←getMsg()
5. si (m = Extension: (Γ_i^-, $Level_i$)) alors
6. pour (j=0; j<$|\Gamma_i^-|$; j++) faire
7. si $Level_i[Agent_j]$< $Level_i$[Self] alors
8. $\Gamma^+ \leftarrow \Gamma^+ \cup \{Agent_j\}$
9. si $Level_i[Agent_j]$> $Level_i$[Self] alors
10. $\Gamma^- \leftarrow \Gamma^- \cup \{Agent_j\}$
11. $Level_{self}[Agent_j] \leftarrow Level_i[Agent_j]$
12. tri de Γ^- en fonction de Level
Fin

4. Généralisation de DIBT

Dans [7] les auteurs ont proposé l'algorithme DIBT en utilisant le niveau de granularité le plus fin. Chaque agent a une seule variable. Dans [8], l'auteur a signalé que la généralisation à plusieurs variables est directe. Chaque agent propose simultanément à ses suivants une solution à son sous-problème. C'est à dire, qu'il suffit d'affecter à la variable **myValue** [7] la solution locale recherchée par la généralisation de la primitive **getValue()**. La section suivante fait le point sur cette généralisation et propose une version optimale en envoi de message.

4.1. Généralisation Optimale de DIBT (OGDIBT)

Nous présentons ici notre mécanisme de minimisation des envois de message. Cela nous conduit à modifier l'algorithme DIBT généralisé pour en produire un autre optimal OGDIBT. Avant de détailler la méthode, nous illustrons le principe de minimisation que nous proposons.

Principe et illustration :

Chaque agent possède un ensemble de variables. De plus, il possède le savoir relatif à chacune de ses variables (domaines et contraintes). Ce savoir n'est donc pas strictement local. Tout se passe comme si les relations de contraintes inter-agents permettaient le partage d'une partie des buts de chacun. Ce partage permet de calculer localement l'impact d'un changement d'instanciation sur son environnement (caractérisé par Γ). Si cet impact est positif (dans le sens où les changements d'instanciation entraînent des changements d'instanciation pour certains agents suivants), l'envoi de messages est donc nécessaire pour les agents concernés par ces changements. Sinon cela n'est pas nécessaire.

Supposons qu'un agent A change l'instanciation ((i,a)..(j,b)..) par ((i,c)..(j,d)..). C'est à dire seules les variables i et j ont changé de valeurs. Dans DIBT, l'agent A devra informer tous les agents suivants. Dans notre méthode, l'agent A n'informe que les agents connectés avec lui par des contraintes qui portent sur i et j. Ce qui entraîne le gain des messages envoyés vers les agents non concernés.

Propriété 1 :

Un agent A informe un agent B du changement d'instanciation (..(i,a)..) à (..(i,b)..) si $\exists j \in X$ tq possède(B,j), $C_{ij} \in C$ et $a \neq b$.

Chaque agent devra donc localement vérifier la propriété avant de décider si l'information d'une accointance est nécessaire ou non (voir exemple 4.3).

Problème :

Un agent A reçoit un message de backtrack d'un agent B. A est un précédent de B. L'ensemble des variables de l'agent A, reliées par des contraintes avec les variables de l'agent B, devra changer d'instanciation. Sinon A n'informant pas l'agent B (propriété 1) ; alors la terminaison de l'algorithme, basé sur la propriété 1, sera remise en cause. En effet, puisque l'agent B qui a envoyé le message de retour arrière à A attend la réponse de celui-ci pour qu'il puisse trouver une instanciation à son sous-problème. Cette réponse n'arrivant pas. Pour palier à ce problème, nous introduisons la propriété suivante.

Propriété 2 :
Soient deux agents A et B. A est le plus proche précédent de B. Si A reçoit un message de backtrack de B, la sous-instanciation, des variables de A reliées par des contraintes à B, devra changer (voir exemple 4.3).

4.2. Algorithme OGDIBT

4.2.1. Principe

Sur la base de la propriété précédente, nous avons proposé une modification de l'algorithme DIBT [1]. Dans cet algorithme nommé OGDIBT, chaque agent Self (l'agent générique) utilise les mêmes primitives et structures de données suivantes :

- **D_{self}** est le domaine de Self et MyInst c'est l'instanciation courante de Self.
- **Inst[]** est la table stockant les instanciations prises par les accointances supérieures.
- **GetFirst (S)**, retourne le premier élément de l'ensemble S.
- **GetInst(type)** : Si type = info, Self est informé par ses accointances supérieures où bien il s'instancie à une première valeur, retourne la première instanciation de D_{self} compatible avec les instanciations de Inst[]. Si type = bt, Self reçoit un message de BackTrack, retourne la première instanciation de D_{self} située après MyInst, compatible avec les instanciations de Inst[] en tenant compte de la propriété 2 .
- **merge(A,B)**, retourne l'ensemble ordonné union des ensembles A et B .
- La fonction **Compare**, fait la comparaison de deux instanciations et renvoie les variables qui ont les valeurs différentes. C'est une fonction de comparaison de deux listes .
- La fonction **Localise**, permet la localisation des agents suivants qui partagent avec Self des contraintes dont les variables de Self ont changé de valeurs.

OptimalSendMsg, fonction qui s'occupe de l'envoi des messages aux agents en respectant la propriété 1 (voir algorithme: OptimalSendMsg ci-dessous).

Après le partitionnement de Γ et l'ordonnancement de Γ^{-} (algorithme ci-dessous : ligne 1), Self cherche sa première solution et l'affecte à sa variable **MyInst** (ligne 3).

Cette dernière est ensuite envoyée à l'ensemble Γ^+ par un message de type **infoVal** (ligne 4): un message de type infoVal comprend Γ^+, l'instanciation de Self myInst et le nom de Self ; Self informe ses suivants par son instanciation. La boucle commence par la réception de message **m** (ligne 7). Si le message est de type **infoVal** (ligne 9), Self stocke l'information dans la table **Inst[]** (ligne 10). Ensuite, il fait appel à la fonction **OptimalSendMsg(MyInst, Info)** (voir description ci-dessous). Un message de type **btSet** (BackTrack) comprend Γ^-, la table Value et les positions des agents de Γ^- dans la hiérarchie. Si le message est de type **btSet** (ligne 12), Self compare son instanciation courante à l'instanciation rapportée dans le message (ligne 13). S'il y a égalité, le message est valide. Self fait appel à **OptimalSendMsg(MyInst, bt)**.

OptimalSendMsg démarre par la recherche d'une nouvelle instanciation compatible avec la table **Inst[]** suivant le type de message (ligne 2) (voir algorithme ci-dessous). Si la nouvelle instanciation (**MyInst**) est différente de l'ancienne (**OldInst**) et aussi de l'ensemble vide (ligne 3) : Self fait appel à la fonction **Compare** (ligne 5). Cette dernière permet la localisation des variables qui ont des valeurs différentes dans les deux instanciations. Self envoie des messages de type **infoVal** aux agents localisés par la fonction **Localise** (ligne 6). Cette fonction permet de trouver les agents qui partagent avec self des contraintes qui portent sur des variables changeant de valeurs. Si la nouvelle instanciation (MyInst) est égale à l'ensemble vide :

- Si **type = info** (ligne 10), un message de type **btSet** (ligne 11)est adressé à l'agent **Nearest** (algorithme OGDIBT : ligne 2) qui est le supérieur le plus proche. Un message de type btSet comprend Γ^-, Inst[] et les positions des agents de Γ^- dans la hiérarchie.
- Sinon, si Self est une source ($\Gamma^- = \varnothing$) et si l'ensemble **Set** = $\varnothing$ (ligne 13), l'**insolubilité** est détectée (ligne 15). Sinon, un retour arrière est construit en réutilisant l'ensemble de retour transmis, **Set**. Self construit l'ensemble de retour **FollowSet** (ligne 19), union ordonnée de Γ^- et du Set (Set sans le premier élément). Un message **btSet** est alors transmis au premier élément de **FollowSet**, **Follow** (lignes 20-23). Si la demande de **backtrack** n'est pas adressée à un agent de Γ^- (ligne 24), Self se réinstancie à une instanciation compatible avec la table **Inst[]** et appellera **OptimalSendMsg(MyInst, info)** (ligne 27).

4.2.2. Complétude de OGDIBT

Dans [5], l'auteur a montré que DIBT [8] est incomplet. Par conséquent, OGDIBT, proposé dans [1], l'est aussi. En effet, lorsqu'il vérifie la validité d'un contexte d'échec, OGDIBT ne prend en compte que la valeur de l'agent qui reçoit le message, et pas celles des agents de Γ^-. Cela peut conduire à oublier ou sauter des solutions. Considérons cela sur l'exemple de la figure 2 où chaque agent n'a qu'une seule variable :

A_1 propose à ses suivants A_2 et A_3 la première solution à son sous-problème {(A_1, a)}. A_2 choisit c, et informe son fils. A_3 génère un message à A_2 de type (btSet: {A_2, A_1}, {(X_1, a), (X_2, c)}).

Pendant ce temps, A_1 reçoit un message d'un de ses parents, qui élimine la valeur a. A_1 réinstancie sa variable par b, et envoie un message de type (infoVal: ({(X_1, b)}, A_1)) à A_2 et A_3.

Le retour arrière initié par A_3 atteint A_2. Le contexte d'échec est obsolète (X_1= b et non a), mais A_2 le considère comme valide puisqu'il ne teste que sa propre valeur. A_2 envoie un message (btSet : {A_1}, {(X_1, b)}) à A_1.

A_1 génère un message de type btSet à son plus proche agents (pas de solution à son sous-problème). Des solutions ont été oubliées. L'algorithme est incomplet.

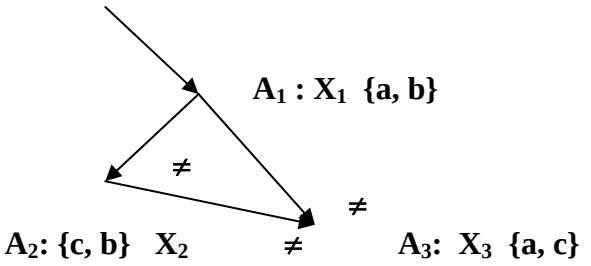

Figure2 : *DCSP avec des contraintes de différences*

Solution :

Dans OGDIBT, après avoir testé la pertinence de la demande de retour en arrière, le récepteur propose s'il le peut une valeur inédite, sinon, s'occupe de prolonger le retour arrière. Ce prolongement s'effectue en effectuant une union ordonnée entre son propre contexte de retour arrière Γ^- et le reste du contexte envoyé Set (i.e le contexte envoyé privé de son premier élément). Ce retour sera vers le

premier élément **Follow** de l'union ordonnée de Γ^- et **Set**. Nous proposons comme solution la propriété suivante.

Propriété 3 :

Si **Follow** $\in$ **Set** $\wedge\ \Gamma^-$ (voir algorithme ci-dessous) alors, avant de prolonger le retour en arrière, nous devons tester l'égalité suivante : **Insts [Follow] = Inst[Follow]** (Insts[] est la table des instanciations du contexte de retour arrière, Set. Inst[] est la table de l'agent récepteur).

En effet, si Insts[Follow] $\neq$ Inst[Follow] l'agent récepteur ne devra pas prolonger le backtrack à Follow. Parce que, Follow a changé d'instanciations et il a envoyé déjà un message de type InfoVal aux agents suivants. Parmi les agents suivants de Follow la cause de Backtrack : Follow $\in$ Set $\wedge\ \Gamma^-$.

Pour l'exemple de la figure 2 et dans l'étape 3, A_2 décide de ne pas prolonger le retour arrière parce que : Follow = $A_1 \in$ Set$\wedge\Gamma_2^-$ et (Insts[A_1]= a) $\neq$ (Inst[A_1] = b).

Sur la base de la propriété 3, la procédure OptimalSendMsg modifie celle de l'algorithme OGDIBT proposé dans [1].

4.3. Exemple d'illustration

(Voir la modélisation de l'exemple d'illustration de la section 2.)

Agent$_{Casa}$:

X_C={UC, A, D} ;

D_C={a, b, c, d} ;

C_C={(UC<D), (UC<A), (A=D), (D=S), (A=S), (D=CL), (A=CL), (EC<UC)}.

Agent$_{Tunis}$:

X_T={S, CL};

D_T={b, d} ;

C_T={(CL=S), (D=S), (A=S), (D=CL), (A=CL)}.

Agent$_{Dakar}$:

X_D={EC} ;

D_D={a, b, d} ;

C_D={(EC<UC)}

La structuration des agents par EDisAO en utilisant l'heuristique MaxDeg est :

$Agent_{Casa}$: $\Gamma^{-}=\varnothing$, $\Gamma^{+}=\{Agent_{Tunis}, Agent_{Dakar}\}$;

$Agent_{Tunis}$: $\Gamma^{-}=\{Agent_{Casa}\}$, $\Gamma^{+}=\varnothing$;

$Agent_{Dakar}$: $\Gamma^{-}=\{Agent_{Casa}\}$, $\Gamma^{+}=\varnothing$;

Lors du backtracking distribué, $Agent_{Casa}$ propose à ses suivants la première solution à son sous-problème {(UC,a), (A,b), (D,b)}. Après la réception de (infoVal: ({(UC,a),(A,b),(D,b)}, Casa)), $Agent_{Tunis}$ vérifie la compatibilité de sa solution locale {(S,b),(CL,B)} et la solution locale d'$Agent_{Casa}$. $Agent_{Tunis}$ n'envoie pas de message de type btSet.

Après la résolution de son sous-problème {(EC,a)}, $Agent_{Dakar}$ vérifie sa compatibilité avec celle d'$Agent_{Casa}$. $Agent_{Dakar}$ génère un message à $Agent_{Casa}$ de type (btSet:{Casa},{(UC, a), (A, b), (D, b)}.

Si nous ne tenons pas compte de la propriété 2, $Agent_{Casa}$ reçoit ce dernier message et réinstancie ses variables par {(UC, a), (A, c), (D, c)}. La variable UC n'a pas changé de valeur. Donc $Agent_{Casa}$ devra envoyer un message de type (infoVal: ({(UC, a), (A, c), (D, c)}, Casa)) tout simplement à $Agent_{Tunis}$ parce que la fonction Localise renvoie comme résultat l'ensemble $\{Agent_{Tunis}\}$. $Agent_{Dakar}$ attend toujours la réponse…

Algorithm : Optimal Generalization Distributed Intelligent BackTracking (OGDIBT)

```
1. EDisAO(f, Γ, op) // on ordonne les agents
2. Nearest← getFirst(Γ⁻)
3. MyInst← getInst(info)
4. broadcast(Γ⁺, infoVal : ( MyInst, Self))
5. end← false
6. tant que (!end) faire
7.    m← getMsg()
8.    Si (m = stop) Alors end← true
9.    Si (m = infoVal : (localInst,j)) Alors
10.          Inst[j]← localInst
11.          optimalsendMsg(MyInst, info)
12.  Si (m = btSet : (Set, Insts)) Alors
13.       Si (Insts[Self]=MyInst) Alors  optimalsendMsg(MyInst, bt)
Fin.
```

Algorithm : OGDIBT OptimalSendMsg

OptimalSendMsg(in: MyInst, type)
1. OldInst← MyInst
2. MyInst← getInst(type)
3. Si (MyInst ≠ OldInst) and (MyInst) Alors
4. début
5. differentes← Compare(MyInst, OldInst)
6. broadcast(Localise(Γ^+, differentes), infoVal : (MyInst, Self))
8. Sinon
9. début
10. Si (MyInst=∅) and (type= info) Alors
11. broadcast(Nearest, btSet : (Γ^-, Inst[Γ^-]))
12. Si (MyInst = ∅) and (type = bt) Alors
13. Si (Γ^- =∅ et Set =∅) Alors
14. début
15. broadcast(system, stop)
16. end← true
17. Sinon
18. début
19. FollowSet← merge(Γ^-, Set)
20. Follow← getFirst(FollowSet)
21. Si Follow ∈ Set ∧ Γ^- Alors
22. Si Insts[Follow]= Inst[Follow] Alors
23. broadcast(Follow, btSet : (FollowSet, Inst[Γ^-]∪Insts))
24. Si (Follow ∉ Γ^-) alors
25. début
26. broadcast(Follow, btSet : (FollowSet, Inst[Γ^-]∪Insts))
27.OptimalSendMsg(MyInst, info)
28. Finsi ;
29. finsinon ;
30. finsi ;
31. finsinon ;
32. finsi ;
33. Fin.

En utilisant la propriété 2, $Agent_{Casa}$ réinstancie ses variables par {(UC,b), (A,c), (D,c)}. Il envoie le message (infoVal : ({(UC,b), (A,c), (D,c)}, Casa)) à $Agent_{Tunis}$ et $Agent_{Dakar}$. Après la résolution de son sous-problème {(EC,a)}, $Agent_{Dakar}$ vérifie sa compatibilité avec celle d'$Agent_{Casa}$. $Agent_{Dakar}$ ne génère aucun message. $Agent_{Tunis}$ envoie un message à $Agent_{Casa}$ de type (btSet:{Casa},{(UC, b), (A, c), (D, c)} parce qu'il n'a pas trouvé une instanciation à son sous-problème compatible avec celle de $Agent_{Casa}$. Ce dernier reçoit ce dernier message et réinstancie ses variables par {(UC,b), (A,d), (D,d)}. Dans DIBT, $Agent_{Casa}$ devra envoyer un message de type (infoVal: ({(UC, b), (A, d), (D, d)}, Casa)) à $Agent_{Tunis}$ et $Agent_{Dakar}$. Dans OGDIBT, la variable UC n'a pas changé de valeur. Donc $Agent_{Casa}$ devra envoyer un message de type (infoVal: ({(UC,b), (A, d), (D, d)}, Casa)) tout simplement à $Agent_{Tunis}$. $Agent_{Tunis}$ réinstancie ses variables par {(S,d), (CL,d)}. Tout le monde est satisfait. Donc, nous avons une solution.

Les agents s'échangent des assignements solutions à leurs sous-problèmes. Au total, le problème est résolu par (EDisAO+OGDIBT) avec 12 messages point à point. Par contre, si nous utilisons EDisAO et DIBT nous aurons 19 messages point à point.

4.4. Complexité

Dans le pire des cas, le DCSP comprend p = n agents. Chaque agent possède donc une unique variable locale connectée à n-1 autres.

Complexité spatiale

Le stockage de l'instanciation locale fait que $|myInst| = 1$ et $|D| = d$. Un agent et connecté par relation de contrainte à n-1 autres agents donc, la table $|Inst[]| = n-1$. Chaque agent nécessite un espace O(n) pour stocker sa part du DCSP. Si nous considérons la totalité du système distribué, l'espace nécessaire au stockage est $O(n^2)$.

Complexité temporelle

EDisAO : La première phase partitionne Γ en deux sous-ensembles. Dans le pire cas, cette phase admet une complexité temporelle d'ordre O(n). Pour calculer sa propre position, chaque agent reçoit de ses suivants leur position. Dans le pire

cas, le graphe possède n niveaux. C'est à dire un agent par niveau. Dans cette deuxième phase l'agent de niveau i utilise n-i messages et n-i opérations locales. Si nous considérons la totalité du système, la deuxième phase utilise $O(n^2)$ messages et O(n) opérations locales. De la même façon, dans la troisième phase, chaque agent de niveau i reçoit i-1 messages de ses précédents et utilise i-1 opérations locales. Donc, nous avons $O(n^2)$ messages et O(n) opérations locales.
Dans la phase Extension, chaque agent de niveau i reçoit n-i messages d'extension. Pour l'extension, il utilise dans le pire des cas (n-i)(n-i+1) comparaison de niveaux, car la longueur de la première boucle est n-i et celle de la deuxième est n-i+1. Donc pour le système distribué, nous avons $O(n^2)$ messages et $O(n^2)$ opérations locales.
OGDIBT : Considérons le pire des cas, où p = n agents. Chaque agent possède donc une unique variable locale connectée à n-1 autres. Dans ce cas, la complexité de OGDIBT sera de $O(d^n)$. La démonstration est simple, il suffit de procéder par des retours arrières des niveaux les plus bas vers les niveaux les plus hauts. En particulier, du niveau 1 jusqu'au niveau n.
Avec ces retours arrières, nous risquons d'explorer le produit cartésien de tous les domaines. Avec n variables et $|D| = d$, la complexité sera $O(d^n)$.

4.5. Expérimentations

Ces expérimentations mettent en œuvre EDisAO, DIBT généralisé (plusieurs variables par agents) et OGDIBT dans un environnement distribué utilisant une granularité p<n (p agents et n variables).
Une approche plus théorique et couramment utilisée consiste à tester les algorithmes sur un échantillon de problèmes tirés aléatoirement (voir algorithme ci-dessous) ; on prend habituellement pour mesure la moyenne des performances réalisées sur l'échantillon testé. Ceci implique : La description du modèle des DCSP générés (paramètres et algorithme de génération). Les tests pour chaque valeur du ou des paramètres de génération.
Il reste enfin à définir les unités de mesure utilisées. Puisque nous cherchons l'optimalité d'envoi de messages, nous avons choisi l'unité couramment utilisée : la moyenne des temps d'exécution.

Notre générateur de problèmes distribués aléatoires est l'extension du générateur de Van Beek [11] (la bibliothèque de programmes concernant les CSP de Van Beek) au cadre de DCSP. Il nécessite cinq paramètres : le nombre de variables **n**, le nombre de valeurs par domaine **d**, la proportion (probabilité) de contraintes dans le réseaux $\mathbf{p_1}$, la proportion de paires interdites dans une contrainte $\mathbf{p_2}$ et le nombre d'agents **p** < **n**. Nous avons généré deux types de problèmes : des problèmes faiblement contraints ou **peu denses** pour les petites valeurs de p_1 et des problèmes ou graphes complets pour p_1=1.

Après la création du CSP global (algorithme ci-dessous : lignes 1-8), nous l'avons distribué comme suit. Dans un premier temps, nous avons initialisé chaque sous-problème CSP_j (CSP_j est le sous-problème de l'agent j, $1 \leq j \leq p$) par la variable $i \in \{1,.., n\}$ telle que i=j ainsi que son domaine D_i et l'ensemble des relations R_{ik} telle que $1 \leq k \leq n$ et $k \neq i$ (lignes 12,13). L'extension de ces sous-problèmes CSP_j est faite suivant la valeur d'une variable x tirée aléatoirement entre 0 et 1 (lignes 14-20).

Algorithm : Générateur aléatoire de DCSP

In : n, d, p_1, p_2, p
Début
1. Pour chaque $1 \leq i < j \leq n$ faire
2. $x \leftarrow$ valeur aléatoire entre 0 et 1 ;
3. Si $x < p_1$ alors
4. {la contrainte C_{ij} est créée, initialement vide}
5. $R_{ij} \leftarrow \varnothing$;
6 Pour chaque $(a,b) \in D_i \times D_j$ faire
7. $y \leftarrow$valeur aléatoire entre 0 et 1
8. Si $y<p_2$ alors $R_{ij} \leftarrow R_{ij} \cup \{(a,b)\}$
9. Finp
10. Finsi
11. Finp
12. Pour chaque agent $1 \leq j \leq p$ faire Insérer (j, CSP_j) ;
13. {insère la variable j, D_j et les R_{ij} dans CSP_j}
14. Pour chaque agent $1 \leq j \leq p-1$ faire
15. Pour chaque variable $p+1 \leq i \leq n$ faire
16. $x \leftarrow$ valeur aléatoire entre 0 et 1 ;
17. Si $x < p/n$ alors Insère (i, CSP_j)
18. Finp
19. Finp ;
20. Ajouter le reste des variables à CSP_p
Fin.

Pour chacune des méthodes que nous avons testé, nous avons utilisé comme **plateforme** un Pentium IV, 2.5 Ghz et comme **langage** de développement le C Lisp sous Windows. la démarche est comme suit : chaque agent est activé à tour de rôle par un processus système, pour un cycle durant lequel l'agent peut lire tous les messages qui lui ont été adressés au cycle précédent, effectuer localement les calculs qui s'y rapportent et envoyer les messages adéquats. Lorsqu'il y a stabilisation de tout les processus, le processus système détecte la terminaison. Puisque tout les processus utilisent un seul processeur, nous avons mesuré le temps processeur permettant la résolution de chaque problème. Nous avons généré deux types de problèmes pour différentes valeurs de $\mathbf{p_2}$:

- Problèmes denses **<p, n, d, p_1, p_2>=<5,15, 5, 1, p_2>** ;
- Problèmes peu denses **<p, n, d, p_1, p_2>=<5,15, 5, 0.3, p_2>**,.

Sur chaque instanciation de p_2, nous avons comparé les performances de DIBT généralisé (plusieurs variables par agents) et d'OGDIBT, en utilisant l'heuristique MaxDeg pour ordonner les agents.

La **figure 3** montre nos résultats sur un ensemble de problèmes denses en faisant varier p_2 par pas de 0.1. OGDIBT s'est révélé nettement meilleur que DIBT pour p_2 dans [0.3,.., 0.5]. Ceci s'explique par le fait que le degré de connectivité des variables est de 100% (p1=1) et si p_2 appartient à l'intervalle [0.3,.., 0.5], le nombre d'envoi de messages augmente parce que le nombre de contextes de retour en arrière s'élève en balayant l'ensemble de valeurs avec presque 50% de valeurs interdites. Ceci explique l'écart en temps de calcul dans cet intervalle.

La **figure 4** montre aussi nos résultats sur des problèmes peu denses avec une variation de p_2 par pas de 0.1. Dans l'intervalle [0.5,.., 0.8] de p_2, OGDIBT se comporte encore mieux que DIBT. Cet intervalle est supérieur à celui des problèmes denses parce que le degré de connectivité est inférieur à celui de ces derniers.

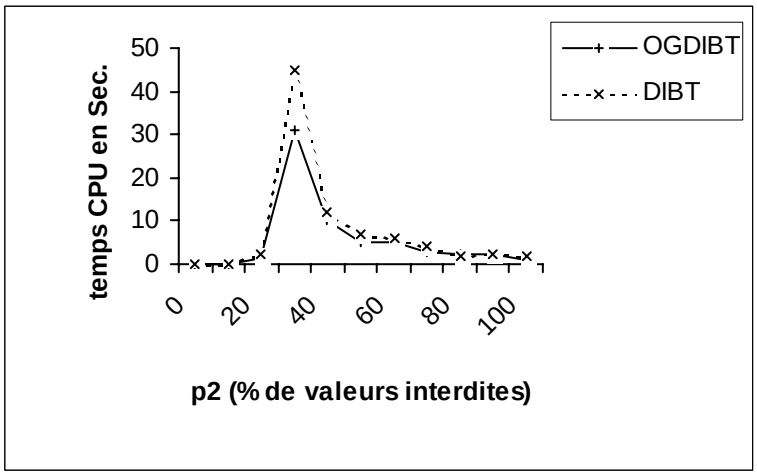

Figure 3 *: Comparaison entre OGDIBT et DIBT sur des problèmes denses (p_1=100%).*

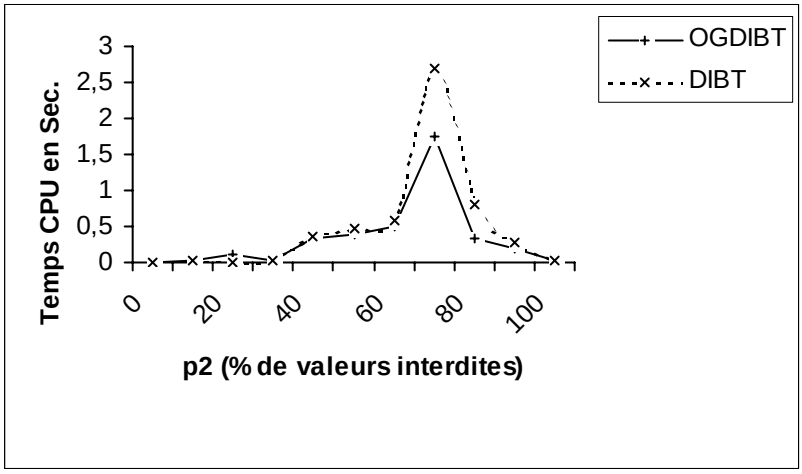

Figure 4 : *Comparaison entre OGDIBT et DIBT sur des problèmes peu denses (p_1=0.3).*

5. Conclusion

Dans le domaine des Problèmes de Satisfaction de Contraintes Distribués (DCSP), nous avons consacré notre étude à la généralisation optimale en envois de message de l'algorithme de backtracking distribué DIBT (Distributed Intelligent BackTracking) [7]. Nous avons d'abord défini une extension de l'algorithme d'ordonnancement des agents DisAO (Distributed Agents Ordering). Cette extension EDisAO (Extension DisAO) est utilisée pour accroître les performances des algorithmes de résolution des DCSP.

Nous avons ensuite défini notre mécanisme de minimisation des envois de message. Cela nous a conduit à modifier l'algorithme DIBT généralisé pour en produire un autre optimal OGDIBT (Optimal Generalization DIBT) ayant une complexité $O(d^n)$. Pour illustrer cet algorithme, un exemple simple est traité ; la Gestion de Ressources Distribuées. Les expérimentations mettent en œuvre DIBT généralisé [8] et notre méthode dans un environnement distribué utilisant une granularité $p<n$ (p agents et n variables).

Bibliographie

[1] Belaissaoui, M., Bouyakhf, EH., Le backtracking dans les réseaux de contraintes distribués, Journées Nationale de Modèles de Raisonnement (JNMR'01), Arras, France, (www.cril.univ-artois.fr/jnmr01/).

[2] Belaissaoui, M., Bouyakhf, EH., Les problèmes de satisfaction de contraintes distribués et le backtracking, EJNDP, Volume 12, sept.-01, (rerir.univ-pau.fr/rerir2.html).

[3] Belaissaoui, M., Bouyakhf, EH., Les problèmes de satisfaction de contraintes distribués et le backtracking, JFIADSMA'01.

[4] Belaissaoui, Mustapha, Contribution à l'étude de Raisonnement Temporel et au Traitement des Problèmes de Satisfaction de Contraintes Distribués, thèse de doctorat, université Mohammed V Agdal, Faculté des Sciences Rabat, 2002.

[5] Bessière, C., Maestre, A. et Meseguer, P., Distributed dynamic backtracking. In M. Silaghi, editor, Proceedings of the IJCAI'01 workshop on distributed constraint reasoning, Seattle, pages 9-16, 2001.

[6] Eisenberg, C., Integrating distributed and heterogeneous organisation using constraint programming, CP'2000.

[7] Youssef, H., Bessière, C., Backtracking in distributed constraint networks, ECAI'98.

[8] Youssef, H., Traitement des problèmes de satisfaction de contraintes distribués, thèse de doctorat, université Montpellier II, 99.

[9] Hyuckchul, J. et al., On modeling argumentation as distributed constraint satisfaction, CP'2000.

[10] Queloz, P., Projet ISACOM, rapport pour le FNRS, 98.

[11] Van Beek, P., Sources for CSP programming. (http://web.cs.ualberta.ca:80/vanbeek/)

[12] Yokoo, M., Ishida, T., et Kubawara, K. (1990). Distributed constraint satisfaction for DAI problems. In Huhns, M. N., editor, Proc. of the 10th International Workshop on Distributed Artificial Intelligence, chp 9.

[13] Yokoo, M., Durfee, E., Ishida, T., et Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In 12th Int. Conf. on Distributed Computing Systems, 614-624.

[14] Yokoo, M. et Hirayama, K. (1998). Distributed constraint satisfaction algorithm for complex local problems. In ICMAS, 372-379.