Le Backtracking dans les Réseaux de Contraintes Distribuées

Mustapha Belaïssaoui

ENCG
Université Hassan 1er
Km 3, Route Casa, B.P 658,
Settat, Maroc
Email :
Belaissa@Ibnsina.uh1.ac.ma
Tel : (212)(23) 40 10 63
Fax : (212)(23) 40 13 62

El Houssine Bouyakhf

Faculté de Sciences,
Université Mohamed V
Av. I.Batouta, B.P 1014
Rabat, Maroc

Bouyakhf@mtds.com
Tel : (212)(37) 77 89 73
Fax: (212)(37) 77 89 73

Résumé L'expression et le traitement de la distributivité ou de partage de Problèmes de Satisfaction de Contraintes (CSP) a fait l'objet de diverses extensions du cadre CSP classiques à des CSP Distribués (DCSP). Malheureusement, de tels problèmes sont en général NP-complets. Dans ce cadre nous avons développé une extension de l'algorithme d'Ordonnancement Distribué des Agents [HAM 98]. A partir de l'algorithme Intelligent BackTracking Distribué (DIBT) [HAM 98], nous avons proposé une Généralisation Optimale en envoi de messages (OGDIBT).
Mots-clés : Problèmes de Satisfaction de Contraintes Distribués, Backtracking Distribué, Problèmes NP-complets.

1 Introduction

Les Problèmes de Satisfaction de Contraintes Distribués (DCSP) constituent un outil intéressant de modélisation et de résolution pour beaucoup de problèmes d'Intelligence Artificielle et de Planification. Malheureusement, de tels problèmes sont en général NP-complets. Parmi les nombreuses méthodes étudiées afin de contourner cette difficulté, la méthode Backtracking distribuée est une technique de résolution des DCSP. Dans ce cadre, nous proposons une généralisation optimale en envois de message de l'algorithme DIBT (Distributed Intelligent BackTracking) [HAM 98]. Cette généralisation OGDIBT (Optimal Generalization DIBT) est basée sur l'ordonnancement selon une heuristique des agents d'un DCSP. Pour établir cet ordonnancement, nous proposons une Extension EDisAO de la méthode DisAO (Distributed Agents Ordering) [HAM 98]. EDisAO est défini pour accroître les performances des algorithmes de résolution des DCSP, en particulier OGDIBT.

2 Problèmes de Satisfaction de Contraintes Distribués (DCSP)

Un DCSP [HYU 2000] est un CSP dans lequel variables et contraintes sont distribuées entre plusieurs agents. Le problème consiste alors à trouver une combinaison consistante d'actions des agents. Différents modèles de résolution de ces CSP ont été récemment proposés. De nombreux problèmes d'application entrent dans ce cadre : allocation de ressources distribuée, ordonnancement distribué [QUE 98], recherche d'informations distribuées [EIS 2000]...

Définition 1 Un problème distribué de satisfaction de contraintes (DCSP) est un quadruplet P=(X, D, C, A) où :
- X est un ensemble fini de n variables $\{X_1,.., X_n\}$;
- D est l'ensemble des domaines associés à ces variables $\{D_1,.., D_n\}$;
- C est un ensemble fini de m contraintes sur les variables $\{C_1,.., C_m\}$;
- A est un ensemble fini d'agents $\{A_1,.., A_p\}$.

Chaque agent possède un sous ensemble des variables du problème de telle façon que A soit une partition de X. Chaque agent possède l'ensemble des données du problème relatives à son sous ensemble des variables (domaines, contraintes). Formellement, notons possède (a, x) : l'agent *a* possède l'élément de problème *x*.

$$\forall a \in A, \forall i \in X, \text{ possède}(a, i) \text{ ssi } (\text{possède}(a, D_i) \wedge \text{possède}(a, C_{i..j}).$$

Exemple 1 DCSP entre 2 agents A et B
A possède $X_A=\{ x_1,x_2 \}$, $D_A=\{\{1,2\},\{2\}\}$ et $C_A=\{(x_1 \neq x_3), (x_2=x_1)\}$
B possède $X_B=\{x_3\}$, $D_B=\{\{1,2\}\}$ et $C_B=\{(x_1 \neq x_3)\}$

3 Backtracking Distribué

Dans [YIK 90], [YDI 92] et [YH 98] les auteurs définissent les DCSPs comme des problèmes distribués que l'on cherche à résoudre. Il ne s'agit donc plus de CSP classiques que l'on distribue dans l'espoir de traitements plus efficaces. Leur approche s'insère dans le cadre du Cooperative Distributed Problem Solving (CDPS). Les auteurs définissent informellement les DCSP comme des CSP binaires où les variables sont réparties parmi différents agents. Ces agents communiquent par envoi de messages 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 [HAM 99].

3.1 Structuration des agents: 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 de la totalité 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, [HAM 98] a proposé son algorithme d'ordonnancement distribué DisAO (Distributed Agents Ordering).

3.1.1 Algorithme DisAO

Au départ, une phase d'échanges locaux permet aux agents de connaître la valeur de l'heuristique d'ordonnancement de chaque agent connecté. Ensuite, chaque agent (Self : l'agent exécutant l'algorithme) 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 : cet ensemble représente l'ensemble des accointances de l'agent Self, c'est à dire l'ensemble des agents du système avec lesquels Self partage au moins une contrainte inter-variable ;
- Level : table d'entiers destinée au tri des accointances supérieures ;
- 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.

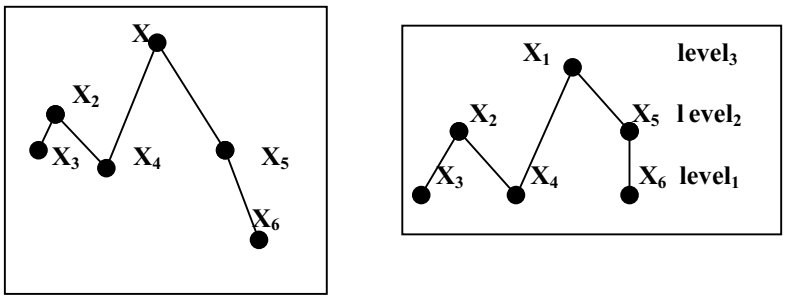
Algorithm: Distributed Agent Ordering

```
DisAO(in : f, Γ, op ; out: Γ⁻ et Γ⁺)
Début
  % partition de Γ
  Γ⁻ ←∅ ; Γ⁺←∅
  pour Agent_j∈ Γ faire
    si (f(Agent_j) op f(self)) alors
          Γ⁺← Γ⁺∪{Agent_j}
    sinon
          Γ⁻ ← Γ⁻∪{Agent_j}
  % ordonnancement de Γ⁻
  max←0
  pour (i=0; i<|Γ⁺|; i++) faire
     m←getMsg()
     si (m= value : v) alors
          si (max < v) alors max←v
  max++
  broadcast(Γ⁻, value : max)
  broadcast(Γ⁺, position : (max, self))
  pour (i=0; i<|Γ⁻|; i++) faire
     m←getMsg()
     si (m= position : (p, j)) alors level_j←p
  tri de Γ⁻ en fonction de level
Fin
```

DisAO prend en entrée Γ, f et op, il retourne en sortie Γ^- et Γ^+ qui sont respectivement les ensembles des agents précedents et des agents suivants. Après la construction des deux ensembles Γ^- et Γ^+, le calcul du niveau de l'agent dans la hiérarchie peut débuter. Ce niveau est représenté par l'entier max. Chaque agent conserve le niveau maximum transmis par Γ^+ et calcul son propre niveau en rajoutant 1 au maximum. Ensuite, chaque agent envoie sa propre position dans la hiérarchie vers Γ^- (message de type value) et vers Γ^+ (message de type position). Enfin, l'ordonnancement consiste dans la récupération et le stockage de chaque position dans la table level. Γ^- est ensuite trié en fonction de ces positions stockées dans la table level.

Exemple 2 $p=|X|$, chaque agent à une seule variable. $\forall i \in X, D_i=\{a,b\}$.

FIG. 1: Un DCSP et sa structuration utilisant l'heuristique max-degree



Le résultat de l'appel à DisAO par chacun des agents est le suivant :
$Agent_1$: $\Gamma^-=\emptyset$, $\Gamma^+=\{Agent_4, Agent_5\}$;
$Agent_2$: $\Gamma^-=\emptyset$, $\Gamma^+=\{Agent_3, Agent_4\}$;
$Agent_3$: $\Gamma^-=\{Agent_2\}$, $\Gamma^+=\emptyset$;
$Agent_4$: $\Gamma^-=\{Agent_2, Agent_1\}$, $\Gamma^+=\emptyset$;
$Agent_5$: $\Gamma^-=\{Agent_1\}$, $\Gamma^+=\{Agent_6\}$;
$Agent_6$: $\Gamma^-=\{Agent_5\}$, $\Gamma^+=\emptyset$.

3.2 L'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 effectuant 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^+ peut éventuellement modifier l'instanciation courante des agents informés. Cependant, la considération unique du lien unissant un agent à ses suivants ne suffit pas. En effet, considérons cela sur l'exemple 2 :

Supposons qu'initialement les cinq variables soient instanciées à la première valeur de domaine, a. Considérons maintenant, un retour initié par X_5, ce retour conduisant l'agent possédant X_1 à l'instanciation X_1=b. Il faut que cet agent informe maintenant ses suivants. L'information directe de X_5 n'est pas problématique car X_5 ne dépend que de X_1. L'information directe de X_4 l'est. En effet, X_2 est connectée à X_4 et précède X_1 dans l'ordre total Γ^- utilisé par X_4 lors de ces retours. Donc, en plus d'informer ses suivants X_4 et X_5, l'agent X_1 doit s'assurer de la remise à zéro de l'ensemble des agents précédents X_4et X_5, situés avant X_1 dans l'ordre total local Γ_4^- et Γ_5^-. Cet exemple permet d'illustrer que la relation d'accointances entre agents de niveaux l et l-1 ne suffit pas à assurer l'exhaustivité de la recherche. Nous étendons ici les deux ensembles Γ^- et Γ^+, afin d'assurer la complétude de l'exploration de l'arborescence [HAM 99]. 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 :

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

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

Dans l'exemple 2 (voir FIG. 1), les agents possèdent les ensembles suivants :

$Agent_1$: $\Gamma^-=\varnothing$, $\Gamma^+=\{Agent_2, Agent_4, Agent_5\}$;

$Agent_2$: $\Gamma^-=\{Agent_1\}$, $\Gamma^+=\{Agent_3, Agent_4\}$;

$Agent_3$: $\Gamma^-=\{Agent_2\}$, $\Gamma^+=\varnothing$;

$Agent_4$: $\Gamma^-=\{Agent_2, Agent_1\}$, $\Gamma^+=\varnothing$;

$Agent_5$: $\Gamma^-=\{Agent_1\}$, $\Gamma^+=\{Agent_6\}$;

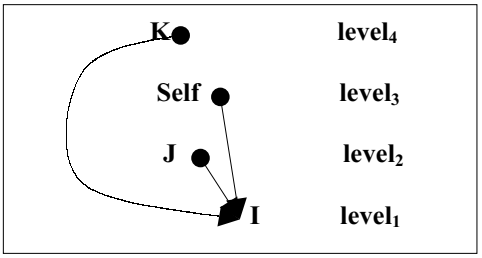
$Agent_6$: $\Gamma^-=\{Agent_5\}$, $\Gamma^+=\varnothing$.

3.2.1 Extension de DisAO: EDisAO

Après le partitionnement de Γ à Γ^- et Γ^+ par un appel à DisAO, ces deux ensembles sont ensuite étendus comme ci-dessus. L'algorithme décrit dans [HAM 99] est comme suit : Après l'application de DisAO, Self 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^-.

Analyse comportementale de l'extension Après que Self envoie à ses accointances précédentes son ensemble ordonné Γ^- (voir FIG. 2), il reçoit Γ_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^-. $Agent_k$ reçoit Γ_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 message d'insertion et l'agent self en a envoyé une. Donc, au total nous avons 3 demandes d'insertion.

FIG. 2: structure du graphe à 4 agents après l'application de DisAO



Communication optimale (minimisation du nombre de messages) Nous remarquons que l'Agent$_i$ envoie à Self, Agent$_j$ et l'Agent$_k$ son ensemble Γ_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^+ l'Agent$_j$ et dans Γ^- l'Agent$_k$ sans demande d'insertion. De même, L'Agent$_k$ affecte à son ensemble Γ_k^+ l'Agent$_j$ et Self. L'Agent$_j$ affecte à son ensemble Γ_j^- l'Agent$_k$ et Self. Les messages de demande d'insertion sont inutiles. En utilisant les insertions directes, suivant les niveaux, nous avons pu réduire le nombre de messages de demande d'insertion.

Algorithme EDisAO (Extension Distributed Agent Ordering) Dans chaque système, chaque agent Self utilise les mêmes structures de données que DisAO, ainsi que les messages de type extension (extension: Γ^-, level$_{self}$). level$_{self}$ est utilisé par les précédents de Self pour ordonner leur ensemble d'accointances précédentes.

Algorithm: Extension Distributed Agent Ordering

```
EDisAO(in : f, Γ, op ; out: Γ⁻ et Γ⁺)
  DisAO(f, Γ, op)
  broadcast(Γ⁻, Extension: (Γ⁻, level_self ))
  pour (i=0; i<|Γ⁺|; i++) faire  % Extension de Γ⁻ et Γ⁺
     m←getMsg()
     si (m= Extension: (Γ_i⁻ , level_i)) alors
        pour (j=0; j<|Γ_i⁻ |; j++) faire
           si level_i[Agent_j]< level_i[Self] alors
                    Γ⁺← Γ⁺∪{Agent_j}
           si level_i[Agent_j]> level_i[Self] alors
                    Γ⁻ ← Γ⁻∪{Agent_j}
                    level_self[Agent_j]← level_i[Agent_j]
  tri de Γ⁻ en fonction de level
Fin
```

3.3 Généralisation de DIBT

[HAM 98] a proposé son algorithme DIBT en utilisant le niveau de granularité le plus fin. Chaque agent a une seule variable ($p=|X|$). [HAM 99] a signalé que la généralisation à plusieurs variables est directe. Chaque agent propose simultanément à ses suivants une solution à son sous-problème. Il suffit donc d'affecter à la variable myValue (voir [HAM 98]) la solution locale recherchée par la généralisation de la primitive getValue().

3.3.1 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 allons apporter.

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.

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 qui 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.

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 3 ci-dessous).

3.3.2 Algorithme OGDIBT

Sur la base de la propriété précédente, OGDIBT modifie l'algorithme DIBT. Dans OGDIBT, chaque agent Self utilise les mêmes primitives et structures de données suivantes :

- D_{self} est le domaine de Self et myInstanciation c'est la courante instanciation de Self ;
- Instanciation, table stockant les instanciations prises par les accointances supérieures ;
- GetFirst (S), retourne le premier élément de l'ensemble S ;
- GetInstanciation(type) :

  - Si type = info, retourne la première instanciation de D_{self} compatible avec les instanciations de la table Instanciation ;
  - Si type = bt, retourne la première instanciation de D_{self} située après myInstanciation, compatible avec les instanciations de la table Instanciation ;
- merge(A,B), retourne l'ensemble ordonné union des deux ensembles A et B ;
- La fonction Compare, comparaison de deux instanciations et renvoie les variables qui ont les valeurs différentes. C'est une fonction de comparaison de deux listes.
- 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 précédente (voir algorithme: OptimalSendMsg).

Après le partitionnement de Γ et l'ordonnancement de Γ^-, Self cherche sa première solution et l'affecte à sa variable myInstanciation. Cette dernière est ensuite envoyée à l'ensemble Γ^+ par un message de type infoVal. La boucle commence par la réception de message m. Si le message est de type infoVal, Self stocke l'information dans la table Instanciation. Ensuite, il fait appel à la fonction OptimalSendMsg(myInstanciation, Info) (voir description ci-dessous). Si le message est de type btSet, Self compare son instanciation courante à l'instanciation rapportée dans le message. S'il y a égalité, le message est valide. Self fait appel à OptimalSendMsg(myInstanciation, bt).

OptimalSendMsg démarre par la recherche d'une nouvelle instanciation compatible avec la table Instanciation suivant le type de message (voir algorithme ci-dessous).

Si la nouvelle instanciation (myInstanciation) est différente de l'ancienne (oldinstanciation) et aussi de l'ensemble vide :

- Self fait appel à la fonction Compare. 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. 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 (myInstanciation) est égale à l'ensemble vide :

- Si type = info, un message de type btSet est adressé à l'agent nearest qui est le supérieur le plus proche. Un message de type btSet comprend Γ^-, la table Instanciation et les positions des agents de Γ^- dans la hiérarchie ;
- Sinon, si l'agent est une source ($\Gamma^-=\varnothing$) et si l'ensemble Set=$\varnothing$, l'insolubilité est détectée. Sinon, un retour arrière est construit en réutilisant l'ensemble de retour transmis. Self construit l'ensemble de retour followSet, union ordonnée de Γ^- et du Set (Set sans le premier élément). Un message btSet est alors transmis au premier élément de followSet. Si la demande de backtrack n'est pas adressée à un agent de Γ^-, Self se réinstancie à une instanciation compatible avec la table Instanciation et appellera d'une manière récursive OptimalSendMsg(myInstanciation, info).

Algorithm : Optimal Generalization Distributed Intelligent BackTracking (OGDIBT)

```
Début
  EDisAO(f, Γ, op) // on ordonne les agents
  nearest← getFirst(Γ⁻)
  myInstanciation← getInstanciation(info)
  broadcast(Γ⁺,infoVal:( myInstanciation, self))
  end← false
  tant que (!end) faire
    m← getMsg()
    si (m=stop) alors end← true
    si (m=infoVal: (localInstanciation,j)) alors
          instanciation[j]← localInstanciation
          optimalsendMsg(myInstanciation, info)
    si (m=btSet: (set, Instanciations)) alors
          si (Instanciations[self]=myInstanciation) alors
          optimalsendMsg(myInstanciation, bt)
Fin
```

Algorithm:OGDIBT OptimalSendMsg

```
OptimalSendMsg(in:  myInstanciation, type)
Début
   oldinstanciation← myInstanciation
   myInstanciation← getInstanciation(type)
   si (myInstanciation≠oldinstanciation) and (myInstanciation) alors
        differentes← compare(myInstanciation, oldinstanciation)
        broadcast(localise(Γ⁺,differentes), infoVal:( myInstanciation, self))
   sinon
        si (myInstanciation=∅) and (type= info) alors
             broadcast(nearest, btSet:( Γ⁻, Instanciation[Γ⁻]))
        si (myInstanciation=∅) and (type= bt) alors
             si (Γ⁻=∅ et Set =∅) alors
                broadcast(system, stop)
                end← true
             sinon
                followSet← merge(Γ⁻, Set)
                follow← getFirst(followSet)
                broadcast(follow, btSet:( followSet, instanciation[Γ⁻]∪
                          instanciations))
                si (follow∉ Γ⁻) alors OptimalSendMsg(myInstanciation,info)
Fin
```

Exemple 3 DCSP entre 3 agents A, B et C :
A : X_A={ A_1,A_2 }, D_A={{a,b},{b,c}} et C_A={(A_1≠A_2), (A_1=B_1), (A_1=C_1), (A_2=C_2)}
B : X_B={B_1,B_2)}, D_B={{a,b},{a,b}} et C_B={(B_1=B_2), (B_1=A_1)}
C : X_C={C_1,C_2)}, D_C={{a,c},{a,c}} et C_C={(C_1≠C_2), (A_1=C_1), (A_2=C_2)}

La structuration des agents par EDisAO en utilisant l'heuristique Max-degree :
$Agent_A$: Γ^-=∅, Γ^+={$Agent_B$, $Agent_C$};
$Agent_B$: Γ^-={$Agent_A$}, Γ^+=∅;
$Agent_C$: Γ^-={$Agent_A$}, Γ^+=∅ ;

Lors du backtracking distribué, l'agent A propose à ses suivants B et C la première solution à son sous-problème {(A_1,a),(A_2,b)}. Après la réception de (infoVal: ({(A_1,a),(A_2,b)},A)), l'agent B vérifie la compatibilité de sa solution locale {(B_1,a),(B_2,a)} et la solution locale de l'agent A. C_{A1B1}(a,a)=true donc, l'agent B n'envoie pas de message de type btSet.

Après la résolution de son sous-problème {(C_1,a),(C_2,c)}, l'agent C vérifie sa compatibilité avec celle de l'agent A. l'agent C génère un message à l'agent A de type (btSet:{A},{(A_1,a),(A_2,b)}. L'agent A reçoit ce dernier message et réinstancie ces variables par {(A_1,a),(A_2,c)}. Si nous appliquons la propriété précédente, la variable A_1 n'a pas changé de valeur et A_2 l'a changé. Donc l'agent A devra envoyer un message de type (infoVal: ({(A_1,a),(A_2,c)}, A)) tout simplement à l'agent C par ce que la fonction localise renvoie comme résultat l'ensemble {Agent C}.

3.3.3 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.

Spatiale |D|=d. Un agent est connecté par relation de contrainte à n-1 autres agents donc, la table |Instanciation|=n-1. Chaque agent nécessite donc un espace O(n) pour stocker son sous-problème du DCSP. Si nous considérons la totalité du système distribué, l'espace nécessaire au stockage est $O(n^2)$.

Temporelle 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 Conclusion

Dans le domaine des Problèmes de Satisfaction de Contraintes Distribués (DCSP), nous avons consacré notre étude à la généralisation optimale en envoi de message de l'algorithme de backtracking distribué DIBT (Distributed Intelligent BackTracking) [HAM 98]. 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). L'algorithme à une complexité $O(d^n)$.

Références

[YIK 90] 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, chapter 9.

[YDI 92] 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, pages 614-624.

[HAM 98] Youssef, H., Backtracking in distributed constraint networks, ECAI'98.

[QUE 98] P.Queloz, projet ISACOM, rapport pour le FNRS, 98.

[YH 98] Yokoo, M. et Hirayama, K. (1998). Distributed constraint satisfaction algorithm for complex local problems. In ICMAS, pages 372-379.

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

[HYU 2000] Hyuckchul jing et al, on modeling argumentation as distributed constraint satisfaction, CP'2000, (voir liawww.epfl.ch/cp2000dcs)

[EIS 2000] Eisenberg, C., Integrating distributed and heterogeneous organisation using constraint programming, CP'2000, (voir liawww.epfl.ch/cp2000dcs)