

ACADÉMIE DE MONTPELLIER
UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

Mémoire de Stage de Master

SPÉCIALITÉ : **Recherche en Informatique**
Mention : **Informatique, Mathématiques, Statistiques**

effectué au LIRMM/INFO

—
sous la direction de CHRISTIAN BESSIÈRE

Les CSP distribués
Confidentialité et évaluation

par

Arnault Ioualalen

Soutenu le 18 juin 2007

Résumé

Les CSP distribués (DisCSP) proposent un modèle de résolution de contraintes impliquant des agents distincts reliés par des contraintes. Toutefois à ce jour peu d'algorithmes de CSP distribués prennent en compte la confidentialité de valeurs ou de contraintes que les agents souhaiteraient avoir. Le but est de trouver une solution consistante pour les agents sans toutefois révéler ni les valeurs prises par les agents, ni les contraintes les reliant. Dans ce rapport nous présentons un protocole définissant de nouveaux algorithmes pour les CSP distribués qui permettent de protéger à la fois les valeurs de variables contrôlées par un agent, et les contraintes qui les relient. Ce protocole se décline en trois algorithmes : l'algorithme initial et de 2 améliorations successives de celui ci, chacune présentant une nouvelle protection améliorant la précédente. Ces 3 algorithmes sont étudiés avec le système VPS [9] qui permet de quantifier la connaissance d'information privée qu'un agent peut acquérir de ses concurrents. Un modèle d'analyse complémentaire est présenté pour lever partiellement l'incertitude de réception des messages dû à l'asynchronisme des algorithmes. Enfin nous verrons dans les différentes améliorations possibles à ce protocole afin de diminuer encore la quantité d'information dévoilée.

Table des matières

I	État de l’art	3
1	Problématique	3
2	Définitions et outils	4
2.1	Les agents	4
2.2	Le CSP	4
2.3	Le DisCSP	5
2.4	Le modèle de <i>Contraintes Distribuées Non-partagées</i>	6
2.5	La mesure de confidentialité : le système VPS	6
2.5.1	Connaissance qu’a un agent de ses concurrents	7
2.5.2	Connaissance qu’ont des concurrents sur un agent	7
2.5.3	Quantification de l’évolution de la connaissance	8
2.5.4	Unification	8
3	“La famille ABT”	9
3.1	L’algorithme ABT	9
3.1.1	Présentation	9
3.1.2	Principe	9
3.1.3	Confidentialité dans ABT	10
3.2	Les différentes adaptations d’ABT	10
II	Le protocole 3	12
4	Présentation	12
4.1	Principe du protocole 1	12
4.2	Principe du protocole 2	13
4.3	Le protocole 3	13
5	Analyse de la confidentialité du protocole 3	15
5.1	Analyse du comportement des agents	15
5.2	Application de VPS	16
5.2.1	Remarques	16
5.2.2	Application pour un agent supérieur	17
5.2.3	Application pour un agent inférieur	18
5.2.4	Cas particulier d’acquisition de connaissance simplifié	19
5.3	Connaissance des “demi-contraintes” d’un agent	19
5.4	Conclusion	20
III	L’algorithme DisFC-1ph	21

6	Présentation	21
6.1	Principe	21
6.2	Algorithme de DisFC	22
7	Modèle d’analyse du comportement des agents	24
7.1	Principe	24
7.2	Définitions	25
7.3	La segmentation	25
7.4	Applications	27
7.4.1	Application pour un agent supérieur	27
7.4.2	Application pour un agent inférieur	29
7.4.3	Analyse des relations \mathcal{R}	30
7.5	Conclusion	30
IV	L’algorithme Ring_AB_T	32
8	Présentation	32
8.1	Principe	32
8.2	Algorithme de Ring_AB_T	35
9	Analyse de l’algorithme Ring_AB_T	40
9.1	Quantification des nogoods éliminés	40
9.2	Connaissance qu’a un agent supérieur	41
9.3	Connaissance qu’ont les agents sur la chaîne	41
9.3.1	Les “nogoods de chaîne” et la segmentation :	41
9.3.2	Connaissance des agents sur la chaîne hormis le dernier	42
9.3.3	Connaissance du dernier agent de la chaîne	43
9.4	Conclusion	43
V	Perspectives	45
10	Adaptation du protocole 3 aux problèmes d’optimisation	45
11	Concernant l’ordre des agents	45
11.1	Une permutation sur la chaîne d’agents	45
11.2	Un ordre dynamique d’agent	45
12	Une analyse de déviance de comportement	46
A	Algorithme d’AB_T	47
B	Algorithme du protocole 3	50

Première partie

État de l’art

1 Problématique

Aujourd’hui de nombreux problèmes du monde réel peuvent être résolus en un temps humainement acceptable grâce à l’utilisation de CSP (cf. 2.2). Contrairement à un style de résolution naïf où les solutions sont énumérées, les CSPs utilisent des techniques de filtrage qui permettent de limiter l’exploration de l’espace de recherche. Ces techniques viennent de nombreux travaux destinés à mieux modéliser les problèmes traités, et donc filtrer plus efficacement les solutions. Toutefois certains problèmes restent encore à résoudre : par exemple dans un problème où deux personnes veulent se donner un rendez-vous, leur rencontre peut rentrer en conflit avec d’autres rendez-vous. Une approche centralisée avec un CSP classique ne peut pas deviner à priori quels conflits apparaîtront et devra donc demander systématiquement des informations aux personnes mis en cause. Quand on sait que toute personne est à six poignées de mains de n’importe qui dans le monde on comprend que cette approche peut rapidement impliquer une bonne partie de la population mondiale ! À l’inverse une approche distribuée (i.e. avec un DisCSP, cf. 2.3) peut être plus judicieuse, car les changements et les conflits locaux peuvent être traités et résolus de manière locale par des agents communicants (cf. 2.1).

Une autre motivation concernant l’usage de DisCSP se situe dans la possibilité de donner aux agents en plus de leur autonomie (i.e. asynchronisme) une part de confidentialité. En effet un domaine classique d’application des DisCSP est le problème du rendez-vous (“meeting scheduling problem”). Chaque agent dispose d’un emploi du temps (celui d’un président par exemple) qu’il ne souhaite pas révéler aux autres agents (i.e. à d’autres présidents) car il constitue une information sensible. Le but est de trouver un lieu et un créneau libre pour chaque agents qui soit compatible avec tous les agents, en tenant compte des contraintes temporelles de déplacement pour atteindre le lieu choisit. On comprend ici qu’il est nécessaire de protéger à la fois les valeurs possédées par les agents (i.e. l’emploi du temps qu’ils contrôlent) et les contraintes qui les relient (car elle permettent de déduire des informations sur les valeurs dans cet exemple).

Des premiers travaux [4] ont permis de montrer la balance implicite entre l’efficacité d’un algorithme et la confidentialité qu’il octroie aux agents impliqués dans le processus. En effet plus un algorithme protège des données (i.e. diminue leur partage) au plus la propagation de contraintes diminue, de ce fait l’algorithme passe du temps à explorer des parties de l’espace de recherche qui aurait été éliminées d’office si la propagation avait pu se faire totalement. Toutefois il est possible de pallier cette perte d’efficacité par l’asynchronisme des algorithmes. En effet le temps perdu à explorer des parties inutiles de l’espace de recherche peut être négligé si d’autres parties, plus prometteuses, de l’espace

de recherche sont explorées au même instant.

La confidentialité des algorithmes de résolution de DisCSP fut en premier lieu exprimée en terme de moyens cryptographiques ([22]) par Yokoo *et al.* Toutefois l'usage de la cryptographie pose le problème du temps nécessaire au cryptage et au décryptage de l'information à cacher . Ces opérations s'effectuent généralement par des exponentiations et ne permettent pas une implémentation efficace sur des ordinateurs. Il semble logique donc de chercher de nouvelles méthodes de protection plus orientées vers l'algorithmique et des structures de données plus efficaces. C'est dans cette optique que le protocole 3 fut développé, sa présentation fait l'objet de la partie 4.

2 Définitions et outils

2.1 Les agents

Les agents résultent de la modélisation informatique d'entités concrètes comme des personnes ou des machines. Les agents sont des entités informatique indépendantes disposant de leur propres ressources (mémoire, processeur, moyen de communication, ...) et d'un comportement que l'on considère en générale comme unique (ce qui ouvre une variété quasi infinie de comportement dans une société d'agents). Les agents sont disposés dans un environnement avec lequel il peuvent interagir via des actions que leur comportement peut choisir, par exemple les agents peuvent prendre un objet dans l'environnement afin de s'en servir pour une action, ou bien y déposer quelque chose (une phéromone par exemple). Un point crucial pour une société d'agents est la communication ; En effet les agents peuvent interagir entre eux soit directement par des envois de messages, soit via leur environnement en y laissant une trace de leur action qui sera utile aux autres agents.

Dans le cadre de ce rapport les agents sont des programmes informatiques capables de communiquer avec leur voisins sur le graphe de contrainte, et qui exécutent un algorithme de résolution de contraintes sur les variables que l'agent contrôle (cf. 2.3).

2.2 Le CSP

Un CSP se définit comme un triplet de 3 ensembles :

- $\chi = \{X_1, \dots, X_N\}$ un ensemble de variables, cet ensemble est totalement ordonné.
- $D = \{D(X_1), \dots, D(X_N)\}$ une collection de domaines finis, ils représentent l'ensemble de valeurs que chaque agent peut prendre au court du processus.
- C un ensemble de contraintes, chacune portant sur un sous ensemble de variables de χ . Soit si $C_i \in C$ est d'arité k alors $var(C_i) = (X_i, \dots, X_{i+(k-1)})$. Chaque contrainte $C_i \in C$ définit un ensemble de tuples de valeurs autorisés pour les variables de $var(C_i)$. Une contrainte d'arité k de C_i se compose de tuples $(v_i, \dots, v_{i+(k-1)})$ tel que $\forall j \in \{0..k-1\} \Rightarrow v_{i+j} \in D(X_j)$.

Une solution d'un CSP est affectation de valeur à chacune des variables telle qu'aucune contrainte ne soit violée (pour un problème de décision), ou que le coût de violation de cette affectation soit minimal (pour un problème d'optimisation). La littérature est abondante sur la manière de procéder à cette affectation, deux mécanismes principaux sont à retenir :

- la propagation : dès qu'une variable est affectée à une valeur on propage cette instantiation afin de retirer les valeurs d'autres variables qui rentrent en conflit avec cette affectation.
- le BackTrack : quand une variable à un domaine vide car les affectations précédentes ont interdites toutes ses valeurs, alors le système modifie une affectation déjà existante.

Il est à noter que de nombreuses heuristiques ont été mis en place pour améliorer la propagation, le mécanisme de BackTrack.

2.3 Le DisCSP

Un DisCSP est un CSP où les variables sont distribuées entre plusieurs agents distincts. Un DisCSP se définit comme un tuple de taille 5 noté (χ, D, C, A, ϕ) , où χ, D, C se définissent comme pour les CSP classiques et où A et ϕ sont définis par :

- $A = \{1, \dots, p\}$ un ensemble ordonné d'agents.
- $\phi : \chi \rightarrow A$ une fonction qui associe chaque variable à l'agent qui la possède.

Les agents communiquent par des envois de messages ayant les postulats suivants [20] :

- Un agent ne peut envoyer un message que s'il connaît l'adresse du destinataire.
- Le délai de réception d'un message est borné mais aléatoire. Pour une paire d'agents donnée, les messages sont reçus dans l'ordre dans lequel ils ont été envoyés.

On désigne par C_{ij} une contrainte reliant les agents X_i et X_j . La distribution des variables de χ forme une bipartition de C en $C_{intra} = \{C_{ij}/\phi(X_i) = \phi(X_j)\}$ et $C_{inter} = \{C_{ij}/\phi(X_i) \neq \phi(X_j)\}$. On parle alors d'ensemble de contraintes *intra*-agents et d'ensemble de contraintes *inter*-agents. Il est à noter qu'une contrainte C_{ij} de C_{intra} n'est connue que de l'agent détenteur de X_i et X_j . On considère généralement que les contraintes *inter*-agents C_{ij} sont connues de $\phi(X_i)$ et $\phi(X_j)$ ([20], [8]).

Dans le cadre de ce rapport nous considérons ϕ comme une fonction bijective de ce fait on désigne par le terme X_i à la fois la variable et l'agent qui la possède.

L'ensemble A étant totalement ordonné pour chaque agent on définit les ensembles Γ^- et Γ^+ . $\Gamma^-(X_i)$ désigne l'ensemble des agents contraints par X_i et qui apparaissent plus haut que X_i dans l'ordre. Inversement $\Gamma^+(X_i)$ désigne l'ensemble des agents contraints par X_i qui apparaissent plus bas que X_i dans l'ordre.

Comme dans le cadre d'une résolution centralisée d'un CSP, une solution d'un

DisCSP est une affectation des variables qui ne viole aucune contrainte. Toutefois les DisCSPs sont résolus par l'action collective et coordonnée des agents de A , chacun effectuant en parallèle un processus de satisfaction de contraintes. Toutefois du point de vue de la confidentialité le modèle des DisCSPs n'est pas suffisamment expressif, et des modifications doivent être apportées, ces modifications font l'objet de la section 2.4.

2.4 Le modèle de *Contraintes Distribuées Non-partagées*

Le modèle de contrainte des DisCSPs n'est pas suffisant pour modéliser efficacement un processus collaboratif où chaque agent dispose de préférences propres (i.e. de contraintes propres) et où le but est de trouver une affectation des variables qui soit consistante avec toutes les préférences de chaque agent. Considérons la contrainte $C_{ij} \in C$ portant sur l'agent X_i et X_j , les tuples contenues dans C_{ij} représentent les préférences des deux agents en une seule table. Pourtant il semble logique du point de vue de la confidentialité que les agents mutuellement contraints ne souhaitent pas à priori présenter à leurs concurrents leur table de tuples rejetés. Il est plus intuitif d'exprimer les contraintes par les tuples qu'elles interdisent car du point de vue de la confidentialité il est plus juste de considérer que les agents s'interdisent des valeurs plutôt que de s'en imposer.

Le modèle de *Contraintes Distribuées Non-partagées*, permet de scinder les préférences de chaque agent tout en conservant la structure du graphe de contraintes. En outre chaque $C_{ij} \in C$ se décompose en deux "demi-contraintes" notées : $C_{i(j)}$ (possédée par X_i) et $C_{(i)j}$ (possédée par X_j). Ces deux demi-contraintes se définissent de la manière suivante :

- $C_{i(j)}$ contient la liste des tuples de $D(X_i) \times D(X_j)$ que X_i interdit.
- $C_{(i)j}$ contient la liste des tuples de $D(X_i) \times D(X_j)$ que X_j interdit.
- $C_{i(j)} \cup C_{(i)j} = C_{ij}$ (la fusion des deux contraintes redonne la contrainte initiale)

Ce modèle est une adaptation du modèle *Partially Known Constraints* (PKC, [3]) toutefois nous avons préféré considérer qu'une demi-contrainte $C_{i(j)}$ ne contienne que l'ensemble de tuples interdits (par l'agent X_i en l'occurrence) plutôt que 3 ensembles de tuples (les tuples admis, les tuples interdits et les tuples inconnus).

2.5 La mesure de confidentialité : le système VPS

Le système *Valuation Per State* (VPS, [9]) considère des agents distincts possédant des informations privées que l'on nomme "états", les agents communiquent et s'échangent des informations, le but du système VPS est de formaliser la connaissance que les agents peuvent obtenir à propos de leurs concurrents durant la communication. Le système VPS considère que les agents possèdent un nombre fini d'états différents qui sont initialement connus de tous les agents, toutefois l'état courant choisit par un agent durant le processus constitue l'information privée.

2.5.1 Connaissance qu'a un agent de ses concurrents

On note par S_i les états possibles de l'agent X_i , $S_i = \{v_1, \dots, v_p\}$
L'ensemble de tous les états possibles pour les concurrents de X_i se note :

$$\mathbb{S}_i = S_1 \times \dots \times S_{i-1} \times S_{i+1} \times \dots \times S_N$$

On note $K_i = \text{card}(\mathbb{S}_i) = \prod_{z \neq i} \text{card}(S_z)$, le nombre de combinaisons d'états possibles.

Pour chaque agent X_i on construit un vecteur de probabilité noté $\mathbb{P}^i(\mathbb{S}_i)$ tel que pour tout $E_i \in \mathbb{S}_i$ on associe une probabilité que l'on note $P^i(E_i)$. $P^i(E_i)$ exprime la probabilité que les agents $X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_N$ soient dans l'état E_i .

D'où la définition de $\mathbb{P}^i(\mathbb{S}_i) = [P^i(E_1) \dots P^i(E_{K_i})]$.

Il est à noter de par la nature des $P^i(E_k)$ que :

- $\forall k \in [1..K_i], 1 \geq P^i(E_k) \geq 0$
- $\sum_{k \in [1..K_i]} P^i(E_k) = 1$
- initialement on pose $\forall k \in [1..K_i] \Rightarrow P^i(E_k) = \frac{1}{K_i}$, en effet tous les états sont équiprobables au début de la négociation.

On note $\pi_i E_k$ la projection d'un tuple E_k de \mathbb{S}_i sur un agent X_i . Cet opérateur permet de récupérer la valeur d'un agent (ici X_i) en fonction d'un état de \mathbb{S}_i donné.

Le système considère donc pour un agent donné (ici X_i) les probabilités de toutes les combinaisons possibles d'états que peuvent prendre tous les autres agents. Il est important de noter qu'il est inefficace de chercher à réduire l'ensemble $(S)_i$ en ne considérant que les agents voisins de X_i dans le graphe de contraintes. En effet il est possible qu' X_i reçoive des informations concernant un agent auquel il n'est pas relié dans le graphe (par un nogood par exemple), il est donc utile de considérer tous les agents à la fois.

2.5.2 Connaissance qu'ont des concurrents sur un agent

Réciproquement on peut exprimer la connaissance qu'a l'agent X_j de l'agent X_i en fonction des états possibles de X_i , on note cette connaissance $\mathbb{P}_i^j(S_i)$.

De manière symétrique à X_i on pose un vecteur de probabilité pour chaque état de X_i , que l'on note $\mathbb{P}_i^j(S_i) = [P_i^j(v_1) \dots P_i^j(v_p)]$.

On définit chaque $P_i^j(v_k)$ en considérant la probabilité de tous les états de $(S)_j$ où X_i est effectivement dans l'état v_k et à en faire la somme. Formellement on pose

$$\forall v_k \in S_i : P_i^j(v_k) = \sum_{E \in \mathbb{S}_j : \pi_i E = v_k} P^j(E)$$

En généralisant la définition ci dessus, on peut caractériser la connaissance qu'ont tous les agents de l'agent X_i . On l'exprime sous la forme d'un vecteur

de probabilités que l'on note $\mathbb{P}_i(S_i)$, et qui se définit comme suit :

$$\mathbb{P}_i(S_i) = [\mathbb{P}_i^1(S_i), \dots, \mathbb{P}_i^{i-1}(S_i), \mathbb{P}_i^{i+1}(S_i), \dots, \mathbb{P}_i^N(S_i)]$$

Le système VPS permet donc aussi de modéliser la connaissance qu'ont les concurrents de l'agent considéré. De la sorte un agent est capable de déterminer quel est son concurrent le mieux informé, ou inversement de connaître le concurrent dont l'agent connaît le mieux l'état. L'étape suivante consiste donc à définir la fonction permettant de quantifier ces connaissances.

2.5.3 Quantification de l'évolution de la connaissance

Le système VPS permet d'explicitier à une date donnée la connaissance possédée par les agents, il est logique ensuite de chercher à mesurer l'évolution de cette connaissance durant la négociation. Pour cela on considère le nombre d'états dont les probabilités sont devenues nulles durant le processus. Les états dont la probabilité est nulle expriment le fait que l'agent sait avec certitude que l'état concerné n'est pas celui dans lequel se trouve ses concurrents. Quand un agent a pu éliminer tous les états sauf un il sait avec certitude dans quel état se trouve ses concurrents (l'état restant a une probabilité de 1 car $\sum_{k \in [1..K_i]} P^i(E_k) = 1$).

On utilise pour dénombrer le nombre de probabilités nulles une fonction indicative, notée I , et que l'on définit comme suit : $I : [0..1] \rightarrow \{0, 1\}$

Pour une probabilité non nulle I renvoie 0, et pour une probabilité nulle I renvoie 1.

Les probabilités nulles nous permettent de quantifier la réduction de l'ensemble des combinaisons d'états possibles. Toutefois d'autres fonctions indicatives peuvent être utilisées, par exemple on peut construire des fonctions dont le seuil n'est pas à zéro mais à une valeur α donnée par l'utilisateur.

Quelque soit la fonction indicative I que l'utilisateur souhaite utiliser il lui faut ensuite construire la fonction d'agrégation qui permet de valeur globalement l'évolution de la connaissance des agents.

2.5.4 Unification

On note \mathbb{V} la fonction d'agrégation qui évalue l'évolution globale des probabilités de chaque agent à l'aide de la fonction indicative I . Cette fonction permet de quantifier l'évolution globale de la connaissance d'un agent. Le système VPS ne propose pas de fonction unique d'agrégation, il laisse l'utilisateur libre d'adapter la fonction à ses exigences.

Une définition possible de \mathbb{V} pour un agent X_i est de poser :

$$\mathbb{V}_i(\mathbb{P}_i(S_i)) = \sum_{j \neq i} \sum_{v_i \in S_i} I_{\{\mathbb{P}_i^j(v_i) > 0\}}$$

Autrement dit, l'agent X_i comptabilise le nombre d'états concernant tous ses concurrents dont la probabilité est nulle.

Enfin on peut évaluer l'évolution de la connaissance qu'a un agent après une négociation en la comparant avec la connaissance avant la négociation, i.e. en considérant : $\mathbb{V}_{i,end}(\mathbb{P}_i(S_i)) - \mathbb{V}_{i,begin}(\mathbb{P}_i(S_i))$.

On constate que le but d'un algorithme censé protéger l'état prit par un agent X_i devra donc minimiser la différence entre $\mathbb{V}_{i,begin}$ et $\mathbb{V}_{i,end}$.

3 “La famille ABT”

3.1 L'algorithme ABT

3.1.1 Présentation

La plupart des recherches sur les algorithmes de résolution de DisCSP ont été focalisées sur un principe d'exploration complète et asynchrone de l'espace de recherche. Ces algorithmes font agir les agents en parallèle et leurs communications s'effectuent, si besoin est, pour maintenir la consistance des affectations de variables qu'ils contrôlent. L'algorithme fondateur est nommé “Asynchronous BackTracking” (ABT) et fut publié par Yokoo [21]. L'algorithme ABT effectue une exploration complète de l'espace de recherche. Cet algorithme est considéré comme une référence pour la conception de tout nouvel algorithme de résolution de DisCSP.

ABT fut initialement publié pour résoudre des problèmes de décision de manière asynchrone il fut par la suite étendu pour résoudre des problèmes d'optimisation ([14], [15]). La version concernant les problèmes d'optimisation a servi de fondement pour de nombreux algorithmes de résolutions de DisCSP qui sont détaillés dans la section 3.2. Cependant très peu d'effort ont été porté sur ces améliorations pour y intégrer une notion de confidentialité.

Dans toute la suite de ce rapport nous considérons des réseaux de contraintes binaires car l'algorithme ABT est formulé pour des réseaux de ce type.

3.1.2 Principe

L'algorithme ABT est totalement asynchrone et procède par des envois de messages qui contiennent soit :

- l'affectation courante de l'agent émetteur. Ces messages sont notés **ok?**.
- la justification d'un BackTrack. Ces messages sont appelés nogoods et notés **ngd**.
- une demande de lien. Ces messages particuliers peuvent être évités [2] toutefois ils permettent de ne pas ré explorer plusieurs fois une même partie de l'espace de recherche en rajoutant dynamiquement des liens dans le graphe de contraintes. Ces messages sont appelés “add-link” et notés **adl**.

Dans ABT les agents sont totalement ordonnés selon un ordre de priorité. Cet ordre permet de déterminer le receveur d'un nogood (i.e. l'agent devant changer sa valeur à cause d'un BackTrack). Quand un agent choisit une valeur il la communique à ses voisins inférieurs dans le graphe de contrainte via un message **ok**?. Le choix de cette valeur fait en sorte de ne pas violer les contraintes reliant l'agent à ses voisins supérieur dans le graphe de contrainte. Quand un agent reçoit un message **ok**? il vérifie que sa valeur ne rentre pas en conflit avec la nouvelle valeur reçue, sinon il change sa valeur afin de ne plus être en conflit et prévient ses voisins inférieurs du changement. Si l'agent a son domaine qui est vidé par la nouvelle valeur reçue il envoie un nogood au plus proche voisin supérieur dans le graphe de contraintes qui est responsable de cet échec.

L'algorithme exacte d'ABT est présenté en annexe A.

3.1.3 Confidentialité dans ABT

L'algorithme ABT n'est pas conçu pour fournir une notion de confidentialité aux agents qui sont mis en jeu. C'est pour cela que l'algorithme présenté en annexe A ne prend pas en compte le modèle de "demi contraintes" (cf. 2.4). En outre ni la valeur des variables ni les contraintes des agents ne sont protégées. En effet les agents s'échangent leurs valeurs en clair et les contraintes sont entièrement connues des agents inférieurs qui adaptent leurs valeurs en fonction des valeurs reçues de leur prédécesseur dans l'ordre.

On peut noter que dès la version initiale d'ABT le contenu d'un nogood permet à un agent de recevoir des informations qui concernent des concurrents inconnus pour lui (en terme de voisinage de graphe). Ce cas de figure justifie l'usage des messages **adl** ("add-link") qui permettent de connecter des agents non contraints jusque là.

On constate donc que l'algorithme dévoile systématiquement aux agents voisins les contraintes et les valeurs prises, mais que ce voisinage peut s'étendre dynamiquement lors de l'exécution, ce qui augmente encore la perte de confidentialité. Toutefois il est possible de se passer des messages **adl** [2] en perdant un peu d'efficacité de l'algorithme qui risque d'explorer plusieurs fois la même portion d'espace.

3.2 Les différentes adaptations d'ABT

ABT sans ajout de lien : dans cette version [2] d'ABT il est possible de ne plus traiter l'ajout de lien (message de type **adl**), si après avoir envoyé un nogood tous les nogoods stockés qui sont reliés à une variable inconnue sont immédiatement retirés de l'ensemble de nogoods stockés. Toutefois si ces nogoods stockés étaient nécessaire pour trouver une solution, l'algorithme les re-générera ultérieurement, de ce fait l'algorithme explorera à nouveau une partie de l'espace de recherche et deviendra donc moins efficace. Néanmoins cette version de l'algorithme ABT a le mérite d'éviter de mettre des agents initialement non mutuellement contraints en relation, ce qui apporte un gain sur la confidentialité.

Adopt : L’algorithme *Adopt* [12] permet de résoudre des problèmes d’optimisation sur des DisCSPs. Il propose une décomposition du graphe de contrainte en un “DFS tree“ (Depth First Search tree) ce qui permet de garantir que les nogoods générés transiteront uniquement sur la branche d’agents qui ont induit ce nogood. De plus un mécanisme de seuil est implémenté pour les BackTracks afin d’éviter de traiter successivement le nogood un trop grand nombre de fois.

OptAPO : l’algorithme *Optimal Asynchronous Partial Overlay* [10] propose initialement d’identifier les zones de l’espace de recherche particulièrement dures à traiter et de les résoudre de manière centralisée grâce à un agent médiateur. Cet agent médiateur s’intercale entre les agents présents dans la partie de l’espace ciblé et calcule localement la solution optimale pour lui et ses voisins en utilisant un algorithme de “Branch and Bound” centralisé. Ce système octroie un gain significatif de temps comparé à l’algorithme *Adopt* quand le délai d’acheminement d’un message sur le réseau est élevé.

AAS : l’algorithme *Asynchronous Aggregation Search* ([16], [17]) propose un modèle dual des DisCSPs où les agents contrôlent les contraintes plutôt que les variables. Pour ce faire les agents divisent l’espace des combinaisons de valeur de leur contraintes en sous groupes équivalents tels que pour toutes combinaisons d’un même sous-groupe, chacune d’elle implique un coût identique. Enfin les agents s’échangent des agrégations de plusieurs valeurs afin de diminuer le nombre de messages. Cet algorithme permet de gagner plusieurs ordres de magnitude sur des problèmes générés aléatoirement [17]. Il est à noter que les agrégations peuvent faire l’objet de méthode de cryptage comme présenté pour l’algorithme SSDPOP [25].

AWC : l’algorithme *Asynchronous Weak – Commitment search* permet de ré agencer dynamiquement l’ordre des agents, afin de cibler et traiter en priorité les parties difficiles de l’espace de recherche. Pour cela les agents qui envoient un BackTrack sont automatiquement placés en premier dans l’ordre. Le problème principale de cette amélioration est la place exponentielle requise pour stocker tous les nogoods afin de garantir la terminaison et la complétude de l’algorithme.

ABTR et ABT_DO : les algorithmes ABTR [18] et plus récemment ABT_DO [24] sont des extensions de AWC [23]. Dans ces algorithmes les agents ne sont capables que de réordonner les agents qui sont inférieurs à eux dans l’ordre. Ces algorithmes ne stockent plus un nombre exponentiel de nogoods toutefois il ne permettent pas pour autant d’obtenir un gain d’efficacité significatif par rapport à un algorithme centralisé.

DisFC-1ph l’algorithme DisFC-1ph [3] fait l’objet de la section III de ce rapport. Il est le premier algorithme basé sur le protocole 3 permettant d’inclure enfin à ABT une notion de confidentialité de valeurs **et** de contraintes.

Deuxième partie

Le protocole 3

4 Présentation

Le protocole 3 a été conçu afin de permettre premièrement aux agents de posséder leur contraintes propres sans avoir à les présenter aux agents concurrents, et deuxièmement de protéger la valeur prise par les agents en ne la faisant plus transiter directement dans les messages **ok**?. Ces deux améliorations permettent de pallier aux défauts de l'algorithme ABT exposés en section 3.1.3. Le protocole 3 est issu de la fusion de deux premiers protocoles exposés ci après.

4.1 Principe du protocole 1

Le but du protocole 1 est de protéger la valeur prise par un agent. Dans l'algorithme ABT les agents s'échangent directement leur valeur via les messages **ok**?. Ils adaptent ensuite en fonction des valeurs reçues dans les messages **ok**? leur propres valeurs de manière à être consistant avec leurs voisins supérieurs. Quand deux agents X_i et X_j souhaitent accorder leur valeur selon une contrainte C_{ij} ils peuvent aussi s'échanger des propositions jusqu'à atteindre un accord, plutôt que de dévoiler leur valeur et déduire automatiquement un accord possible. Pour ce faire l'agent X_i peut proposer à l'agent X_j l'ensemble de valeurs de $D(X_j)$ qui sont compatibles avec la valeur v_i prise par X_i . Afin de déduire cet ensemble de valeurs compatibles, l'agent X_i doit connaître la contrainte C_{ij} le reliant à X_j .

Le protocole 1 fonctionne selon ce principe : les agents choisissent leur valeur dans l'ensemble de valeurs que leur proposent leurs agents supérieurs, et envoient ensuite les ensembles de valeurs compatibles avec leur choix de valeur à ses agents inférieurs. Quand un agent n'a plus de valeur valide, par exemple s'il reçoit deux ensembles disjoints de valeurs compatibles, un mécanisme de BackTrack est émis comme dans ABT afin de faire changer la valeur de l'agent supérieur le plus proche en cause.

Formellement on a pour chaque agent X_i la propriété suivante sur les ensembles de valeurs compatibles notés $D_{v_i}(\cdot)$ de ses voisins inférieurs :

$$(\forall i \in A / X_i = v_i) \wedge (\forall X_j \in \Gamma^+(X_i) / X_j = v_j) \Rightarrow v_j \in D_{v_i}(X_j)$$

On constate que les agents dans le protocole 1 s'imposent les valeurs compatibles afin de garantir la consistance des affectations, sans toutefois présenter directement leurs valeurs à leurs voisins inférieurs.

Le principale défaut de ce protocole réside dans le fait que pour déduire une ensemble $D_{v_i}(X_j)$ l'agent X_i qui a prit la valeur v_i doit connaître intégralement la contrainte C_{ij} qui le relie à l'agent X_j . De plus en terme de confidentialité le fait que les agents s'imposent mutuellement les valeurs compatibles leur permet de déduire l'ensemble de valeurs dans lequel l'agent ne sera jamais, ce qui limite la connaissance que les agents ont les uns des autres sans toutefois l'annuler.

4.2 Principe du protocole 2

Le but du protocole 2 est de protéger les “demi-contraintes” (cf. 2.4) que possède chaque agent. Dans l’algorithme ABT les contraintes $C_{ij} \in C$ sont entièrement connues des agents inférieurs, dans le modèle de “demi-contraintes” un agent X_i possède sa propre table de tuples que X_i rejette, la fusion de chaque “demi-contrainte” $C_{i(j)}$ avec son homologue $C_{(i)j}$ permet de reconstituer la contrainte C_{ij} .

Dans le protocole 2 les agents ne partagent plus leur contraintes, mais ils s’échangent leur valeur afin de garantir la consistance de leur affectation. Chaque agent choisit une valeur compatible avec ses agents supérieurs en fonction des “demi-contraintes” auquel il est relié, ensuite il envoie à tous ses voisins la valeur choisie afin que ceux ci puissent vérifier si la valeur reçue est compatible avec la leur.

Formellement quand un agent X_i de valeur v_i reçoit un message de $X_h \in \Gamma^-(X_i)$ contenant la valeur v_h il vérifie que $(v_i, v_h) \notin C_{i(h)}$ (i.e. le tuple n’est pas interdit par X_i). Si le tuple (v_i, v_h) est interdit X_i change de valeur afin d’être consistant avec tous ses agents supérieurs ou il envoie un BackTrack car aucune affectation consistante n’est possible. On peut exprimer cette propriété plus généralement de la manière suivante :

$$(\forall i \in A / X_i = v_i) \wedge (\forall X_h \in \Gamma^-(X_i) / X_h = v_h) \Rightarrow (v_i, v_h) \notin C_{i(h)}$$

Symétriquement quand un agent X_i de valeur v_i reçoit un message de $X_j \in \Gamma^+(X_i)$ il vérifie que $(v_i, v_j) \notin C_{i(j)}$, si ce n’est pas le cas (i.e. le tuple (v_i, v_j) est interdit par X_i) il envoie un nogood à l’agent X_j afin de lui faire changer de valeur. L’agent X_j adaptera sa valeur et renverra un nouveau message **ok** ? ou bien renverra un BackTrack.

On constate le protocole 2 permet bien de ne plus dévoiler systématiquement les “demi-contraintes” en permettant aux agents de vérifier eux mêmes la consistance des affectations qui transitent dans le réseau. Toutefois contrairement au protocole 1 il dévoile les valeurs prises par un agent à ses concurrents. En terme de confidentialité dévoiler les valeurs n’est pas avantageux car les tuples de chaque “demi-contrainte” qui sont censées être protégées, sont des combinaisons des valeurs qui transitent dans le réseau, de ce fait les agents peuvent déterminer si certains tuples sont admis ou rejetés selon les réponses qu’ils reçoivent.

4.3 Le protocole 3

Le protocole 3 est issue de la fusion du protocole 1 et du protocole 2, dans le sens où il protège à la fois les valeurs prises par les agents (comme le protocole 1) et les “demi-contraintes” (comme dans le protocole 2). Pour ce faire les agents possèdent chacun leurs “demi-contraintes” et s’échangent des ensemble de valeurs. Toutefois contrairement au protocole 1 les agents ne s’échangent plus des ensembles de valeur compatibles (notés $D.(.)$) mais des ensembles de valeurs

incompatibles (notés $\Delta_*(.)$). Cette modification mineure ne change que très peu le principe et l'implémentation du protocole 1, en outre un agent X_i au lieu de vérifier que sa valeur v_i se trouve dans l'ensemble $D_*(X_i)$, il vérifie que v_i n'est pas dans $\Delta_*(X_i)$.

Le protocole 3 est à la fois similaire au protocole 1 dans les structures de données utilisées (les ensembles de valeurs), et au protocole 2 dans la manière de procéder aux échanges de messages (par des propositions qui donnent lieu à des réponses jusqu'à accord). Formellement on a pour deux agents X_i et X_j l'ensemble $\Delta_{v_i}(X_j) = \{v_j \in D(X_j) / (v_i, v_j) \in C_{i(j)}\}$

Le protocole 3 ré-utilise les mêmes types de messages que l'algorithme ABT, à savoir :

- des messages **ok?** : pour faire transiter les propositions des agents.
- des messages **ngd** : pour représenter soit un BackTrack quand la négociation entre les agents échoue, soit une réponse négative lors de la négociation (cette nuance est expliquée ci après).
- des messages **adl** : pour créer des liens temporaires afin de ne pas explorer plusieurs fois des parties de l'espace de recherche suite à un BackTrack.

Le protocole 3 toutefois utilise un nouveau type de message en plus des trois présents dans ABT. Ce nouveau type de messages nommé **yes** permet de signifier à un agent inférieur que sa proposition a été acceptée par l'agent supérieur qui lui envoie ce message.

L'idée directrice dans l'implémentation du protocole 3 consiste à forcer les agents à être consistant avec leur voisins supérieurs quitte à les faire changer de valeur pour l'être. Les agents inférieurs énumèrent les valeurs possibles qui ne leur sont pas interdites par leur concurrent, jusqu'à ce que leurs agents supérieurs valident tous une des propositions qui leurs sont faites. Si aucune proposition n'est validée alors l'agent inférieur peut envoyer un BackTrack comme dans ABT pour forcer un agent supérieur à changer de valeur.

Détail de l'utilisation des messages ok? : Les messages **ok?** sont émis quand un agent X_i choisit une (nouvelle) valeur v_i . Ces messages sont destinés aux voisins d' X_i dans le graphe de contraintes, chaque message **ok?** destiné à un agent $X_j \in \Gamma^+(X_i) \cup \Gamma^-(X_i)$ contient l'ensemble $\Delta_{v_i}(X_j)$ de valeurs incompatibles pour X_j .

Un agent X_i recevant un message **ok?** d' X_j ne réagit pas de la même façon selon si $X_j \in \Gamma^+(X_i)$ ou $X_j \in \Gamma^-(X_i)$:

- si $X_j \in \Gamma^-(X_i)$ alors X_i élimine les valeurs contenues dans $\Delta_{v_j}(X_i)$ de son domaine. Si sa valeur v_i est éliminée, il change de valeur afin d'être consistant avec tous ses voisins supérieurs. Si X_i n'a plus de valeur disponible il envoie un BackTrack.
- si $X_j \in \Gamma^+(X_i)$ alors X_i vérifie si $v_i \in \Delta_{v_j}(X_i)$. Si c'est le cas il envoie un nogood à X_j pour lui faire changer de valeur. Si ce n'est pas le cas il répond un message **yes** pour signifier à X_j que sa proposition est acceptée.

Les messages **ok ?** permettent donc à la fois d'éliminer les valeurs impossibles chez les agents inférieurs et acheminer les propositions des agents inférieurs aux agents supérieurs. Comme les messages **ok ?** contiennent des ensembles qui peuvent être potentiellement identiques, chaque message **ok ?** est estampillé par l'estampille courante de l'agent qui l'envoie.

Détail de l'utilisation des messages ngd : Comme nous l'avons précisé précédemment les nogoods peuvent caractériser soit le refus d'une proposition, soit l'envoi d'un BackTrack. La distinction entre ces deux comportements s'effectue en considérant la place dans l'ordre de l'agent qui a émis ce nogood.

- si le nogood vient d'un agent $X_j \in \Gamma^+(X_i)$ alors le nogood correspond à un BackTrack qui oblige donc l'agent X_i à changer sa valeur. Ces nogoods sont nommés “implicites” par [3].
- si le nogood vient d'un agent $X_j \in \Gamma^-(X_i)$ alors le nogood correspond au refus d'une proposition de X_i , i.e. $X_j = v_j \in \Delta_{v_i}(X_j)$. Ces nogoods sont nommés “explicites” par [3].

Les nogoods contiennent tous l'estampille courante de l'agent qui l'a initié et l'estampille la plus récente de l'agent qui a causé son émission.

Les nogoods permettent donc, de manière générale, de faire changer la valeur de l'agent à qui il est destiné. Toutefois les nogoods “explicites”, n'étant pas relayés par les agents, sont responsables de plus grandes pertes de confidentialité (voir section 7 pour plus de précisions). Les nogoods implicites pouvant être relayés il ne permettent pas à l'agent le recevant de déterminer à quel endroit dans le graphe de contraintes se trouve l'agent responsable de l'échec de la négociation.

Détail de l'utilisation des messages yes : Les messages **yes** permettent à un agent supérieur de valider la proposition que lui a fait un agent inférieur. De ce fait ce type de message ne transite que d'un agent supérieur à un agent inférieur. Les messages **yes** contiennent l'estampille de la proposition qui a été validée afin de permettre à l'agent inférieur de savoir si la proposition qui a été acceptée n'était pas déjà obsolète.

Le détail de l'algorithme issu du protocole 3 est disponible en annexe B.

5 Analyse de la confidentialité du protocole 3

5.1 Analyse du comportement des agents

Comme nous l'avons vu le protocole 3 fonctionne par des propositions entraînant des réponses positives (messages **yes**) ou des réponses négatives (messages **ngd**). Un point important dans le protocole 3 (qui nous sera utile pour son analyse avec VPS) est que chaque proposition que fait un agent inférieur implique

une réponse de l'agent supérieur. De ce fait les agents inférieurs peuvent toujours mettre en correspondance les propositions qui sont faites et les réponses reçues. Toutefois cela ne s'applique pas aux agents supérieurs, car une proposition d'un agent supérieur n'implique pas forcément un changement de valeur de l'agent inférieur à qui la proposition est destinée (et donc l'émission d'un message).

Les réponses de X_j qui peuvent être reçues suite à une proposition de X_i sont de 3 types :

- **yes** : si $X_j \in \Gamma^-(X_i)$ et que $v_i \notin \Delta_{v_j}(X_i)$. Cela signifie que la proposition d' X_i a été accepté par X_j .
- **ngd** : on distingue deux cas possibles d'émission d'un nogood.
 - si $X_j \in \Gamma^+(X_i)$ et $D(X_j) = \emptyset$. Cela signifie qu' X_j n'a plus aucune valeur de valide et a envoyé un BackTrack.
 - si $X_j \in \Gamma^-(X_i)$ et que $v_i \in \Delta_{v_j}(X_j)$. Cela signifie que la proposition d' X_i a été rejetée par X_j .
- **ok ?** : si $X_j \in \Gamma^+(X_i)$. Cela signifie qu' X_j a changé sa valeur suite à la réception de l'ensemble $\Delta_{v_i}(X_j)$ car $v_j \in \Delta_{v_i}(X_j)$. Ce dernier cas n'est pas automatiquement émis car X_j ne change pas obligatoirement de valeur.

Dans la section suivante nous allons détailler quelles informations sont révélées et comment un agent peut les exploiter.

5.2 Application de VPS

Le système VPS nous permet de modéliser les états possibles qu'ont pu choisir les concurrents d'un agent. Dans le cadre de ce rapport les états possibles sont les valeurs prises par les agents au fil du processus de résolution du DisCSP. Nous verrons que la découverte des tuples rejetés par les "demi-contraintes" est déduite de la connaissance de la valeur prise par un agent. Cette étude fait l'objet de la section 5.3.

5.2.1 Remarques

Remarque concernant la découverte d'une solution : Une première constatation concerne l'arrêt du processus suite à la découverte d'une solution au DisCSP qui est traité. En effet quand une solution est trouvée cela implique que la dernière proposition que chaque agent a fait, a été acceptée par les agents voisins. Comme les propositions contiennent des ensembles de valeurs interdites cela signifie que tous les agents voisins n'avaient pas pris leur dernière valeur dans l'ensemble transmis qui les concernait.

Formellement on peut exprimer ce résultat de la façon suivante :

$$\forall i \in A \wedge (\forall X_j \in \Gamma^+(X_i) \cup \Gamma^-(X_i) / X_j = v_j) \Rightarrow v_j \notin \Delta_{v_i}(X_j)$$

En terme de VPS cela signifie que les valeurs contenues dans chaque ensemble $\Delta_{v_i}(X_j)$ sont toutes associées à une probabilité nulle. Soit donc pour un agent X_i :

$$\forall X_j \in \Gamma^+(X_i) \cup \Gamma^-(X_i) \wedge (\forall v_t \Delta_{v_i}(X_j)) \Rightarrow P_j^i(v_t) = 0$$

Remarque concernant l’obsolescence et le cumul de connaissance :

Le modèle VPS nous sert à modéliser la valeur qu’a pris un agent à une certaine date. Quand un agent change de valeur, et envoie donc un message **ok ?**, il est logique de rendre obsolète la connaissance qui concerne la valeur précédente de l’agent. En terme de VPS cela se traduit par rendre équiprobables toutes les probabilités concernées par l’agent qui a changé de valeur. Il est à noter que la connaissance concernant les “demi-contraintes” (voir section 5.3) n’est jamais obsolète, car les agents ne modifient pas leurs “demi-contraintes” au fil du processus.

Toutefois tant qu’un agent ne change pas de valeur et continue à envoyer des messages, la connaissance issue de ses messages se cumule. Nous verrons que ce cumul est justement l’atout majeur d’un agent inférieur pour découvrir la valeur de ses agents supérieurs (voir section 5.2.3).

Remarque concernant les connaissances complémentaires : Il est important de noter que dans le cadre de l’algorithme du protocole 3 la connaissance qu’obtient un agent X_i sur la valeur v_j de son concurrent X_j peut être vue de deux manières différentes. En effet comme X_i connaît le domaine d’ X_j l’assertion suivante est vérifiée :

$$v_j \notin \Delta_{v_i}(X_j) \Leftrightarrow v_j \in D(X_j) \setminus \Delta_{v_i}(X_j)$$

On peut donc intervertir l’une ou l’autre écriture sans changer la sémantique de la propriété. On constatera que l’algorithme Ring-ABT présenté dans la section IV permet d’affaiblir cette équivalence.

5.2.2 Application pour un agent supérieur

. Dans cette section nous considérons 2 agents X_i, X_j tels que X_i soit supérieur à X_j . On se place donc du point de vue d’ X_i afin d’expliciter la connaissance qu’il peut obtenir au sujet d’ X_j .

Comme nous l’avons précisé précédemment les messages **ok ?** envoyés par X_i à X_j n’impliquent pas forcément une réponse d’ X_j , sauf si l’agent X_j doit changer sa valeur car X_i lui a interdite. Il faut donc distinguer plusieurs cas de figure qui pourront faire l’objet d’heuristiques permettant de les discerner avec plus ou moins d’exactitude. La connaissance qu’obtient un agent supérieur d’un agent inférieur n’est donc pas “exacte”, dans le sens où elle est liée une heuristique. Cette notion d’heuristique sera plus amplement étudiée dans la section 5.2.2 à l’occasion de l’utilisation du modèle d’analyse d’incertitude de réception de messages pour les agents supérieurs.

Cas n°1 : X_i prend la valeur v_i et envoie un message **ok ?** à X_j , quand X_j le reçoit il vaut la valeur v_j , or $v_j \in \Delta_{v_i}(X_i)$. Donc X_j change sa valeur en v'_j et renvoie un message **ok ?** à X_i contenant le nouveau ensemble $\Delta_{v'_j}(X_i)$. En terme

de VPS cela signifie pour X_i qu'avant qu' X_j ne change de valeur, X_j avait pris une valeur $v_j \in \Delta_{v_i}(X_j)$. De ce fait on a $\forall v_t \in D(X_j) \setminus \Delta_{v_i}(X_j) \Rightarrow P_j^i(v_t) = 0$, et symétriquement $\sum_{v_t \in \Delta_{v_i}(X_j)} P_j^i(v_t) = 1$ car $v_j \in \Delta_{v_i}(X_j)$.

Cas n°2 : X_i prend la valeur v_i et envoie un message **ok?** à X_j , quand X_j le reçoit il vaut la valeur v_j telle que $v_j \in \Delta_{v_i}(X_i) \wedge D(X_j) \setminus \Delta_{v_i}(X_j) = \emptyset$ (X_j n'a plus aucune valeur de valide). Étant donné qu' X_j n'a plus de valeur possible X_j constitue un nogood et l'envoi au plus proche agent supérieur responsable de cet échec. Supposons que cet agent soit X_i , quand X_i reçoit ce nogood il peut vérifier que l'estampille le concernant dans le nogood est bien la dernière qu'il a émis. X_i peut supposer ensuite que $v_j \in \Delta_{v_i}(X_i)$ puisque X_i est responsable du nogood. En terme de VPS cela se traduit comme dans le cas N°1 par une annulation des probabilités associées aux valeurs non-contenues dans l'ensemble $\Delta_{v_i}(X_j)$.

Cas n°3 : X_i prend la valeur v_i et envoie un message **ok?** à X_j , quand X_j le reçoit il vaut la valeur v_j telle que $v_j \notin \Delta_{v_i}(X_i)$. X_j ne change pas sa valeur, et n'envoie donc aucun message. En terme de VPS cela signifie que les probabilités associées aux valeurs contenues dans $\Delta_{v_i}(X_j)$ sont nulles, i.e. $\forall v_t \in \Delta_{v_i}(X_j) \Rightarrow P_j^i(v_t) = 0$.

On constate que deux types de connaissances peuvent être extraites selon si v_j se trouve ou non dans l'ensemble $\Delta_{v_i}(X_j)$. De plus on constate que le nombre d'éléments contenus dans l'ensemble $\Delta_{v_i}(X_j)$ permet de déduire le nombre de probabilités qui sont annulées, c'est donc un facteur important du point de vue de la confidentialité. Le nombre de probabilité annulées est maximal quand $v_i \in \Delta_{v_i}(X_j)$ et $card(\Delta_{v_i}(X_j))$ est minimal, et de la même manière le nombre de probabilités annulées est maximal quand $v_i \notin \Delta_{v_i}(X_j)$ et $card(\Delta_{v_i}(X_j))$ est maximal.

Enfin il est important de noter qu'en dépit du fait qu' X_i ne dispose pas d'hypothèses exactes cela ne nuit pas au fait que la connaissance qu'il formule est cumulable. Par exemple si X_i envoie p propositions, notés $\Delta_{v_{i_1}}, \dots, \Delta_{v_{i_p}}$ à X_j . Supposons que l'heuristique d' X_i considère qu' X_j n'a émis aucune réponse en rapport avec ces p propositions, i.e. aucune de ses propositions n'a impliqué un changement de valeur d' X_j . Alors X_i sait que v_j n'est dans aucun des ensembles proposés. i.e. $v_i \notin \cup_{k \in \{1..p\}} \Delta_{v_{i_k}}(X_j)$.

5.2.3 Application pour un agent inférieur

Dans cette section nous considérons 2 agents X_i, X_j tels que X_i soit supérieur à X_j . On se place du point de vue d' X_j afin d'explicitier la connaissance qu'il peut obtenir au sujet d' X_i . Contrairement aux agents supérieurs les agents inférieurs disposent d'hypothèses exactes concernant leurs agents supérieurs. La connaissance est exacte car chaque proposition d'un agent X_j implique une réponse de l'agent X_i . Cet état de fait est dû aux estampilles contenues dans les messages **yes** et **ngd**, et qui permettent à X_j de mettre directement ses propo-

sitions et les réponses reçues en correspondance.

Les réponses d' X_i peuvent être de deux types :

- X_i répond un message **yes** : alors $v_i \notin \Delta_{v_j}(X_i)$, de ce fait toutes les probabilités des valeurs contenues dans $\Delta_{v_j}(X_i)$ sont annulées, i.e. $\forall v_t \in \Delta_{v_j}(X_i) \Rightarrow P_i^j(v_t) = 0$.
- X_i répond un message **ngd** : alors $v_i \in \Delta_{v_j}(X_i)$, de ce fait toutes les probabilités des valeurs contenues dans $D(X_i) \setminus \Delta_{v_j}(X_i)$ sont annulées, i.e. $\forall v_t \in D(X_i) \setminus \Delta_{v_j}(X_i) \Rightarrow P_i^j(v_t) = 0$.

Il est important de noter ici que tant qu' X_i n'envoie pas un message **ok** ? la connaissance que tire X_j de toutes les réponses d' X_i se cumule. Par contre dès qu' X_i change de valeur et envoie un message **ok** ? la connaissance concernant l'état d' X_i devient obsolète.

Considérons qu' X_j envoie p propositions à X_i et qu'il reçoive successivement $p - 2$ nogoods, et deux messages **yes**. On note $\Delta_{v_{j_1}} \dots \Delta_{v_{j_p}}$ les p propositions. X_j peut donc déduire que v_i se trouve dans l'intersection des $p - 2$ premières propositions, et que v_i ne se trouve ni dans la $(p - 1)^{ime}$ ni la p^{ime} , i.e.

$$v_i \in (\cap_{k \in \{1..p-1\}} \Delta_{v_{j_k}}(X_i)) \setminus (\cup_{k \in \{p-1,p\}} \Delta_{v_{j_k}}(X_i))$$

Il est intéressant de remarquer que plus le nombre de réponses augmente plus la connaissance qu'a X_j de v_i se précise. Il est possible que cette connaissance devienne exacte si l'expression ci dessus se réduit à un singleton. En terme de VPS cela signifie que toutes les probabilités sauf une sont nulles, i.e. $\exists v_i \in D(X_i) / \forall v_k \neq v_i \rightarrow P_i^j(v_k) = 0 \wedge P_i^j(v_i) = 1$.

5.2.4 Cas particulier d'acquisition de connaissance simplifié

Les agents agissant en parallèle il est particulièrement difficile de savoir si les messages que X_i reçoit d'un agent X_j sont dus aux propositions d' X_i ou bien de celles d'un tierce agent X_k venu s'intercaler dans le processus de négociation entre X_i et X_j . Toutefois si l'agent X_j est un sommet pendant du graphe de contrainte (i.e. $\Gamma^+(X_j) \cup \Gamma^-(X_j) = \{X_i\}$), alors il est impossible qu'un agent X_k soit intervenu pour modifier le comportement d' X_j . De ce fait X_i peut analyser les réactions d' X_j en sachant pertinemment que tous messages qu'envoie X_j est causé par un message antérieur d' X_i . L'acquisition d'information concernant X_j est donc grandement simplifiée. Il est à noter que ce cas particulier s'applique aussi pour les algorithmes DisFC-1ph, et Ring-ABT.

5.3 Connaissance des “demi-contraintes” d'un agent

L'application du modèle VPS permet d'explicitier la connaissance que les agents peuvent avoir de la valeur prise par leurs voisins. Il est important de rappeler que les ensembles de valeurs interdites qui sont transmis sont déterminés par la valeur de l'agent émetteur et la “demi-contrainte” qui le relie avec le receveur. Formellement si v_i est la valeur de l'agent X_i , et X_j son voisin alors $\Delta_{v_i}(X_j) = \{v_j \in D(X_j) / (v_i, v_j) \in C_{i(j)}\}$. De ce fait si X_j connaît la valeur v_i d' X_i alors il peut déduire l'ensemble des tuples interdits par $C_{i(j)}$ associé à v_i .

X_j peut déduire ces tuples de la manière suivante : $(\{v_i\} \times \Delta_{v_i}(X_j)) \subseteq C_{i(j)}$. Si toutefois X_j ne connaît pas la valeur d' X_i avec certitude, i.e. $\forall v_i \in D(X_i) \Rightarrow P_i^j(v_i) < 1$, alors l'agent X_j peut uniquement supposer que parmi les probabilités non encore nulles l'une d'entre elles a permis de produire l'ensemble $\Delta_{v_i}(X_j)$, i.e.

$$\exists v_i \in D(X_i)/P_i^j(v_i) \neq 0 \Rightarrow \{v_i\} \times \Delta_{v_i}(X_j) \subseteq C_{i(j)}$$

5.4 Conclusion

Nous avons vu que le protocole 3 octroie des gains disymétriques de connaissance entre les agents supérieurs et les agents inférieurs. Les agents supérieurs disposent d'hypothèses moins fortes que les agents inférieurs. On peut donc conclure que les agents inférieurs sont les plus à même de déduire exactement la valeur de leurs agents supérieurs, et par ce fait la liste des tuples interdits de la "demi-contrainte" les reliant. Il semble donc nécessaire de restreindre la connaissance que peuvent acquérir les agents inférieurs. Pour cela il est possible de supprimer l'usage des messages **yes** utilisés par les agents supérieur, de la sorte les agents inférieurs n'obtiendront plus automatiquement d'accusé de réception pour chacune de leurs propositions. Cette modification fait l'objet de l'algorithme DisFC-1ph présenté dans la section suivante.

Troisième partie

L'algorithme DisFC-1ph

6 Présentation

L'algorithme DisFC-1ph développé par I. Brito et P. Meseguer [3] est une amélioration du protocole 3. En outre cet algorithme supprime l'usage des messages **yes** qui servaient à confirmer l'acceptation d'une proposition faite par un agent inférieur à un de ses agents supérieurs. Nous avons vu dans la section 5.2.3 que le gain d'information entre les agents était disymétrique, en outre les agents inférieurs disposent de meilleurs hypothèses concernant l'état de leurs concurrents supérieurs. En effet les agents inférieurs recevaient des messages pouvant faire office d'accusé de réception de leur proposition (messages **yes** et **nogood**). L'algorithme DisFC-1ph permet de pallier en partie à ce problème en rajoutant une part d'incertitude dans la réception des propositions que font les agents inférieurs. La connaissance qu'obtiennent les agents supérieurs est quant à elle identique à celle présentée dans le protocole 3 (section 5.2.2).

6.1 Principe

Le protocole 3 ne nécessite que très peu de modifications pour produire l'algorithme DisFC-1ph. La principale modification s'effectue dans la méthode *ProcessInfo(msg)* qui assure la gestion des messages **ok ?** reçus par l'agent. En effet DisFC-1ph se passe de l'utilisation des messages **yes**, les agents supérieurs ne répondent plus à leurs agents inférieurs que leurs propositions ont bien été acceptées par celui-ci. Nous verrons que de la sorte les agents inférieurs ne peuvent plus savoir si leurs propositions ont été acceptées avant ou après un changement de valeur de leur agent supérieur, rendant du coup plus incertaine la connaissance de l'état de leurs agents supérieurs. Un modèle d'analyse de cette incertitude est présenté à cet effet dans la section 7.

Il est à noter que cette modification n'entraîne un changement de comportement que du point de vue d'un agent supérieur, il est donc logique que les agents inférieurs soient exposés exactement aux mêmes problèmes de confidentialité que dans le protocole 3 (cf. section 5.2.2).

L'algorithme de DisFC-1ph est présenté dans la section ci-après.

6.2 Algorithme de DisFC

Les procédures manquantes peuvent être trouvées en annexe B.

Algorithm 1 Protocole 3.1 : Main procedure

```
myValue ← ChooseValue();  
end ← false;  
timestamp ← randomInt();  
FilteredDomain ←  $\emptyset$ ;  
Compute  $\Gamma^+$ ,  $\Gamma^-$ ;  
CheckAgentView();  
while  $\neg$ end do  
  msg ← GetMsg();  
  switch (msg.type)  
    ok? : ProcessInfo(msg); break;  
    ngd : ResolveConflict(msg); break;  
    adl : SetLink(msg); break;  
    stp : end ← true;  
end while
```

Algorithm 2 Protocole 3.1 : ProcessInfo(msg)

```
0: UpdateAgentView(msg);
if ( $msg.sender \in \Gamma^+(self)$ ) then
  FilteredDomain[msg.sender]  $\leftarrow msg.domain$ ;
  if ( $myValue \in msg.domain$ ) then
    # si l'agent inférieur interdit la valeur de self, self lui demande de changer
    de valeur
    SendMsg :ngd(msg.sender,  $self = self.timestamp \Rightarrow msg.sender \neq$ 
     $msg.timestamp$ );
  else
    return;
    # self ne répond plus un message yes à l'agent inférieur
  end if
end if
if ( $msg.sender \in \Gamma^-(self)$ ) then
  UpdateDomain(msg);
  if ( $myValue \in msg.domain$ ) then
    # si la valeur de self est interdite par un agent supérieur
    # self tente de changer sa valeur
    CheckAgentView();
  else
    # sinon il propose l'ensemble de valeurs interdites à son agent supérieur
    SendMsg :ok?(msg.sender, timestamp, incompatible(D(sender), myValue));
  end if
end if
```

7 Modèle d'analyse du comportement des agents

Le modèle que nous présentons ici permet de modéliser l'incertitude de réception des messages dont les agents inférieurs sont victimes dans l'algorithme DisFC-1ph. En effet considérons deux agents X_i et X_j tels que $i < j$ dans l'ordre, supposons qu' X_j envoie 3 messages **ok?** à X_i , à la suite de cela X_j reçoit un message **ok?** d' X_i , puis le processus s'arrête car une solution est finalement trouvée. Sans les accusés de réception X_j ne peut pas savoir si ses propositions ont toutes été acceptées par X_i avant ou après l'émission du message **ok?** d' X_i , il se peut même que les deux premières aient été acceptées avant et la troisième après.

Le but du modèle présenté ici est donc de déterminer les différents scénarios possibles pour l'acceptation des propositions faites par un agent inférieur. Nous verrons que les nogoods qui sont émis par un agent supérieur constituent la clé de voûte pour la construction de ce modèle, en effet la quantité de nogoods émis permettra de mesurer l'efficacité de ce modèle.

7.1 Principe

Pour appliquer notre modèle nous utilisons les estampilles qui transitent dans les nogoods. Les nogoods sont formés à partir de l'*AgentView* de l'agent émetteur, dans le cadre du protocole 3 cette structure de données se résume à stocker pour chaque agent voisin : le dernier ensemble transmis dans un message **ok?** ainsi que l'estampille qui y correspond. Les nogoods contiennent la dernière estampille de chaque agent ayant entraîné l'épuisement du domaine de l'agent qui envoie le nogood.

Formellement un nogood correspond à une expression du type :

$$X_1 = timestamp_1 \wedge \dots \wedge X_k = timestamp_k \Rightarrow X_i \neq v_i$$

où X_i est l'agent qui envoie le nogood, et $\{X_1, \dots, X_k\}$ est un sous ensemble d'agents voisins d' X_i dont les ensembles correspondant aux estampilles $[timestamp_1 \dots timestamp_k]$ sont responsables du nogood.

Supposons que l'agent X_1 reçoive ce nogood. Il est possible qu'entre le moment où ce nogood est reçu et le moment où il a été émis, X_1 ait changé de valeur et s'attribue l'estampille $timestamp_1$. Toutefois X_1 sait que son nouveau message **ok?** n'a pas encore été reçu par X_i car le nogood d' X_i n'en fait pas mention (l'estampille n'est pas à jour chez X_i de ce fait le nogood sera ignoré par X_1 car il est obsolète).

On constate donc que les nogoods permettent de poser certaines restrictions sur les correspondances possibles entre messages, en outre un message d'estampille k ne peut pas être mis en relation avec un autre si aucun nogood n'atteste de la réception d'une estampille inférieure ou égale à k .

Symétriquement on peut constater que des messages ne peuvent pas être mis en correspondance si un nogood reçu antérieurement atteste de la réception d'une estampille ultérieure.

Ces restrictions permettent de définir des sous ensembles de messages qui sont potentiellement compatibles, ces sous ensembles sont formés par un processus de segmentation des ensemble de messages (présenté dans la section 7.3) qui utilise les estampilles contenues dans les nogoods reçus.

7.2 Définitions

Afin d'appliquer notre modèle quelques définitions concernant les ensembles de messages sont nécessaires.

Soient deux agents X_i et X_j , tels que $i < j$ dans l'ordre des agents, et tels qu'ils soient mutuellement contraints par $C_{i(j)}$ et $C_{(i)j}$. Les valeurs prises par X_i sont notées v_{i_1}, \dots, v_{i_k} , et les valeurs prises par X_j sont notées v_{j_1}, \dots, v_{j_r} .

On note $Emis_j^i$ l'ensemble des messages émis par X_i destinés à X_j , et on note $Recus_j^i$ l'ensemble des messages reçus par X_i provenant d' X_j . Symétriquement on définit $Emis_i^j$ et $Recus_i^j$ pour X_j .

Ces ensembles sont totalement ordonnés par une relation de précédence temporelle notée $<$ qui correspond à l'ordre de réception et d'émission des messages. Conformément aux messages que s'échangent les agents dans DisFC-1ph, ces ensembles contiendront :

- des messages **ok ?** qui seront notés : $OK(v_{j_k})$ (resp. $OK(v_{i_k})$) pour signifier que le message **ok ?** contenait l'ensemble $\Delta_{v_{j_k}}(X_i)$ (resp. $\Delta_{v_{i_k}}(X_j)$).
- des nogoods qui seront notés : $NGD(v_{i_k}, v_{j_t})$, qui est la forme réduite du nogood $X_i = timestamp_k \wedge \dots \wedge X_{i+l} = timestamp_{k_l} \Rightarrow X_j \neq timestamp_t$ où X_i est l'agent le plus proche d' X_j dans l'ordre.

On désignera par le terme *Msg* un élément quelconque d'un de ces ensembles.

7.3 La segmentation

Comme nous l'avons dit précédemment les messages peuvent être mis en correspondance que sous certaines conditions que les nogoods imposent. L'étape de segmentation permet donc de limiter l'explosion combinatoire des correspondances possibles entre messages, en les limitant sur un partition de sous ensembles compatibles.

En effet un message Msg_1 émis ne peut être la cause d'un message Msg_2 reçu si un nogood ultérieur à Msg_2 n'atteste pas de la réception d'un message d'estampille supérieure ou égale à celle transmise dans Msg_1 .

Symétriquement un message Msg_1 reçu ne peut être la conséquence d'un message Msg_2 émis si un nogood antérieur à Msg_2 n'atteste pas de la réception d'une estampille inférieur ou égale à celle transmise dans Msg_2 .

Ces restrictions sont illustrées par l'exemple suivant, dans cet exemple on se place du point de vue de X_j :

On constate que le message $OK(v_{j_4})$ de X_j , bien qu'antérieur à la réception du message $OK(v_{i_2})$, ne peut avoir un lien avec l'émission du message $OK(v_{i_2})$, car le nogood $NGD(v_{i_2}, v_{j_3})$ indique qu'à cette date le message $OK(v_{j_4})$ n'avait

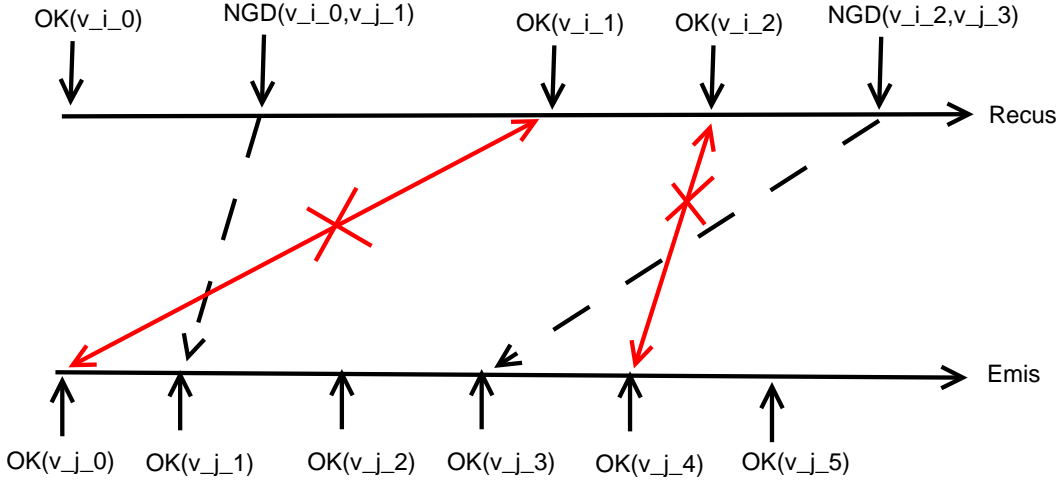


FIG. 1 – Un exemple de l'utilisation de nogood pour la segmentation

pas encore été reçu par X_i .

De là même manière on constate que le message $OK(v_{j_0})$ ne peut avoir un lien avec le message $OK(v_{i_1})$ car le nogood $NGD(v_{i_0}, v_{j_1})$, qui est antérieur à l'émission du message $OK(v_{i_1})$, atteste de la réception et du traitement du message $OK(v_{j_1})$.

La segmentation permet donc de mettre en relation des messages de *Recus* contenus entre 2 nogoods **adjacents** (que l'on note $NGD(v_{i_x}, v_{j_i})$ et $NDG(v_{i_y}, v_{j_r})$), qu'avec des messages émis dont l'estampille est contenue dans l'intervalle d'estampilles suivant : $[v_{j_i}, \dots, v_{j_r}]$.

Formellement pour un agent X_i ayant reçu p nogoods de la part d' X_j on définit $p-1$ paires de sous-ensembles de $Emis_j^i$ et de $Recus_j^i$. On note ces paires $\{(Emis_j^i(1), Recus_j^i(1)), \dots, (Emis_j^i(p), Recus_j^i(p-1))\}$.

On peut remarquer que la segmentation ne tient pas compte des messages d'estampilles antérieures au tout premier nogood reçu, et les messages d'estampilles ultérieures au tout dernier reçu. Il est tout à fait possible d'étendre la segmentation sur les paires d'ensembles correspondant.

On définit donc $(Emis_j^i(0), Recus_j^i(0))$ à partir de l'estampille d' X_i contenue dans le premier nogood reçu d' X_j , notons ce nogood $NGD(v_{i_k}, v_{j_r})$. Alors on construit les ensembles $Emis_j^i(0)$ et $Recus_j^i(0)$ de la manière suivante :

- $Emis_j^i(0) = \{Msg \in Emis_j^i / Msg.timestamp < OK(v_{i_k}).timestamp\}$
- $Recus_j^i(0) = \{Msg \in Emis_j^i / Msg < NGD(v_{i_k}, v_{j_r})\}$

Symétriquement on définit $(Emis_j^i(p), Recus_j^i(p))$ à partir de l'estampille d' X_i contenue dans le dernier nogood reçu d' X_j , notons ce nogood $NGD(v_{i_l}, v_{j_r})$. Soit donc :

- $Emis_j^i(p) = \{Msg \in Emis_j^i / Msg.timestamp > OK(v_i).timestamp\}$
- $Recus_j^i(p) = \{Msg \in Emis_j^i / Msg > NGD(v_i, v_{j_r})\}$

On obtient donc une segmentation qui partitionne entièrement les ensembles $Emis_j^i$ et $Recus_j^i$ en $p + 1$ sous ensembles. Ces partitions vont nous permettre de définir par morceaux des relations de correspondance de messages sur les sous ensembles compatibles (i.e. ayant le même indice).

7.4 Applications

7.4.1 Application pour un agent supérieur

Considérons deux agents X_i et X_j , tel que $i < j$ dans l'ordre. On se place du point de vue de X_i . Dans l'algorithme DisFC-1ph l'agent X_i de valeur v_i peut envoyer deux types de messages à X_j :

- des messages **ok ?** : pour interdire à X_j un certain ensemble de valeurs déduites de la valeur v_i choisie par X_i .
- des messages **ngd** : pour signifier à X_j que sa proposition interdit la valeur courante de X_i , et lui demander donc de changer de valeur.

Les messages **ok ?** peuvent entraîner un changement de valeur de X_j si $v_j \in \Delta_{v_i}(X_j)$, et donc l'émission d'un nouveau message **ok ?** de la part d' X_j , dans le cas contraire X_j ne répond rien à X_i . Les nogoods impliquent obligatoirement un changement de valeur de X_j , et donc l'envoi d'un message **ok ?**.

Dans cette partie nous cherchons à construire une relation \mathfrak{R}_1 de causalité entre les messages émis par X_i et les messages **ok ?** répondus par X_j . Pour cela nous appliquons l'étape de segmentation (cf. 7.3) sur les ensemble de messages et définissons notre relation \mathfrak{R}_1 par morceaux sur les paires de sous ensembles compatibles. Dans la section 5.2.2 nous avons parlé d'heuristiques permettant à un agent supérieur de déterminer l'impact de ses propositions. Nous voyons ici donc un moyen de spécifier précisément comment construire ces heuristiques grâce à la modélisation de \mathfrak{R}_1 .

Comme \mathfrak{R}_1 est une relation de causalité entre les messages, deux propriétés de chronologie sont donc respectées :

D'une part un message émis ne peut être la cause que d'un message reçu ultérieurement. Plus formellement on a la propriété suivante :

$$\forall (Msg_1, Msg_2) \in \mathfrak{R}_1 / Msg_1 \in Emis_j^i(k) \wedge Msg_2 \in Recus_j^i(k) \Rightarrow Msg_1 < Msg_2$$

D'autre part un message Msg_1 émis ne peut être la cause que d'un message reçu Msg_2 dont un des prédécesseur n'est pas causé par un message émis ultérieurement à Msg_1 . Cette propriété vient du fait que les messages émis sont reçus dans le même ordre que leur émission. Plus formellement on a la propriété suivante :

$$\forall \{(Msg_1, Msg_2), (Msg_1', Msg_2')\} \subseteq \mathfrak{R}_1 \Rightarrow (Msg_1 < Msg_1' \wedge Msg_2 < Msg_2') \vee (Msg_1 > Msg_1' \wedge Msg_2 > Msg_2')$$

Enfin comme la cause d'un message est toujours le traitement d'un autre message la relation \mathfrak{R}_1 est donc injective.

Les éléments en relation dans \mathfrak{R}_1 peuvent être de plusieurs types :

- des paires $(OK(v_i), OK(v'_j))$, qui signifient qu'après qu' X_i ai envoyé le message **ok ?**, X_j avait sa valeur $v_j \in \Delta_{v_i}(X_j)$. X_j a donc changé sa valeur en v'_j telle que $v'_j \notin \Delta_{v_i}(X_j)$.
- des paires $(NGD(v_i, v_j), OK(v'_j))$, qui signifient qu'après qu' X_i ai envoyé le nogood à X_j , l'agent X_j a changé sa valeur en v'_j telle que $v'_j \notin \Delta_{v_i}(X_j) \wedge v'_j \neq v_j$
- des paires $(NGD(v_i, v_j), NGD(v_i, v_j))$ ou $(OK(v_i), NGD(v_i, v_j))$ qui constituent des BackTrack venant d' X_j . Ces nogoods reçus par X_i servent à délimiter les ensembles de messages qui sont compatibles (voir section 7.3).

Utilisation des paires (OK, OK) : Ce premier type de paire de \mathfrak{R}_1 permet de déterminer quand X_j à changer de valeur suite à une proposition d' X_i . De plus on peut par contraposé déterminer quels messages n'ont donc pas entraîné de changement de valeur. En outre tous les messages **ok ?** de X_i compris entre deux paires (OK, OK) **adjacentes** de \mathfrak{R}_1 ont donc tous été acceptés par X_j , i.e. la valeur d' X_j n'était comprise dans aucun des ensembles transmis. En terme de VPS cela on peut donc annuler toutes les probabilités en rapport avec les valeurs contenues dans ces ensembles.

Formellement on exprime cette propriété ainsi :

$$\forall OK(v_{i_k}) \in Emis_j^i(k) / OK(v_{i_k}) \notin Domain(\mathfrak{R}_1) \wedge (\exists OK(v_{i_l}), OK(v_{i_r}) \in Domain(\mathfrak{R}_1) / OK(v_{i_l}) < OK(v_{i_k}) < OK(v_{i_r})) \Rightarrow v_j \notin \Delta_{v_{i_k}}(X_j).$$

En terme de VPS on peut exprimer le résultat ci dessus à l'aide de l'équivalence suivante :

$$v_j \notin \Delta_{v_{i_k}}(X_j) \Leftrightarrow \forall v_t \in \Delta_{v_{i_k}}(X_j), P_j^i(v_t) = 0.$$

Utilisation des paires (NGD, OK) : Les nogoods qu'envoie X_i à X_j ont pour but de faire changer la valeur v_j de X_j , ils se traduisent donc par l'envoi d'un message **ok ?** de X_j si ce changement de valeur est possible (i.e. X_j dispose encore d'autres valeurs valides que v_j). Toutefois il n'est pas évident pour X_i de reconnaître le message **ok ?** de X_j qui a été émis après le traitement du nogood. Les paires (NGD, OK) permettent de supposer que le message **ngd** émis par X_i a causé l'envoi du message **ok ?** de X_j .

En terme de VPS cela permet de déterminer à quel moment X_j change de valeur et donc à partir de quel message la connaissance concernant l'ancienne valeur v_j d' X_j devient obsolète. Il est à noter toutefois que X_j choisit une nouvelle valeur qui n'est toujours pas contenue dans l'ensemble de valeurs interdites transmis par X_i lors de son plus proche message **ok ?** antérieur à l'envoi du nogood. En terme de VPS on a donc la propriété suivante si v'_j est la nouvelle valeur de X_j et v_i la dernière valeur choisie par X_i :

$$v'_j \notin \Delta_{v_i}(X_j) \Rightarrow \forall v_t \in \Delta_{v_i}(X_j), P_j^i(v_t) = 0$$

7.4.2 Application pour un agent inférieur

On considère ici encore deux agents X_i et X_j , tels que $i < j$ dans l'ordre. Toutefois on se place du point de vue de X_j . Dans l'algorithme DisFC-1ph l'agent X_j de valeur v_j peut envoyer deux types de messages à X_i :

- des messages **ok?** : ces messages sont des propositions contenant chacune un ensemble de valeurs de $D(X_i)$ qui sont incompatibles avec v_j .
- des messages **ngd** : ces nogoods sont des BackTracks signifiant que la dernière proposition qu'a reçu X_j a vidé son domaine, et qu' X_i est le plus proche agent supérieur en cause dans cet échec.

Dans le protocole 3 nous avons bâti notre modèle d'analyse pour les agents inférieurs (cf. 5.2.3 sur l'utilisation des messages **yes** et les nogoods que X_i répondait à chaque proposition de X_j . Ces messages servaient d'accusés de réception de chacune des propositions que formulait X_j . Dans l'algorithme DisFC-1ph les messages **yes** ne sont plus utilisés, de ce fait quand une proposition d' X_j est acceptée par X_i (i.e. $v_i \notin \Delta_{v_j}(X_i)$) l'agent X_j n'en n'est plus informé. Il est possible de même que l'agent X_i change de valeur pendant qu' X_j fait des propositions successives. Si on suppose que ces propositions n'entraînent aucun nogood de la part d' X_i , l'agent X_j pour autant ne peut pas savoir si ses propositions ont été acceptées avant le changement de valeur d' X_i , ou après, ou même une partie avant et une partie après.

Il est important de noter que les nogoods que reçoit X_j de la part d' X_i octroient un gain d'information strictement identique au protocole 3 (cf. 5.2.2), nous ne les traiterons donc pas à nouveau dans cette section. D'autre part on peut remarquer que les nogoods qu'émet X_j et qui ont pour but de faire changer de valeur X_i sont sujets à une incertitude qui concerne le moment où ils sont effectivement traités par X_i . Si X_i change de valeur avant l'arrivée du nogood il sera considéré comme obsolète et ignoré par X_i , si au contraire le nogood contient la bonne estampille d' X_i alors il forcera X_i à changer de valeur et envoyer un nouveau message **ok?**. Toutefois dans l'un ou l'autre cas l'agent X_j n'en tire aucune information utile au sens de VPS, nous ne considérerons pas donc ces cas dans la relation \mathfrak{R}_2 .

La problématique principale pour les agents inférieurs est donc de reconstituer les messages **yes** qui leur permettraient de connaître avec exactitude l'état dans lequel X_i avait interprété et accepté leurs propositions. Pour cela on construit une relation \mathfrak{R}_2 sur $Emis_i^j(k) \rightarrow Recus_i^j(k)$ qui modélise le fait que la proposition $OK(v_j)$ a été acceptée alors que X_i se trouvait dans l'état v_i correspondant à son message $OK(v_i)$.

On peut remarquer que contrairement à la relation \mathfrak{R}_1 la relation \mathfrak{R}_2 ne respecte qu'une seule propriété de chronologie. En effet une proposition d' X_j peut être acceptée par un état de X_i répertorié antérieurement à la proposition. Toutefois si une proposition d' X_j est acceptée par un état v_i de X_i alors une proposition ultérieure d' X_i ne peut en aucun cas être acceptée par un état antérieur d' X_i . Formellement la propriété qui est respectée est la suivante :

$$\forall\{(Msg_1, Msg_2), (Msg'_1, Msg'_2)\} \subseteq \mathfrak{R}_1 \Rightarrow (Msg_1 < Msg'_1 \wedge Msg_2 < Msg'_2) \vee (Msg_1 > Msg'_1 \wedge Msg_2 > Msg'_2)$$

De même la relation \mathfrak{R}_2 n'est pas injective car plusieurs propositions peuvent être acceptées sans qu' X_i ne change de valeur.

La relation \mathfrak{R}_2 permet de définir pour chaque paire d'éléments, une affirmation du type suivant :

$$\begin{aligned} \forall(OK(v_j), OK(v_i)) \in Re_2 &\Rightarrow v_i \notin \Delta_{v_j}(X_i) \\ \text{ce qui s'écrit avec l'aide du système VPS :} \\ \forall(OK(v_j), OK(v_i)) \in Re_2 &\Rightarrow \forall v_t \in \Delta_{v_j}(X_i) \Rightarrow P_i^j(v_t) = 0 \end{aligned}$$

On constate qu'il est donc possible de construire une relation permettant de reproduire les messages **yes** qu'émettait l'agent X_i dans le protocole 3. De ce fait il est possible une fois la relation construite d'appliquer la même analyse avec VPS que dans le protocole 3 concernant les agents inférieurs. Nous avons vu comment définir les relations, il est maintenant important de les énumérer et de choisir celle qui est la plus probable, cette étude fait l'objet de la section suivante.

7.4.3 Analyse des relations \mathfrak{R}

Nous avons vu que pour passer outre l'incertitude de réception des messages de l'algorithme DisFC-1ph, nous pouvons modéliser des relations qui représentent les différents comportements que les agents ont pu adopter. Nous avons pu réduire l'aspect exponentiel du nombre de correspondances possibles entre messages en construisant par morceaux nos relations. Chaque paire de sous ensembles où sont définis les relations est indépendante des autres (cf. 7.3), les correspondances sont donc à construire indépendamment pour chaque segment. Si les ensembles segmentés contiennent peu d'éléments il est possible pour un agent d'énumérer assez rapidement toutes les relations \mathfrak{R}_1 (resp. \mathfrak{R}_2) possibles. Une fois les relations énumérées il est possible d'étudier quelle est la relation \mathfrak{R}_1^* (resp. \mathfrak{R}_2^*) qui est la plus probable.

Une fois cette relation \mathfrak{R}_1^* (resp. \mathfrak{R}_2^*) explicitée alors l'agent X_i (resp. X_j) peut appliquer la même analyse de confidentialité présentée pour le protocole 3 (cf. section 5).

7.5 Conclusion

L'algorithme DisFC-1ph est une évolution du protocole 3 qui permet de réduire la quantité de messages qui transitent dans le réseau. Nous avons vu que les messages supprimés servaient auparavant aux agents inférieurs d'accusés de réception, leur permettant d'obtenir de la connaissance sur l'état de leurs concurrents supérieurs. Les supprimer permet donc de réduire la connaissance que peut obtenir un agent inférieur car l'asynchronisme de l'algorithme DisFC-1ph rend l'analyse du comportement plus complexe. Toutefois nous avons vu qu'il est possible d'une part de borner les espaces de correspondances entre messages grâce à l'utilisation des estampilles contenues dans les nogoods reçus

par les agents. Et d'autre part il est possible de chercher à se ramener à l'étude de la confidentialité du protocole 3 en énumérant les relations possibles entre messages afin de sélectionner le scénario le plus probable, ces relations décrivent les différents scénarios de comportement de l'agent. L'énumération des relations est un processus exponentiel en temps, toutefois le fait de pouvoir borner les espaces de correspondances permet de limiter l'explosion combinatoire. Comme les nogoods reçus par les agents inférieurs permettent d'appliquer ce modèle efficacement nous allons ensuite nous intéresser à limiter ce nombre de nogoods.

Quatrième partie

L'algorithme Ring_ABT

8 Présentation

8.1 Principe

Dans le protocole 3 nous avons vu que les agents inférieurs se servaient des réponses **yes** et **ngd** de leurs agents supérieurs pour obtenir des informations concernant leur valeur. Dans l'algorithme DisFC-1ph les messages **yes** disparaissent permettant aux agents supérieurs de protéger le fait qu'une proposition venant d'un agent inférieur a été acceptée. Toutefois il était encore possible d'énumérer les comportements possibles de l'agent supérieur et choisir le plus probable. Comme nous l'avons vu dans la section 7.3 l'efficacité de l'application de cette analyse repose sur le nombre de nogoods qu'un agent supérieur envoie à ses agents inférieurs, plus ce nombre est élevé plus l'espace des correspondances entre messages est réduit, et plus il est simple d'énumérer tous les comportements possibles. Pour diminuer encore la connaissance qu'un agent inférieur peut acquérir de ses agents supérieurs on peut donc chercher à limiter le nombre de nogoods qu'envoie un agent supérieur à ses agent inférieurs. Cette idée a été le point de départ de l'algorithme Ring_ABT.

Le principe de l'algorithme Ring_ABT consiste à faire collaborer les agents inférieurs à X_i afin de leur faire prendre des valeurs telles que les ensembles de valeurs incompatibles correspondant ne couvrent pas tout le domaine de X_i . Illustrons cette idée par l'exemple suivant :

On considère 4 agents X_i, X_j, X_k et X_l , tels que X_i soit inférieur dans l'ordre à tous les autres agents. Dans l'algorithme DisFC-1ph une fois que X_i a pris la valeur v_i il envoie successivement :

- $\Delta_{v_i}(X_j)$ à X_j
- $\Delta_{v_i}(X_k)$ à X_k
- $\Delta_{v_i}(X_l)$ à X_l

Les agents X_j, X_k et X_l propose ensuite à X_i les ensembles de valeurs incompatibles en fonction de la valeur choisie. Ce procédé est illustré par le schéma suivant :

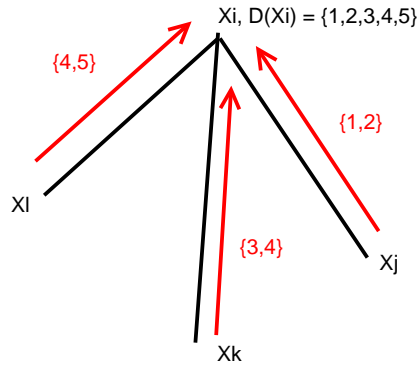


FIG. 2 – Illustration d’une négociation dans DisFC-1ph

On constate que l’union des ensembles de valeurs incompatibles proposés par X_j , X_k et X_l couvre l’ensemble du domaine de X_i , de ce fait v_i se trouve obligatoirement dans au moins l’un des ensembles transmis. En conséquence X_i sera contraint d’envoyer un nogood aux agents qui interdisent v_i . Or on constate que si les agents inférieurs avait négocié entre eux avant de faire chacun leur proposition ils auraient pu éviter de proposer des ensembles qui couvraient tout le domaine d’ X_i . Pour modéliser cette négociation avant de faire chacun leur proposition à X_i , on commence par ordonner totalement les agents inférieurs sur une “chaîne d’agents”. Les agents sur cette chaîne font transiter un “jeton” (message **rng**, “Ring” en anglais) qui contient l’ensemble des valeurs incompatibles pour tous les agents qui ont fait transiter le jeton jusque là. Si les agents sur la chaîne sont X_j , X_k , X_l on note alors cet ensemble $\Delta_{v_j, v_k, v_l}(X_i)$. Ce procédé est illustré par le schéma suivant :

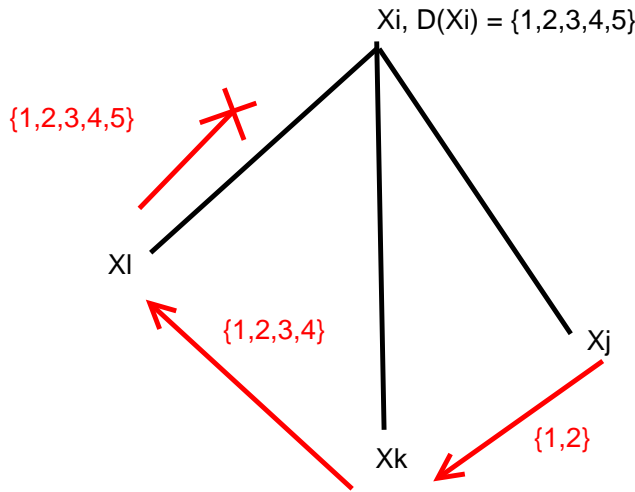


FIG. 3 – Illustration du procédé pour construire $\Delta_{v_j, v_k, v_l}(X_i)$

Le jeton transite successivement d’ X_j à X_k puis à X_l , l’ensemble $\Delta_{\dots}(X_i)$ fusionnant incrémentalement les interdictions de valeurs des agents inférieurs. Quand un agent X_k sur la chaîne remarque que l’ensemble $\Delta_{\dots, v_k}(X_i)$ couvre

tout le domaine de X_i alors l'agent X_k énumère ses valeurs restantes pour en trouver une telle que $\Delta_{\dots,v_k}(X_i) \neq D(X_i)$. Si l'agent X_k ne dispose d'aucune valeur compatible alors le jeton repart en sens inverse sous la forme d'un "nogood de chaîne" obligeant l'agent le précédant à changer sa valeur. Ce procédé de "nogood de chaîne" s'apparente à un mécanisme de BackTrack classique qui toutefois s'effectue sur les agents de la chaîne.

Si le jeton arrive au bout de la chaîne sans couvrir tout le domaine d' X_i alors il est transmis à X_i qui peut alors vérifier si $v_i \notin \Delta_{v_j,v_k,v_l}(X_i)$.

Si $v_i \notin \Delta_{v_j,v_k,v_l}(X_i)$ alors la négociation entre X_i et ses agents inférieurs s'arrête car une affectation consistante est trouvée, sinon X_i renvoie au dernier agent sur la chaîne un "nogood de chaîne", afin de faire trouver aux agents inférieurs un autre ensemble $\Delta_{v_j,v_k,v_l}(X_i)$ qui ne couvre pas entièrement $D(X_i)$.

Toutefois il est possible qu'aucune combinaison de valeurs des agents X_j , X_k et X_l ne conduise à produire un ensemble $\Delta_{v_j,v_k,v_l}(X_i) \neq D(X_i)$ et tel que $v_i \notin \Delta_{v_j,v_k,v_l}(X_i)$, dans ce cas le jeton finira par remonter via des "nogoods de chaîne" d'agent en agent et sera renvoyé à l'agent X_i par le premier agent de la chaîne (en l'occurrence ici X_j). Ce nogood sera alors considéré comme un BackTrack classique destiné à X_i lui demandant de changer sa valeur. Une fois qu' X_i aura changé de valeur alors X_i imposera de nouveaux ensembles de valeurs interdites à ses agents inférieurs et enverra un nouveau jeton qui transitera sur la chaîne d'agent.

8.2 Algorithme de Ring_ABT

Les procédures manquantes peuvent être trouvées en annexe B.

Algorithm 3 Protocole 3.2 : Main procedure

```
myValue ← ChooseValue();  
end ← false;  
timestamp ← randomInt();  
myRings ← [];  
Compute  $\Gamma^+$ ,  $\Gamma^-$ ;  
CheckAgentView();  
while  $\neg$ end do  
  msg ← GetMsg();  
  switch (msg.type)  
    ok? : ProcessInfo(msg); break;  
    ngd : ResolveConflict(msg); break;  
    rng : ProcessRing(msg); break;  
    adl : SetLink(msg); break;  
    stp : end ← true;  
end while
```

Algorithm 4 Protocole 3.2 : ChooseValue(sender)

```
for each value  $\in D(\textit{self})$  notEliminatedBy nogoodstore do
  # self teste toutes ses valeurs pour en trouver une compatible
  valueCompatible  $\leftarrow \textit{True}$ 
  for each agent  $\in \Gamma^-(\textit{self})$  do
    # self teste si "value" est compatible pour tous ses jetons
    newarray  $\leftarrow \textit{myRings}[\textit{agent}].\textit{domain} \cup \textit{incompatible}(D(\textit{agent}), \textit{value})$ ;
    if (newarray =  $D(\textit{agent})$ ) then
      valueCompatible  $\leftarrow \textit{False}$ 
      break;
    end if
  end for
  if (valueCompatible) then
    return value;
  else
    # si value n'est pas compatible on la stock dans un nogood
     $\textit{Store}(\textit{sender} = \textit{sender.timestamp} \Rightarrow \textit{self} \neq \textit{value}, \textit{nogoodstore})$ ;
  end if
end for
return  $\emptyset$ ;
```

Algorithm 5 Protocole 3.2 : ProcessInfo(msg)

```
0:  $\textit{UpdateAgentView}(\textit{msg})$ ;
  if (msg.sender  $\in \Gamma^+(\textit{self})$ ) then
    #self ne fait rien, car ce sont des messages rng qui arriveront de  $\Gamma^+(\textit{self})$ 
  end if
  if (msg.sender  $\in \Gamma^-(\textit{self})$ ) then
     $\textit{UpdateDomain}(\textit{msg})$ ;
    if (myValue  $\in \textit{msg.domain}$ ) then
      # si la valeur de self est interdite par un agent supérieur
      # self change sa valeur
       $\textit{CheckAgentView}(\textit{msg.sender})$ ;
    else
      return;
      # self n'envois pas l'ensemble de valeurs incompatibles,
      # il attend que le jeton les concernant arrive
    end if
  end if
```

Algorithm 6 Protocole 3.2 : ProcessRing(msg)

```
if ( $myRings[msg.agentconcerned].timestamp \leq msg.timestamp$ ) then
  # si ce jeton n'est pas obsolète
   $myRings[msg.agentconcerned] \leftarrow msg$ ;
  if ( $msg.sender \in \Gamma^+(self)$ ) then
    if ( $msg.agentconcerned = self$ ) then
      # si self est l'initiateur de ce jeton
      if ( $msg.timestamp = self.timestamp$ ) then
        # et si ce jeton est bien le dernier a avoir été émis
        if ( $myValue \in msg.domain$ ) then
          # si un des agents inférieurs interdit à self sa valeur
          # self demande à recevoir un nouvel ensemble de valeurs interdites
          SendMsg :ngd( $Last(\Gamma^+(self))$ , self, msg.timestamp,  $self = self.timestamp \Rightarrow rng \neq msg.timestamp$ );
        end if
      end if
    else
      # si ce jeton doit juste être traité et transiter par self
       $newarray \leftarrow myRings[msg.agentconcerned].domain \cup incompatible(D(msg.agentconcerned), myValue)$ ;
      if ( $newarray = D(msg.agentconcerned)$ ) then
        # si le nouvel ensemble de valeurs incompatibles couvre le domaine
        # self tente de changer sa valeur
        CheckAgentView(msg.sender);
      else
        # sinon self fait transiter le jeton avec le nouvel ensemble de valeurs interdites
        SendMsg :rng(msg.nextagent(), msg.agentconcerned, msg.timestamp, newarray);
      end if
    end if
  end if
end if
```

Algorithm 7 Protocole 3.2 : ResolveConflict(msg)

```
if (coherent(msg)) then
  Store(msg, nogoodstore);
  if ( $msg.agentconcerned = self$ ) then
    CheckAgentView();
  else
    ProcessRingNogood(msg);
  end if
end if
```

Algorithm 8 Protocole 3.2 : CheckAgentView(sender)

```
myValue ← ChooseValue();
if (myValue ≠ empty) then
  # si self a trouvé une nouvelle valeur compatible avec tous les agents supérieurs
  timestamp = timestamp + 1;
  for each agent ∈  $\Gamma^+(self)$  do
    # self propose à chaque agent inférieur son nouvel ensemble de valeurs interdites
    SendMsg :ok?(agent, timestamp, incompatible(D(agent), myValue));
  end for
  # ensuite self envoie le jeton au premier agent inférieur afin de le faire transiter
  SendMsg :rng(First( $\Gamma^+(self)$ ), self, self.timestamp,  $\emptyset$ );
  for each agent ∈  $\Gamma^-(self)$  do
    # comme self a changé de valeur il faut re-expédier tous les autres jetons
    # self est sur que sa valeur ne produira pas un ensemble couvrant tout un domaine,
    # car la méthode ChooseValue() l'assure.
    rng ← myRings[agent]
    newarray ← rng.domain ∪ incompatible(D(agent), myValue);
    SendMsg :rng(rng.nextagent(), rng.agentconcerned, rng.timestamp, newarray);
  end for
else
  BackTrack();
end if
```

Algorithm 9 Protocole 3.2 : ProcessRingNogood(msg)

```
if (myValue isEliminatedBy nogoodstore) then
  myValue  $\leftarrow$  ChooseValue();
  if (myValue  $\neq$  empty) then
    # si self a trouvé une nouvelle valeur compatible avec tous les agents
    # supérieurs
    timestamp = timestamp + 1;
    for each agent  $\in$   $\Gamma^+(self)$  do
      # self propose à chaque agent inférieur son nouvel ensemble de valeurs
      # interdites
      SendMsg :ok?(agent, timestamp, incompatible(D(agent), myValue));
    end for
    # ensuite self envoie le jeton au premier agent inférieur afin de le faire
    # transiter
    SendMsg :rng(First( $\Gamma^+(self)$ ), self, self.timestamp, D(self));
    for each agent  $\in$   $\Gamma^-(self)$  do
      # comme self a changer de valeur il faut re-expedier tous les autres
      # jetons
      # self est sur que ca valeur ne produira pas un ensemble couvrant tout
      # un domaine,
      # car la méthode ChooseValue() l'assure.
      rng  $\leftarrow$  myRings[agent]
      newarray  $\leftarrow$  rng.domain  $\cup$  incompatible(D(agent), myValue);
      SendMsg :rng(rng.nextagent(), rng.agentconcerned, rng.timestamp,
        newarray);
    end for
  else
    # si self n'a plus de valeur disponible il "BackTrack" le jeton sur la chaîne
    rng  $\leftarrow$  myRings[msg.agentconcerned];
    SendMsg :ngd(rng.previousagent(), rng.agentconcerned, rng.timestamp,
      rng.domain);
  end if
end if
```

9 Analyse de l’algorithme Ring_AB_T

Hormis le gain de confidentialité des agents supérieurs qui peuvent éviter d’envoyer trop de nogoods à leurs agents inférieurs, d’autres bonnes propriétés du point de vue de la confidentialité peuvent être mises à jour avec l’utilisation des jetons transitant sur les chaînes d’agents. Nous allons donc caractériser premièrement quels nogoods peuvent être évités (section 9.1) et ensuite dans quelle mesure les messages **rng** modifient la connaissance que peuvent acquérir les agents (section 9.2 et 9.3).

9.1 Quantification des nogoods éliminés

Du point de vue de la confidentialité nous avons pu voir que les nogoods émis d’un agent supérieur à un agent inférieur était le paramètre essentiel pour l’efficacité de la segmentation (cf. section 7.3). Le but de l’algorithme Ring_AB_T est de diminuer le nombre de nogoods émis qui ne sont pas “nécessaires”. Nous entendons par “nécessaire” le fait que des nogoods peuvent être évités si une concertation des agents inférieurs à lieu avant qu’ils n’émettent leur proposition à leur agent supérieur.

Les nogoods qui sont éliminés correspondent à des ensembles de propositions des agents inférieurs qui conduisent à interdire à leur supérieur commun toutes ses valeurs possibles. Formellement ils sont donc caractérisés par un ensemble de n-uplets de valeurs des agents inférieurs qui conduisent à interdire toute valeur pour X_i . Si on note les agents de $\Gamma^+(X_i) : X_1, \dots, X_k$ alors l’ensemble des nogoods qui peuvent être éliminés durant le processus correspond à l’ensemble suivant :

$$EliminatedNogoods = \{(v_1, \dots, v_k) \in D(X_1) \times \dots \times D(X_k) / \cup_{j \in [1..k]} \Delta_{v_j}(X_i) = D(X_i)\}$$

Il est important de rappeler que le nombre des tuples correspondant aux nogoods effectivement éliminés durant le processus peut être différent à chaque exécution de l’algorithme sur un même problème. En effet l’asynchronisme des algorithmes ne permet pas de déterminer à l’avance le fil de l’exécution et l’ordre dans lequel les tuples seront énumérés. La mesure du nombre de tuples éliminés ne peut donc se faire qu’en moyenne sur plusieurs instances du même problème.

Le nombre maximal de tuples qui peuvent être éliminés correspond au cardinal de l’ensemble *EliminatedNogoods* et chaque tuple de cet ensemble pouvant être généré pendant l’exécution du processus. On peut constater qu’il existe une balance entre le nombre de nogoods potentiellement éliminés et le nombre de nogoods effectivement éliminés. En effet si on considère que le réseau de contrainte est composé de contraintes dures (i.e. qui interdisent en moyenne beaucoup de tuples) alors le nombre de nogoods potentiellement éliminés est grand. En effet un grand nombre de combinaisons de valeurs conduisent à interdire tout le domaine d’ X_i . A l’inverse si les contraintes sont dures le nombre de nogoods effectivement éliminés diminuent car pour être généré un n-uplet de valeur de l’ensemble *EliminatedNogod* doit avant tout être autorisé par tous les agents

ce qui est moins souvent le cas quand les contraintes sont dures. On peut reformuler cette balance en remarquant que des contraintes dures ont tendance à faire rapidement grossir les ensembles $\Delta_{v_1, \dots, v_k}(X_i)$, et causer donc rapidement un “BackTrack du jeton” sur la chaîne d’agents, mais en même temps comme beaucoup de valeurs sont interdites pour chaque agent il y a moins de combinaisons possibles à énumérer.

9.2 Connaissance qu’a un agent supérieur

Les agents supérieurs ayant sensiblement le même comportement que dans l’algorithme DisFC-1ph leur gain d’information est presque similaire. En effet les agents supérieurs disposent des mêmes hypothèses concernant la valeur prise par leurs voisins inférieurs. Néanmoins leur connaissance des “demi-contraintes” possédées par les agents inférieurs est grandement diminuée. En effet dans l’algorithme DisFC-1ph une fois que l’agent supérieur dispose de la valeur de l’agent inférieur il lui suffit de la combiner avec l’ensemble de valeur interdites qu’il a reçu pour reconstituer une partie des tuples interdits par la “demi-contrainte” (cf. 5.3).

Dans l’algorithme Ring-ABT les agents supérieurs reçoivent un ensemble de valeurs interdites par au moins un des agents inférieurs, néanmoins ils ignorent quel(s) agent(s) est(sont) responsable(s) de l’interdiction de chaque valeur.

Les valeurs contenues dans un ensemble $\Delta_{v_1, \dots, v_k}(X_i)$ sont par définition interdites par au moins un agent de la chaîne. Il y a donc pour chaque valeur interdite $card(\Gamma^+(X_i))!$ possibilités d’agents en cause dans cette interdiction. On a donc au total

$(card(\Gamma^+(X_i)))^{card(\Delta_{v_1, \dots, v_k}(X_i))}$ possibilités si on considère toutes les valeurs interdites, on constate donc qu’il est quasiment impossible de les énumérer toutes. Par exemple un message **rng** contenant 40 valeurs interdites provenant d’une chaîne composée de seulement 5 agents produit plus de combinaisons que le nombre d’atomes présumé de l’univers ($\simeq 10^{80}$).

9.3 Connaissance qu’ont les agents sur la chaîne

Dans cette section nous détaillons les différentes hypothèses que les agents inférieurs peuvent formuler à propos de leur agent supérieur commun. Nous constaterons que la force de leurs hypothèses est liée à leur position sur la chaîne, en effet nous verrons que le gain d’information est croissant sur la chaîne d’agents. Dans cette section on note par $\{X_1, X_2, \dots, X_{k-1}, X_k\}$ la chaîne d’agents liée à l’agent X_i .

9.3.1 Les “nogoods de chaîne” et la segmentation :

Les “nogoods de chaîne” bénéficient des bonnes propriétés, du point de vue de la confidentialité, des nogoods représentant les BackTracks classiques, dans la mesure où ils peuvent être relayés et dûs à des causes différentes qui sont indistinguables. En effet dans l’algorithme Ring-ABT il y a 2 raisons possibles pour la réception d’un “nogood de chaîne” par un agent $X_j \neq X_k$:

- soit $\Delta_{v_1, \dots, v_k}(X_i) \neq D(X_i) \wedge v_i \in \Delta_{v_1, \dots, v_k}(X_i)$, i.e. le message **rng** est remonté jusqu’à X_i et X_i l’a rejeté. Ensuite les agents ultérieurs à X_j sur la chaîne n’ont pas réussi à faire accepter une autre proposition à X_i .
- soit $\forall (v_{j+1}, \dots, v_k) \in D(X_{j+1}) \times \dots \times D(X_k) / v_{j+1}, \dots, v_k$ **NotEliminatedByNogoodStore** $\Rightarrow \Delta_{v_1, \dots, v_k}(X_i) = D(X_i)$
i.e. toutes les valeurs autorisées qui sont prises par les agents ultérieurs à X_j conduisent à former un ensemble $\Delta_{v_1, \dots, v_k}(X_i)$ qui couvre systématiquement $D(X_i)$.

Une différence fondamentale, du point de vue de la confidentialité, distingue ces deux cas : dans le premier cas X_i a refusé la proposition révélant donc des informations concernant sa valeur, dans le second cas X_i n’a rien pu révéler car la proposition n’a pas pu remonter jusqu’à lui. Un agent sur la chaîne ne peut distinguer si le “nogood de chaîne” qu’il reçoit est représentatif de l’un ou l’autre cas.

Cette constatation transforme profondément la segmentation qui était utilisé pour l’analyse de DisFC-1ph (cf. 7.3), en effet les nogoods qui étaient reçus par les agents inférieurs leur permettaient de connaître jusqu’où leur agent supérieur avait reçu leurs propositions. De la sorte il était possible de construire les sous ensembles de messages compatibles. Étant donné que dans Ring-ABT les agents ne peuvent savoir si l’échec qui leur est signalé par un “nogood de chaîne” est dû à X_i ou à un agent ultérieur sur la chaîne, les agents inférieurs ne peuvent donc plus segmenter avec certitude leurs ensembles de messages.

De ce fait pour les agents de la chaîne hormis le dernier, l’application de la relation \mathfrak{R}_2 leur permettant de savoir si l’agent X_i a accepté leur propositions doit tout être précédée par une segmentation plus complexe que pour DisFC-1ph. En effet la segmentation doit être construite à l’aide d’une heuristique permettant de déterminer si la cause de l’échec sur la chaîne est due à X_i ou non.

Une fois cette segmentation achevée les agents peuvent construire les relations \mathfrak{R}_2 sur leur sous ensembles comme pour DisFC-1ph en remplaçant les paires (OK, OK) par des paires (RNG, OK) .

On constate que de la sorte on diminue les performances de la relation \mathfrak{R}_2 en contraignant son application par l’usage d’une heuristique.

9.3.2 Connaissance des agents sur la chaîne hormis le dernier

Nous considérons ici que les agents inférieurs hormis le dernier ont déjà construit leur segmentation à l’aide d’une heuristique. Nous nous intéressons aux différentes hypothèses que peuvent formuler les agents de la chaîne, sauf le dernier, car seul le dernier dispose d’un cadre identique à DisFC-1ph pour formuler ses hypothèses.

Le gain des agents sur la chaîne est dissymétrique. En effet lors du transit d’un message **rng** un agent combine l’ensemble de valeurs interdites jusque là, avec l’ensemble de valeurs que l’agent interdit lui même. Toutefois il est important de remarquer que les valeurs qui ne sont pas encore interdites par un agent

sur la chaîne peuvent le devenir à cause d'un agent ultérieur sur la chaîne.

Formellement pour un agent $X_j \neq X_k$ on a :

$$\forall v_t \notin \Delta_{v_1, \dots, v_j}(X_i) \not\Rightarrow v_t \notin \Delta_{v_1, \dots, v_k}(X_i)$$

A l'inverse comme les ensembles $\Delta_{v_1, \dots}(X_i)$ ne peuvent que s'agrandir les agents peuvent supposer :

$$\forall v_t \in \Delta_{v_1, \dots, v_j}(X_i) \Rightarrow v_t \in \Delta_{v_1, \dots, v_k}(X_i)$$

On constate donc que le gain d'information est croissant au fur et à mesure que l'ensemble $\Delta_{v_1, \dots, v_j}(X_i)$ grossit. En effet en terme de VPS la taille de cet ensemble détermine le nombre de probabilités qui seront annulées en cas d'acceptation du message **rng** par X_i (i.e. si $v_i \notin \Delta_{v_1, \dots, v_k}(X_i)$). Les agents de la fin de la chaîne ont plus de chance de disposer d'informations plus précises que ceux du début.

9.3.3 Connaissance du dernier agent de la chaîne

Dans la section précédente nous avons pu voir que les “nogoods de chaîne” ne permettaient pas aux autres agents de déterminer si l'échec venait de X_i ou d'un agent ultérieur sur la chaîne. Nous avons vu de même que les valeurs qui n'étaient pas interdites par les agents sur la chaîne pouvaient le devenir à cause d'un agent ultérieur. Ces deux constatations ne sont évidemment plus valables pour le dernier agent (X_k) de la chaîne car : d'une part il est le seul à savoir si la proposition a été transmise à X_i et d'autre part l'ensemble des valeurs interdites que transmet X_k est maximal (i.e. plus aucune autre valeur ne pourra être interdite ensuite).

L'agent X_k dispose donc d'hypothèses bien plus fortes que les autres agents sur la chaîne. En outre il est le seul à pouvoir construire sa segmentation et appliquer la relation \mathfrak{R}_2 de la même manière que pour DisFC-1ph. Toutefois comme le nombre de nogoods provenant de X_i est réduit sa segmentation sera moins efficace que dans l'algorithme DisFC-1ph.

9.4 Conclusion

L'algorithme Ring-ABT permet d'assurer plus de confidentialité pour les agents de différentes manières :

Tout d'abord les agents inférieurs exposent moins les tuples de leurs “demi-contraintes” car en fusionnant leurs ensembles de valeurs interdites il est quasi impossible pour l'agent supérieur de discerner les agents responsables de l'interdiction de chaque valeur.

Ensuite les agents supérieurs exposent moins leurs valeurs aux agents de la chaîne. En effet tous les agents de la chaîne excepté le dernier ne peuvent plus appliquer directement la relation \mathfrak{R}_2 car ils doivent dorénavant construire leur segmentation avec l'aide d'une heuristique. Cette heuristique sert à déterminer si les “nogoods de chaîne” reçus sont dûs au refus des propositions par l'agent supérieur, et peuvent donc servir d'accusés de réception.

Enfin les agents supérieurs exposent moins leurs valeurs au dernier agent de leur chaîne car même si ces agents peuvent appliquer la relation \mathfrak{R}_2 comme dans DisFC-1ph, leur segmentation est toutefois moins efficace car une partie

des nogoods émis par l'agent supérieur sont éliminés.
On peut conclure que l'algorithme Ring-ABT rend donc en priorité plus difficile l'acquisition de connaissance des agents inférieurs, afin de réduire le gain disproportionné de connaissance qu'accordait l'algorithme DisFC-1ph à ceux-ci.

Cinquième partie

Perspectives

10 Adaptation du protocole 3 aux problèmes d'optimisation

Les différents algorithmes du protocole 3 ne concernent que les problèmes de décision. Il est tout à fait envisageable de les étendre aux problèmes d'optimisation. En effet les messages **ok?** ou **rng** peuvent contenir en plus des ensembles de valeurs incompatibles le coût de l'instanciation courante. Il est possible de remplacer le test de consistance avec les valeurs reçues dans les messages **ok?** ou **rng**, par le test du coût de l'instanciation courante, et envoyer d'un nogood que dans le cas où ce coût dépasserait le coût maximal autorisé.

11 Concernant l'ordre des agents

11.1 Une permutation sur la chaîne d'agents

Cette idée d'amélioration concerne l'algorithme Ring-ABT. Dans cette partie nous considérons l'agent X_i comme l'agent supérieur initiateur des messages **rng**.

Comme nous l'avons vu le gain d'information entre agents présents sur une même chaîne est disymétrique, le dernier agent dispose toujours de meilleurs hypothèses concernant X_i . En outre le dernier agent de chaque chaîne est le seul à pouvoir construire une segmentation et appliquer les relations \mathfrak{R}_2 comme dans DisFC-1ph (cf. 5.2.3). Afin de limiter la connaissance du dernier agent il est possible à chaque envoi d'un message **rng** de l'agent X_i , de permuter l'ordre des agents de $\Gamma^+(X_i)$ sur la chaîne. De la sorte à chaque changement de valeur de X_i les agents n'auront pas le même gain d'information, et le dernier agent ne sera plus le même.

Toutefois cette première approche continue de favoriser le dernier agent de la chaîne tant que X_i ne change pas de valeur (et n'envoie pas donc un nouveau message **rng**).

Une seconde approche serait de permuter également les agents à chaque nogood qu'envoie X_i . Le dernier agent de la chaîne le recevant ne serait donc plus à chaque fois le même. Cependant cette méthode risque de ne pas faire générer tout les n-uplets possibles de valeurs d'agents sur la chaîne, et faire perdre du même coup la complétude de l'algorithme.

11.2 Un ordre dynamique d'agent

Le protocole 3 octroie des gains disymétriques d'information entre un agent supérieur, et ses agents inférieurs. Une nouvelle voie de développement d'algorithme pourrait conduire à dynamiquement changer l'ordre des agents afin de réduire les écarts de gain d'information en les nivelant vers le bas. De ce fait les agents ne seraient plus en permanence soit en position de faiblesse soit en

position de force pour obtenir des informations. Ce type de solutions ont déjà été entreprises dans [23] et [18], pour augmenter l'efficacité de l'exploration de l'espace de recherche. Toutefois ces méthodes entraînent soit un nombre exponentiel de place pour stocker les nogoods ([23]), soit octroient un gain d'efficacité peu significatif [18] par rapport à une version centralisée.

En poussant l'idée de ré agencement de l'ordre des agents à l'extrême, il serait possible de concevoir un système sans aucun ordre, où la solution émergerait à la manière des travaux concernant la vie artificielle (cf. [19]).

12 Une analyse de déviance de comportement

Une voie d'intérêt dans les applications distribuées consiste à identifier un comportement déviant d'un agent par rapport au protocole qu'il est censé utilisé. En effet dans le protocole 3 rien n'empêche un agent X_j de continuer à faire des propositions à l'agent X_i même si X_j et X_i ont déjà des valeurs consistantes pour $C_{i(j)}$ et $C_{(i)j}$. De la sorte l'agent X_j peut continuer à préciser sa connaissance de l'état d' X_i . Afin de détecter ce genre de comportement il est envisageable d'adapter le système de "Vickrey-Clarke-Groves tax" (VCG) aux DisCSPs. Ce système permet d'établir un système de taxe entre les agents afin d'éviter que l'un d'entre eux ne fasse trop de propositions par rapport aux informations qu'il révèle aux autres concurrents.

Ce système fut employé pour la première fois dans un contexte de DisCSP dans [26], de même une adaptation de l'algorithme DPOP fut effectuée avec le système VCG dans [13]. Toutefois des travaux restent à faire pour l'adapter aux différentes versions du protocole 3.

A Algorithmes d'ABT

Algorithm 10 ABT : Main procedure

```
myValue ← empty;  
end ← false;  
Compute  $\Gamma^+$ ,  $\Gamma^-$  ;  
CheckAgentView() ;  
while  $\neg$ end do  
  msg ← GetMsg() ;  
  switch (msg.type)  
    ok? : ProcessInfo(msg) ; break ;  
    ngd : ResolveConflict(msg) ; break ;  
    adl : SetLink(msg) ; break ;  
    stp : end ← true ;  
end while
```

Algorithm 11 ABT : CheckAgentView(*msg*)

```
if ( $\neg$ consistent(myValue, myAgentView)) then  
  myValue ← ChooseValue() ;  
  if (myValue) then  
    for each child child  $\in \Gamma^+(self)$  do  
      SendMsg :ok?(msg.sender, myValue) ;  
    end for  
  else  
    BackTrack() ;  
  end if  
end if
```

Algorithm 12 ABT : ProcessInfo(*msg*)

```
UpdateAgentView(msg.assign) ;  
CheckAgentView() ;
```

Algorithm 13 ABT : ResolveConflict(*msg*)

```
if (coherent(msg.Nogood,  $\Gamma^-(self) \cup self$ )) then  
  CheckAddLink(msg) ;  
  add(msg.Nogood, myNogoodStore) ;  
  myValue ← empty ;  
  CheckAgentView() ;  
else  
  if (coherent(msg.Nogood, self)) then  
    SendMsg :ok?(msg.sender, myValue) ;  
  end if  
end if
```

Algorithm 14 ABT : BackTrack()

```
newNogood  $\leftarrow$  solve(myNogoodStore);  
if newNogood = empty then  
  end  $\leftarrow$  true;  
  sendMsg :stp(system);  
else  
  sendMsg :ngd(newNogood);  
  UpdateAgentView(rhs(newNogood)  $\leftarrow$  unknow);  
  CheckAgentView();  
end if
```

Algorithm 15 ABT : ChooseValue()

```
for each  $v \in D(self)$  not eliminated by myNogoodStore do  
  if (consistent( $v$ , myAgentView[ $\Gamma^-(self)$ ])) then  
    return  $v$ ;  
  else  
    add( $x_j = val_j \Rightarrow self \neq v$ , myNogoodStore);  
  end if  
end for  
return empty;
```

Algorithm 16 ABT : UpdateAgentView(newAssig)

```
add(newAssig, myAgentView);  
for each  $ng \in myNogoodStore$  do  
  if  $\neg$ (coherent(lhs( $ng$ ), myAgentView)) then  
    remove( $ng$ , myNogoodStore);  
  end if  
end for
```

Algorithm 17 ABT : Coherent(nogood, agents)

```
for each  $var \in nogood \cup agents$  do  
  if  $nogood[var] \neq myAgentView[var]$  then  
    return false;  
  end if  
end for  
return true;
```

Algorithm 18 ABT : SetLing(msg)

```
add(msg.sender,  $\Gamma^+(self)$ );  
sendMmsg :ok?(msg.sender, myValue);
```

Algorithm 19 ABT : CheckAddLink(msg)

```
for each var  $\in$  lhs(msg.Nogood) do  
  if var  $\notin \Gamma^-(self)$  then  
    SendMsg :adl(var,self);  
    add(var, $\Gamma^-(self)$ );  
    UpdateAgentView(var  $\leftarrow$  varValue);  
  end if  
end for
```

B Algorithme du protocole 3

Algorithm 20 Protocole 3 : Main procedure

```
myValue ← ChooseValue();  
end ← false;  
timestamp ← randomInt();  
FilteredDomain ←  $\emptyset$ ;  
Compute  $\Gamma^+$ ,  $\Gamma^-$ ;  
CheckAgentView();  
while  $\neg$ end do  
  msg ← GetMsg();  
  switch (msg.type)  
    ok? : ProcessInfo(msg); break;  
    yes : break;  
    ngd : ResolveConflict(msg); break;  
    adl : SetLink(msg); break;  
    stp : end ← true;  
end while
```

Algorithm 21 Protocole 3 : UpdateDomain(*msg*)

```
for each val ∈ msg.domain do  
  #on retire toutes les valeurs incompatibles du domaine  
  add(msg.sender = msg.timestamp ⇒ self ≠ val, mygoodstore);  
end for
```

Algorithm 22 Protocole 3 : incompatible(*agentDomain*, *myValue*)

```
array ←  $\emptyset$ ;  
for each val ∈ agentDomain do  
  if ((myValue, val) ∉  $C_{self,agent}$ ) then  
    array ← array ∪ {val};  
  end if  
end for  
return array;
```

Algorithm 23 Protocole 3 : ProcessInfo(msg)

```
0: UpdateAgentView(msg);
  if ( $msg.sender \in \Gamma^+(self)$ ) then
     $FilteredDomain[msg.sender] \leftarrow msg.domain$ ;
    if ( $myValue \in msg.domain$ ) then
      # si l'agent inférieur interdit la valeur de self, self lui demande de changer
      de valeur
      SendMsg :ngd(msg.sender,  $self = self.timestamp \Rightarrow msg.sender \neq$ 
       $msg.timestamp$ );
    else
      # si l'agent inférieur n'interdit pas la valeur de self, self accepte la pro-
      position
      SendMsg :yes(msg.sender,  $self = self.timestamp$  compatible with
       $msg.sender = msg.timestamp$ );
    end if
  end if
  if ( $msg.sender \in \Gamma^-(self)$ ) then
    UpdateDomain(msg);
    if ( $myValue \in msg.domain$ ) then
      # si la valeur de self est interdite par un agent supérieur
      # self tente de changer sa valeur
      CheckAgentView();
    else
      # sinon il propose l'ensemble de valeurs interdites à son agent supérieur
      SendMsg :ok?(msg.sender, timestamp, incompatible(D(sender), myVa-
      lue));
    end if
  end if


---


```

Algorithm 24 Protocole 3 : CheckAgentView()

```
if (myValue isEliminatedBy nogoodstore) then
  myValue  $\leftarrow$  ChooseValue();
  if (myValue  $\neq$  empty) then
    timestamp = timestamp + 1;
    for each agent  $\in \Gamma^+(self) \cup \Gamma^-(self)$  do
      # self propose à tous les agents voisins leur nouvel ensemble de valeurs
      interdites
      SendMsg :ok?(agent, timestamp, incompatible(D(agent), myValue));
    end for
    for each agent  $\in \Gamma^+(self)$  do
      if (myValue  $\in$  FilteredDomain[agent]) then
        # self previent tous les agents inférieurs dont les propositions ne
        sont plus valides
        SendMsg :ngd(agent, self = self.timestamp  $\Rightarrow$  agent  $\neq$ 
        agent.timestamp);
      end if
    end for
  else
    BackTrack();
  end if
end if
```

Références

- [1] The Asynchronous Backtracking Family. C. Bessiere, I. Brito, A. Maestre, P. Meseguer. 2003
- [2] Asynchronous Backtracking without Adding Links : A New Member in the ABT Family. C. Bessière, I. Brito, A. Maestre, P. Meseguer. 2005
- [3] Thesis : Distributed Constraint Satisfaction. I.Brito, 2007
- [4] Privacy/Efficiency Tradeoffs in Distributed Meeting Scheduling by Constraint-Based Agents. C.Freuder, M. Minca and R.J. Wallace. 2001.
- [5] Multi-agent constraint systems with preferences : Efficiency, solution quality and privacy loss. Franzin, M.S., F. Rossi, and R.Wallace. 2004
- [6] Analysis of Privacy loss in Distributed Constraint Optimization. Greenstadt R., Pearce P. and Tambe M. 2006
- [7] Collaborative scheduling : Threats and promises. In Workshop on Economics and Information Security. Greenstadt R., and Smith M. 2006
- [8] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In Proceeding ECAI'98, pages 219-223, Brighton, UK, 1998.
- [9] Privacy Loss in Distributed Constraint Reasoning : A quantitative Framework for Analysis and its Applications. Maheswaran et al. 2005
- [10] Solving Distributed Constraint Optimization Problems using Cooperative Mediation. Proceeding of the 3rd AAMAS, pp. 438-445. R.Mailler and V.Lesser. 2004.
- [11] P. Meseguer and M.A. Jiménez. Distributed forward checking. In Proceedings CP'00 workshop on Distributed Constraint Satisfaction, Singapore, 2000.
- [12] An Asynchronous Complete Method for General Distributed Constraint Optimization. J; Modi, W.M. Shen, M. Tambe, and M. Yokoo.
- [13] MDPOP : Faithful Distributed Implementation of Efficient Social Choice Problems. Proceedings of the 5th AAMAS. A. Pectou, B. Faltings and D. Parkes. 2006.
- [14] Asynchronously Solving Distributed problems with Privacy Requirement. Ph.D. Thesis 2601, EPFL, M.Silaghi. 2002
- [15] Nogood-based Asynchronous Distributed Optimization (ADOPT-ng). Proceedings of the 5th AAMAS. M. Silaghi, and M. Yokoo. 2006
- [16] Asynchronous Search with Aggregations. Proceedings of the 17th AAAI, pp. 917-922. M. Silaghi, D. Sam-Haroud and B.Faltings. 2000
- [17] Asynchronous Aggregation and Consistency in Distributed Constraint Satisfaction. Artificial Intelligence **161**(1-2) pp.25-54. M.Silaghi and B.Faltings. 2005
- [18] ABT with Asynchronous Reordering. Proceedings of the 2nd A-P Conference on Intelligent Agent Technology IEEE press, pp.54-63. M. Silaghi, D.Sam-Haroud and B.Faltings. 2001.

- [19] Solving Permutation Constraint Satisfaction Problems with Artificial Ants. Christine Solnon, 2000
- [20] The distributed constraint satisfaction problem : Formalization and algorithms. M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara. 1998
- [21] Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. Proceedings of the 12th ICDCS, pp.614-624. M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara. 1992
- [22] Secure Distributed Constraint Satisfaction : Reaching Agreement without Revealing Private Information. *Artificial Intelligence* **161**(1-2), pp.229-246. M.Yokoo, K.Suzuki, and K.Hirayama. 2005
- [23] Weak-commitment Search for Solving Constraint Satisfaction Problems. Proceedings of the 12th AAAI, pp.313-318. M. Yokoo. 1994
- [24] Dynamic Ordering for Asynchronous Backtracking on DisCSPs. Proceedings of the 11th CP, Springer LNCS 3709, pp.32-46. R.Zivan and A.Meisels. 2005
- [25] SSDPOP : Using secret sharing to improve the privacy of DCOP. AAAI-07, page 78
- [26] Incentive-compatible Open Constraint Optimization. Proceedings of 4th ACM Conference on Electronic Commerce. 2003.