Université de Valenciennes et du Hainaut-Cambrésis

Thèse de doctorat

Pour obtenir le grade de Docteur de l'Université de

VALENCIENNES ET DU HAINAUT-CAMBRESIS

Discipline, spécialité selon la liste des spécialités pour lesquelles l'Ecole Doctorale est accréditée :
Informatique

Présentée et soutenue par Pierre, Monier.

Le 12/03/2012, à Valenciennes

Ecole doctorale :
Sciences Pour l'Ingénieur (SPI)

Equipe de recherche, Laboratoire :
Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH)

DBS multi-variables pour des problèmes de coordination multi-agents

JURY

Président du jury
- Koukam, Abderrafiâa. Professeur des Universités. Université de Belfort.

Rapporteurs
- Gleizes, Marie-Pierre. Professeur des Universités. Université Paul Sabatier de Toulouse (IRIT).

- Quinqueton, Joël. Professeur des Universités. Université de Montpellier III (LIRMM).

Examinateurs
- Koukam, Abderrafiâa. Professeur des Universités. Université de Belfort.

- Guessoum, Zahia. Maître de conférences HDR. Université Paris VI.

Directeur de thèse

- Mandiau, René. Professeur des Universités. Université de Valenciennes et du Hainaut-Cambrésis.

Co-directeur de thèse : Piechowiak, Sylvain. Professeur des Universités. Université de Valenciennes et du Hainaut-Cambrésis.

Co-encadrant : Vion, Julien. Maître de conférences. Université de Valenciennes et du Hainaut-Cambrésis.

Membres invités
- Doniec, Arnaud. Maître assistant. Ecole des Mines de Douai.

Remerciements

Je tiens à remercier etc ...

« Beaucoup encore il te reste à apprendre. »
M.Yoda

Table des matières

Table des figures

Résumé des notations utilisées

- $\mathscr{X}$: ensemble des variables X_i,
- $\mathrm{dom}(X_i)$: domaine de la variable X_i ,
- $\mathscr{D}$: ensemble des domaines $\mathrm{dom}(X_i)$ de chaque variable X_i ,
- $\mathscr{C}$: ensemble des contraintes C_i,
- $\mathscr{A}$: ensemble des agents A_i,
- $\mathrm{var}(A_i)$: ensemble des variables encapsulées par un agent A_i,
- $\mathrm{var}(C)$: ensemble des variables impliquées par une contrainte,
- d : la taille maximale des domaines des variables du DisCSP,
- e : le nombre de contraintes du DisCSP,
- δ_{global} : la densité des contraintes inter-agents (compris entre 0 et 100 %),
- t_{global} : la dureté des contraintes inter-agents (compris entre 0 et 100 %),
- δ_{local} : la densité des contraintes intra-agent,
- t_{local} : la dureté des contraintes intra-agent,
- m : le nombre d'agents du DisCSP,
- n : le nombre maximal de variables d'un agent,
- N : le nombre de variables du DisCSP (on a donc $N \leq nm$),
- $\succ$: ordre total de priorité entre les agents,
- $\Gamma(A_i)$: ensemble des accointances d'un agent A_i,
- $\Gamma(A_i)^-$: ensemble des accointances de priorité supérieure d'un agent A_i,
- $\Gamma(A_i)^+$: ensemble des accointances de priorité inférieure d'un agent A_i.

Introduction générale

De nombreuses applications distribuées (demandant l'utilisation d'outils collaboratifs telles que la prise de rendez-vous de différentes personnes ayant leur propre agenda, la gestion d'avions dans un aéroport, la simulation de trafic routier, etc.) nécessitent des mécanismes de distribution et de partage de données (éventuellement confidentielles). Ce besoin de décentralisation du contrôle et des données soulèvent de nombreuses problématiques scientifiques allant de la modélisation aux outils informatiques en passant par des aspects de formalisation. De nombreux chercheurs tentent de répondre à ces besoins selon différentes facettes, par exemple la protection des données privées, la gestion des bases de données hétérogènes, le passage à l'échelle, la confiance des données transmises et reçues, etc. Depuis une trentaine d'années, des chercheurs du domaine des systèmes multi-agent (SMA) essaient de proposer des solutions.

Les études en SMA s'intéressent principalement aux mécanismes liés aux problématiques de coordination et d'interaction entre différentes entités autonomes. La coordination inter-agents peut s'effectuer de différentes manières. Nous considérons, dans le travail présenté ici, que la coordination peut être définie comme un processus de recherche basé sur la résolution de DisCSP (*Distributed Constraint Solving Problem*) [Yokoo, 2001]. Les DisCSP sont situés à l'intersection des domaines des SMA et des CSP (*Constraint Solving Problem*).

Dans de tels contextes, la coordination multi-agent peut être vue comme la décomposition d'un problème en sous-problèmes, la résolution des sous-problèmes, les mécanismes d'échanges de résultats partiels ou la diffusion des solutions des sous-problèmes aux agents, jusqu'à l'obtention de la (des) solution(s) globale(s). Les raisonnements intra-agent et inter-agents sont basés sur un ensemble de relations entre différentes variables. La résolution du problème s'effectue grâce aux agents qui interagissent afin d'obtenir une solution globale à partir des solutions locales.

De nombreux algorithmes permettant la résolution de DisCSP existent dans la littérature tels que *Multi-ABT* [Hirayama *et al.*, 2000, Hirayama *et al.*, 2004], *Multi-AWC* [Yokoo et Hirayama, 1998] et *AFC* [Meisels et Zivan, 2007]. Ces algorithmes présentent plusieurs limites :

- les agents n'interagissent pas complètement de manière asynchrone,
- les agents peuvent créer des agents virtuels pour chacune de leurs variables locales, augmentant ainsi considérablement la complexité du problème si les agents virtuels ne sont pas correctement gérés,
- des créations de liens entre les agents apparaissent au cours de la recherche,
- des nogoods coûteux en calculs sont utilisés par les agents.

Le travail présenté ici porte sur la proposition d'un algorithme complet de résolution de DisCSP multi-variables par agent, nommé *Distributed Backtracking with Sessions* (DBS) dont l'objectif est de prendre en compte les limites présentées ci-dessus.

Le chapitre 1 présentera de manière générale les systèmes multi-agents. Nous y expliquons ce qu'est un agent, un SMA ainsi que les différents éléments qui le composent. Puis le formalisme des CSP est détaillé. Nous présentons différents algorithmes de filtrage, de résolution ainsi que

des heuristiques permettant de réduire les temps de résolution. Enfin, nous nous focalisons sur les DisCSP, domaine intersectant celui des SMA et des CSP. Nous détaillerons ainsi différents algorithmes de filtrage distribué, des heuristiques permettant d'ordonner les agents ainsi que différentes extensions possibles aux DisCSP.

Le chapitre 2 sera consacré à la résolution des DisCSP. Nous verrons, pour commencer, des définitions formelles décrivant les DisCSP. Nous exposerons différents problèmes pouvant être formalisés et résolus grâce aux DisCSP. Nous verrons que la principale difficulté consiste à gérer l'asynchronisme entre les agents de manière à obtenir un comportement inter-agents cohérent. Pour réaliser cette gestion, il est indispensable d'affecter un contexte à chaque message échangé. Nous commencerons par nous intéresser à la résolution de DisCSP où le problème local d'un agent ne contient qu'une unique variable. Nous détaillerons différents algorithmes de recherche asynchrone tels que *ABT* pour *Asynchronous Backtracking* et *AWC* pour *Asynchronous Weak-Commitment*. Puis, nous nous concentrerons sur la résolution de DisCSP où le problème local de chaque agent est dit « complexe ». Nous verrons différentes méthodes permettant de généraliser un algorithme de résolution de DisCSP mono-variable par agent. Puis nous détaillerons différents algorithmes dont *Multi-ABT* qui est la généralisation de *ABT* ainsi que *Multi-AWC* qui est la généralisation de *AWC*. Nous verrons ensuite en détail les limites des algorithmes existants.

Le chapitre 3 sera consacré à la présentation de notre proposition nommée « *Distributed Backtracking with Sessions* (DBS) » permettant de résoudre des DisCSP multi-variables par agent. Nous présenterons les hypothèses et notations de cet algorithme, initialement présenté dans [Doniec *et al.*, 2005] puis étendu dans [Monier *et al.*, 2009a, Monier *et al.*, 2009b]. Puis nous nous intéresserons à la résolution du CSP local propre à chaque agent, au choix de la solution locale à communiquer aux accointances en fonction du contexte propre à chaque agent. Les propriétés de l'algorithme *DBS* seront détaillées. Nous exposerons la preuve de correction de cet algorithme avant de présenter la preuve de complétude. Puis la complexité spatiale et la complexité temporelle de *DBS* seront fournies. Enfin, nous détaillerons différents filtres que chaque agent peut effectuer sur sa liste de messages en attente de traitement dans sa boîte de réception et ceci, sans remettre en cause la preuve de complétude.

Le chapitre 4 portera sur l'évaluation de *DBS*. Nous commencerons par présenter les résultats obtenus dans le cas général où le problème local de chaque agent est complexe. Nous comparerons notre proposition aux algorithmes *Multi-ABT*, *Multi-AWC* et *AFC*. Afin d'évaluer ces algorithmes, nous utiliserons les deux principaux benchmarks de notre domaine : les DisCSP aléatoires et les problèmes de coloration de graphes distribués. Puis, nous traiterons le cas particulier des DisCSP où le problème local de chaque agent est composé d'une unique variable.

Puis, une autre évaluation de DBS sera réalisée sur un problème réel étudié en système multi-agent, à savoir le problème d'exploration multi-robots. Nous commencerons par présenter une manière de formaliser ce problème sous la forme d'un DisCSP. Ensuite, nous détaillerons les métriques que nous utiliserons pour évaluer notre proposition sur ce problème et nous présenterons les différents résultats obtenus. Pour ce cas particulier de DisCSP mono-variable par agent, nous nous comparerons aux algorithmes *ABT* et *AWC*.

Le mémoire s'achèvera avec des perspectives envisageables.

1

Etat de l'art : SMA et CSP

Sommaire

1.1 Introduction

Les études en système multi-agent (SMA) s'intéressent aux mécanismes liés aux problématiques de coordinations et d'interactions entre différentes entités [Ferber, 1995]. Nous considérons, dans ce cadre, que la coordination peut être définie comme un processus de recherche dans des contextes de résolution de DisCSP (Distributed Constraint Solving Problem) [Yokoo, 2001]. Les DisCSP peuvent être utilisés, par exemple pour résoudre de nombreux problèmes naturellement distribués tels que les problèmes d'emploi du temps [Tsuruta et Shintani, 2000], de trafic routier [Doniec *et al.*, 2008] ou d'exploration multi-robots [Monier *et al.*, 2010b]. Dans de tels contextes, la coordination peut être vue comme la décomposition d'un problème en sous-problèmes, la résolution des sous-problèmes, les mécanismes d'échanges de résultats partiels ou la diffusion des solutions des sous-problèmes aux agents, jusqu'à l'obtention de la (des) solution(s) globale(s). Les raisonnements intra-agent et inter-agents sont basés sur un ensemble de relations entre différentes variables. La résolution du problème s'effectue grâce aux agents qui interagissent afin d'obtenir une solution globale à partir des solutions locales. Les DisCSP sont parfaitement

adaptés pour résoudre des problèmes utilisant des données physiquement réparties et ne pouvant être résolus de manière centralisée.

Nous allons présenter de manière générale, en section 1.2, quelques généralités sur les systèmes multi-agents. Nous expliquerons ce qu'est un agent, un SMA ainsi que les différents éléments qui le composent. Puis la section 1.3 détaillera le formalisme des CSP (*Constraint Solving Problem*). Nous présenterons différents algorithmes de filtrage, de résolutions ainsi que des heuristiques permettant de réduire les temps de résolution. Enfin, nous insisterons en section 1.4 sur les DisCSP, domaine intersectant celui des SMA et des CSP. Nous détaillerons ainsi différents algorithmes de filtrage distribué, des heuristiques permettant d'ordonner les agents ainsi que différentes extensions possibles.

1.2 Approche multi-agents

Le domaine de l'Intelligence Artificielle Distribuée (IAD) s'intéresse aux problèmes de coordination et d'interaction entre différentes entités appelées *agents*. Un SMA est un système composé d'un ensemble d'agents, situés dans un environnement et interagissant afin d'atteindre un objectif. Les travaux en SMA se focalisent sur l'étude des comportements collectifs et sur l'émergence d'une intelligence à partir d'un ensemble d'agents plus ou moins autonomes.

Afin de définir de manière formelle un SMA, nous utilisons l'approche Voyelles [Demazeau, 1995] décrivant un SMA sur quatre points : Agent, Environnement, Interaction et Organisation : un SMA peut être vu comme un ensemble d'agents (A), situés dans un environnement (E), interagissant (I) dans une organisation (O) afin d'atteindre un but.

1.2.1 Eléments d'un système multi-agents

Agent

Le terme *agent* est très souvent utilisé, notamment en IA. Cependant, il n'existe pas une définition unique de ce terme. De nombreux articles ont proposé des définitions mais, à ce jour, aucune d'entre-elles ne fait l'unanimité dans ce domaine. La définition qui nous paraît la mieux adaptée pour le cadre des DisCSP est celle qu'a proposée Ferber [Ferber, 1995] :

Définition 1 (Agent) *On appelle agent une entité informatique qui :*

1. *se trouve dans un système informatique ouvert (ensemble d'applications, de réseaux et de systèmes hétérogènes),*
2. *peut communiquer avec d'autres agents,*
3. *est mue par un ensemble d'objectifs propres,*
4. *possède des ressources propres,*
5. *ne dispose que d'une représentation partielle des autres agents,*
6. *possède des compétences (services) qu'elle peut offrir aux autres agents,*
7. *a un comportement tendant à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose et en fonction de ses représentations et des communications qu'elle reçoit.*

Afin d'être autonome, un agent dispose d'un mécanisme lui permettant de réagir aux informations perçues de son environnement. Ce mécanisme cyclique est composé de trois phases : la perception, la décision et l'action, comme le montre la figure 1.1.

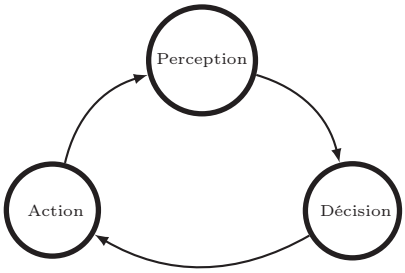

FIGURE 1.1 – Mécanisme de décision d'un agent

La phase de perception permet à un agent d'obtenir des informations sur son environnement. Dans le cas d'agents communicants (par exemple, les agents d'un DisCSP), la phase de perception consiste à recevoir les messages des autres agents. Les informations obtenues durant cette phase sont recueillies puis formalisées. Jusqu'à récemment, la phase de perception d'un agent était considérée comme relativement facile à réaliser, comparée à la phase de décision. Cependant, lorsqu'il s'agit d'agents physiques et non plus virtuels, cette phase est extrêmement difficile à réaliser [Coltin *et al.*, 2010].

La phase de décision, appelée aussi phase de cognition, permet à un agent d'effectuer un raisonnement à partir des informations qu'il a perçues de son environnement. Ce raisonnement a pour but de planifier les actions qu'il va effectuer. La phase d'action consiste à ce que l'agent effectue les actions qu'il a planifiées durant la phase de décision.

Les agents sont classés en deux principales catégories : les agents réactifs et les agents cognitifs. Les SMA basés sur des agents réactifs permettent d'obtenir un comportement global considéré comme intelligent alors que les raisonnements individuels sont simples, voire limités [Brooks, 1986]. Contrairement aux agents réactifs, les agents cognitifs possèdent une mémoire et peuvent utiliser un mécanisme de raisonnement complexe basé sur une représentation symbolique de leur environnement. C'est le cas, par exemple, des agents cognitifs utilisés dans des simulations de football [Ros *et al.*, 2009].

La séparation entre agents réactifs et cognitifs n'est pas évidente. Il existe des agents hybrides (étant à la fois cognitif et réactif) utilisant l'avantage de chaque approche comme le montre la figure 1.2 issue de [Ferber, 1995].

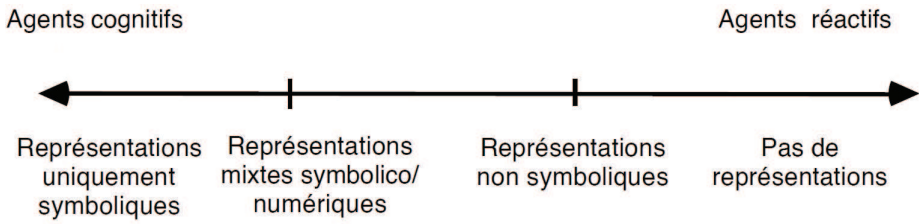

FIGURE 1.2 – Agents cognitifs, réactifs et hybrides

Environnement

Les agents d'un SMA font partie d'un environnement. Il s'agit d'un espace commun à tous les agents permettant à ces derniers d'interagir. Les agents peuvent se déplacer dans un environnement s'il s'agit d'agents situés. Les agents peuvent communiquer grâce à cet environnement en

laissant, par exemple, des informations sur ce dernier. Ils peuvent déplacer des objets présents dans cet environnement. Du point de vue d'un agent, l'environnement peut être constitué, en plus des objets présents, des autres agents du SMA. La composition d'un environnement dépend donc de la nature du problème.

Dans [Russell et Norvig, 1995], différentes propriétés caractérisant un environnement sont présentées. Un environnement est *déterministe s'il est possible de prévoir, pour toutes actions possibles, le résultat sur cet environnement. Dans le cas contraire, il s'agit d'un environnement non déterministe.*

Un environnement est dit statique si un état ne dépend que des états antérieurs et des actions réalisées par le système. À l'inverse, lorsqu'un état a été obtenu par les actions des agents sur l'environnement ou par une entité extérieure au SMA, il s'agit d'un environnement *dynamique.*

Un environnement est dit accessible lorsque chaque agent peut obtenir une vision globale de chaque élément composant l'environnement (y compris les agents eux-mêmes). Cependant, en général, lorsqu'un SMA est utilisé pour une application, les agents n'ont pas de vision globale.

L'environnement est considéré comme *ouvert lorsqu'il est possible d'ajouter des éléments extérieurs au SMA. Si ce n'est pas le cas, l'environnement est* fermé. Par exemple, pour les agents d'un DisCSP (définition 16 du chapitre 2), l'environnement est considéré comme ouvert s'il est possible d'ajouter en cours de résolution des variables, des contraintes ou des valeurs aux domaines.

Interaction

Les agents interagissent si nécessaire pour l'obtention de leurs buts. Ferber [Ferber, 1995], propose que l'interaction soit définie comme étant dépendante de trois composants : les buts des agents compatibles ou incompatibles, les ressources suffisantes ou insuffisantes et les compétences suffisantes ou insuffisantes. À partir de ces trois composantes, huit types de situations peuvent être obtenues : indépendance, collaboration simple, encombrement, collaboration coordonnée, compétition individuelle pure, compétition collective pure, conflits individuels pour des ressources et conflits collectifs pour des ressources.

Les agents peuvent interagir sans communiquer [Rosenschein, 1985], en utilisant un support de communications tels que les tableaux noirs [Erman *et al.*, 1980], en utilisant l'environnement [Drogoul, 1993] ou en utilisant un protocole de communication. Il existe de nombreux protocoles pour communiquer tels que la communication point à point (envoi de messages d'un agent à un autre agent), le broadcast (envoi de messages à tous les agents) ou alors l'envoi de messages à certains groupes d'agents (par exemple, ceux proches de l'agent émetteur de l'information). Des normes sont apparues afin d'établir un contexte à chaque message échangé comme par exemple la norme FIPA (Foundation for Intelligent Physical Agents) qui est utilisée dans la plateforme multi-agents JADE [Bellifemine *et al.*, 2000].

Dans le cadre des DisCSP (Cf. chapitre 2), les agents ont des buts compatibles : ils essayent de satisfaire l'ensemble des contraintes inter et intra agents afin d'obtenir une solution globale consistante construite à partir des solutions locales de chaque agent. Les agents ont des compétences insuffisantes pour atteindre leurs buts car ils ne peuvent pas obtenir seuls une solution globale. En effet, ils n'ont pas accès aux variables des autres agents. Les *ressources* des agents sont suffisantes si le DisCSP dispose d'au moins une solution, il s'agit dans ce cas d'une situation de collaboration simple. Lorsque le DisCSP ne contient aucune solution, les ressources sont alors insuffisantes et il s'agit d'une situation d'encombrement.

Organisation

De nombreuses définitions du terme « organisation » existent. Nous avons choisi de présenter celle de [Morin, 1977] :

Définition 2 *Une organisation peut être définie comme un agencement de relations entre composants ou individus qui produit une unité, ou système, dotée de qualités inconnues au niveau des composants ou individus. L'organisation lie de façon inter relationnelle des éléments ou événements ou individus divers qui dès lors deviennent les composants d'un tout. Elle assure solidarité et solidité relative, donc assure au système une certaine possibilité de durée en dépit de perturbations aléatoires.*

Une organisation peut être utilisée dans un SMA afin de structurer l'ensemble des agents. L'organisation peut être statique et définie au lancement du SMA ou émerger [Drogoul, 1993] au bout d'un temps fini. Par exemple, des organisations émergentes peuvent être observées dans des SMA composés d'agents réactifs tels que les SMA modélisant des sociétés animales [Dorigo, 1992].

Dans le cadre des DisCSP, l'organisation peut être statique ou dynamique. Lorsqu'elle est statique, chaque agent dispose d'une priorité définie avant de commencer la recherche de solutions [Yokoo, 2001]. Lorsqu'elle est dynamique [Yokoo, 2001] la priorité de chaque agent est modifiée en cours de résolution. Durant la résolution du DisCSP, l'agent de plus haute priorité essayera d'imposer sa solution courante aux agents de priorité inférieure qui devront s'adapter.

1.2.2 Du SMA au DisCSP

Un DisCSP [Yokoo, 2001] peut être vu comme un SMA dont les raisonnements intra-agent et inter-agents sont basés sur un ensemble de relations entre différentes variables. La résolution du problème s'effectue grâce aux agents qui interagissent afin d'obtenir une solution globale à partir des solutions locales de chaque agent. Les DisCSP sont parfaitement adaptés pour résoudre des problèmes utilisant des données physiquement réparties et ne pouvant être résolus de manière centralisée.

La figure 1.3 présente un SMA selon trois vues : la vue globale, la vue sociale et la vue locale :

- du point de vue global, nous avons un SMA plongé dans un environnement.
- du point de vue social, nous pouvons observer le SMA composé d'un ensemble d'agents interagissant entre eux. Pour des agents résolvant un DisCSP, ces interactions sont des envois de messages permettant aux agents de s'échanger leurs solutions locales dans le but d'obtenir une solution globale composée des solutions locales de chaque agent. Cette solution globale doit satisfaire des contraintes inter-agents reliant des variables de différents agents. Ces contraintes sont représentées ici par les flèches reliant les agents deux à deux.
- du point de vue local, nous observons le mécanisme de raisonnement d'un agent basé sur les trois phases de perception, décision et d'action. La perception s'effectue en recevant des messages concernant des valeurs des variables d'autres agents. La phase de décision s'effectue grâce à la résolution d'un CSP. L'agent essaye de trouver une solution locale qui soit consistante vis à vis des valeurs des variables des autres agents. Enfin, durant la phase d'action, l'agent émet des messages concernant la solution locale qu'il a choisie ou qu'il n'a pas pu choisir.

Nous allons maintenant détailler des notions de CSP, lesquels sont résolus durant la phase de décision d'un agent.

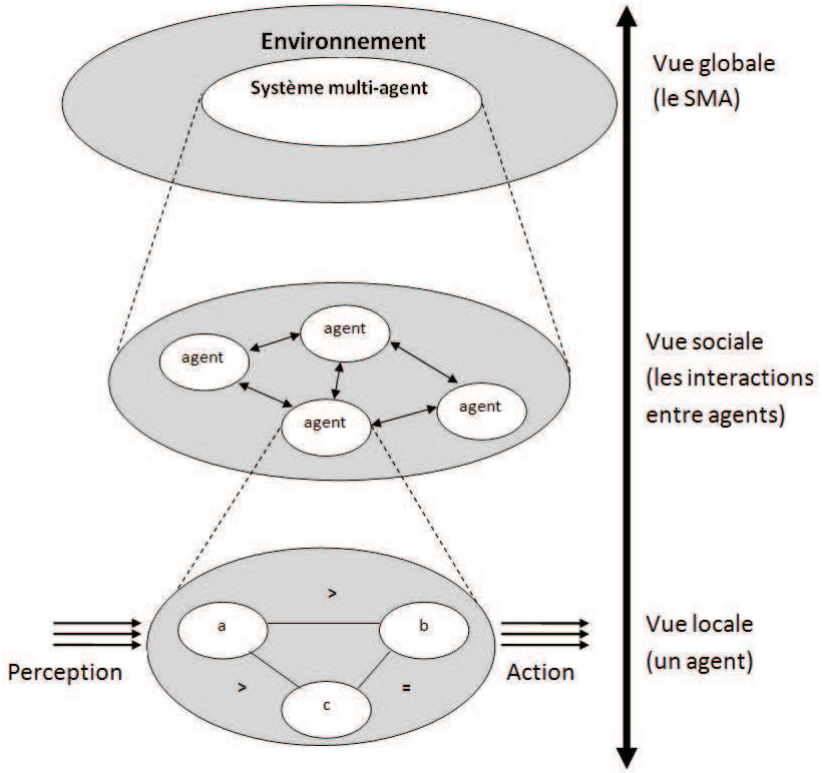

FIGURE 1.3 – Vue globale, locale et sociale d'un SMA

1.3 Problèmes de satisfaction de contraintes

Les problèmes de satisfaction de contraintes (CSP) sont des problèmes étudiés en Intelligence Artificielle. Il s'agit d'un formalisme permettant de représenter un problème sous la forme d'un ensemble de variables et de contraintes. Afin de résoudre un CSP, l'objectif est d'affecter une valeur à chaque variable de manière à ce que toutes les contraintes soient satisfaites. Les variables prennent leurs valeurs dans un domaine qui peut être discret ou continu. Les contraintes peuvent être unaires, binaires ou n-aires. En général, le but est de trouver une solution au problème mais il peut être demandé de fournir toutes les solutions du problème, celles maximisant une fonction objectif, ou encore le nombre de contraintes satisfaites. Il ne s'agit plus dans ce cas de CSP mais de COP pour *Constraint Optimization Problem.*

1.3.1 Définition

Un CSP centralisé peut être vu comme un ensemble de variables prenant leurs valeurs dans un domaine et devant satisfaire un ensemble de contraintes. Une définition plus formelle d'un CSP est :

Définition 3 (Problème de Satisfaction de Contraintes - CSP) *Un Problème de Satisfaction de Contraintes (CSP) est un triplet* $\langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$ *où :*

- $\mathscr{X} = \{X_1, X_2, \ldots, X_N\}$ *est un ensemble fini de* N *variables.*
- $\mathscr{D} = \{\mathrm{dom}(X_1), \mathrm{dom}(X_2), \ldots, \mathrm{dom}(X_N)\}$ *est l'ensemble des domaines associés à chaque variable.*

– $\mathscr{C} = \{C_1, C_2, \ldots, C_e\}$ *est un ensemble fini de e contraintes. Chaque contrainte* C_i *porte sur un sous-ensemble des variables du problème.*

Les contraintes C_i peuvent être définies en extension, en représentant l'ensemble des couples interdits (ou autorisés) ou alors en intention, en précisant la fonction caractéristique de la contrainte.

Définition 4 (Affectation partielle) *Une affectation (ou assignation/instanciation) partielle* A *d'un ensemble de variables* $vars(A) = \{X_1, X_2, \ldots, X_k\} \subseteq \mathscr{X}$ *affecte à chaque variable de* $vars(A)$ *une valeur de son domaine. Une affectation d'une de ces variable correspond à un k-uplet de* $\mathrm{dom}(X_1) \times \cdots \times \mathrm{dom}(X_k)$.

Lorsque toutes les variables de l'ensemble $\mathscr{X}$ sont affectées, il s'agit d'une affectation totale.

Définition 5 (Affectation partielle localement consistante) *Une affectation partielle d'un ensemble de variables* $X_A \in \mathscr{X}$ *est localement consistante si et seulement si elle satisfait toutes les contraintes portant uniquement sur des variables de* X_A.

Définition 6 (Solution d'un CSP) *Une solution* S *d'un CSP* $P = \langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$ *est une instanciation des variables de* $\mathscr{X}$ *qui satisfait la totalité des contraintes de* $\mathscr{C}$. *On dit alors que l'instanciation* S *satisfait* P.

Un CSP disposant d'au moins une solution est dit satisfiable ou cohérent. Si aucune solution n'existe, le CSP est alors insatisfiable, incohérent ou sur-contraint.

Définition 7 (Nogood) *Un nogood est une affectation partielle incohérente, c'est-à-dire ne pouvant apparaître dans aucune solution.*

Nous verrons par la suite, que les nogoods peuvent être à la fois utilisés dans des algorithmes de résolution de CSP centralisés et DisCSP.

De nombreux problèmes académiques sont résolus efficacement par des algorithmes de résolution de CSP. Nous pouvons citer les problèmes de coloration de graphes, de CSP aléatoires, le problème des n-dames, le problème d'ordonnancement, etc. Nous présentons, figure 1.4, un problème de coloration de graphe composé de trois variables $\mathscr{X} = \{X_0, X_1, X_2\}$ prenant leurs valeurs dans un domaine $\mathrm{dom}(X_i) = \{1, 2, 3\}$ et trois contraintes $X_0 \neq X_1$, $X_0 \neq X_2$, $X_1 \neq X_2$. Une solution possible de ce CSP est $\{X_0 = 1, X_1 = 2, X_2 = 3\}$.

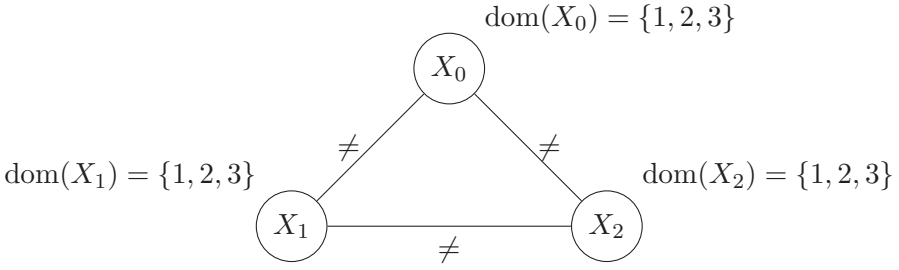

FIGURE 1.4 – Exemple de CSP centralisé.

1.3.2 Algorithmes de filtrage

Si toutes les contraintes peuvent être vérifiées en temps et espace polynomial, la recherche de solutions dans un CSP est un problème NP-Complet. Afin de simplifier au maximum le problème avant de le résoudre, des algorithmes de filtrage permettant de réduire la taille du problème et donc l'espace de recherche peuvent être utilisés. Ces algorithmes de filtrage exploitent la définition des contraintes dans le CSP afin de supprimer des valeurs des domaines qui ne pourront appartenir à aucune solution [Dechter, 2003].

Différents degrés de consistance peuvent être obtenus. L'*arc consistance* correspond au degré le plus généralement recherché. Afin d'introduire cette notion, nous devons au préalable définir la notion de support :

Définition 8 (Support) *Les supports $SpI(C, a)$ avec $a \in \text{dom}(X_i)$ et $X_i \in vars(C)$ d'une valeur a dans une contrainte d'arité quelconque est l'ensemble des instanciations (I) satisfaisant la contrainte C telles que $(X_i, a) \in I$.*

Dans le cas des contraintes binaires, on définit la notion de valeur support : les valeurs supports d'une valeur $a \in \text{dom}(X_i)$ dans une contrainte binaire C telle que $vars(C) = \{X_i, X_j\}$, sont l'ensemble $SpV(C, a)$ des valeurs $b \in \text{dom}(X_j)$ telles que $\{(X_i, a), (X_j, b)\} \in SpI(C, a)$.

Définition 9 (Arc consistance) *Soit un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$. Une valeur $a \in \text{dom}(X_i)$ est arc-consistante si et seulement si a dispose d'au moins un support dans toutes les contraintes impliquant la variable X_i.*

Un réseau P est arc-consistant si et seulement si toutes les valeurs de chaque variable sont arc-consistantes.

Un algorithme d'arc-consistance très utilisé est $AC3$ [Mackworth, 1977]. Celui-ci est présenté à l'algorithme 1. $AC3$ consiste à utiliser une file (Q) pour mémoriser les arcs à réviser. Cette dernière est mise à jour de manière à ce que les contraintes à vérifier soient celles qui étaient reliées à une variable dont le domaine a été modifié.

Algorithm 1: AC3($\mathscr{X}, \mathscr{D}, \mathscr{C}$)

```
1 Q ← 𝒞;
2 tant que Q ≠ ∅ faire
3     sélectionner et supprimer une contrainte C de Q;
4     si Revise(C) alors
5         Q ← Q ∪ {C' ≠ C | vars(C) ∩ vars(C') ≠ ∅};
```

Algorithm 2: Revise(C)

```
1 supprime ← faux;
2 pour chaque (X_i, a) | X_i ∈ vars(C), a ∈ dom(X_i) faire
3     si SpI(C, a) = ∅ alors
4         supprimer a de dom(X_i);
5         supprime ← vrai ;
6 retourner supprime
```

L'algorithme AC4 [Mohr et Henderson, 1986] modifie la propagation au niveau des valeurs de chaque variable. Afin de gérer cette propagation, la notion de support (Cf. déf 8) est utilisée. Un support est associé à chaque valeur de chaque variable. Lorsqu'un support d'une valeur devient vide, la valeur peut être supprimée du domaine, entraînant ainsi une mise à jour des autres variables.

L'algorithme AC6 [Bessière, 1994] a la particularité de n'utiliser qu'un seul support pour une valeur. Le prochain support est calculé lors de la suppression de ce support. Grâce à cela, les complexités spatiales et temporelles moyennes sont grandement améliorées.

D'autres algorithmes de filtrage existent. Parmi eux, nous pouvons citer AC5 [Hentenryck *et al.*, 1972] qui est un algorithme d'arc consistance générique et exploitant la sémantique des contraintes. L'algorithme AC7 [Bessière *et al.*, 1999] est, comme pour AC4 et AC6, basé sur le calcul et l'exploitation des supports. Cependant, la bidirectionnalité des contraintes est utilisée pour améliorer la création de support.

Nous avons vu dans cette section que de nombreux algorithmes de filtrage peuvent être utilisés afin de simplifier au maximum un CSP avant de le résoudre. Nous allons maintenant nous intéresser à l'étape suivante consistant à la résolution de ces CSP.

1.3.3 Algorithmes de résolution

Nous présentons dans cette section différentes méthodes permettant de résoudre des CSP. Nous commencerons par détailler le **B**ack**T**racking Chronologique (BT) qui est l'algorithme le plus simple de backtracking [Bitner et Reingold, 1975]. Il est considéré comme l'algorithme de base pour la résolution de CSP. Puis nous présenterons les algorithmes Forward-Checking et Backjumping.

Le backtracking chronologique

Dans BT, les variables X_i sont assignées chronologiquement de manière à ce que toutes les contraintes soient satisfaites. Supposons que les variables X_0 à X_i soient assignées. La prochaine variable devant être assignée sera la variable X_{i+1}. Lorsqu'il est impossible d'assigner une variable car toutes ses valeurs sont inconsistantes (une valeur inconsistante est une valeur conduisant à la violation d'au moins une contrainte) avec les variables précédemment assignées, la valeur de la dernière variable assignée devra être modifiée. L'algorithme BT se termine de deux façons différentes : soit toutes les variables sont assignées et le CSP est résolu, soit BT aura parcouru l'ensemble de l'arbre de recherche et le CSP est inconsistant. BT est présenté ci-dessous (algorithme 3). L'algorithme effectue des appels récursifs (ligne 7) et s'arrête lorsque toutes les variables sont instanciées (lignes 1 et 2).

Algorithm 3: BT(A, $\langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$)

```
1 si toutes les variables sont assignées dans A alors
2 |  retourner l'assignation courante A
3 sinon
4 |  soit X_i une variable non assignée de 𝒳 ;
5 |  pour chaque valeur possible v de dom(X_i) faire
6 |  |  X_i ← v;
7 |  |  si estConsistant(A ∪ {X_i ← v}) alors
8 |  |  |  BT(A ∪ {X_i ← v})
9 |  |  X_i ← null
```

BT peut être amélioré de différentes manières. Diverses heuristiques pour le choix des valeurs ou des variables à affecter peuvent être utilisées à l'initialisation de l'algorithme (Cf. section 1.3.4). L'algorithme présenté ici est statique. Cependant il existe un BT dynamique où l'ordre des variables à assigner évolue durant toute la résolution du CSP [Ginsberg, 1993].

Cet algorithme peut être amélioré en supprimant, dans le domaine des variables non assignées, les valeurs incompatibles avec les variables déjà assignées. Il s'agit d'une technique d'anticipation utilisée, par exemple, dans l'algorithme Forward-Checking.

Le Forward-Checking

Le Forward-Checking (FC) [Haralick et Elliot, 1980] est la technique la plus simple de prévision d'échecs pour résoudre des CSP. Les bases sont les mêmes que pour BT. Une étape de filtrage est ajoutée. L'algorithme FC s'obtient en modifiant la fonction *estConsistant* de l'algorithme 3. Le domaine de chaque variable non instanciée sera mis à jour de manière à ne conserver que les valeurs consistantes avec la solution partielle courante. Si le domaine d'une variable devient vide, cela signifie que la solution partielle courante ne peut être étendue en une solution complète. La fonction $FC\text{-}estConsistant(A, y)$ retourne alors $faux$. À chaque fois qu'une instanciation échoue, les domaines des variables non encore instanciées seront restaurés à leurs états précédents.

Algorithm 4: FC-estConsistant(A,X_i)

```
1 // X_i est une variable non assignée (∉ A);
2 pour chaque valeur v de dom(X_i) faire
3 |  si ¬ estConsistant(A ∪ {X_i ← v}) alors
4 |  |  supprimer v du domaine;
5 si dom(X_i) = ∅ alors
6 |  retourner faux;
7 retourner vrai;
```

Dans BT ou FC, à chaque retour arrière, l'algorithme passe de la variable X_i à la variable précédente X_{i-1} alors qu'il n'existe pas forcément de contraintes entre ces deux variables. Une amélioration de ces algorithmes consiste à effectuer un retour arrière dit « intelligent » vers la variable fautive : c'est le cas dans l'algorithme Backjumping.

Le Backjumping

Lorsque l'affectation d'une variable échoue, BT effectue un retour arrière vers la dernière variable instanciée. Cependant, il se peut que la dernière variable instanciée ne soit pas la cause de cet échec. En effet, lorsqu'une affectation d'une variable échoue, cela est dû à une contrainte C. Tant que les autres valeurs des variables appartenant à la contrainte C ne seront pas modifiées, le conflit ne sera pas résolu. Il faut donc que le retour arrière porte sur l'une des variables appartenant à cette contrainte C. Un algorithme de backjumping fera un saut vers la valeur de la dernière variable affectée présente dans la contrainte.

Dans l'algorithme Graph-based Backjumping [Dechter, 1990a], le retour arrière s'effectue vers les variables des contraintes reliées à la variable fautive sans vérifier si la contrainte était bel et bien violée. Dans l'algorithme du Conflict-directed Backjumping [Prosser, 1993, Chen et Beek, 2001], le retour arrière s'effectue vers une variable ayant engendrée une violation de contrainte. L'algorithme CDB décrit le Conflict-directed backjumping.

Algorithm 5: CDB(A)

```
1  si toutes les variables sont assignées dans A alors
2      retourner la solution A;
3  sinon
4      soit x une variable non assignée et S ← null;
5      pour chaque valeur possible v de x faire
6          x ← v;
7          cs ← CBJ-Evaluer(A ∪ {x ← v});
8          si cs est vide alors
9              cs ← CBJ(A ∪ {x ← v});
10         si x ∉ cs alors
11             x ← null;
12             retourner cs;
13         S ← S ∪ cs;
14     retourner S − {x};
```

Algorithm 6: CBJ-Evaluer(A)

```
1  si A est consistant alors
2      retourner null ;
3  sinon
4      soit C une contrainte violée par A;
5      retourner scope(c) ;
```

1.3.4 Heuristiques

Durant la résolution d'un CSP, l'ordre dans lequel les instanciations sont effectuées est très important. Des heuristiques d'ordonnancement peuvent être utilisées au niveau de l'ordre d'instanciation des variables et de l'ordre d'instanciation des valeurs.

L'ordre d'instanciation des variables peut avoir un réel impact sur l'efficacité de l'algorithme de recherche [Bessière *et al.*, 2002]. En effet, il est plus avantageux d'instancier en priorité les variables présentes dans la partie difficile du réseau de contraintes. Il existe des méthodes d'ordonnancement de variables statiques basées sur la structure du réseau de contraintes. Nous pouvons indiquer, par exemple, l'heuristique largeur-minimum (*min-width*) [Freuder, 1982] permettant d'obtenir un ordre minimisant la largeur de l'arbre de recherche à explorer. L'heuristique min-domain [Haralick et Elliot, 1980] consiste à choisir en priorité les variables dont le domaine est le plus petit.

Les heuristiques d'ordonnancement de valeurs ne sont utiles que si le but est de trouver une solution au CSP ou si les branchements sont binaires. En effet, elles n'ont aucun impact si le CSP est inconsistant ou si l'on recherche toutes les solutions puisque la totalité de l'arbre de recherche devra être parcouru. Il existe de nombreuses heuristiques d'ordonnancement des valeurs. Par exemple, nous pouvons citer l'heuristique *min-conflict* qui favorise les valeurs qui violent le moins de contraintes.

1.3.5 Extensions possibles

Il existe différentes extensions possibles aux CSP. Parmi elles, nous pouvons citer les CSP dynamiques dont la définition est la suivante :

Définition 10 *Un CSP dynamique (DCSP) est une suite de CSP statiques* P_0, ..., P_α, $P_{\alpha+1}$, ..., P_X *chacun résultant du précédent par l'ajout ou le retrait d'une contrainte (ces opérations sont respectivement appelées restriction et relaxation). Par conséquent si on note* $P_\alpha = (\mathscr{X}, \mathscr{D}, \mathscr{C}_\alpha)$*, on a* $P_{\alpha+1} = (\mathscr{X}, \mathscr{D}, \mathscr{C}_{\alpha+1})$ *avec* $\mathscr{C}_{\alpha+1} = \mathscr{C}_\alpha \pm C_{ij}$ *où* C_{ij} *est une contrainte. Au départ on a* $P_0 = (\mathscr{X}, \mathscr{D}, \{\})$.

Des algorithmes de filtrage sont apparus dans le cadre des CSP dynamiques tel que *DnAC*4 [Bessière, 1991]. Nous avons vu précédemment que l'algorithme *AC*4 est facilement adaptable aux restrictions (ajout de contraintes). Cependant *AC*4 ne peut pas remettre des valeurs supprimées précédemment lors d'une relaxation. Afin de retirer une contrainte, nous devons procéder à une restriction sur le problème initial. L'idée principale de l'algorithme *DnAC*4 est de sauvegarder les informations permettant de connaître les valeurs des domaines à remettre lors du retrait d'une contrainte.

Il existe de nombreuses extensions des DCSP. Citons, par exemple, les CCSP (Composite CSP) [Sabin et Freuder, 1996], les CSP* [Amilphastre, 1999], les CSPe (CSP à états) [Veron et Aldanondo, 2000] ou encore les ACSP (Activity CSP) [Geller et Veksler, 2005]. Une autre extension possible des CSP consiste à valuer les contraintes [Schiex *et al.*, 1997] :

Définition 11 (CSP valué) *Un CSP valué (VCSP) est un CSP classique* $P =< \mathscr{X}, \mathscr{D}, \mathscr{C} >$ *doté d'une structure de valuation* $S = (E, \preceq, \oplus)$ *et d'une application* φ *de* C *dans* E*, qui associe une valuation à chaque contrainte du CSP. On le note comme un 5-uplet* $< \mathscr{X}, \mathscr{D}, \mathscr{C}, S, \varphi >$.

Dans le cadre des VCSP, le but n'est plus de trouver une solution qui satisfasse toutes les contraintes comme pour les CSP mais une solution qui minimise la somme des valuations des contraintes violées.

Nous allons maintenant nous intéresser à une autre extension des CSP : les CSP distribués.

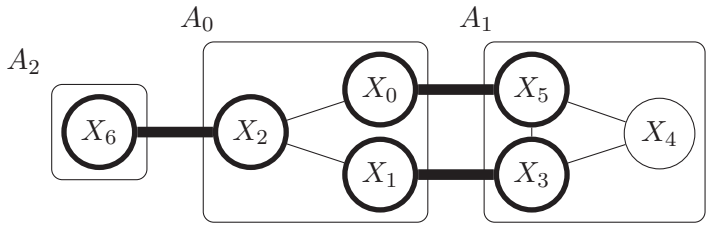

FIGURE 1.5 – Exemple de CSP Distribué comportant 3 agents.

1.4 Problèmes de Satisfaction de Contraintes Distribués

Nous présentons ici le formalisme des Problèmes de Satisfaction de Contraintes Distribuées (DisCSP). Il s'agit d'une extension des Problèmes de Satisfaction de Contraintes Centralisés (voir déf 3). Les DisCSP ont été formalisés pour la première fois par Yokoo et al. [Yokoo *et al.*, 1990]. Intuitivement, un DisCSP est un CSP où les variables sont distribuées entre plusieurs agents (voir déf 1).

1.4.1 Généralités

Un DisCSP peut être défini comme suit :

Définition 12 (Problème de Satisfaction de Contraintes Distribués - DisCSP) *Un Problème de Satisfaction de Contraintes Distribués (DisCSP) est un quadruplet* $< \mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{A} >$ *où :*

- $\mathscr{X} = \{X_1, X_2, \ldots, X_N\}$ *est un ensemble fini de* N *variables.*
- $\mathscr{D} = \{\mathrm{dom}(X_1), \mathrm{dom}(X_2), \ldots, \mathrm{dom}(X_N)\}$ *est l'ensemble des domaines associés à chaque variable.*
- $\mathscr{C} = \{C_1, C_2, \ldots, C_e\}$ *est un ensemble fini de* e *contraintes. Chaque contrainte* C_i *porte sur un sous-ensemble des variables du problème.* $\mathscr{C}$ *est composé de deux sous-ensembles disjoints : les contraintes inter-agents* $\mathscr{C}_{inter}$ *portant sur des variables appartenant à des agents différents et les contraintes intra-agent* $\mathscr{C}_{intra}$ *portant sur des variables appartenant au même agent.*
- $\mathscr{A} = \{A_1, A_2, \ldots, A_m\}$ *est un ensemble fini de* m *agents. Les agents encapsulent des sous-parties exclusives de* $\mathscr{X}$*. L'ensemble des variables encapsulées par un agent* A *est nommé* $\mathrm{var}(A)$*.* n *correspond au plus grand nombre de variables encapsulées par un agent (*$N \leq nm$*).*

Définition 13 (Variable d'interface) *Les variables apparaissant dans les contraintes inter-agents* $\mathscr{C}_{inter}$ *sont appelées* variables d'interface.

La figure 1.5 représente un DisCSP comportant 3 agents $\mathscr{A} = \{A_0, A_1, A_2\}$. Ce DisCSP contient 7 variables réparties entre ces 3 agents. L'agent A_0 dispose des variables X_0, X_1 et X_2, l'agent A_1 des variables X_3, X_4 et X_5 et l'agent A_2 de la variable X_6. Les arcs reliant les différentes variables représentent des contraintes. Les arcs fins représentent les contraintes intra-agents et les arcs épais représentent les contraintes inter-agents. Toutes les variables, exceptée la variable X_4, sont des variables d'interface.

Les agents d'un DisCSP sont munis d'un ordre de priorité noté $\succ$ dont la définition est la suivante :

Définition 14 ($\succ$) *Soient deux agents A_i et A_j. $A_i \succ A_j$ signifie que A_i possède une priorité supérieure à celle de A_j.*

Par extension, une variable X_i possède une priorité supérieure à X_j ssi $X_i \in A_i, X_j \in A_j$ et $A_i \succ A_j$. Cet ordre, $\succ$, est utilisé lors de la propagation pour répondre aux problèmes de bouclage qui peuvent survenir, par exemple, lorsqu'un changement pour un agent A_1 entraîne un changement pour un agent A_2 qui entraîne un changement pour A_1.

Définition 15 (accointance) *Les agents qui partagent une contrainte commune avec l'agent A_i sont appelés accointances de A_i. Les accointances de A_i ayant une priorité supérieure (au sens de $\succ$) à A_i sont appelées accointances supérieures ($\Gamma^-(A_i)$). Celles ayant une priorité inférieure sont appelées accointances inférieures ($\Gamma^+(A_i)$).*

Par exemple, pour le problème décrit à la figure 1.5, $\Gamma^-(A_1) = \{A_0\}$ et $\Gamma^+(A_1) = \{\}$ car il n'existe aucun agent de priorité inférieure à celle de A_1 relié par une contrainte inter-agents à A_1.

La plupart des algorithmes de résolution de DisCSP ne s'intéressent qu'aux DisCSP binaires qui sont des DisCSP où les contraintes portent au plus sur deux variables.

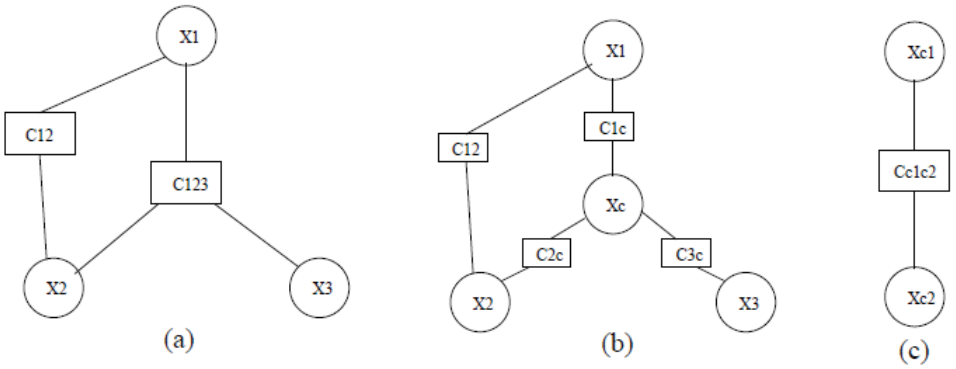

FIGURE 1.6 – Transformation des contraintes n-aires en contraintes binaires.

Afin de résoudre un DisCSP disposant de contraintes inter-agents n-aires, deux choix sont possibles. Soit le DisCSP est résolu directement, soit il est transformé en DisCSP binaire [Dechter, 1990b, Dechter et Pearl, 1989]. Il existe deux principales techniques permettant cette transformation :

- la première transformation réutilise les variables initiales du problème. Chaque contrainte k-aire (avec $k \geq 2$), $C_{X_1,X_2,...,X_k}$, peut être transformée en k contraintes binaires. Cette transformation implique l'ajout d'une nouvelle variable, X_c, qui sera reliée à chaque variable apparaissant initialement dans $C_{X_1,X_2,...,X_k}$ par une contrainte binaire. $\mathrm{dom}(X_c)$ contiendra le produit cartésien des domaines des variables impliquées dans $C_{X_1,X_2,...,X_k}$. Il s'agit d'une méthode peu pratique car si une contrainte relie 5 variables X_i (avec $|\,\mathrm{dom}(X_i)| = 10$, alors $|\,\mathrm{dom}(X_c)|$ contiendrait 100 000 valeurs. Une contrainte unaire restreindra le domaine de la variable X_c qui est appelée *variable encapsulée*. Grâce à cette technique, la solution du problème initial peut être directement obtenue à partir du problème transformé.
- la seconde transformation ne réutilise pas les variables initiales. Chaque contrainte k-aire (avec $k \geq 2$), $C_{X_1,X_2,...,X_k}$, peut être substituée par une nouvelle variable encapsulée dont le domaine contiendra le produit cartésien des domaines des variables impliquées dans

$C_{X_1,X_2,...,X_k}$. Ce domaine est réduit par la contrainte initiale. Les différentes variables encapsulées sont reliées entre-elles par des contraintes binaires d'égalité. Le problème obtenu dispose d'un faible nombre de variables. Cependant, la solution globale du problème doit être extraite de l'ensemble des variables encapsulées.

La figure 1.6 représente en (a) le problème initial, ainsi que sa transformation en problème binaire avec réutilisation des variables initiales (b) et sans utilisation des variables initiales (c).

1.4.2 Algorithmes de filtrage Distribué

Nous avons vu, dans la section 1.3.2 que l'utilisation d'algorithmes de filtrage avant la résolution d'un CSP centralisé permettait de réduire considérablement la complexité du problème. Il en est de même avec les DisCSP.

Algorithm 7: disAC4

```
1  List ← null;
2  Supports_<_,_> ← null;
3  InitComWorker();
4  ToSendList ← null;
5  pour chaque (i, j) ∈ arc(G)|i ∈ MyNodes(x) faire
6      pour chaque v ∈ dom(i) faire
7          counter[< i, j >][j] ← 0;
8          si C_ij(v, w) alors
9              counter[< i, j >][j] + +;
10             Support_<j,w> ← Support_<j,w> ∪ {< i, v >};
11         si counter[< i, v >][j] = 0 alors
12             List ← List ∪ {< i, v >}; dom(i) = dom(i) − {v};
13             ToSendList ← ToSendList ∪ {< i, v >};
14 tant que ¬done faire
15     pour chaque < j, w >∈ List faire
16         List ← List − {< j, w >};
17         pour chaque < i, v >∈ Support_<j,w>|v ∈ dom(i) faire
18             counter[< i, v >][j] − −;
19             si counter[< i, v >][j] = 0 alors
20                 List ← List ∪ {< i, v >}; dom(i) ← dom(i) − {v};
21                 ToSendList ← ToSendList ∪ {< i, v >};
22     SendMessage(ToSendList);
23     ReceiveMessage(List);
```

L'algorithme $DisAC4$ [Nguyen et Deville, 1995], présenté en algorithme 7, reprend le principe de l'algorithme $AC4$ mais, cette fois-ci, dans un cadre distribué. Cet algorithme utilise k processus ($k \geq 1$) appelés *workers* se répartissant équitablement les N variables du problème. Comme pour $AC4$, chaque *worker* effectue deux tâches. La première consiste à mettre à jour de manière locale et indépendante des compteurs et des structures de supports [Nguyen et Deville, 1995]. Durant la seconde étape, les labels inconsistants sont traités localement. Puis chaque *worker* doit transmettre ses labels inconsistants qui ont été détectés localement aux autres *workers* tout en

recevant les labels inconsistant des autres *workers*.

L'algorithme *DisAC*9 [Hamadi, 1999a] est une approche optimale en nombre de messages échangés. À l'initialisation, chaque agent calcule les labels inconsistants. Ensuite, les labels pouvant induire d'autres suppressions de labels sont transmis vers les autres agents. Après cette étape d'initialisation, les agents mettent à jour les supports de labels suite à la réception de messages. Lorsque des nouveaux labels sont supprimés, de nouveaux messages sont envoyés.

Nous avons présenté dans cette section différents algorithmes de filtrage distribué permettant de simplifier des DisCSP avant leurs résolutions que nous présenterons au chapitre suivant. Nous allons maintenant nous intéresser à des heuristiques pouvant être utilisées lors de la résolution de ces problèmes afin de réduire considérablement les temps de résolution.

1.4.3 Heuristiques

Afin d'optimiser la résolution d'un DisCSP, différentes heuristiques peuvent être utilisées. Les heuristiques vues par les CSP centralisés portant sur le choix des valeurs et des variables (voir section 1.3.4) peuvent être utilisées dans le cadre des DisCSP. En plus de ces heuristiques, nous pouvons utiliser des heuristiques d'ordonnancement des agents.

Différentes heuristiques d'ordonnancement des agents sont présentées dans [Armstrong et Durfee, 1997]. Lorsqu'une unique variable est affectée par agent, les heuristiques permettant d'ordonner les variables dans un CSP centralisé peuvent être reprises pour ordonner les agents.

L'algorithme *DisAO* [Hamadi *et al.*, 1998] pour Distributed Agent Ordering, permet d'obtenir un ordonnancement efficace des agents. Cet algorithme s'applique lorsque chaque agent ne possède qu'une seule variable. Selon les auteurs, la généralisation à plusieurs variables par agent est directe mais non détaillée. *DisAO* permet donc de créer les accointances supérieures (Γ^-) et inférieures (Γ^+) de chaque agent et prend en entrée :

- f : une fonction heuristique d'évaluation d'un agent du système (qui attribue, par exemple, un score élevé aux agents les plus contraints),
- Γ : l'ensemble des accointances de l'agent ($\Gamma = \Gamma^+ \cup \Gamma^-$),
- *op* : un opérateur de comparaison entre deux agents du système.

DisAO est présenté en algorithme 8. À l'initialisation de *DisAO*, chaque agent construit les ensembles Γ^- et Γ^+ (lignes 3 à 7) grâce à l'opérateur *op* et à la fonction heuristique f.

Puis, chaque agent calcule sa priorité (son niveau) dans la hiérarchie. La priorité de l'agent correspond à la valeur de max (ligne 9). La priorité d'un agent sera celle du niveau maximal reçu par les agents de Γ^+ auquel est ajouté 1.

Les priorités des agents sont attribuées du bas vers le haut en commençant par les agents de plus petites priorités (i.e, disposant d'une priorité valant 1).

Algorithm 8: disAO(in : f, Γ, op ; out : Γ^+, Γ^-)

```
1  Γ+ ← null ;
2  Γ− ← null ;
3  pour chaque Agent_j ∈ Γ faire
4      si f(Agent_j) op f(self) alors
5          Γ+ ← Γ+ ∪ {Agent_j}
6      sinon
7          Γ− ← Γ− ∪ {Agent_j}
8  max ← 0;
9  pour chaque i = 0; i < |Γ+|; i++ faire
10     m ← getMsg();
11     si m = value : v alors
12         si max < v alors
13             max ← v
14 max++;
15 communiquer (Γ, value : max);
16 communiquer(Γ+, position : (max, self));
17 pour chaque i = 0; i < |Γ|; i++ faire
18     m ← getMsg();
19     si m = position : (p, j) alors
20         level_j ← p;
21 tri de Γ en fonction de level;
```

Lorsque le CSP local à chaque agent contient plus d'une variable, les agents peuvent, par exemple, être ordonnés selon le nombre moyen de valeurs pouvant être prises par les variables d'un agent (utilisation de l'heuristique *min-domain*). Cet algorithme donne la priorité aux agents disposant de peu de choix pour ses variables. Les agents peuvent aussi être ordonnés selon le nombre de solutions locales. Cependant, ceci nécessite que toutes les solutions locales soient calculées avant de commencer la recherche globale.

Il existe aussi des heuristiques d'ordonnancement dynamique des agents comme dans l'algorithme AWC [Yokoo, 2001]. Dans AWC, chaque agent A_i doit respecter les contraintes inter-agents reliant A_i aux agents de priorité supérieure à A_i. Cependant, lorsqu'A_i ne peut vérifier ses contraintes, il crée un nogood et sa priorité devient supérieure à toutes les priorités de ses accointances. Ainsi, comme A_i devient l'agent de plus haute priorité, quelle que soit la valeur qu'il a choisie, A_i ne pourra pas violer de contraintes inter-agents, le reliant à des agents de priorité supérieure à la sienne. A_i choisi alors une valeur minimisant le nombre de contraintes violées avec les agents ayant une priorité inférieure.

1.5 Conclusion

Dans ce chapitre, nous avons présenté de manière générale les systèmes multi-agents, les problèmes de satisfaction de contraintes et un domaine reliant à la fois les CSP et les SMA : les CSP distribués.

Pour commencer, nous avons énoncé quelques généralités sur les SMA. Nous avons décrit

les différents points caractérisant un SMA, à savoir les agents, l'environnement, les interactions et l'organisation. Puis nous avons vu que le raisonnement intra-agent et inter-agents pouvait être basé sur un ensemble de relations entre différentes variables. La résolution du problème peut s'effectuer grâce à des agents interagissant afin d'obtenir une solution globale à partir des solutions locales des CSP affectés à chaque agent.

La seconde partie a été consacrée aux CSP. Après avoir donné différentes définitions, nous nous sommes intéressés à des algorithmes de filtrage. Ces derniers permettent de transformer un CSP en un CSP équivalent disposant des mêmes solutions mais dont les domaines des variables ont été réduits, permettant ainsi de réduire la taille de l'arbre de recherche. Puis nous avons présenté des algorithmes de recherche de solutions pour CSP centralisés tels que BT, BJ et CDB. Nous avons vu brièvement différentes heuristiques permettant d'ordonnancer efficacement les valeurs et les variables et nous avons présenté différentes extensions possibles aux CSP dont les DisCSP.

La dernière partie s'est focalisée sur les DisCSP. Nous avons présenté ce formalisme puis nous avons vu les différents algorithmes de filtrage distribué tels que $DisAC4$ et $DisAC9$. Enfin, nous avons présenté des heuristiques d'ordonnancement d'agent.

Dans le chapitre suivant, nous présenterons des méthodes permettant la résolution de DisCSP. Nous verrons les différents algorithmes permettant une recherche asynchrone que ce soit pour des DisCSP contenant une unique variable par agent ou contenant un CSP local composé de différentes variables et contraintes intra-agents.

2

Résolution des CSP Distribués

Sommaire

2.1 Introduction

Dans le chapitre précédent, nous avons introduit de manière générale les notions de systèmes multi-agents et de problèmes de satisfaction de contraintes. Puis, nous avons fait le lien entre ces deux domaines de l'intelligence artificielle. Nous avons ainsi vu que les raisonnements intra-agent et inter-agents pouvaient être formalisés par un ensemble de relations entre différentes variables. On ramène ainsi le raisonnement distribué à un problème de satisfaction de contraintes distribué. Ce formalisme permet d'appréhender simplement mais de manière efficace des problèmes naturellement distribués. Chaque sous-problème, représenté par un ensemble de variables et de contraintes intra-agent, est affecté à un agent, puis les différents agents collaborent afin de respecter les contraintes inter-agents.

La seconde section de ce chapitre sera consacrée à la présentation des DisCSP. Nous verrons, pour commencer, des définitions formelles portant sur les DisCSP. Nous exposerons différents problèmes pouvant être formalisés grâce aux DisCSP. Puis, nous détaillerons les difficultés rencontrées pour résoudre des DisCSP grâce à l'utilisation d'algorithmes de recherche asynchrones.

Nous verrons que la principale difficulté consiste à gérer l'asynchronisme entre les agents de manière à obtenir un comportement inter-agents cohérent. Nous verrons alors l'importance qu'il y a à affecter un contexte à chaque message échangé. Nous discuterons de la gestion de la détection de la terminaison des algorithmes. Enfin, nous verrons différentes représentations distribuées possibles d'un DisCSP.

La troisième section portera sur des algorithmes de résolution de DisCSP où le problème local d'un agent est réduit à une seule variable. Nous détaillerons différents algorithmes de recherche asynchrones tels que *Asynchronous Backtracking* (ABT) et *Asynchronous Weak-Commitment* (AWC). Les hypothèses et notations utilisées par ces algorithmes seront précisées. Enfin, nous détaillerons ces algorithmes avant d'observer leurs comportements sur des exemples.

La quatrième section s'intéressera à la résolution de DisCSP où le problème local de chaque agent est dit « complexe », c'est-à-dire que chaque problème local sera constitué d'un ensemble de variables et de contraintes intra-agent. Nous verrons différentes méthodes permettant de généraliser un algorithme de résolution de DisCSP mono-variable par agent. Puis nous détaillerons différents algorithmes dont *Multi-ABT* et *Multi-AWC*. Enfin, nous étudierons les limites des algorithmes existants avant de conclure.

2.2 Généralités sur les DisCSP

Le problème de satisfaction de contraintes distribué (DisCSP) est un domaine assez jeune de l'Intelligence Artificielle. Ce domaine est apparu vers le début des années 1990 afin de modéliser efficacement de nombreux problèmes de nature distribuée dans lesquels les données sont physiquement réparties. Ces problèmes ne peuvent pas être résolus de manière classique et centralisée pour des raisons diverses :

- Le temps de rapatriement des données, sur un site unique, peut être trop long.
- L'espace mémoire nécessaire pour stocker l'ensemble du problème peut être trop important pour un site unique.
- Certains sites peuvent limiter les échanges de leurs données afin de préserver leur confidentialité.

L'approche distribuée consiste à répartir le problème sur un ensemble d'agents. Ceux-ci interagissent afin de faire émerger une solution globale à partir des solutions locales de chaque agent. Les DisCSP peuvent être utilisés pour résoudre de nombreux problèmes naturellement distribués tels que les problèmes d'emploi du temps [Tsuruta et Shintani, 2000] ou de trafic routier [Doniec *et al.*, 2008]. Les travaux précurseurs de Yokoo et al. [Yokoo *et al.*, 1990, Yokoo *et al.*, 1992, Yokoo, 1995] ont permis de créer des algorithmes permettant la recherche de solutions dans les DisCSP. Nous allons commencer, dans la section suivante, par énoncer des définitions sur les DisCSP.

2.2.1 Définitions

Définition 16 (DisCSP) *Un CSP Distribué est un tuple* $(\mathscr{X}, \mathscr{D}, \mathscr{C}, \mathscr{A})$ *où :*

- $\mathscr{X} = \{X_1, X_2, \ldots, X_N\}$ *est un ensemble fini de* N *variables,*
- $\mathscr{D} = \{\mathrm{dom}(X_1), \mathrm{dom}(X_2), \ldots, \mathrm{dom}(X_N)\}$ *est l'ensemble des domaines finis* $\mathrm{dom}(X_i)$ *de taille maximale* d *que les variables peuvent prendre,*
- $\mathscr{C} = \{C_1, C_2, \ldots, C_e\}$ *est un ensemble fini de* e *contraintes,*
- $\mathscr{A} = \{A_1, A_2, \ldots, A_m\}$ *est un ensemble fini de* m *agents.*

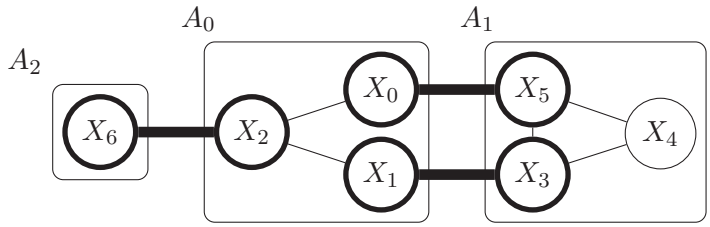

FIGURE 2.1 – Exemple de CSP Distribué.

Chaque contrainte C_i peut porter sur plusieurs variables ($\text{var}(C_i) \subseteq \mathscr{X}$). Les agents encapsulent des sous-ensembles exclusifs de $\mathscr{X}$. L'ensemble des variables encapsulées par un agent A est nommé $\text{var}(A)$. n correspond au plus grand nombre de variables encapsulées par un agent (on a donc $N \leq nm$). $\mathscr{C}$ est composé de deux sous-ensembles disjoints : les contraintes inter-agents $\mathscr{C}_{inter}$ portant sur des variables appartenant à des agents différents et les contraintes intra-agent $\mathscr{C}_{intra}$ portant sur des variables qui n'appartiennent qu'à un seul agent.

Définition 17 (Solution d'un DisCSP) *Une solution d'un DisCSP est une instanciation de toutes les variables de l'ensemble des agents satisfaisant, à la fois, les contraintes intra-agent et inter-agents.*

Parmi l'ensemble des variables affectées aux agents, certaines jouent un rôle particulier :

Définition 18 (Variable d'interface) *Les variables apparaissant dans les contraintes inter-agents $\mathscr{C}_{inter}$ sont appelées variables d'interface.*

Les variables d'interface sont les variables qui vont permettre de faire le lien entre les CSP affectés aux différents agents. En effet, afin d'obtenir une solution globale consistante, les agents s'échangeront des messages portant sur les valeurs de leurs variables d'interface afin de vérifier les contraintes inter-agents.

La figure 2.1 représente un DisCSP comportant 3 agents $\mathscr{A} = \{A_0, A_1, A_2\}$. Ce DisCSP contient 7 variables réparties entre ces 3 agents. L'agent A_0 comporte les variables X_0, X_1 et X_2, l'agent A_1 les variables X_3, X_4 et X_5 et l'agent A_2 la variable X_6. Les arcs reliant les différentes variables représentent des contraintes. Les arcs fins représentent les contraintes intra-agent et les arcs épais représentent les contraintes inter-agents. Toutes les variables, exceptée la variable X_4, sont des variables d'interface.

Pour simplifier la présentation, nous ne nous intéresserons qu'à des DisCSP binaires. Ceci permet de représenter simplement chaque problème sous la forme d'un graphe où les nœuds sont les variables et les arcs sont des contraintes. Un DisCSP disposant de contraintes inter-agents n-aires peut être modifié en un DisCSP équivalent avec des contraintes inter-agents binaires même si cette transformation est très coûteuse en temps et en espace (problème NP-difficile) [Dechter et Pearl, 1989, Dechter, 1990b].

À notre connaissance, dans tous les algorithmes de résolution de DisCSP, les agents sont munis d'un ordre de priorité que nous notons $\succ$. Cet ordre est utilisé lors de la propagation pour pallier aux problèmes de bouclage qui surviennent lorsque les chaînes de changement de comportement des agents comportent des cycles. Dans la plupart des algorithmes de DisCSP, cet ordre est statique. Il peut être fixé de manière lexicographique ou selon des heuristiques telles que *DisAO* [Hamadi *et al.*, 1998] ou celles présentées dans [Armstrong et Durfee, 1997]. D'autres algorithmes, notamment *AWC* [Yokoo, 2001], utilisent un ordonnancement dynamique entre les agents. Nous utilisons par la suite la notion d'accointance définie ci-dessous :

Définition 19 (Accointance) *Les agents qui partagent une contrainte commune avec un agent A_i sont appelés accointances de A_i. Les accointances de A_i ayant une priorité supérieure (au sens de $\succ$) à A_i sont appelées accointances supérieures ($\Gamma^-(A_i)$). Celles ayant une priorité inférieure sont appelées accointances inférieures ($\Gamma^+(A_i)$).*

Par exemple, pour le problème décrit à la figure 2.1, où les priorités des agents sont attribuées de manière lexicographique, $\Gamma^-(A_1) = \{A_0\}$ et $\Gamma^+(A_1) = \{\}$ car il n'existe aucun agent de priorité inférieure à celle de A_1 relié par une contrainte inter-agents à A_1.

Dans la majorité des algorithmes de résolution de DisCSP, les agents échangent des messages $\langle OK?\rangle$ et $\langle BT\rangle$. Comme nous pouvons l'observer à la figure 2.2, les messages $\langle OK?\rangle$ (contenant les valeurs des variables instanciées à communiquer) sont transmis d'un agent de priorité supérieure vers un agent de priorité inférieure. Les messages de backtrack $\langle BT\rangle$ (permettant d'effectuer un retour-arrière) sont transmis d'un agent de priorité inférieure vers un agent de priorité supérieure. Nous allons maintenant exposer différents problèmes concrets distribués qui peuvent être formalisés puis résolus grâce aux DisCSP.

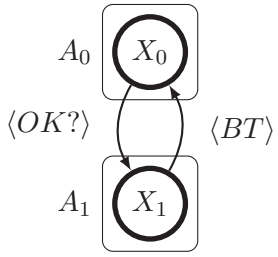

FIGURE 2.2 – Envoi de messages entre les agents résolvant un DisCSP.

2.2.2 Exemple d'application des DisCSP

Le SensorDisCSP

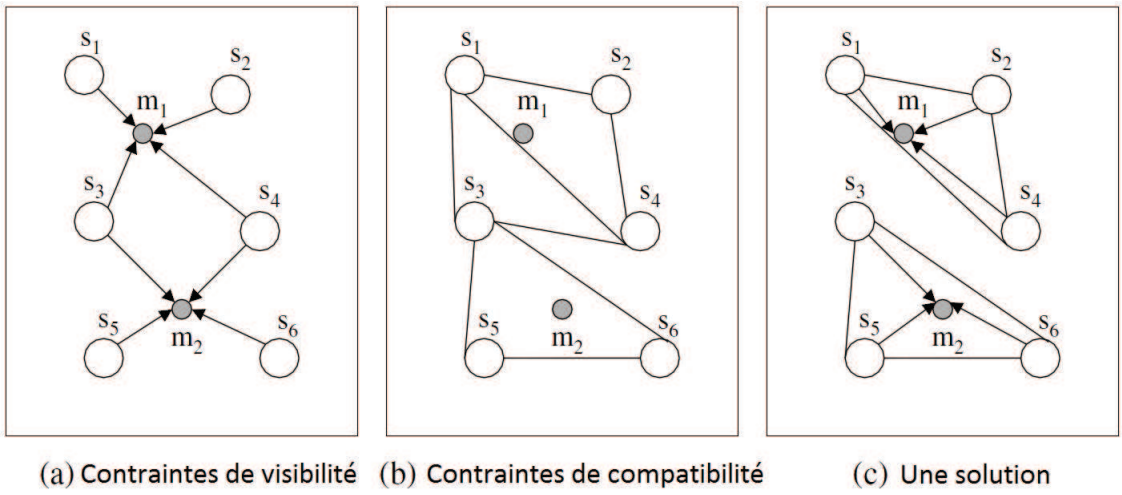

FIGURE 2.3 – Un problème de type SensorDisCSP.

Les DisCSP permettent de modéliser de manière simple et efficace de nombreux problèmes naturellement distribués. Parmi eux, nous pouvons citer le SensorDisCSP (pour *Distributed Sensor-*

Mobile problem [Fernandez *et al.*, 2002]). Il s'agit d'un problème contenant un ensemble de détecteurs et un ensemble de mobiles. Les détecteurs doivent suivre les mobiles. Le but de ce problème est de faire en sorte que chaque mobile soit en permanence suivi par un ensemble de détecteurs (en général trois pour ce problème). Chaque détecteur ne peut suivre qu'un seul mobile. Afin de résoudre ce SensorDisCSP, une solution globale doit être trouvée où chaque mobile est suivi par trois détecteurs distincts. Les contraintes de visibilité et les contraintes de compatibilité doivent être satisfaites.

Ce problème peut être formalisé sous la forme d'un DisCSP de la manière suivante. Chaque agent représente un mobile et dispose de trois variables représentant les trois détecteurs devant suivre le mobile. Le domaine des variables contient un ensemble de détecteurs compatibles. Les contraintes intra-agent permettent de vérifier que les détecteurs assignés au mobile sont bien différents et compatibles. Enfin, les contraintes inter-agents permettent de s'assurer que chaque détecteur n'est affecté qu'à un mobile au maximum.

La figure 2.3 illustre un SensorDisCSP composé de six détecteurs $\{s_1, \ldots, s_6\}$ et deux mobiles m_1 et m_2. La figure 2.3-a permet de connaître quels sont les détecteurs qui peuvent observer les mobiles : le mobile m_1 peut être vu par les détecteurs s_1, s_2, s_3 et s_4 et le mobile m_2 peut être vu par s_3, s_4, s_5 et s_6. Il s'agit des contraintes de visibilité. La figure 2.3-b présente les contraintes de compatibilité entre les différents détecteurs. Par exemple, comme il n'y a pas de contraintes entre s_2 et s_3, cela signifie que ces deux détecteurs ne peuvent pas suivre le même mobile.

Une solution au problème est présentée sur la figure 2.3-c. Le but du problème est d'affecter trois détecteurs à chaque mobile. Nous pouvons observer que les détecteurs s_1, s_2 et s_4 suivent le mobile m_1 et que les détecteurs s_3, s_5 et s_6 suivent le mobile m_2.

Le problème d'exploration multi-robots

L'exploration d'un environnement inconnu tout en vérifiant des contraintes de communication est considérée comme un problème difficile [Rooker et Birk, 2007], [Vazquez et Malcolm, 2004]. Dans le problème d'exploration multi-robots [Doniec *et al.*, 2009], les robots disposent de capacités de communication (wifi par exemple) et de capacités de détection (par exemple camera ou laser). Deux robots peuvent communiquer directement s'ils se situent à l'intersection de leurs portées de communication. Chaque robot ne peut observer que l'environnement qui se situe à sa portée de détection.

L'objectif des robots est d'explorer complètement l'environnement dans lequel ils évoluent. Au cours de cette exploration, les robots doivent respecter deux contraintes :

- ne pas perdre le contact avec les autres (ce qui arrive si les robots sont trop éloignés),
- ne pas entrer en collision avec les autres (ce qui arrive si les robots sont trop proches).

Ce problème peut être formalisé sous la forme d'un DisCSP de la manière suivante. Chaque agent représente un robot constitué d'une unique variable. Le domaine de chaque variable est composé des huit directions que peut suivre le robot lors du pas suivant $\{N, NE, E, SE, S, SO, O, NO\}$. Les contraintes inter-agents représentent des contraintes de distance entre les robots :

- la future position d'un robot ne doit pas « casser » la connectivité du réseau.
- la future position d'un robot ne doit pas créer de chevauchements avec les capteurs des autres robots.

Cette application sera reprise dans le chapitre 4 afin d'évaluer différents algorithmes de résolution de DisCSP où le problème local d'un agent se résume à une unique variable. Il existe de nombreux problèmes, non présentés ici, pouvant être résolus par des DisCSP tels que le problème d'emploi du temps scolaire [Burke *et al.*, 2007] ou encore le problème d'ordonnancement dans les systèmes de fabrication [Sousa et Ramos, 1999].

Après avoir présenté deux applications pouvant être modélisées grâce aux DisCSP, nous allons maintenant aborder les difficultés rencontrées par les algorithmes de résolution de DisCSP en commençant par la gestion de l'asynchronisme.

2.2.3 Gestion de l'asynchronisme

Un algorithme de résolution de DisCSP asynchrone est un algorithme où les agents agissent de manière concurrente. C'est-à-dire qu'un agent A_i ne doit pas attendre qu'un agent A_{i-1} ait fini d'effectuer ses propres calculs avant de pouvoir effectuer les siens. La résolution d'un DisCSP de manière asynchrone est un problème difficile. La difficulté provient du fait que les agents travaillent simultanément et émettent des messages de manière asynchrone. Lorsqu'il reçoit un message, un agent doit d'abord s'assurer que ce message n'est pas obsolète. En effet, durant la résolution d'un DisCSP, un agent peut communiquer différentes solutions locales et recevoir, par la suite, plusieurs messages lui demandant de modifier la valeur d'une de ses variables. Or, il est impossible de savoir si tous ces messages doivent être traités sachant qu'ils peuvent porter soit sur la solution courante de l'agent soit sur des solutions antérieures. Une solution consiste à attribuer un contexte à chaque demande émise permettant de savoir si ce message est toujours d'actualité.

Afin d'établir un contexte aux différents messages échangés, la plupart des algorithmes de résolution de DisCSP utilisent des nogoods que les agents joignent aux demandes de backtrack. Ainsi, au lieu de ne joindre que la valeur de la variable devant être modifiée à un message de backtrack, l'agent émetteur du message fournit le contexte ayant abouti à cette demande de backtrack. Chaque agent dispose d'un ensemble appelé $agentView$ contenant les valeurs des variables reçues des agents de priorité supérieure. Lorsqu'un agent ne peut trouver de solution consistante vis-à-vis de cette $agentView$, il joint à la demande de backtrack un sous-ensemble de l'$agentView$ ayant abouti à cette inconsistance. Ce sous-ensemble est un nogood qui servira de contexte. Nous allons maintenant nous intéresser à un autre point délicat de la résolution des DisCSP : comment les agents s'aperçoivent-ils qu'ils ont effectivement trouvé une solution globale à partir des résolutions locales ? C'est le problème de la terminaison.

2.2.4 Gestion de la terminaison

Comme nous l'avons vu dans les sections précédentes, les agents agissent de manière asynchrone afin de résoudre un DisCSP. Cela induit des difficultés dans la coordination inter-agents. À cause de cet asynchronisme, la détection de la fin de la résolution dans un DisCSP est difficile. Cette détection de la fin est difficile uniquement lorsqu'une solution est trouvée car lorsqu'aucune solution n'existe au DisCSP, un agent le détectera et stoppera l'algorithme.

Lors de la résolution d'un problème de manière asynchrone, aucun agent ne peut s'apercevoir que la résolution est terminée (avec une solution trouvée) ou si la recherche globale est toujours en cours. En effet, du point de vue d'un agent, si celui-ci ne reçoit plus aucun message et ne travaille plus depuis un certain temps, il ne connaît pas l'état des autres agents résolvant le DisCSP. La difficulté pour terminer l'algorithme de résolution de DisCSP est d'être certain qu'il n'y ait plus de messages échangés, que tous les agents soient en attente de messages et qu'ils ne travaillent pas. Différentes méthodes de détection de la terminaison existent :

– Un agent coordinateur peut observer les communications entre les agents. Ainsi lorsqu'il détecte qu'il n'y a plus de messages échangés sur une certaine période de temps, il peut annoncer la fin de la résolution. Cependant, rien ne peut indiquer que tous les agents soient bel et bien en attente de messages. En effet, un agent peut être en train d'effectuer des

calculs aboutissant à l'émission d'un message après cette période de temps.
– Un agent coordinateur peut envoyer toutes les t secondes un message à tous les agents exécutant l'algorithme afin de connaître leurs états. Puis lorsqu'un agent reçoit ce message, il indique à cet agent coordinateur par un message s'il est en attente de messages depuis x secondes ou bien s'il est en train de traiter des messages. Ainsi, lorsque l'agent coordinateur aura reçu de chaque agent un message indiquant qu'ils sont tous en attente de messages depuis x secondes, l'agent coordinateur pourra affirmer qu'une solution a été trouvée.

Nous allons maintenant nous intéresser à la distribution du DisCSP entre les différents agents.

2.2.5 Représentation distribuée du problème

Cette section présente des méthodes permettant la distribution d'un réseau de contraintes entre différents agents. Pour chacune de ces méthodes, des algorithmes de résolution de DisCSP ont été proposés.

Nous présentons trois méthodes illustrées dans la figure 2.4 qui représentent la distribution explicite (à gauche), la distribution canonique (au centre) et la distribution canonique sans ajout d'agent (à droite) d'un même problème. Nous expliquons ces trois méthodes de distribution grâce à un exemple de DisCSP contenant deux agents où le premier agent A_0 dispose des variables X_0, X_1 et X_2 et le second agent A_1 dispose des variables X_3, X_4 et X_5. Les variables grisées sur la figure sont des variables additionnelles nécessaires à la distribution. Les contraintes inter-agents initiales sont représentées par des traits plus épais que les autres contraintes.

La distribution explicite (à gauche de la figure 2.4) consiste à répartir simplement les variables et les contraintes intra-agent entre les différents agents. Ces agents seront reliés entre eux par les contraintes inter-agents. Il s'agit de la méthode la plus utilisée et la majorité des algorithmes de résolution de DisCSP tels que DDB [Bessière *et al.*, 2001], DIBT [Hamadi *et al.*, 1998], DB [Yokoo et Hirayama, 1996] suppose que cette distribution du problème a été utilisée entre les agents. Nous avons donc six contraintes intra-agent reliant X_0 à X_1, X_0 à X_2, X_1 à X_2, X_3 à X_4, X_3 à X_5, X_4 à X_5 et deux contraintes inter-agents reliant X_1 à X_3 et X_2 à X_4.

La distribution canonique (au centre de la figure 2.4) consiste à ajouter un « agent superviseur » au système. Cet agent possède une copie de toutes les variables d'interface de chaque agent ainsi que de l'ensemble des contraintes inter-agents. Ainsi, la résolution du CSP de l'agent superviseur permet de respecter toutes les contraintes inter-agents. Chaque agent est relié à l'agent superviseur grâce à une contrainte d'égalité reliant leurs variables d'interface à leurs copies présentes dans l'agent superviseur. Il y a de nombreuses variables qui ont été créées. La résolution du problème peut être plus rapide car elle est en grande partie centralisée mais l'aspect naturellement distribué du problème est perdu.

La distribution canonique sans ajout d'agent (à droite de la figure 2.4) permet de déplacer les contraintes inter-agents initiales à l'intérieur des différents agents. Pour cela, les variables d'interface sont dupliquées pour apparaître dans les agents situés à l'autre extrémité des contraintes inter-agents. De nouvelles contraintes inter-agents égalitaires sont créées permettant de lier les variables initiales à leurs « clônes ». Cette distribution peut être utilisée avant l'utilisation d'algorithmes tels que [Prosser et al., 1992] ou [Lemaître et Verfaillie, 1997].

La majorité des algorithmes de résolution de DisCSP utilisent la distribution explicite. Dans la section suivante, nous présentons ces algorithmes de résolution de DisCSP.

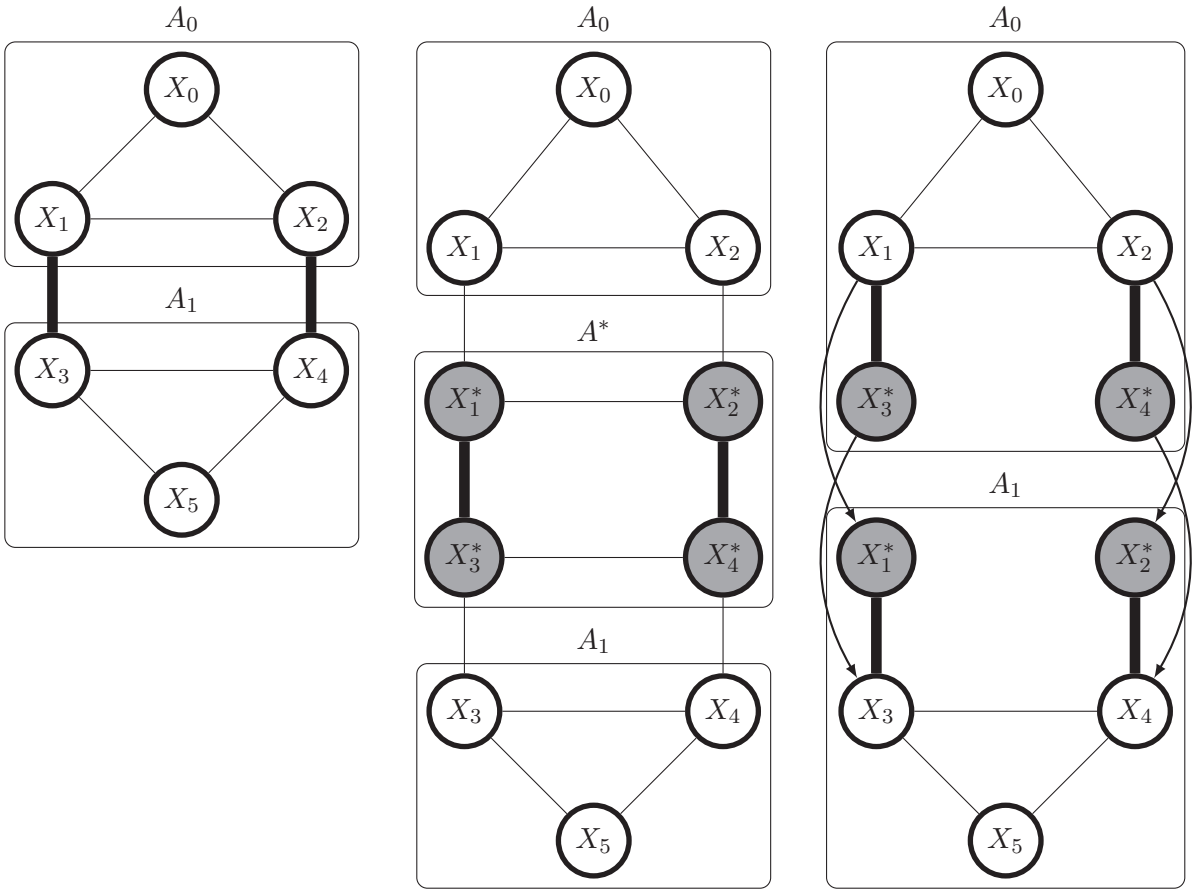

FIGURE 2.4 – Distribution explicite (à gauche), distribution canonique (au centre) et distribution canonique sans ajout d'agent (à droite).

2.3 Algorithmes de DisCSP mono-variable

De nombreux algorithmes de résolution de DisCSP existent tels que *ABT* [Yokoo, 2001], *DIBT* [Hamadi *et al.*, 1998, Hamadi, 1999b], *DDB* [Bessière *et al.*, 2001], *Distributed Breakout* [Yokoo et Hirayama, 1996, Yokoo, 2001] ou *AWC* [Yokoo, 1995]. Citons également les algorithmes d'optimisation (DCOP) tels que le *Dynamic Branch and Bound Distribué* [Benelallam *et al.*, 2008] ou *DyBop* [Ezzahir *et al.*, 2008]. Ces algorithmes permettent de résoudre des DisCSP où chaque agent ne gère qu'une seule variable.

Parmi tous ces algorithmes, *ABT* joue un rôle particulier et sert d'algorithme de référence pour de nombreux travaux dans ce domaine [Bessière *et al.*, 2001, Meisels et Zivan, 2007] ou encore [Muscalagiu, 2005]. Il nous semble donc important de le présenter ici.

2.3.1 Asynchronous Backtracking

L'algorithme Asynchronous Backtracking (*ABT*) est le premier algorithme permettant de résoudre des DisCSP de manière asynchrone. Tel qu'il est présenté dans [Yokoo *et al.*, 1992], cet algorithme permet de résoudre des DisCSP où chaque agent contient une unique variable. Les agents sont ordonnés selon un ordre arbitraire (généralement attribué pendant une phase d'initialisation).

Chaque agent assigne de manière concurrente une valeur à sa variable puis envoie cette assignation à ses accointances inférieures grâce à des messages de type ⟨OK?⟩. Dès qu'un agent ne peut pas trouver d'instanciation possible avec les assignations reçues, il en informe ses accointances supérieures via un message de type $\langle BT \rangle$. Ces dernières essaieront alors de modifier leur valeur courante.

Plaçons-nous au niveau d'un agent appelé A_i. A_i dispose d'un ensemble $agentView$ contenant les différentes assignations proposées par les accointances supérieures. Si aucune assignation n'est possible, en accord avec $agentView$, A_i construit un *nogood*. Il s'agit d'un sous-ensemble des valeurs de l'$agentView$ pour lesquelles A_i ne peut pas instancier sa variable : $\{(A_j, v_j), ..., (A_k, v_k)\}$. Le *nogood* est envoyé à l'agent de plus faible priorité A_k appartenant à ce nogood (A_k dispose obligatoirement d'une priorité supérieure à A_i). Il s'agit d'une demande de backtrack.

Chaque agent travaille de manière asynchrone. De ce fait, lorsqu'un message est émis, il se peut qu'il soit obsolète (par exemple si l'agent destinataire de ce message a modifié sa valeur courante entre temps). Il est donc nécessaire de joindre au message une information qui permet de savoir s'il doit être traité ou non. Ainsi, lorsqu'un agent A_i reçoit un *nogood* $\{(A_i, v_i), (A_j, v_j), ..., (A_k, v_k)\}$, il en tient compte uniquement si la valeur courante de sa variable est identique à celle du nogood (v_i) et si les valeurs présentes dans $agentView$ sont identiques à celles du *nogood*. Dans certains cas, un agent A_i peut recevoir un *nogood* contenant une variable inconnue appartenant à un agent A_j, A_i demande alors une création de lien avec l'agent A_j.

Nous allons maintenant observer le comportement des agents exécutant l'algorithme ABT pour résoudre le DisCSP présenté à la figure 2.5. Il s'agit d'un DisCSP contenant trois agents A_0, A_1 et A_2 disposant respectivement des variables X_0, X_1 et X_2. Ce disCSP comporte deux contraintes inter-agents, à savoir $X_1 \neq X_2$ et $X_0 \neq X_2$. La priorité des agents a été attribuée de manière lexicographique, donc l'agent A_0 est l'agent de plus haute priorité, suivi des agents A_1 et A_2.

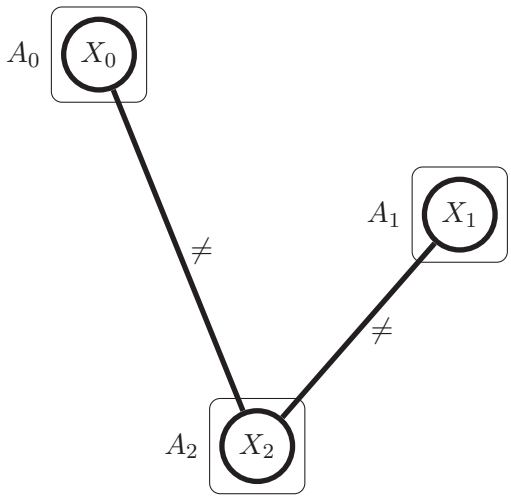

FIGURE 2.5 – Exemple de DisCSP mono-variable par agent comportant 3 agents.

Nous présentons, à la figure 2.6, un exemple possible d'exécution d'ABT où les agents sont activés à tour de rôle, en commençant par l'agent A_0. Durant la phase d'initialisation, l'agent A_0 choisit la valeur 0 pour sa variable A_0 et la communique ensuite grâce au message M_0. L'agent A_1 choisit la valeur 1 pour sa variable X_1 et la transmet grâce au message M_1. L'agent A_2 reçoit le message M_0 et met à jour son $agentView$. Celle-ci devient $agentView = \{(X_0, 0)\}$. Sa solution locale $X_2 = 1$ vérifie la contrainte inter-agents le reliant à A_0. Puis il reçoit le message M_1 et

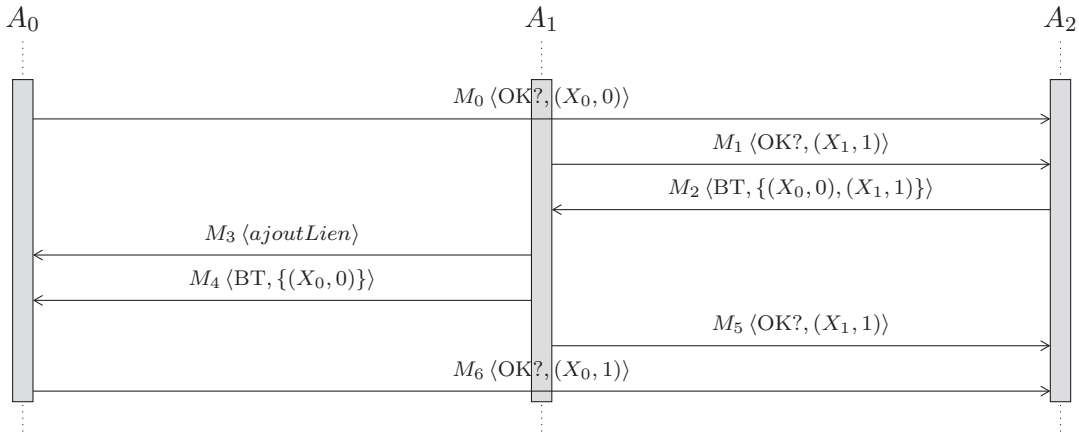

FIGURE 2.6 – Echange de messages entre les agents exécutant ABT.

met à jour son $agentView$ qui devient $agentView = \{(X_0, 0), (X_1, 1)\}$. Puis, A_2 cherche une valeur pour sa variable vérifiant les contraintes inter-agents le reliant aux agents A_0 et A_1 et n'en trouve pas. Il crée un nogood représentant la sous-partie de son $agentView$ pour laquelle il ne trouve pas de solution locale consistante : $\{(X_0, 0), (X_1, 1)\}$. Ce nogood est envoyé grâce au message M_2 au plus petit agent contenu dans le nogood, à savoir A_1.

Lorsque l'agent A_1 reçoit le message M_2 contenant un agent qu'il ne connaît pas (l'agent A_0), il envoie une demande d'ajout de lien à l'agent A_0. Puis, comme il ne peut modifier sa valeur courante, il poursuit la demande de backtrack vers l'agent A_0 grâce au message M_4 et émet sa solution locale à l'agent A_2 grâce au message M_5. En recevant la demande de backtrack M_4, l'agent A_0 choisit la valeur 1 pour sa variable puis la transmet à A_2 grâce au message M_6. L'agent A_2 reçoit ce dernier message, met à jour son $agentView$ qui vaut $agentView = \{(X_0, 1), (X_1, 1)\}$ et choisit la valeur 0 pour sa variable qui vérifie toutes les contraintes inter-agents. Plus aucun message n'est envoyé et la solution au DisCSP est trouvée : $\{(X_0, 1), (X_1, 1), (X_2, 0)\}$.

Après avoir détaillé l'algorithme ABT et présenté sur un exemple simple les différents messages pouvant être envoyés, nous allons nous intéresser à AWC, une extension d'ABT qui permet un ordonnancement dynamique des agents.

2.3.2 Asynchronous Weak-Commitment

L'algorithme AWC [Yokoo, 1995] est une variante de l'algorithme ABT. Contrairement à ABT, les agents se voient attribuer une priorité qui évolue au cours du temps. À l'initialisation de AWC, la priorité de chaque agent est fixée à zéro. En cas d'égalité entre deux variables, l'ordre lexicographique s'impose.

De manière similaire à ABT, des messages de types $\langle OK? \rangle$ et $\langle BT \rangle$ sont émis entre les agents. Pour tenir compte de l'évolution des priorités des agents, les messages de type $\langle OK? \rangle$ sont de la forme $\langle$ variable, valeur, priorité de l'agent émetteur $\rangle$. Lorsqu'un message de type $\langle OK? \rangle$ est reçu, contenant un triplet (variable, valeur, priorité), ce triplet est sauvegardé dans l'$agentView$ de l'agent.

Lorsque la valeur courante d'un agent vérifie les contraintes reliant cet agent aux agents de plus haute priorité alors la valeur courante est considérée comme consistante.

Puis, lorsqu'un agent A_i ne peut pas instancier sa variable en respectant les contraintes avec les agents de priorité supérieure, la priorité de l'agent A_i change et devient la plus élevée parmi

ses accointances.

Lorsqu'un agent doit instancier sa variable, il utilise l'heuristique min-conflit [Minton *et al.*, 1992] pour le choix de la valeur. La valeur choisie est celle qui respecte les contraintes avec les agents de priorité supérieure et qui viole le moins de contraintes possibles avec les agents de priorité inférieure.

Cet algorithme permet de modifier plus rapidement la variable qui provoque l'inconsistance. Dans ABT, il aurait fallu que toutes les valeurs de chaque agent de priorité inférieure aient été testées avant de modifier la variable appartenant à l'agent de plus haute priorité.

Nous allons maintenant observer les comportements d'AWC lors de la résolution du DisCSP présentée à la figure 2.5. Nous présentons, à la figure 2.7, un exemple possible d'exécution d'AWC où les agents sont activés à tour de rôle en commençant par l'agent A_0.

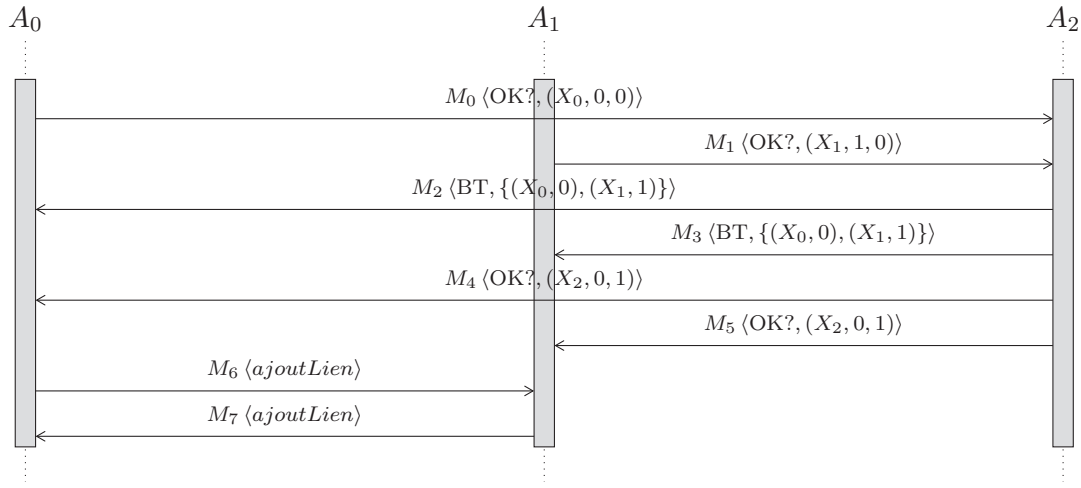

FIGURE 2.7 – Echange de messages entre les agents exécutant AWC.

Comme pour l'algorithme ABT, durant la phase d'initialisation, l'agent A_0 choisit la valeur 0 pour sa variable A_0 et la communique grâce au message M_0. L'agent A_1 choisit la valeur 1 pour sa variable X_1 et l'envoie grâce au message M_1. L'agent A_2 reçoit les messages M_0 et M_1 et met à jour son $agentView$ qui devient $agentView = \{(X_0, 0, 0), (X_1, 1, 0)\}$. A_2 cherche en vain une valeur pour sa variable vérifiant les contraintes inter-agents le reliant aux agents A_0 et A_1. Comme pour ABT, A_2 crée un nogood représentant la sous-partie de son $agentView$ pour laquelle il ne trouve pas de solution locale consistante : $\{(X_0, 0), (X_1, 1)\}$. Ce nogood est transmis grâce aux messages M_2 et M_3 aux agents A_1 et A_2. Puis, contrairement à ABT, comme l'agent a envoyé une demande de backtrack, sa priorité devient la plus élevée parmi ses accointances. La priorité de A_0 étant de 0 et celle de l'agent A_1 valant 0, la priorité de l'agent A_2 passe à 1. L'agent A_2 communique sa solution courante à ces accointances de priorité inférieure grâce aux messages M_4 et M_5.

L'agent A_0 reçoit la demande de backtrack M_2, ne modifie pas sa solution courante car toutes les contraintes le reliant à des agents de priorité supérieure sont respectées. Il demande un ajout de lien à l'agent A_1, qu'il ne connaît pas, présent dans ce message avec le message M_6. Puis A_0 reçoit M_4. Son $agentView$ devient alors $\{(X_2, 0, 1)\}$ et il choisit la valeur 0 pour sa variable afin de satisfaire la contrainte $X_0 \neq X_1$.

L'agent A_1 reçoit la demande de backtrack M_3, ne modifie pas sa solution courante et demande un ajout de lien à l'agent A_0, qu'il ne connaît pas, présent dans M_6. Puis A_1 reçoit M_5. Son $agentView$ devient alors $\{(X_2, 0, 1)\}$ et il garde son unique valeur 0 qui vérifie la

contrainte $X_1 \neq X_2$. Les agents reçoivent les demandes d'ajout de lien. Il n'y a plus aucun message échangé entre les agents. La résolution est donc terminée et *AWC* trouve la même solution que *ABT*, c'est-à-dire $\{(X_0, 1), (X_1, 1), (X_2, 0)\}$.

Nous avons décidé de détailler, dans cette section, l'algorithme *ABT* car il sert d'algorithme de référence pour de nombreux travaux dans le domaine de DisCSP. Puis nous avons présenté l'algorithme *AWC* qui a l'intérêt d'utiliser un ordonnancement dynamique entre les agents. Nous allons maintenant observer, dans la section suivante, une généralisation possible de ces deux algorithmes afin de traiter des DisCSP où le problème local de chaque agent est composé de plusieurs variables et plusieurs contraintes intra-agent. De plus, nous présenterons l'algorithme *AFC*.

2.4 Algorithmes de DisCSP multi-variables

Nous présentons, dans cette section, des algorithmes permettant de résoudre des DisCSP disposant de problèmes locaux complexes, c'est-à-dire composés d'un ensemble de variables et de contraintes intra-agent. Nous commençons par présenter deux méthodes pouvant être utilisées :

- Une première méthode consiste à ce que chaque agent, qui traite un problème local à n variables, crée n agents virtuels. Chaque agent virtuel s'occupe d'une variable locale. Ainsi les agents n'ont pas besoin de trouver toutes les solutions de leurs problèmes locaux avant d'effectuer la recherche globale. Cependant, ceci augmente considérablement le nombre d'agents. L'algorithme *Multi-AWC* [Yokoo et Hirayama, 1998] reprend ce principe.

- Une seconde méthode consiste à reformaliser le problème de manière à ce que chaque agent dispose d'une unique variable pouvant prendre comme valeur une solution de son problème. Par exemple, l'unique variable d'un agent A_i pourrait prendre la valeur $A_i = \{(X_0 = 1), (X_1 = 1), (X_2 = 0)\}$ si le CSP local de A_i contenait les trois variables X_0, X_1 et X_2. Le domaine de l'unique variable peut être calculé à l'initialisation de l'algorithme même si c'est très coûteux. Ce domaine peut aussi être calculé en parallèle de la recherche locale.

Nous allons maintenant détailler une généralisation de l'algorithme *ABT* utilisant la première méthode présentée ici.

2.4.1 Multi Asynchronous Backtracking

L'algorithme *ABT* a été étendu en *Multi-ABT* [Hirayama *et al.*, 2000, Hirayama *et al.*, 2004] afin que chaque agent gère plusieurs variables. Cet algorithme reprend les bases de l'algorithme *ABT* présenté dans [Yokoo *et al.*, 1998] et suit les points suivants :

- Comme dans *ABT*, un ordre de priorité est établi entre les différents agents. Cependant, pour cette version multi-variables, un ordre de priorité est aussi établi entre les variables internes à chaque agent.
- Lorsqu'un agent A_i reçoit un message de type $\langle OK? \rangle$, il met à jour son $agentView$ en sauvegardant la valeur des variables qu'il vient de recevoir puis il exécute pour chacune de ses variables, notées X_j^i (i représente le numéro de l'agent et j le numéro de la variable), la séquence suivante :

  1. Il choisit une valeur pour X_j^i qui soit à la fois consistante avec les valeurs des variables

présentes dans l'$agentView$ et les valeurs des variables internes de priorité supérieure à celle de X_j^i. Si la valeur de X_j^i est déjà consistante, elle n'est pas modifiée. Puis, l'agent passe à la variable interne suivante dans l'odre de priorité.

2. S'il n'y a aucune valeur consistante pour X_j^i alors un nogood est généré en utilisant la méthode décrite dans [Hirayama et Yokoo, 2000]. Ce nogood est un ensemble de triplets $(agent, variable, valeur)$ pour lesquels aucune valeur pour X_j^i ne peut être consistante. Si le nogood est vide, *Multi-ABT* peut alors affirmer qu'aucune solution n'existe.
3. Lorsqu'un nogood est généré par un agent A_i, s'il contient un triplet de ce même agent A_i alors ce nogood est sauvegardé comme une nouvelle contrainte. Puis l'algorithme effectue un retour-arrière vers le plus petit triplet de ce nogood. Par contre, si le nogood ne contient aucune variable de l'agent A_i alors ce nogood est stocké dans un ensemble contenant les nogoods devant être envoyés. A_i essaye ensuite d'instancier la variable suivante X_{j+1}^i.
4. Lorsque toutes les variables d'un agent sont instanciées, l'agent envoie alors les nogoods présents dans l'ensemble contenant les nogoods devant être envoyés. L'agent envoie ensuite les nouvelles valeurs des variables aux agents de priorité inférieure reliés par une contrainte inter-agents.

– Lorsqu'un agent reçoit un message de type $\langle BT \rangle$ contenant un nogood, ce dernier est sauvegardé comme une nouvelle contrainte et l'agent examine la consistance de ses variables de la même manière que lorsqu'il reçoit un message de type $\langle OK? \rangle$. Cependant si ce nogood contient une variable appartenant à un agent qu'il ne connaît pas, il demande à cet agent de lui envoyer la valeur de cette variable et ajoute cet agent à sa liste d'accointances.

Nous allons maintenant observer le comportement de l'algorithme *Multi-ABT*. La figure 2.8 représente un problème de coloration de graphe distribué comportant trois agents A_0, A_1 et A_2. Chaque agent dispose de trois variables pouvant prendre comme valeur *blanc* ou *noir*. Les variables reliées par une contrainte doivent avoir des valeurs différentes. Nous allons maintenant observer le comportement de l'algorithme *Multi-ABT* et un ordre d'envoi possible de messages présentés à la figure 2.9. La priorité de chaque agent est attribuée de manière lexicographique : A_0 a la plus haute priorité et A_2 la plus petite.

L'algorithme s'initialise de la manière suivante, l'agent A_0 choisit la solution locale suivante $\{(X_0, noir), (X_1, blanc), (X_2, noir)\}$. A_0 communique ensuite la valeur de ses variables d'interface grâce aux messages M_1 à M_3. L'agent A_1 choisit la solution locale $\{(X_3, blanc), (X_4, noir), (X_5, blanc)\}$ et la communique grâce au message M_4. L'agent A_2 choisit la solution locale $\{(X_6, noir), (X_7, blanc), (X_8, noir)\}$.

L'agent A_1 reçoit le message M_1. Il vérifie alors l'ensemble de ses variables reliées à cet agent, c'est-à-dire les variables X_4 et X_5. Comme aucune valeur n'est consistante pour X_4 lorsque X_0 vaut *noir* et X_3 vaut *blanc*. L'agent crée donc le nogood $\{(A_0, X_0, noir), (A_1, X_3, blanc)\}$, le sauvegarde et choisit alors *noir* pour X_3 puis *blanc* pour X_4 et *noir* pour X_5. Il envoie alors le message M_5 à l'agent A_2. Puis l'agent A_1 reçoit le message M_2, modifie son $agentView$ et détecte que toutes ses variables sont déjà consistantes. Il n'envoie alors aucun message.

L'agent A_2 reçoit les messages M_3 et M_4 et ne fait rien car sa solution courante est consistante avec sa nouvelle $agentView$ qui est $\{(A_0, X_1, blanc), (A_1, X_5, blanc)\}$. Il reçoit ensuite le message M_5. Il modifie alors la valeur de sa variable X_6 en *blanc* pour vérifier la contrainte inter-agents $X_5 \neq X_6$ et choisit *noir* pour X_7 pour vérifier la contrainte intra-agent $X_6 \neq X_7$. Il détecte qu'il ne peut trouver de valeur consistante pour X_8 et crée alors le nogood $\{(A_0, X_1, blanc), (A_2, X_7, noir)\}$. À cause de ce nogood, il ne peut plus trouver de valeur pour

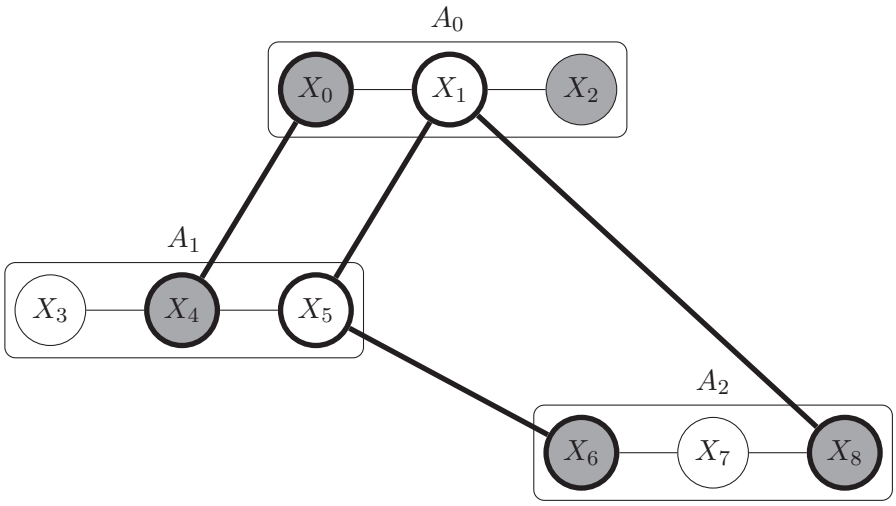

FIGURE 2.8 – Problème de coloration de graphe distribué.

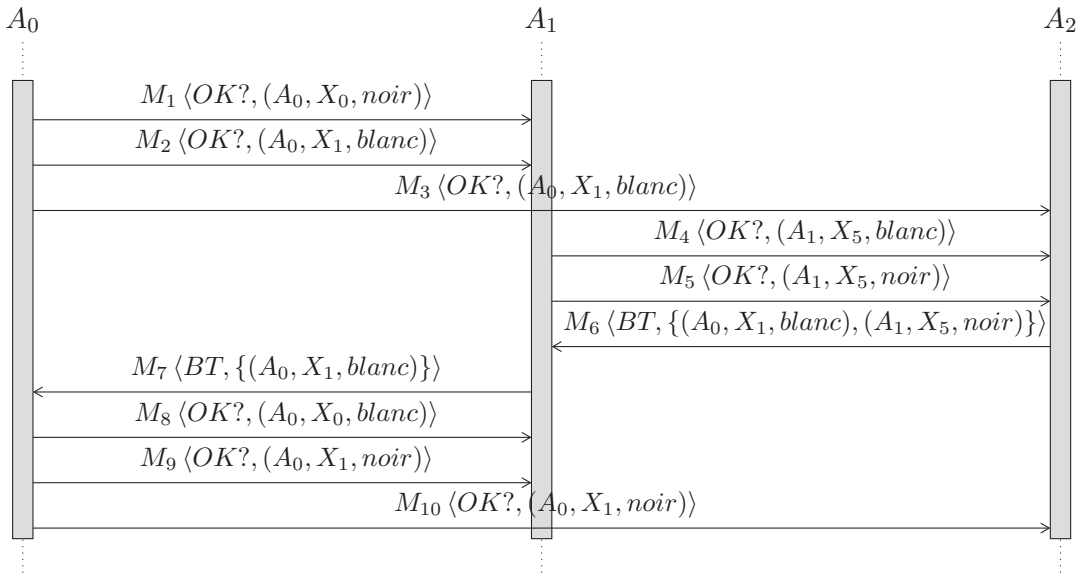

FIGURE 2.9 – Echange de messages entre les agents A_i.

X_7 et crée un nouveau nogood $\{(A_0, X_1, blanc), (A_2, X_6, blanc)\}$. Ce nogood entraîne la création d'un troisième nogood $\{(A_0, X_1, blanc), (A_1, X_5, noir)\}$. Ce nogood est envoyé à l'agent A_1 par le message M_6.

L'agent A_1 reçoit M_6. Comme il ne peut trouver de valeur consistante pour sa variable X_5 vérifiant à la fois la contrainte inter-agents $X_1 \neq X_5$ et le nogood $\{(A_0, X_1, blanc), (A_1, X_5, noir)\}$, il crée le nogood $\{(A_0, X_1, blanc)\}$ qu'il envoie à l'agent A_0 grâce au message M_7.

L'agent A_0 reçoit M_7. Comme la variable X_1 ne peut plus prendre la valeur *blanc* et que la contrainte $X_0 \neq X_1$ doit être satisfaite, il crée alors le nogood local $\{(A_0, X_0, noir)\}$. L'agent choisit *blanc* pour X_0 puis *noir* pour X_1 et enfin *blanc* pour X_2. Il envoie les nouvelles valeurs de ses variables d'interface avec les messages M_8 à M_{10}. De même que précédemment, différents messages sont alors envoyés entre les agents aboutissant à la création du nogood $\{(A_0, X_1, noir)\}$.

En recevant ce message, l'agent A_0 ne peut trouver aucune valeur consistante pour X_1 est crée un nogood vide signifiant que le DisCSP est inconsistant.

Après cette présentation générale de *Multi-ABT*, nous présentons dans la suite *Multi-AWC*.

2.4.2 Multi Asynchronous Weak-Commitment

L'algorithme *AWC* a été étendu en *Multi-AWC* [Yokoo et Hirayama, 1998] afin que chaque agent puisse gérer plusieurs variables. Chaque agent simule un agent virtuel pour chacune de ses variables locales. Cependant, *Multi-AWC* garde l'organisation inter-agents initiale : toutes les contraintes intra-agent initiales doivent être satisfaites avant d'envoyer un message « en dehors » des agents initiaux. L'algorithme *Multi-AWC* diffère de *AWC* sur les points suivants :

- Contrairement à *AWC*, dans *Multi-AWC*, on ne peut plus parler de priorité entre les agents mais uniquement de priorité entre les variables d'un agent. En effet, un agent A_i peut à la fois avoir des variables de priorité supérieure et inférieure à d'autres variables d'un agent A_j.
- Lorsque les agents souhaitent communiquer leurs valeurs, ils envoient un message $\langle OK? \rangle$ contenant un tuple (agent, variable, valeur, priorité).
- Chaque agent change séquentiellement les valeurs de ses variables locales. Pour cela, chaque agent commence par sélectionner la variable ayant la plus haute priorité parmi l'ensemble de ses variables qui violent des contraintes avec des variables de priorité supérieure. Puis il modifie la valeur de cette variable afin de satisfaire ces contraintes tout en minimisant le nombre de contraintes violées le reliant à des variables de priorité inférieure.
- S'il n'existe aucune valeur pour une variable satisfaisant les contraintes avec les variables de priorité supérieure alors l'agent augmente la priorité de cette variable de manière à ce qu'elle soit la plus haute parmi ses accointances.
- Après avoir modifié la valeur d'une variable, l'agent sélectionne la seconde variable par ordre de priorité violant une contrainte avec une variable de priorité supérieure. L'algorithme continue de manière itérative jusqu'à ce qu'il n'y ait plus de variables violant des contraintes avec des variables de priorité supérieure.

Notons que contrairement à *Multi-ABT*, les messages de type $\langle BT \rangle$ sont émis au fur et à mesure que l'agent modifie les valeurs de ses variables alors que dans *Multi-ABT*, les messages sont émis seulement lorsque l'agent ne modifie plus sa solution locale. Nous allons maintenant observer le comportement de *Multi-AWC* en reprenant l'exemple de la figure 2.8 que nous avons déjà utilisé pour *ABT* et nous détaillerons un ordre d'émission possible de messages présenté à la figure 2.10.

Durant la phase d'initialisation, la priorité de chaque variable de chaque agent est fixée à 0. En cas d'égalité entre deux variables, l'ordre lexicographique prime. L'algorithme s'initialise de la même manière que pour *Multi-ABT*. L'agent A_0 choisit la solution locale suivante $\{(X_0, noir)$, $(X_1, blanc)$, $(X_2, noir)\}$. A_0 communique ensuite la valeur de ses variables d'interface grâce aux messages M_1 à M_3. L'agent A_1 choisit la solution locale $\{(X_3, blanc)$, $(X_4, noir)$, $(X_5, blanc)\}$ et la communique via un message M_4. L'agent A_2 choisit la solution locale $\{(X_6, noir)$, $(X_7, blanc)$, $(X_8, noir)\}$.

L'agent A_1 reçoit le message M_1. Il sélectionne la variable X_4 qui est sa variable de plus haute priorité violant une contrainte avec une variable de priorité supérieure. Comme aucune valeur n'est disponible pour X_4 satisfaisant à la fois les contraintes $X_0 \neq X_4$ et $X_3 \neq X_4$, il crée le nogood $\{(A_0, X_0, noir), (A_1, X_3, blanc)\}$, le sauvegarde et le transmet à A_0 par le message M_5. Il modifie alors la priorité de X_4 à 1. Puis l'agent sélectionne la valeur qui viole le moins de contraintes le reliant à des variables de priorité inférieure. Il s'agit de *noir* qui ne viole que

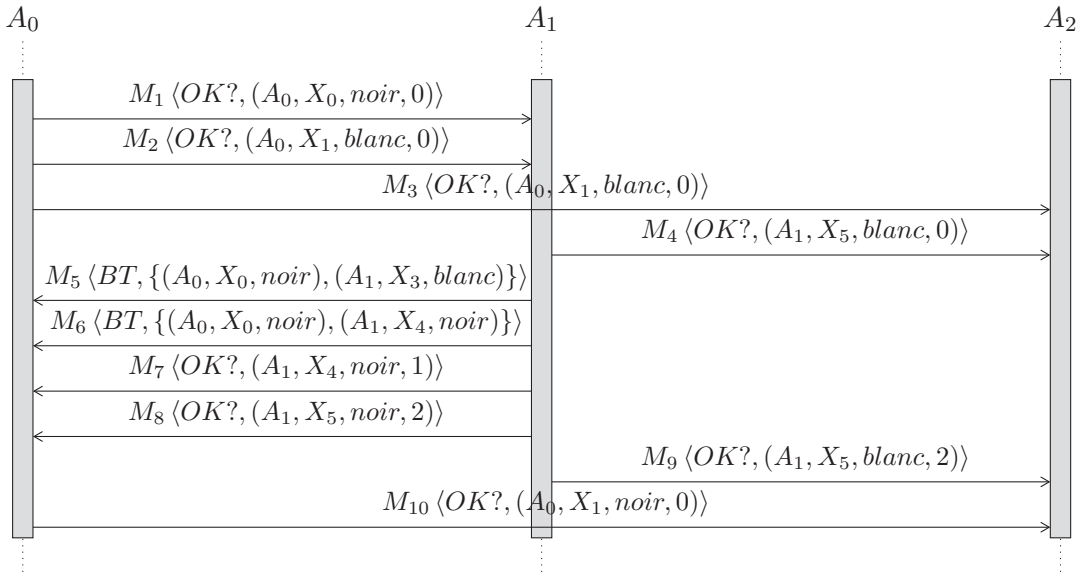

FIGURE 2.10 – Echange de messages entre les agents A_i.

la contrainte $X_0 \neq X_4$. Puis l'agent A_1 choisit à nouveau sa variable de plus haute priorité violant une contrainte : il s'agit de la variable X_3. Comme aucune valeur consistante n'existe pour cette variable, il crée le nogood $\{(A_0, X_0, noir), (A_1, X_4, noir)\}$ et la priorité de X_3 devient 2. Ce nogood est envoyé à l'agent A_0 par le message M_6, A_1 choisit ensuite aléatoirement la valeur *blanc* pour X_3. À ce moment, la solution courante de A_1 satisfait son $agentView$. A_1 communique les changements aux agents de priorité inférieure grâce aux messages M_7 à M_9.

L'agent A_1 reçoit le message M_2. Il sélectionne la variable X_5 qui est sa variable de plus haute priorité violant une contrainte avec une variable de priorité supérieure. Comme il ne peut modifier la valeur de X_5 sans violer les contraintes $X_1 \neq X_5$ et $X_4 \neq X_5$, il crée et sauvegarde le nogood $\{(A_0, X_1, blanc), (A_1, X_4, noir)\}$ et la priorité de X_5 devient 2. A_1 choisit ensuite la valeur *blanc* pour X_5 car toutes les valeurs violent une contrainte. Le nogood est envoyé à l'agent A_0 grâce au message M_6.

L'agent A_2 reçoit les messages M_3, M_4 et M_9 et ne fait rien car sa solution courante est consistante avec sa nouvelle $agentView$ qui est alors $\{(A_0, X_1, blanc, 0), (A_1, X_5, blanc, 2)\}$.

L'agent A_0 reçoit les messages M_5 et M_6. Il sauvegarde ces nouveaux nogoods mais ne modifie pas sa solution locale qui vérifie toutes les contraintes inter-agents car son $agentView$ est actuellement vide. Puis A_0 reçoit M_7. Son $agentView$ devient $\{(A_1, X_4, noir, 1)\}$. La variable X_0 viole le nogood $\{(A_0, X_0, noir), (A_1, X_4, noir)\}$. Il sélectionne la valeur *blanc* pour X_0. Puis A_0 modifie la valeur de X_1 à *noir* afin de vérifier la contrainte intra-agent $X_0 \neq X_1$. De même, il modifie X_2 à *blanc*. La solution courante de A_0 est alors consistante avec son $agentView$. Il communique les changements grâce au message M_{10}. A_0 reçoit M_8 mais ne modifie pas sa solution courante.

L'algorithme *Multi-AWC* aboutira au même résultat que *Multi-ABT*. Un nogood vide sera également créé indiquant qu'aucune solution n'existe au DisCSP. Après avoir expliqué le fonctionnement de *Multi-AWC*, nous allons présenter *AFC*, un algorithme de résolution de DisCSP multi-variables par agent basé sur le Forward-Checking.

2.4.3 Asynchronous Forward Checking

L'algorithme *Asynchronous Forward Checking - AFC* [Meisels et Zivan, 2007] est un algorithme de résolution de DisCSP multi-variables par agent ayant la particularité de rechercher des solutions de manière synchrone tout en propageant les assignations (Forward Checking) de manière asynchrone.

La partie synchrone de *AFC* est la suivante : les assignations dans *AFC* sont réalisées par un seul agent à la fois. Un agent assigne ses variables locales uniquement lorsqu'il reçoit un message nommé CPA pour *Current Partial Assignment*. Ce message contient une affectation partielle du DisCSP que les agents essayent d'étendre pour obtenir une solution contenant toutes les variables du DisCSP. Lorsqu'un agent a affecté ses variables locales, celles-ci sont ajoutées au CPA. Lorsqu'un agent ne peut plus trouver de solution locale en accord avec son $agentView$, il envoie un message nommé $backtrack\_CPA$ au plus petit agent appartenant à son $agentView$.

La partie asynchrone de *AFC* est la suivante : à chaque fois qu'un agent envoie le CPA, il envoie des copies de ce CPA, avec des messages nommés FC_CPA, aux agents qui ne sont pas encore affectés. Puis, lorsqu'un agent reçoit un message FC_CPA, il met à jour le domaine de ses variables locales de manière à ce que les valeurs restantes soient compatibles avec les variables du FC_CPA. Si le domaine d'une variable devient vide, c'est qu'il y a une inconsistance. Dans ce cas, une demande de backtrack est créée contenant un nogood (la plus petite sous-partie de son $agentView$ pour laquelle l'agent ne trouve pas de solution consistante). Ce nogood est envoyé grâce à un message Not_OK vers les agents non présents dans le message FC_CPA reçu. Comme le CPA est unique, même si un agent reçoit plusieurs messages Not_OK, un seul et unique retour-arrière est effectué.

AFC n'utilise pas d'algorithme externe de détection de terminaison. En effet, il détecte seul qu'une solution est trouvée, c'est-à-dire lorsqu'un CPA contenant toutes les variables du DisCSP est créé. Grâce à cela, *AFC* n'a pas besoin d'utiliser d'algorithme de détection de la terminaison. Lorsqu'aucune solution n'existe, un agent le détecte et arrête la résolution du problème.

Nous allons maintenant observer le comportement de *AFC* sur le même exemple que nous avons utilisé pour expliquer les algorithmes *Multi-ABT* et *Multi-AWC*.

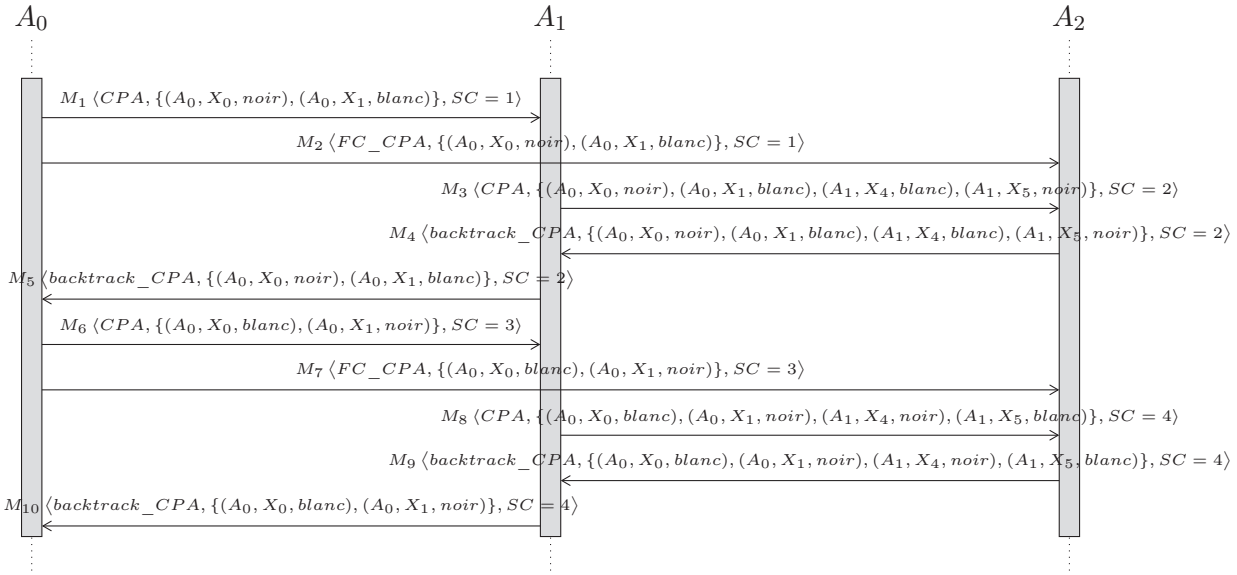

FIGURE 2.11 – Echange de messages entre les agents A_i.

Durant la phase d'initialisation, contrairement aux algorithmes *Multi-ABT* et *Multi-AWC*, un seul agent est activé. Il s'agit de l'agent de plus haute priorité, ici A_0. A_0 génère le CPA et l'assigne avec sa solution courante (afin de simplifier l'exemple, le CPA contient uniquement des variables d'interface). Le CPA contient les tuples suivants : $\{(A_0, X_0, noir), (A_0, X_1, blanc)\}$. A_0 communique ce CPA grâce au message M_1. Puis, A_0 communique, pour la phase de propagation, un message FC_CPA aux agents non assignés dans le CPA via le message M_2.

L'agent A_1 reçoit le message M_1. Il met alors à jour son $agentView$ avec les données présentes dans le CPA. A_1 choisit alors la solution locale $\{(X_3, noir), (X_4, blanc), (X_5, noir)\}$ et l'ajoute au CPA avant de le transmettre grâce au message M_3 à l'agent A_2.

A_2 reçoit M_2. Son $agentView$ devient alors$\{(A_0, X_0, noir), (A_0, X_1, blanc)\}$. Comme il existe une solution locale consistante avec l'$agentView$, la phase de propagation de AFC ne permet pas d'envoyer un message d'échec nommé Not_OK. A_2 reçoit ensuite M_3, met à jour sa vue et envoie le message M_4 car aucune solution locale consistante avec sa nouvelle vue n'existe. A_1 reçoit M_4 et poursuit la demande de backtrack vers l'agent A_0 grâce au message M_5.

En recevant M_5, A_0 modifie sa solution courante en $\{(X_0, blanc), (X_1, noir), (X_2, blanc)\}$ et modifie le CPA avec ces nouvelles valeurs. Ce CPA devient alors $\{(A_0, X_0, blanc), (A_0, X_1, noir)\}$. De même que précédemment, le CPA est transmis à A_1 et un message FC_CPA est envoyé à A_2. Les messages M_6 à M_{10} sont émis. La résolution se termine lorsque A_0 reçoit M_{10}. Il ne peut trouver aucune solution locale consistante. Comme A_0 est l'agent initialisateur pour cet exemple, il ne peut envoyer de demande de backtrack et termine l'algorithme.

2.4.4 Synthèses des approches existantes

Nous présentons, tableau 2.1, une synthèse des différents algorithmes de résolution de DisCSP multi-variables présentés dans ce chapitre. Nous avons indiqué, pour chacun de ces algorithmes si :

- les agents interagissent en terme de comportements de manière synchrone/asynchrone,
- il y a une création d'agents virtuels pour chacune des variables locales,
- il est possible de définir une priorité dynamique entre les différents agents et/ou variables,
- il y a des créations de liens entre les agents au cours de la recherche,
- il y a une utilisation de nogoods entre les agents.

TABLE 2.1 – Synthèses des algorithmes de résolution de DisCSP.

	Comportement asynchrone	Création d'agents virtuels	Priorité dynamique variables/agents	Création de liens entre les agents	Création de nogoods
Multi-ABT	oui	oui	non	oui	oui
Multi-AWC	oui	oui	oui	oui	oui
AFC	oui/non	non	oui	oui	oui

La gestion de l'asynchronisme entre les agents d'un DisCSP est un point très important. En effet, afin de réduire le temps total de la résolution d'un problème, il est important que

les agents soient en permanence en train de traiter ou d'envoyer des messages. Comme nous l'avons présenté dans cette section, les algorithmes *Multi-ABT* et *Multi-AWC* fonctionnent de manière asynchrone. Contrairement à ces algorithmes, *AFC* utilise à la fois du synchronisme et de l'asynchronisme entre les agents. Les messages échangés de manière synchrone servent pour la recherche de solutions tandis que ceux envoyés de manière asynchrone servent uniquement pour la propagation.

Nous avons vu qu'il existe deux principales méthodes de généralisation d'un algorithme mono-variable vers le multi-variables. La première méthode consiste à ce que chaque agent, qui traite un problème local à n variables, crée n agents virtuels. La seconde méthode consiste à reformaliser le problème de manière à ce que chaque agent dispose d'une unique variable pouvant prendre comme valeur une solution de son problème. Nous avons vu, section 2.4, que chacune de ces méthodes avait des avantages et des inconvénients ; mais nous pensons que la création d'agents virtuels induit un très grand nombre d'agents, comme nous le verrons dans le chapitre 4. Nous préférons bénéficier du fait que les variables et les contraintes locales soient centralisées pour utiliser un solveur de CSP centralisé.

Une priorité dynamique entre les agents d'un DisCSP permet d'éviter de nombreuses vérifications de contraintes inutiles en modifiant plus rapidement la cause de l'inconsistance. Cependant, la mise en place d'une priorité dynamique entre les agents tout en conservant la complétude de l'algorithme est très difficile. *Multi-ABT* ne permet pas ce changement dynamique de priorité contrairement à l'algorithme *Multi-AWC* qui a été conçu dans ce but. Il en est de même pour *AFC* pour lequel la priorité dynamique est possible grâce au fait que la phase de recherche de solutions soit synchrone.

De nombreux algorithmes utilisent des créations de liens entre les agents au cours de la recherche. Ces nouveaux liens sont créés lorsqu'un agent reçoit un nogood contenant un agent inconnu. Une demande de lien est alors envoyée permettant à l'agent de recevoir, pour le reste de la résolution du DisCSP, les changements de valeurs de ce nouvel agent [1]. C'est le cas des algorithmes *Multi-ABT* et *Multi-AWC*. Contrairement à ces deux algorithmes, *AFC* considère que chaque agent connaît les autres agents et qu'aucun message d'ajout de liens n'est utilisé.

Les algorithmes *Multi-ABT*, *Multi-AWC* et *AFC* utilisent des nogoods pour établir un contexte à leurs demandes de backtrack. À notre connaissance, peu d'efforts ont été fournis pour minimiser le temps de traitement des demandes de backtrack dans les algorithmes de DisCSP. Nous pensons que c'est un point d'amélioration et nous utilisons un contexte pour ces messages qui ne soit pas des nogoods. En effet, la création d'un nogood est coûteuse (en calculs) même si des méthodes sont apparues pour optimiser leur création. Nous proposons d'utiliser comme contexte un entier, que nous appellerons dans le chapitre suivant une « session ». Cet entier, que nous joindrons à chaque message échangé permettra pour chaque agent de déterminer si ce dernier est obsolète ou non.

Nous présenterons, au chapitre suivant, un algorithme de résolution de DisCSP dont les agents agissent complètement de manière asynchrone. De plus, notre proposition ne créera pas d'agents virtuels pour les raisons que nous avons énoncées précédemment et il n'y aura pas d'ajouts de liens entre les agents. La principale particularité de notre algorithme sera de ne pas utiliser de nogoods.

1. Notons que pour qu'un agent puisse envoyer un message de demande d'ajout de lien à un autre agent, il faut qu'il connaisse au préalable son « adresse » pour lui transmettre ce message.

2.5 Conclusion

Dans la seconde section de ce chapitre, nous avons présenté les DisCSP. Nous avons commencé par fournir différentes définitions relatives au domaine des DisCSP. Nous avons présenté divers problèmes pouvant être résolus grâce aux DisCSP tels que les problèmes de sensor-disCSP ou encore des problèmes d'exploration multi-robots. Puis nous avons expliqué l'une des principales difficultés lors de la résolution des DisCSP, à savoir la gestion de l'asynchronisme entre les agents et une contextualisation des messages échangés entre les agents. Ensuite, nous avons décrit différentes méthodes permettant de détecter la fin de la résolution lorsqu'il n'y a plus de messages échangés entre les agents. Nous avons terminé cette section par différentes manières de représenter un DisCSP telle que la représentation explicite qui est la plus utilisée dans le domaine.

La troisième section a porté sur des algorithmes de résolution de DisCSP où le problème local d'un agent ne contient qu'une unique variable. Nous avons détaillé, pour commencer, le premier algorithme de résolution de DisCSP de manière asynchrone : il s'agit de l'algorithme *ABT*. Nous avons expliqué son fonctionnement et nous avons observé les messages pouvant être échangés entre les agents durant la résolution grâce à un exemple simple de DisCSP comportant trois agents. Puis nous avons expliqué le fonctionnement de l'algorithme *AWC* qui a la particularité d'utiliser un ordonnancement dynamique entre les agents. Nous avons vu qu'après avoir envoyé une demande de backtrack, la priorité de l'agent augmente pour devenir la plus importante parmi ses accointances.

La quatrième section a été consacrée à la résolution des DisCSP disposant de problèmes locaux complexes, c'est-à-dire composés de plusieurs variables et contraintes intra-agent. Nous avons commencé par décrire deux méthodes permettant cette généralisation. Nous avons détaillé l'algorithme *Multi-ABT* qui est une généralisation de *ABT*. Puis nous avons présenté l'algorithme *Multi-AWC* qui est une généralisation de *AWC*. Nous avons vu que cette généralisation avait la particularité de créer autant d'agents virtuels qu'il y a de variables dans le problème initial. Enfin, nous avons détaillé l'algorithme *AFC* qui a la particularité de combiner une partie synchrone pour la recherche de solutions avec une partie asynchrone pour la propagation.

Nous proposons, dans le chapitre suivant, un algorithme, *DBS*, conçu permettant de lever la plupart des limites présentées dans le tableau 2.1. Pour cela, plutôt que d'utiliser un nogood pour établir un contexte pour les messages de backtrack, notre proposition utilise la notion de session. Grâce à ce nouveau contexte, nous verrons que les messages sont créés plus rapidement. D'autre part, nous verrons que *DBS* associe des sessions aux différents messages échangés. La gestion des sessions peut être améliorée en exploitant des propriétés qui réduisent significativement le nombre total de messages tout en conservant la complétude de l'algorithme.

3

Le Backtracking Distribué avec Sessions

Sommaire

3.1 Introduction

Nous avons présenté, dans le chapitre précédent, différents algorithmes de résolution de DisCSP mono-variable et multi-variables par agent. Nous avons vu leur fonctionnement sur des exemples simples ainsi que les mécanismes de coordination inter-agents. Nous avons terminé ce chapitre sur les limites de ces algorithmes.

Dans ce chapitre, nous nous intéressons à une nouvelle méthode permettant d'associer un contexte aux différents messages échangés entre les agents : les sessions. La création d'un contexte pour chaque message échangé durant la résolution d'un DisCSP nous paraît indispensable si l'on souhaite que les agents travaillent de manière asynchrone. En effet, en l'absence de contexte, il se peut qu'un message reçu par un agent soit obsolète.

Contrairement à la plupart des algorithmes tels que *Multi-ABT* [Hirayama *et al.*, 2000] [Hirayama *et al.*, 2004] ou *Multi-AWC* [Yokoo et Hirayama, 1998] dans lesquels un nogood est joint

à chaque demande de backtrack, nous proposons un algorithme où la contextualisation d'un message reposera sur un entier : une session. Notre algorithme, nommé *DBS* pour *Distributed Backtracking with Sessions*, permet la résolution de DisCSP où chaque agent dispose d'un problème local « complexe » [Yokoo et Hirayama, 1998] composé de plusieurs variables et de plusieurs contraintes.

Dans la première section, nous présenterons les hypothèses et les notations de cet algorithme, initialement présentées dans [Doniec *et al.*, 2005] puis complètement revues et étendues dans [Monier *et al.*, 2009b, Monier *et al.*, 2009a]. Les hypothèses utilisées dans la plupart des algorithmes de résolution de DisCSP sont reprises. Puis nous présenterons les différentes structures de données utilisées dans *DBS*.

La seconde section présentera en détail le fonctionnement de l'algorithme *DBS*. Nous nous intéresserons à la résolution du CSP local propre à chaque agent. Ensuite, le choix de la solution locale en fonction du contexte d'un agent sera exposé et le choix des données à communiquer entre les agents sera précisé. Enfin, cette section se terminera avec le pseudo-code détaillé de notre proposition et un exemple simple permettant d'observer les différents messages échangés entre les agents.

Dans la troisième section, nous aborderons les propriétés de l'algorithme *DBS*. Nous commencerons par exposer la preuve de correction de cet algorithme avant de présenter la preuve de complétude. Puis la complexité spatiale et la complexité temporelle de *DBS* seront fournies. Enfin, nous détaillerons les différents traitements que chaque agent peut effectuer sur la liste de messages en attente de traitement dans les boîtes de réception tout en conservant la complétude de *DBS*.

3.2 Généralités

Dans cette section, nous présentons les hypothèses nécessaires au fonctionnement de l'algorithme *DBS*. Puis nous présentons les structures de données utilisées permettant d'établir le contexte de chaque agent.

3.2.1 Hypothèses et notations

Nous proposons un algorithme de résolution de DisCSP multi-variables par agent. De ce fait, contrairement à la majorité des algorithmes de résolution de DisCSP, nous supposons que le problème local d'un agent est complexe, c'est-à-dire composé de plusieurs variables et contraintes intra-agent.

Hypothèse 1 *Un sous-CSP, composé de plusieurs variables et contraintes intra-agent, est attribué à chaque agent exécutant l'algorithme* DBS.

Nous avons vu au chapitre 2 que l'utilisation d'une priorité entre les agents est indispensable si l'on souhaite éviter de manière simple des problèmes de boucles infinies. Afin d'éviter ces problèmes, nous utilisons l'ordre total suivant entre deux agents :

Hypothèse 2 *Un ordre total, noté $\succ$, est établi entre les différents agents. Lorsque $A_i \succ A_j$, cela signifie que A_i a une priorité supérieure à celle de l'agent A_j.*

Dans la suite, nous définissons également $A_i \succeq A_j$ pour $A_i \succ A_j$ ou $A_i = A_j$.

Nous reprenons une hypothèse commune à tous les algorithmes de résolution de DisCSP que nous connaissons dans la littérature.

Hypothèse 3 *Les messages sont reçus dans l'ordre dans lesquels ils ont été envoyés.*

Lors de la résolution d'un DisCSP, chaque agent travaille de manière asynchrone. *DBS* doit parcourir l'arbre de recherche jusqu'à ce qu'une solution soit trouvée. Lors de la recherche, des agents peuvent recevoir, par exemple, des messages de backtrack portant sur des valeurs obsolètes. Afin de permettre à un agent de déterminer si un message est obsolète ou non, nous utilisons la notion de session.

Définition 20 *Une* ***session de travail*** *[Doniec* et al.*, 2005] entre un agent A_i et $\Gamma^+(A_i)$ est un entier indiquant pour chaque élément de $\Gamma^+(A_i)$ l'état de la recherche globale du point de vue de A_i.*

Durant la phase d'initialisation de *DBS*, la session de travail de chaque agent est fixée à 0. Lorsque l'agent A_i reçoit un message de backtrack, noté M_k, ce dernier ne sera traité que si le numéro de session joint à M_k est identique à celui de A_i. Si ce n'est pas le cas, M_k est considéré comme obsolète. Lors de la réception d'un message de type ⟨OK?⟩ (concernant la valeur et la session d'un agent de plus haute priorité), l'agent A_i ferme sa session et incrémente son numéro de session.

Le concept de *sessions de travail* de [Doniec *et al.*, 2005] est très proche de celui des *time-stamp* (appelé aussi *Step-Counter*) utilisé par l'algorithme *AFC* [Meisels et Zivan, 2007] (décrit en section 2.4.3). Dans *AFC*, le *time-stamp* d'un agent A_i n'est pas incrémenté d'une unité à chaque réception d'un message ⟨OK?⟩ comme dans *DBS* mais représente la valeur du plus grand *time-stamp* reçu par A_i. De plus, dans *AFC*, les agents utilisent un message appelé *CPA* pour *Current Partial Assigment* permettant d'échanger une affectation partielle pour l'ensemble des variables du DisCSP de manière synchrone [Meisels et Zivan, 2007]. Contrairement à *DBS* où l'incrémentation de la session d'un agent se fait lorsque ce dernier reçoit un message ⟨OK?⟩, dans *AFC* l'incrémentation du *time-stamp* se fait lorsqu'un agent attribue le *CPA*.

L'algorithme *DBS* utilise trois types de messages :

- Les messages $\langle STOP \rangle$. L'absence de solution au DisCSP sera nécessairement détectée par un agent qui enverra un message $\langle STOP \rangle$ terminant l'algorithme. Si une solution existe, *DBS* la trouvera et entrera dans un état stable : il n'y aura plus d'envoi de message. Dans ce cas, la solution globale est construite à partir des solutions locales de chaque agent.

- Les messages $\langle OK?, (A_i, sol, s_i) \rangle$ permettent à un agent A_i d'envoyer sa solution courante ($sol \in \text{dom}(A_i)$) aux agents de $\Gamma^+(A_i)$. La session courante de l'agent, s_i, est jointe au message. Chaque agent A_i dispose d'un ensemble nommé $agentView(A_i)$ contenant des triplets $(variable, solution, session)$. Lorsque A_i reçoit un message $\langle OK?, (A_i, sol, s_i) \rangle$, il ajoute à $agentView(A_i)$ le triplet (A_i, sol, s_i) contenu dans le message.

- Les messages $\langle BT, (A_j, sol, s_j), listeBt \rangle$ permettent à un agent A_i de demander un retour arrière à un agent A_j de $\Gamma^-(A_i)$. Cette demande de backtrack porte sur la solution *sol* de l'agent A_j durant sa session s_j. Un message BT est traité par A_j uniquement si le numéro de session attaché au message est égal au numéro de session de A_j et si A_j n'a reçu aucune autre demande de backtrack portant sur la même valeur dans sa session courante. $listeBt$ est un ensemble de triplets $(agent, solution, session)$ contenant les triplets reçus des agents A_i de priorité supérieure ($A_i \succ A_j$). Si A_j ne peut trouver de solution locale consistante en accord avec son $agentView$, les triplets contenus dans $listeBt$ permettent à

A_j de poursuivre éventuellement le backtracking vers un agent contenu dans cette liste.

Dans la suite de ce chapitre, nous aurons besoin de comparer des tuples de la forme (*agent*, $(X = v)$, *session*) et de combiner ces tuples. Afin de comparer des tuples t et t', nous introduisons la notation suivante, $t \succeq t'$ tenant compte de la priorité de l'agent inclus dans le tuple (si le tuple porte sur deux agents de priorité identique, alors la comparaison porte sur la priorité de la variable) :

Définition 21 ($\succeq$) $(A_1, (X_1 = v_1), s_1) \succeq (A_2, (X_2 = v_2), s_2) \iff (A_1 \succ A_2) \vee ((A_1 = A_2) \wedge (X_1 \succeq X_2))$.

Nous aurons aussi besoin pour ce chapitre d'utiliser un opérateur de projection $\downarrow$ dont la définition est la suivante :

Définition 22 ($\downarrow$) *Soit une contrainte C et un ensemble de couples (X_i, v_i) nommé S. Nous définissons $S \downarrow_{(C)}$ comme étant la projection de S sur* var(C) *telle que :*
$S \downarrow_{(C)} = \{(X_i, v_i) \in S | X_i \in var(C)\}$.

Afin de combiner des ensembles de la forme (*agent*, $(X = v)$, *session*), nous définissons l'opérateur $\uplus$:

Définition 23 ($\uplus$) *Soit T_1 et T_2 deux ensembles de tuples de la forme $(A, (X = v), s)$, où A est un agent, X est une variable, v est une valeur, s est une session et « _ » dénote toutes les valeurs ou sessions possibles, nous définissons :* $T_1 \uplus T_2 = T_1 \cup \{(A, (X = v), s) \in T_2 \mid \nexists (A, (X = \_), \_) \in T_1\}$.

Après avoir détaillé les principes généraux de l'algorithme *DBS*, nous allons nous intéresser aux structures de données utilisées au niveau de chaque agent.

3.2.2 Structures de données

Dans cette section, nous allons présenter les différentes structures de données permettant de représenter le contexte d'un agent. Le contexte d'un agent, nommé A_i est défini par :

- Le domaine dom(A_i) contenant les solutions locales de A_i.
- La session courante de A_i.
- Un ensemble, appelé *propose* permettant de connaître, pour chaque solution $sol \in$ dom(A_i), si sol a déjà été transmise à $\Gamma^+(A_i)$ durant la session courante de A_i
- Un ensemble appelé $agentView(A_i)$ contenant des triplets (A_i, sol_i, s_i) utilisés pour déterminer la solution sol_i et la session s_i reçus des agents $A_i \in \Gamma^-(A_i)$.
- Un ensemble de solutions, appelé $valBtRecues(A_i)$, permettant de connaître, dans la session courante de A_i, les variables des solutions de dom(A_i) ayant déjà reçues une demande de backtrack.
- Un ensemble nommé $listeBtTotal(A_i)$ contenant des triplets (A_i, sol_i, s_i). Lorsque A_i doit envoyer une demande de backtrack, cet ensemble permet à A_i de transmettre une demande de backtrack adressée à un agent de $listeBtTotal(A_i)$ même si cet agent n'appartient pas à $\Gamma^-(A_i)$.
- Une liste nommée $inconsAV[i]$ composée de triplets $(agent, valeur, session)$. Ces triplets sont ceux de l'$agentView(A_i)$ pour lesquels la solution $D[i]$ est inconsistante. Si $inconsAV[i]$ est vide, cela signifie que $D[i]$ est consistante avec $agentView(A_i)$.

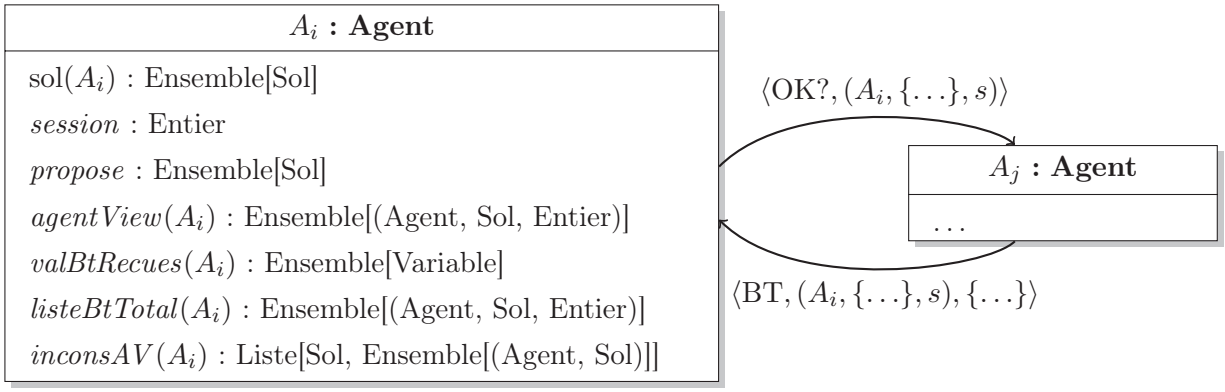

FIGURE 3.1 – Résumé des messages de DBS et des structures des données.

La figure 3.1 est une synthèse du contexte d'un agent et de l'ensemble des messages pouvant être échangés. Le contexte d'un agent évolue au cours de la résolution du DisCSP. Nous verrons dans la section suivante la façon dont le contexte évolue au fur et à mesure des messages reçus.

3.3 Détails de l'algorithme DBS

Nous allons détailler, dans cette section, le comportement de chaque agent exécutant l'algorithme *DBS*. Nous verrons pour commencer la méthode utilisée pour la résolution du CSP local affecté à chaque agent (section 3.3.1). Ensuite, le choix de la solution locale en fonction du contexte d'un agent sera exposé et le choix des données à communiquer entre les différents agents sera spécifié. Puis, nous détaillerons le pseudo-code de l'algorithme *DBS*. Nous terminerons avec un exemple simple permettant d'illustrer les différents messages pouvant être échangés entre les agents.

3.3.1 Résolution du CSP local

Durant la phase de recherche globale de solutions par *DBS*, les différents agents ne communiquent que la valeur de leurs variables d'interface. En effet, seules ces variables d'interface ont un effet sur les autres agents car les autres variables ne jouent aucun rôle pour les contraintes inter-agents. Donc, nous pouvons définir la propriété suivante lorsqu'on ne s'intéresse qu'aux solutions locales différentes au niveau des variables d'interface :

Propriété 1 *Lorsque le solveur local de chaque agent trouve une nouvelle solution locale, nommée SL, il ajoute le nogood $\neg(SL\downarrow_{NV})$ [2] afin de ne pas générer d'autres solutions locales ayant les mêmes valeurs des variables d'interface NV que celles rapportées dans la solution LS.*

Nous avons décidé d'utiliser un solveur local de CSP nommé CSP4J [Vion, 2007] permettant d'ajouter simplement au cours de la recherche de solutions locales des nogoods. Chaque agent résout son CSP local (composé par ses variables et ses contraintes intra-agent) et sauvegarde les solutions dans un ensemble nommé *listeSolutionComplete*. À partir de cet ensemble de solutions, nous créons un nouvel ensemble nommé *domaine* où chaque solution $sol \in domaine$ ne contient que les couples (variable, valeur) des variables d'interface comme le montre la figure 3.2.

2. Par exemple, si $SL = \{(X_0, 0), (X_1, 1), (X_2, 4)\}$ et $NV = \{X_0, X_2\}$, alors $SL\downarrow_{NV} = \{(X_0, 0), (X_2, 4)\}$.

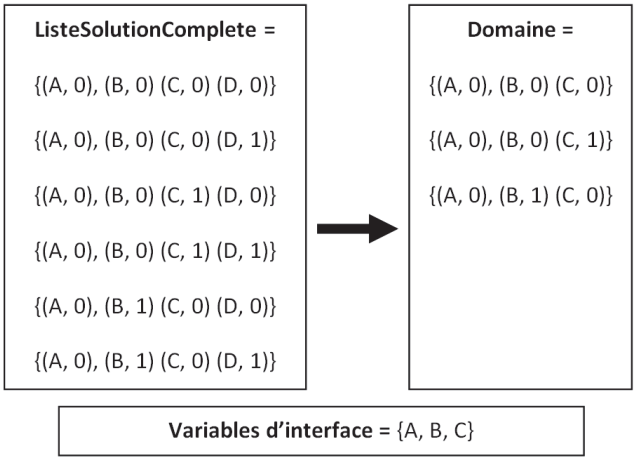

FIGURE 3.2 – Transformation des solutions locales.

Il existe trois différentes méthodes permettant d'effectuer la recherche locale et globale :

- Chaque agent trouve la totalité de ses solutions locales avant de commencer la recherche globale. Ainsi lorsque la recherche globale commence, la liste de solutions locales nommée *domaine* n'est plus modifiée. Il s'agit d'une méthode simple à mettre en œuvre mais qui a l'inconvénient d'être très coûteuse en terme d'espace mémoire lorsque le nombre de solutions locales est important.

- Une seconde méthode consiste à ne sauvegarder aucune solution locale mais à les calculer à chaque fois qu'un agent en a besoin en ajoutant au solveur les contraintes inter-agents que l'agent doit respecter. Cette méthode est intéressante lorsqu'il n'est pas possible, pour des raisons d'espace mémoire, de sauvegarder la totalité des solutions si le nombre de solutions locales est trop important. Cependant, du point de vue de la recherche globale, il est très coûteux de recalculer en permanence à la volée de nouvelles solutions locales.

- La recherche locale et la recherche globale peuvent être exécutées en parallèle. Pour cela, dès qu'une première solution locale est trouvée, l'agent la communique afin de commencer la recherche globale de solutions. À partir de ce moment, la recherche globale s'effectue en parallèle de la recherche locale. Cette méthode a l'avantage d'utiliser les ressources de l'agent pendant qu'il est en attente de messages. Au fur et à mesure que la recherche locale avance, la liste de solutions locales nommée *domaine* augmente. Cependant, si l'agent doit envoyer une demande de backtrack signifiant qu'il n'a trouvé aucune solution locale satisfaisant ses contraintes inter-agents, il doit être certain qu'aucune autre solution consistante non encore communiquée par le solveur local n'existe. L'agent doit donc attendre que la recherche locale soit terminée avant d'envoyer une demande de backtrack.

Nous avons choisi d'utiliser la dernière méthode de recherche de solutions locales qui nous paraît la plus intéressante afin d'optimiser le temps total de résolution de DisCSP. En effet, lorsqu'un agent A_i trouve une solution à son CSP local, il la communique aux agents de $\Gamma^+(A_i)$ puis se met en attente de messages. L'agent A_i peut anticiper un éventuel échec de sa solution courante. Pour cela, pendant l'attente de nouveaux messages, A_i poursuit la recherche de nouvelles solutions pour son CSP local. Ces solutions devront proposer une alternative au niveau des

variables d'interface de l'agent A_i. Cependant, nous pensons que lorsque le CSP local de certains agents comporte un nombre important de solutions, la seconde méthode consistant à calculer les solutions à la volée est préférable. Nous allons maintenant observer la manière dont un agent sélectionne une solution locale en accord avec son contexte.

3.3.2 Choix d'une solution locale

Lorsqu'un agent A_i reçoit une demande de backtrack portant sur une partie de sa solution courante i.e. $\{(X_k = 0)\}$, il faut que A_i évite de proposer, durant sa session courante, une autre solution contenant ces couples. Pour cela, toutes les solutions de son domaine dom(A_i) contenant $\{(X_k = 0)\}$ vont être ajoutées à la liste *propose*. Ainsi, A_i ne pourra pas proposer durant sa session courante une autre solution contenant ce couple. Il s'agit d'une méthode développée initialement dans l'algorithme *OGDiBT* [Belaissaoui et Bouyakhf, 2004].

Lorsque l'ensemble nommé $agentView(A_i)$ est modifié, A_i doit trouver une solution consistante avec tous les triplets de $agentView(A_i)$. La méthode la plus simple consiste à tester chaque solution itérativement jusqu'à l'obtention d'une solution consistante. Lorsque A_i ferme sa session courante (Cf. procédure **fermerSession()** en section 3.3.5), les calculs effectués pour connaître les solutions consistantes sont perdus. Pour contrer ce problème, à chaque solution $D[i]$ du domaine dom(A_i) d'un agent est associé une liste nommée $inconsAV[i]$ de triplets $(agent, valeur, session)$. Il s'agit des triplets de $agentView(A_i)$ pour lesquels la solution $D[i]$ est inconsistante. Si cette liste est vide, cela signifie que $D[i]$ est consistante avec $agentView(A_i)$.

Par exemple, si pour un agent A_i, $D[k] \notin propose$ et $inconsAV[k] = \{(A_0, X_0, 0), (A_1, X_2, 0)\}$, cela signifie que la solution $D[k]$ n'a pas été proposée lors de la session courante et qu'elle n'est pas consistante avec $agentView(A_i)$ (car $inconsAV[k] \neq \{\}$). Lorsqu'un agent utilise la procédure **fermerSession()** (suite à la réception de la nouvelle valeur d'un agent), la liste *propose* est vidée contrairement à la liste $inconsAV[i]$ qui est simplement modifiée de manière à ne contenir que les triplets de l'$agentView$ inconsistants avec la solution $D[i]$. Cette méthode évite un nombre important de vérifications de contraintes mais nécessite, en contrepartie, un certain temps de calcul afin de maintenir à jour les listes $inconsAV[i]$. Pour optimiser ce travail, nous avons repris [Monier *et al.*, 2010a] une méthode présentée dans [Ezzahir, 2008]. Cette méthode porte sur la notion d'interchangeabilité partielle de voisinage utilisant l'opérateur de projection $\downarrow$ (Cf. déf. 22, section 3.2.1) dont la définition est la suivante :

Définition 24 (Interchangeabilité Partielle de Voisinage - VPI)
Deux solutions S_i et S_j locales à un agent sont dites Voisines Partiellement Interchangeables (VPI) par rapport à une contrainte inter-agents donnée C et appartiennent au même ensemble VPI, si $(S_i \downarrow_{(C)}) = (S_j \downarrow_{(C)})$. Lorsque $S_i \in VPI$ viole la contrainte C, alors toutes les solutions de VPI la violent aussi.

Pour chaque contrainte inter-agents C_i, l'ensemble des solutions locales voisines partiellement interchangeables vis à vis de cette contrainte, que nous nommons $C_i.VPI_{set}$, est calculé : $C_i.VPI_{set} = \{VPI_0, VPI_1, ..., VPI_k\}$. Lorsqu'un agent veut vérifier la contrainte C_i, il ne considère que le premier élément de chaque ensemble $C_i.VPI_j$ (avec $0 < j < k$). Si cet élément viole la contrainte C_i, alors toutes les solutions locales contenues dans $C_i.VPI_j$ la violent aussi.

Sur l'exemple de la figure 3.3, l'agent A_2 possède trois variables : X_3, X_4 et X_5. Les contraintes inter-agents C_0, C_1 et C_2 le relient aux agents A_0 et A_1. Pour chaque $C_i \in C_{inter}$, un ensemble de VPI est créé.

Par exemple, $C_0.VPI(A_2) = \{\{S_0, S_1, S_2, S_3\}, \{S_4\}\}$ signifie que si S_0 viole la contrainte C_0 alors les solutions S_1, S_2 et S_3 la violent aussi. Afin de trouver une solution consistante

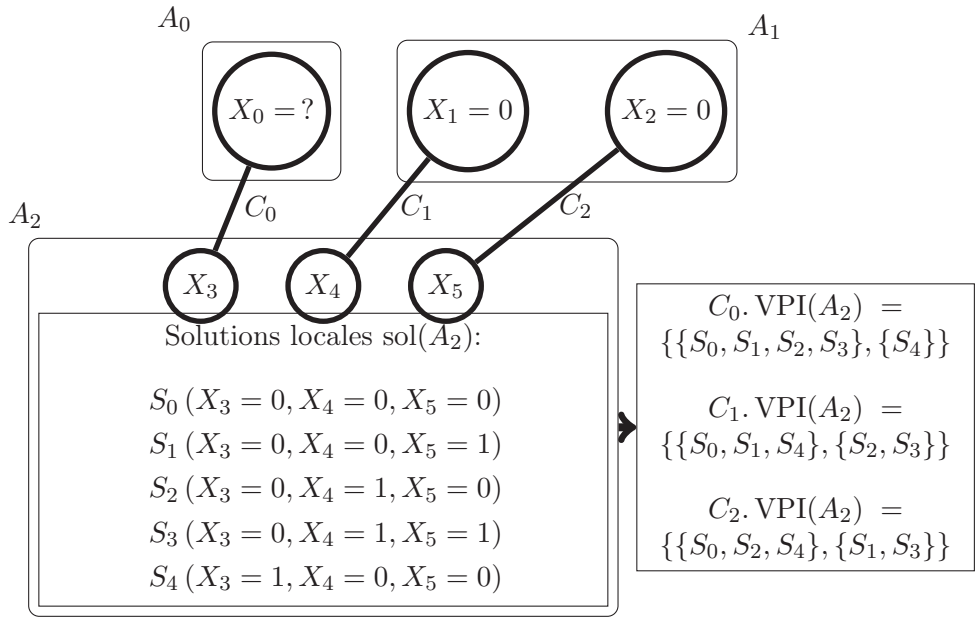

FIGURE 3.3 – Exemple d'utilisation des VPI.

avec $agentView(A_2)$, A_2 vérifie que la solution S_0 respecte toutes les contraintes. A_2 effectue successivement les opérations suivantes permettant de vérifier si toutes ses contraintes inter-agents sont satisfaites :

- Pour chaque contrainte
  - $C_0 : (X_0 = ?)_{A_0} \neq (X_3 = ?)_{A_2}$
    - La contrainte est toujours satisfaite car A_2 ne connaît pas la valeur de la variable X_0 de l'agent A_0.
  - $C_1 : (X_1 = 0)_{A_1} \neq (X_4 = ?)_{A_2}$
    - A_2 vérifie le premier ensemble de $C_1.VPI_{set}$: S_0 viole la contrainte donc S_1 et S_4 aussi.
    - A_2 vérifie le second ensemble de $C_1.VPI_{set}$: S_2 satisfait la contrainte donc S_3 aussi.
    - A_2 met à jour $C_1.VPI_{set}$: il supprime S_0, S_1 et S_4.
  - $C_2 : (X_2 = 0)_{A_1} \neq (X_5 = ?)_{A_2}$
    - A_2 vérifie le premier ensemble de $C_2.VPI_{set}(0)$: S_0 a été supprimé, S_2 viole la contrainte.
    - A_2 vérifie le second ensemble de $C_2.VPI_{set}(1)$: S_1 a été supprimé, S_3 satisfait la contrainte.
- La boucle sur les contraintes est terminée, la solution S_3 n'a pas été éliminée donc elle est la seule solution consistante.

Grâce à l'utilisation des VPI, la mise à jour des listes $inconsAV[i]$ nécessite moins de vérifications de contraintes (Cf. chapitre 4). L'utilisation des VPI est possible car chaque agent sauvegarde l'ensemble des solutions de son CSP local.

Si le nombre de solutions locales est trop important et qu'il n'est pas possible de toutes les sauvegarder, *DBS* devra calculer les solutions à la volée. Dans ce cas, un solveur local tel que CSP4J [Vion, 2007], permettant d'émuler naturellement les VPI via des nogoods locaux ajoutés après chaque solution, peut être utilisé.

Nous allons maintenant observer le comportement de *DBS* lors de la communication de

solutions locales d'un agent à un autre.

3.3.3 Communication d'une solution locale

Nous présentons, dans cette section, les informations échangées par les agents lorsqu'ils souhaitent communiquer leurs nouvelles solutions courantes. Lorsqu'un agent A_i veut communiquer sa solution courante, il est inutile qu'il la transmette entièrement à tous les agents de priorité inférieure reliés par une contrainte C_{inter}. En effet, ces agents n'ont pas besoin de connaître la valeur de toutes les variables d'interface de A_i. Il suffit que A_i communique une partie réduite de sa solution locale courante. La partie de la solution qu'il est utile de communiquer est celle qui respecte la propriété suivante :

Propriété 2 *Soient deux agents A et A', A notifie A' d'un changement d'instanciation $(\dots, X = a, \dots)$ vers $(\dots, X = b, \dots)$ si et seulement si $A \succ A'$ et $\exists X' \in \text{var}(A') \mid \exists C \in \mathscr{C}_{inter} \mid \text{var}(C) = \{X, X'\}$.*

Grâce à cette propriété, les agents ne communiquent à leurs accointances des variables autres que les variables d'interface. Du point de vue de la recherche globale, seules les variables d'interface ont de l'importance pour les agents.

Nous pouvons observer, en figure 3.4, un exemple de DisCSP comportant trois agents : A_0, A_1 et A_2. Supposons que l'agent A_0 souhaite communiquer sa solution courante qui est $\{(X_0 = 0), (X_1 = 0), (X_2 = 0)\}$. En accord avec la propriété 2, l'agent A_0 communique sa solution courante aux agents A_1 et A_2 grâce aux deux messages suivants :

- $\langle OK?, (A_0, \{(X_0 = 0), (X_1 = 0)\}, session = 0)\rangle$ à A_1.
- $\langle OK?, (A_0, \{(X_2 = 0)\}, session = 0)\rangle$ à A_2.

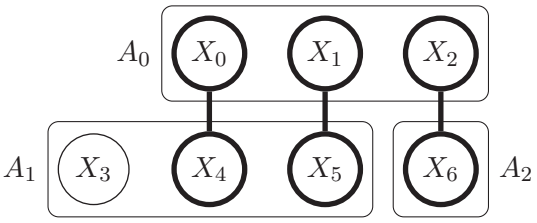

FIGURE 3.4 – Exemple de DisCSP.

Maintenant que le fonctionnement de *DBS* a été spécifié, nous allons nous intéresser à un agent que nous avons ajouté, que nous appelons ici agent coordinateur, permettant d'initialiser les agents de *DBS*, de leur affecter leurs sous-problèmes et de détecter la terminaison de l'algorithme.

3.3.4 L'agent coordinateur

Cette section présente l'agent coordinateur. Comme l'indique la figure 3.5, cet agent intervient durant les phases d'initialisation, de résolution et durant l'affichage de la solution globale.

La première phase consiste à initialiser les agents exécutant *DBS*. Pour cela, l'agent coordinateur lit le problème à résoudre depuis, par exemple, un fichier XML obtenu grâce à un générateur aléatoire de DisCSP. Nous supposons que le problème est déjà formalisé sous la forme d'un DisCSP. Ensuite, l'agent superviseur crée les différents agents qui vont exécuter l'algorithme *DBS*. Il attribue à chacun d'entre eux un CSP composé de variables et de contraintes intra-agent.

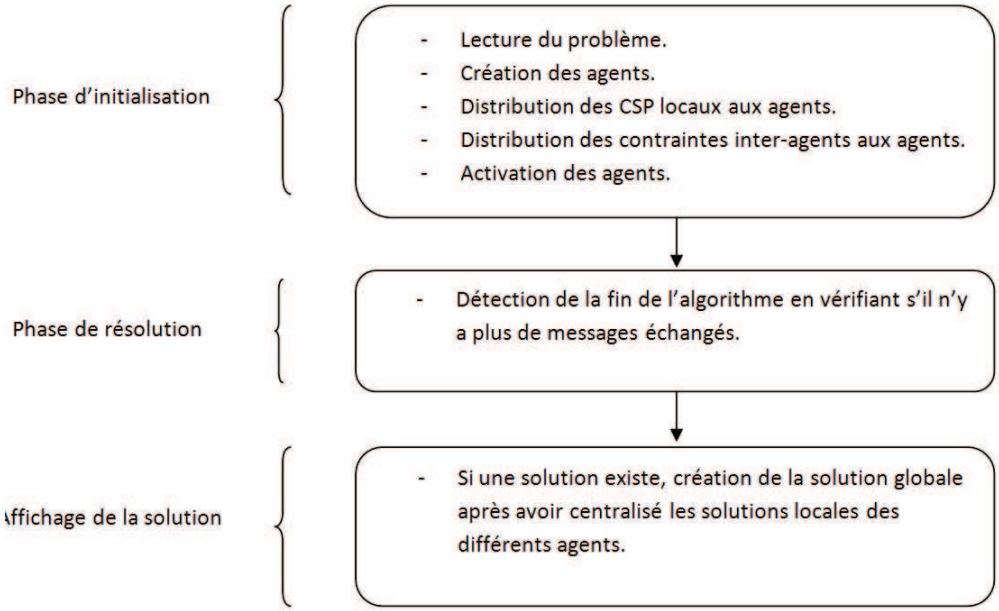

FIGURE 3.5 – Agent coordinateur

Puis il distribue les contraintes inter-agents. Pour cela, il communique chaque contrainte inter-agents binaire à l'agent de plus petite priorité concerné par cette contrainte. Enfin, il active les agents en leur demandant d'exécuter la procédure nommée *dbs* présentée dans la section suivante.

La seconde phase, pour l'agent coordinateur, consiste à détecter la fin de l'algorithme. Rappelons que *DBS* termine de deux manières différentes. Si aucune solution n'existe, un agent enverra un message de type $\langle Fin \rangle$ et si une solution existe, il n'y aura plus de messages échangés. Si l'agent coordinateur reçoit un message $\langle Fin \rangle$, il passe directement à la troisième phase. Sinon, l'agent coordinateur observe les canaux de communication entre les agents. Lorsqu'il voit qu'il n'y a plus de messages échangés entre les agents (Cf. section 2.2), il demande aux agents exécutant *DBS* de s'arrêter et passe ensuite à la troisième phase.

La troisième phase permet à l'agent coordinateur de créer la solution globale du DisCSP si les agents se trouvent dans un état stable. Dans ce cas, il récupère toutes les solutions locales de chaque agent contenant à la fois les variables d'interface et les variables locales. Ensuite il considère l'ensemble des solutions locales pour obtenir la solution globale.

Dans la section suivante, nous allons détailler précisément le pseudo-code de *DBS*.

3.3.5 Pseudo-code

Cette section détaille l'algorithme *DBS*. Celui-ci est composé de deux écouteurs permettant de réagir à la réception des messages $\langle OK? \rangle$ et $\langle BT \rangle$, de quatre procédures et d'une fonction qui communiquent en suivant les flux de données présentés en figure 3.6.

Le point de départ de notre proposition est le suivant. Chaque agent exécute la procédure *DBS*. Pour commencer, les agents lisent le problème qui leur a été affecté (ligne 1). Chaque agent connaît son CSP local composé de différentes variables et contraintes intra-agent. De plus, les agents disposent des contraintes inter-agents portant sur l'une de leurs variables et reliées à des agents de priorité supérieure. Puis, chaque agent crée les ensembles Acc^+ et Acc^- représentant respectivement les accointances inférieures et supérieures (ligne 2). Les agents font ensuite la

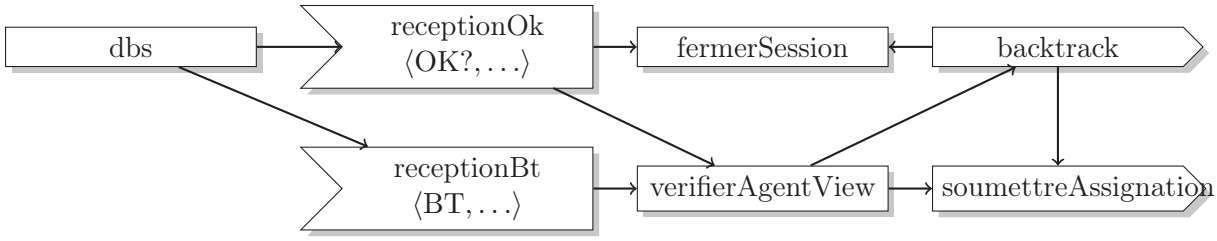

FIGURE 3.6 – Ecouteurs, procédures et fonctions de *DBS*.

différence entre les variables locales et les variables d'interface afin de ne pas transmettre, par la suite, des valeurs des variables locales (ligne 3). La recherche de solutions locales peut débuter (ligne 4) comme nous l'avons présentée dans la section 1.3.3 et dès qu'une première solution est trouvée, elle est communiquée aux agents de Acc^+ (ligne 5). Lorsque l'initialisation de *DBS* est terminée, les agents se mettent en attente de messages. Dès qu'un message $\langle OK?\rangle$ est reçu, l'écouteur receptionOk(message) est appelé (ligne 9). De même, l'écouteur receptionBt(message) est appelé dès qu'un message $\langle BT\rangle$ est reçu (ligne 10). L'algorithme s'achève lorsqu'un message $\langle Fin\rangle$ est reçu (ligne 11).

Algorithm 9: dbs()

```
1  Lecture du problème;
2  Création de Acc+ et Acc−;
3  Création des variables locales et d'interface;
4  Lancement de la recherche locale de solutions;
5  Communication de la première solution locale trouvée;
6  tant que vrai faire
7      msg ← premier message de la boîte de réception ;
8      suivant msg.type faire
9          cas où ⟨OK?⟩ receptionOk(msg);
10         cas où ⟨BT⟩ receptionBT(msg);
11         cas où ⟨Fin⟩ terminer dbs;
```

L'écouteur **receptionOk** ⟨OK?⟩ est appelé lorsqu'un message de type $\langle OK?\rangle$ est reçu. Ces messages sont de la forme $\langle \text{OK?}, (A, sol, s)\rangle$ où A est l'agent émetteur du message, sol représente l'ensemble des variables d'interface de l'agent A qu'il a souhaité communiquer à l'agent $self$ et s correspond à la session courante de l'agent A lorsqu'il a émis le message. L'ensemble $agentView$ de $self$ est alors mis à jour avec les nouvelles valeurs des variables qu'il vient de recevoir (ligne 1). Ensuite, comme nous l'avons décrit dans la section 3.1.1, la session de l'agent se ferme (ligne 2). Enfin, $self$ vérifie si une solution locale consistante vis-à-vis de la nouvelle $agentView$ est possible grâce à l'appel à la procédure $checkAgentView$ (ligne 3).

Écouteur 10: receptionOk $\langle \text{OK?}, (A, sol, s)\rangle$

```
1  mettre à jour agentView avec (A, sol, s);
2  fermerSession();
3  verifierAgentView(A, OK?);
```

L'écouteur **receptionBt** $\langle \text{BT} \rangle$ est appelé lorsqu'un message $\langle \text{BT}, (A, (X = v), s), listeBt \rangle$ est reçu dans lequel A correspond à l'agent $self$, $(X = v)$ correspond à l'affectation de la valeur v à la variable X de l'agent $self$, s représente une session et *listeBt* représente une liste de triplets $(agent, solution, session)$. Pour commencer, l'agent vérifie si la demande de backtrack reçue doit être traitée (ligne 1). Si le numéro de session joint au message est différent du numéro de session de l'agent $self$, alors la demande de backtrack est considérée comme obsolète et n'est pas traitée. Ensuite, les ensembles *valBtRecues*, *listeBtTotal*, *propose* et *currentSol* permettant respectivement de connaître les couples (variable,valeur) ayant reçus une demande de backtrack, les triplets $(agent, solution, session)$ de *listeBt* déjà reçus, les solutions pouvant être proposées et la solution courante sont mis à jour (lignes 2 à 6). Puis l'agent A_i vérifie qu'une valeur consistante avec *agentView* existe.

Écouteur 11: receptionBt $\langle \text{BT}, (A, (X = v), s), listeBt \rangle$

```
1 si s = currentSession ∧ (X = v) ∉ valBtRecues alors
2     valBtRecues ← valBtRecues ∪ {(X = v)} ;
3     listeBtTotal ← listeBtTotal ⊎ listeBt ;
4     si multiVariables alors
5         propose ← propose ∪ {sol ∈ sol(A_i) | (X = v) ∈ sol} ;
6     si (X = v) ∈ currentSol alors
7         currentSol ← null ;
8     verifierAgentView(null, BT);
```

La procédure **fermerSession()** est utilisée pour fermer la session de A_i. Sa valeur courante (i.e solution) est mise à **null** (ligne 1) et son numéro de session est incrémenté (ligne 2). Puis, l'ensemble *valBtRecues* contenant des couples $(variable, valeur)$ qui ont déjà reçu des demandes de backtrack est réinitialisé (ligne 3). Comme A_i se trouve dans une nouvelle session, l'ensemble des valeurs proposées dans la session courante est réinitialisé (ligne 4). Comme $agentView(A_i)$ a été modifié avant que la session ait été fermée, l'ensemble *inconsAV* est mis à jour (ligne 7).

Procedure fermerSession

```
1 solCourante ← null ;
2 sessionCourante ← sessionCourante + 1 ;
3 valBtRecues ← {};
4 propose ← {} ;
5 si multiVariables alors
6     pour chaque sol ∈ sol(A_i) faire
7         mettre à jour inconsAV[sol] ;
```

La fonction **soumettreAssignation()** permet de sélectionner une solution dans la liste des solutions locales et de la communiquer aux accointances inférieures. Pour cela, la première étape (ligne 1) consiste à sélectionner une solution consistante vis à vis de l'ensemble *agentView* et n'ayant pas encore été communiquée dans la session courante. Si aucune solution n'est trouvée (ligne 2), la fonction retourne alors $faux$ (ligne 3). Si une solution est trouvée, la solution courante devient alors la solution venant d'être choisie (ligne 4) et nous ajoutons à la liste contenant les solutions choisies nommée *propose* la solution *sol* (ligne 5). Puis, les lignes 6 à 11 permettent de

communiquer la solution. Si nous sommes dans le cas où chaque agent dispose d'un problème local complexe (ligne 7) alors la solution est communiquée en respectant la propriété 2 de la section 1.3.3 (lignes 8 et 9). Dans le cas où le problème local d'un agent se résume à une unique variable, elle est simplement envoyée aux agents de $\Gamma^+(A_i)$ (ligne 11). Dans ce cas, la fonction retourne alors *vrai* car une nouvelle solution locale a pu être envoyée (ligne 12).

Function soumettreAssignation : boolean

```
1  sélectionner sol ∈ (sol(A_i) − propose) | consistent(sol, agentView) ;
2  si sol est null alors
3  |  return false;
4  SolCourante ← sol ;
5  propose ← propose ∪ {sol} ;
6  foreach A ∈ Γ+(A_i) do
7  |  si multiVariables alors
8  |  |  subset ← {X ∈ var(A_i) | ∃ une contrainte inter-agents entre X et une variable de
   |  |  A};
9  |  |  envoyer ⟨OK?, (A_i, sol↓_subset, sessionCourante)⟩ à A ;
10 |  sinon
11 |  |  envoyer (⟨OK?, (X_{A_i}, solCourante, sessionCourante)⟩) à A
12 return true ;
```

La procédure **verifierAgentView()** est utilisée pour vérifier si une solution est consistante avec l'*agentView* (ligne 1). Si une solution peut être trouvée alors cette procédure permet de l'envoyer en utilisant la fonction *soumettreAssignation* (ligne 3) pour émettre un message ⟨OK?⟩. Si aucune solution ne peut être trouvée alors un message BT (lignes 2 et 4) est émis. La première ligne vérifie si aucune solution n'est possible, c'est-à-dire lorsque toutes les solutions ont déjà été proposées (première partie de la ligne) où si elle sont toutes inconsistantes en ne tenant compte que des tuples de l'*agentView* de priorité supérieure à A_k.

Procedure verifierAgentView(Agent : A_k, MsgType : *type*)

```
1  si type = OK? ∧ (∀sol ∈ sol(A_i), sol ∈ propose ∨ ¬consistent(sol,{(A, _, _) ∈ agentView |
   A ⪰ A_k})) alors
2  |  backtrack(A_k,type) ;
3  sinon si ¬soumettreAssignation() alors
4  |  backtrack(null,type);
```

La procédure **backtrack()** permet d'envoyer une demande de backtrack contenant les informations élaborées dans les lignes 1 à 13. Plusieurs cas sont traités :

- nous sommes dans la procédure backtrack lorsque le paramètre A_k est différent de *null* (ligne 1), ou
- lorsque le type du message est $\langle OK? \rangle$ (ligne 7), ou
- nous sommes dans le cas par défaut (ligne 10).

Quel que soit le cas, deux informations doivent être choisies : le tuple $(agent, (X = v), session)$ à joindre au message ainsi que l'ensemble *listeBt* permettant de poursuivre éventuellement par la suite la demande de backtrack vers un agent contenu dans cet ensemble. Si le tuple choisi

$(agent, (X = v), session)$ est *null* (ligne 14) alors cela signifie que le DisCSP est inconsistant et un message de type $\langle Fin \rangle$ est envoyé (ligne 15) et l'algorithme se termine (ligne 16). Si le tuple est différent de *null*, la demande de backtrack est envoyée (ligne 17).

L'ensemble *listeBtTotale* (ligne 18) est mis à jour. Enfin si le tuple envoyé dans la demande de backtrack appartenait à l'$agentView$ (ligne 19), $agentView$ est mis à jour (ligne 20) sinon si le paramètre *type* vaut BT alors l'agent ferme sa session (ligne 22) puis appelle la fonction *soumettreAssignation* (ligne 23).

Procedure backtrack(Agent : A_k, MsgType : *type*)

```
1  si A_k ≠ null alors
2      si multiVariables alors
3          sélectionner
           (A, (X = v), s) ∈ agentView | A = A_k ∧ (A, (X = _), _) ≽ (A, (X' = _), _);
4      sinon
5          select (A, (X = v), s) ∈ agentView | A = A_k
6      listeBt ← {(A', (X' = v'), s') ∈ agentView ⊎ listeBtTotal | A' ≻ A} ;
7  sinon si type = OK? alors
8      sélectionner (A, (X = v), s) ∈ agentView |
       ∀T ∈ agentView, (A, (X = v), s) ≽ T;
9      listeBt ← {(A', (X' = v'), s') ∈ agentView ⊎ listeBtTotal | A' ≻ A};
10 sinon
11     listeBt ← agentView ⊎ listeBtTotal;
12     sélectionner (A, (X = v), s) ∈ listeBt | ∀T ∈ listeBt, T ≽ (A, (X = v), s);
13     listeBt ← listeBt − {(A, (X = v), s)} ;
14 si (A, (X = v), s) = null alors
15     envoyer un message ⟨STOP⟩;
16     terminer l'algorithme;
17 envoyer ⟨BT, (A, (X = v), s), listeBt⟩ à A;
18 listeBtTotal ← listeBtTotal − listeBt − {(A, (X = v), s)};
19 si (A, (X = _), _) ∈ agentView alors
20     agentView ← agentView − {(A, (X = v), s)} ;
21 sinon si type = BT alors
22     fermerSession() ;
23     soumettreAssignation() ;
```

3.3.6 Illustration du fonctionnement de DBS

La figure 3.7 illustre un problème de coloration de graphe distribué comportant trois agents A_0, A_1 et A_2. Chaque agent comporte trois variables pouvant prendre la couleur *blanc* ou *noir*. Il s'agit de l'exemple avec lequel nous avons présenté le fonctionnement des algorithmes *Multi-ABT*, *Multi-AWC* et *AFC* au chapitre 2. Observons le comportement de l'algorithme *DBS* schématisé dans la figure 3.8. La priorité de chaque agent est attribuée de manière lexicographique : A_0 a la plus haute priorité et A_2 la plus petite.

À l'initialisation de *DBS*, chaque agent résout son CSP local puis choisit une solution (Cf. section 3.3.2). A_0 choisit ainsi la solution $\{(X_0, noir), (X_1, blanc), (X_2, noir)\}$ et la communique

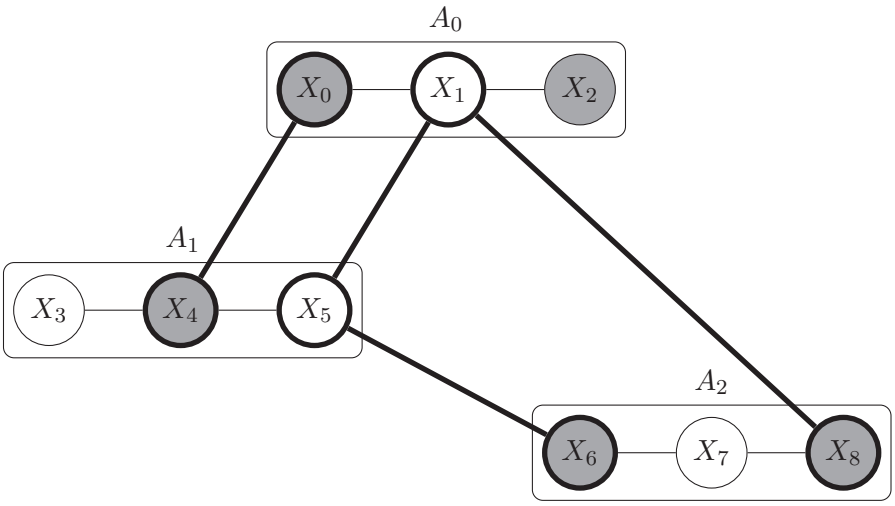

FIGURE 3.7 – Problème de coloration de graphe distribué.

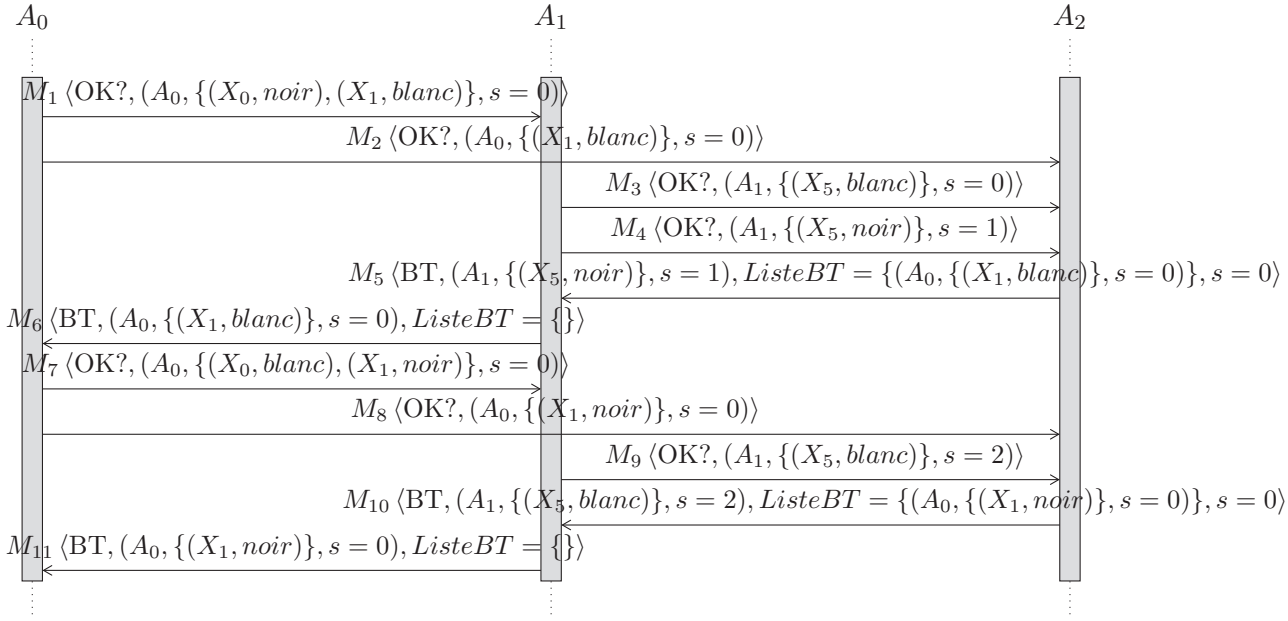

FIGURE 3.8 – Echange de messages entre les agents A_i.

grâce aux messages M_1 et M_2. A_1 choisit la solution $\{(X_3, blanc), (X_4, noir), (X_5, blanc)\}$ et la communique grâce au message M_3. En recevant M_1, A_1 change sa solution locale : $\{(X_3, noir), (X_4, blanc), (X_5, noir)\}$ et envoie M_4.

A_2 reçoit les messages M_2 et M_3. La solution locale courante de A_2 $\{(X_6, noir), (X_7, blanc), (X_8, noir)\}$ est toujours consistante avec son $agentView$ qui vaut $\{(A_0, \{(X_1, blanc)\}, s = 0), (A_1, \{(X_5, blanc)\}, s = 0)\}$. A_2, recevant M_4, met à jour son $agentView$, et ne trouve aucune solution consistante. Il envoie alors une demande de backtrack M_5 à l'agent de plus petite priorité contenue dans son $agentView$: A_1.

A_1 reçoit M_5. Comme il ne peut pas trouver de solution consistante avec son $agentView$ n'ayant pas déjà été proposée dans sa session 1, il poursuit la demande de backtracking vers

l'agent A_0 grâce au message M_6. A_0 reçoit ce message et modifie sa solution courante : $\{(X_0, blanc)$, $(X_1, noir)$, $(X_2, blanc)\}$ puis la communique grâce aux messages M_7 et M_8. A_1 reçoit M_7, ferme sa session (qui vaut alors 2), et change sa solution courante en $\{(X_3, blanc)$, $(X_4, noir)$, $(X_5, blanc)\}$ et la communique grâce au message M_9.

A_2 reçoit les messages M_8 et M_9. Comme il ne trouve pas de solution satisfaisant $agentView = \{(A_0, \{(X_1, noir)\}, s = 0)$, $(A_1, \{(X_0, blanc)\}, s = 2)\}$, il envoie une demande de backtrack M_{10} à A_1. A_1 poursuit cette demande de backtrack vers A_0 grâce au message M_{11}. A_0 reçoit ce message. Comme ses deux solutions locales ont déjà été proposées (car elles appartiennent à la liste *propose*), A_0 affirme qu'il n'y a pas de solution à ce DisCSP et envoie un message $\langle STOP \rangle$ aux agents afin de terminer la recherche.

3.4 Propriétés de DBS

Nous allons démontrer que l'algorithme *DBS* est correct (section 3.4.1), complet et qu'il termine (section 3.4.2). Puis, nous montrerons sa complexité spatiale (section 3.4.3) et temporelle (section 3.4.4). Enfin, différents traitements permettant de supprimer des messages obsolètes sans modifier la preuve de complétude seront présentés dans la section 3.4.5.

3.4.1 Preuve de correction

Théorème 1 DBS *est correct. Il affirme qu'une instanciation est une solution uniquement si toutes les contraintes sont vérifiées et toutes les variables assignées.*

Preuve 1 *Lorsque* DBS *détecte une solution, les agents sont dans un état stable : il n'y a plus d'envoi de message. Cet état est incompatible avec une violation de contrainte qui entraînerait un envoi de message de backtrack.*

3.4.2 Preuve de complétude

Afin de prouver la complétude de *DBS*, nous nous sommes basés sur les preuves de complétude des algorithmes *ABT* [Yokoo, 2001] et *DDB* [Bessière *et al.*, 2001]. La preuve de la complétude de *DBS* est réalisée en trois étapes. La première étape décrit une alternative d'implémentation de l'algorithme *DBS* (similaire à *ABT*) que nous nommons DBS_{all}. Dans DBS_{all}, chaque agent sauvegarde toutes les Combinaisons d'Instanciations Interdites (CII) [3] et chaque agent ne gère qu'une unique variable. Nous allons montrer que DBS_{all} est équivalent à ABT dont la complétude a été prouvée. Puis, dans une seconde étape, nous montrerons qu'il n'est pas nécessaire de stocker toutes les CII au cours de la recherche. Nous verrons alors que *DBS*, dans une version mono-variable par agent, conserve toutes les propriétés portant sur la complétude de DBS_{all}. De ce fait, la preuve de complétude de *DBS* mono-variable par agent sera apportée. Enfin, nous verrons que le fait de gérer plusieurs variables par agent ne remet pas en cause la preuve de complétude.

Preuve de complétude de DBS_{all}

L'algorithme *DBS* sauvegarde temporairement des Instanciations Interdites (II) dans une liste nommée *valBtRecues*. Une II (pour *DBS*) est composée d'une valeur interdite par l'agent A_i. DBS_{all}, quant à lui, stocke les CII dans la liste *listeCII*. Lorsque A_i change de session, cette liste est vidée.

3. Une CII de DBS_{all} est équivalent à un *nogood* de *ABT*.

Pour prouver la complétude de *DBS*, nous allons modifier cet algorithme afin qu'il puisse mémoriser durant toute la résolution les CII. Cela nous permettra d'utiliser la même preuve de complétude que celle utilisée pour ABT [Yokoo, 2001] ou *DDB* [Bessière *et al.*, 2001].

Nous allons maintenant démontrer en quatre étapes que DBS_{all} fonctionne comme ABT.

1 - Sauvegarde des nogoods (CII)

Dans ABT, lorsque A_i reçoit un nogood, il l'ajoute à sa liste de *nogoods*. Dans DBS_{all}, lorsque l'agent A_i reçoit une demande de backtrack $\langle BT, (A_i, ValDefaut, session), listeBt\rangle$, il vérifie $agentView(A_i)$, c'est-à-dire qu'il vérifie que toutes les contraintes portant sur des variables de $agentView(A_i)$ sont satisfaites. Supposons que $agentView(A_i) = \{(X_1, v_1, s_1), ..., (X_k, v_k, s_k)\}$. Contrairement à *DBS* qui ajoute $ValDefaut$ à la liste $valBtRecues$, DBS_{all} complète $ValDefaut$ pour créer une CII. Cette dernière, qui va être stockée dans $listeCII$, est de la forme suivante : $\{(X_1, v_1, s_1)$, ..., (X_k, v_k, s_k), $(X_{A_i}, ValDefaut, session)\}$.

La liste $listeCII$ ne sera jamais vidée. DBS_{all} sauvegarde les CII de la même façon que ABT sauvegarde les *nogoods*.

2 - Contexte d'un message de backtrack

Contrairement à ABT qui joint une liste de *nogoods* à un message de backtrack, DBS_{all} utilise la notion de session. Observons, pour commencer, le comportement d'ABT. Supposons que l'agent A_i reçoive le *nogood* suivant : $\{(X_1, v_1)$, ..., (X_k, v_k), $(X_{A_i}, ValDefaut)\}$

Il y aura un backtrack, pour ABT, si les deux conditions suivantes sont réunies :

1. $ValDefaut$ est égale à la valeur courante de l'agent A_i.
2. pour chaque couple (X_k, v_k) du *nogood*, si X_k appartient à $agentView(A_i)$ alors v_k est égale à la valeur se trouvant dans $agentView(A_i)$.

Observons maintenant le comportement pour DBS_{all} :

Supposons que l'agent A_i reçoive le message $\langle BT, (A_i, ValDefaut, session), listeBt\rangle$. Il prendra en compte cette demande de backtrack si les deux conditions suivantes sont réunies :

1. $ValDefaut$ est égale à la valeur courante de l'agent A_i.
2. le numéro de session attaché au message est égal au numéro de session de l'agent A_i.

Le point 1 pour DBS_{all} est identique au point 1 de ABT. Le point 2 de DBS_{all} est vérifié si l'agent A_i n'a reçu aucun message $\langle OK?\rangle$ entre le moment où il a envoyé sa valeur courante et le moment où il a reçu une demande de backtrack sur cette même valeur (nommée alors $ValDefaut$). Dans ce cas, $agentView(A_i)$ n'aura pas été modifiée (Cela est équivalent au point 2 pour ABT). DBS_{all} utilise donc un contexte pour les messages de backtrack équivalent à celui d'ABT.

3 - Choix du destinataire d'un message de backtrack (suite à la réception d'un message $\langle OK?\rangle$)

ABT et DBS_{all} diffèrent sur le choix du destinataire d'un message de backtrack. Dans ces deux algorithmes, lorsqu'un agent A_i reçoit un message $\langle OK?\rangle$ d'un agent A_k, il ajoute le contenu du message à $agentView(A_i)$. S'il n'y a pas de solution :

1. Dans ABT, A_i construit un *nogood* pour chaque sous-partie d'$agentView(A_i)$ pour lesquelles il n'y a pas de solution. Un message de backtrack est alors envoyé aux agents de plus faible priorité contenu dans ces *nogoods*.
2. Dans DBS_{all}, si A_i ne trouve pas de solution en ne tenant compte, dans $agentView(A_i)$, que des agents de priorité supérieure à A_k (ligne 1 de la procédure *checkAgentView*), le message $\langle BT \rangle$ est envoyé à l'agent A_k. Cependant, si une solution existe, le message de backtrack est adressé à l'agent de plus petite priorité contenu dans $agentView(A_i)$.

ABT envoie directement le message de backtrack à l'agent fautif tandis que DBS_{all} envoie le message de backtrack soit à l'agent fautif, soit à un agent de priorité inférieure à l'agent fautif.

4 - Ajout de liens

L'algorithme ABT ajoute des liens au fur et à mesure de la recherche de manière dynamique afin de poursuivre une demande de backtrack vers un agent n'appartenant pas à $agentView(A_i)$. L'algorithme DBS_{all}, quant à lui, suppose que chaque agent connaît les autres [4].

Dans ABT, le choix du destinataire d'un message $\langle BT \rangle$ suite à la réception d'un message $\langle BT \rangle$ s'établit de la manière suivante. Pour commencer, lorsqu'A_i reçoit un *nogood* contenant un agent A_j qu'il ne connaît pas, A_i ajoute A_j à $agentView(A_i)$ et cherche une solution. S'il n'y a pas de solution, A_i recherche les sous-parties inconsistantes de $agentView(A_i)$, et pour chacune de ces sous-parties, A_i envoie ensuite la demande de backtrack à l'agent de plus petite priorité de $\Gamma^-(A_i)$ et le supprime de $agentView(A_i)$.

Dans DBS_{all}, lorsque A_i reçoit une demande de backtrack dans laquelle $listeBt$ contient un triplet (X_j, v_j, s_j) avec $A_j \notin agentView(A_i)$, A_i n'ajoute pas ce triplet à $agentView(A_i)$ contrairement à ABT. Ensuite, A_i cherche une solution. Le fait de ne pas ajouter A_j à $agentView(A_i)$ revient au même que dans ABT car même si A_i ajoutait le triplet (X_j, v_j, s_j) à $agentView(A_i)$, comme il n'y a pas de contrainte entre A_i et A_j, la recherche de solutions serait identique. Ensuite, A_i cherche une solution et s'il n'en trouve pas, il envoie la demande de backtrack à l'agent de plus faible priorité de $agentView(A_i) \uplus (listeBT_{m_b} \uplus listeBtTotal)$. Cela aboutit au même résultat que pour ABT. Il reste enfin à définir le contenu du message de backtrack. Supposons que le message doive être transmis à A_j. Dans ABT, on joint la valeur v_j de A_j (cette valeur est contenue dans $agentView(A_i)$) au *nogood* à envoyer. Dans DBS_{all}, on met le triplet (X_j, v_j, s_j) contenu dans $agentView(A_i) \uplus (listeBT^* \uplus listeBtTotal)$ où $listeBT^*$ est la $listeBt$ appartenant au message de backtrack reçu par l'agent A_i ayant provoqué l'envoi du message de backtracking vers l'agent A_j. De plus, dans DBS_{all}, l'agent A_i ajoute une liste de triplets (X, v, s) nommée $listeBt$ au message afin de permettre à l'agent A_j de poursuivre la demande de backtrack si nécessaire.

Pour conclure sur la première étape de la preuve de complétude, nous avons proposé une alternative d'implémentation à *DBS*, nommée DBS_{all}, sauvegardant toutes les CII au cours de la recherche. Nous avons montré qu'ABT est identique à DBS_{all} que ce soit au niveau de la sauvegarde des nogoods (des CII pour DBS_{all}), du contexte des messages de backtrack, du choix du destinataire d'un message de backtrack et de l'ajout de liens. Comme l'algorithme ABT est complet [Yokoo, 2001], l'algorithme DBS_{all} l'est aussi. Nous allons maintenant prouver dans la partie suivante que l'algorithme *DBS*, supprimant des instanciations interdites (II) obsolètes, est aussi complet.

4. Dans ABT, comme un agent A_i peut demander un ajout de liens à n'importe quel autre agent, tout se passe comme si A_i connaissait tous les autres agents.

Preuve de complétude de *DBS*

DBS diffère de DBS_{all} sur la sauvegarde des CII. Dans cette section, nous allons prouver que le fait d'éliminer des instanciations interdites obsolètes ne peut pas conduire les agents dans une boucle infinie. Cela permet de prouver la terminaison de *DBS*.

Lemme 1 *Soit A_0 l'agent de plus haute priorité. A_0 ne peut jamais se retrouver dans une boucle infinie même si* DBS *supprime des Instanciations Interdites (II) obsolètes.*

Preuve 2 DBS *réinitialise la liste $valBtRecues$ contenant les II uniquement lorsqu'un agent de $\Gamma^+(A_i)$ envoie un message $\langle OK?\rangle$ à A_i ou lorsque A_i envoie une demande de backtrack à un agent de $\Gamma^+(A_i)$ n'appartenant pas à $agentView(A_i)$. Comme A_0 est l'agent de plus haute priorité, $\Gamma^+_{A_0} = \{\}$. De ce fait, les éléments stockés dans $valBtRecues$ de A_0 ne pouront jamais être supprimés. De plus l'agent A_0 dispose d'un domaine fini, la taille de $valBtRecues$ sera, dans le pire des cas, égale à d (la taille du domaine de la variable de A_0). Comme A_0 ne peut pas réinitialiser $valBtRecues$ et que A_0 ne peut recevoir qu'un nombre fini d de valeurs, il ne peut pas se retrouver dans une boucle infinie.*

Lemme 2 *Si les premiers $(k-1)$ agents, dans l'ordre établi par* DBS*, ne sont pas dans une boucle infinie d'envoi de messages, A_k ne peut pas tomber dans une boucle infinie à cause de la réinitialisation de la liste $BtValRecues$ contenant les Instanciations Interdites (II).*

Preuve 3 *Supposons que A_k se trouve dans une boucle infinie d'envoi de messages. Cela signifie qu'il oublie des II qu'il devrait conserver car ses prédécesseurs modifient la valeur de leur variable. En effet, les II de A_k sont supprimés lorsque :*

- *A_k reçoit un message $\langle OK?\rangle$ d'un agent de $\Gamma^-(A_k)$. La session de A_k est alors incrémentée.*
- *A_k doit envoyer une demande de backtrack à un agent A_i de $\Gamma^-(A_k)$ n'appartenant pas à $agentView_{A_k}$ car A_i a modifié sa valeur. La session de A_k est alors incrémentée.*

Cependant, nous avons supposé que les agents $A_0, ..., A_{k-1}$ ne sont pas en train de « boucler ». Cela signifie qu'ils vont, au bout d'un temps fini, se stabiliser. A_k ne peut donc pas se trouver dans une boucle infinie car il ne recevra plus de messages $\langle OK?\rangle$.

Théorème 2 DBS *est correct, complet et il termine.*

Preuve 4 DBS *hérite des mêmes propriétés que DBS_{all}. Il ne diffère que sur l'élimination des Instanciations Interdites (II) obsolètes. Par récurrence des lemmes 1 et 2, nous pouvons conclure qu'aucun agent ne peut tomber dans une boucle infinie même avec le fait que* DBS *supprime des II qu'il considère comme obsolètes. Nous avons donc prouvé la terminaison de* DBS.

Nous avons décrit une alternative d'implémentation de *DBS* nommé DBS_{all}. Ce dernier est équivalent à *ABT* dont la preuve de complétude a été prouvée [Yokoo, 2001]. Nous en avons donc déduit que DBS_{all} est aussi complet. Puis, nous avons montré que *DBS* conserve les mêmes propriétés que DBS_{all} sur la complétude et qu'il ne peut pas tomber dans des boucles infinies malgré le fait qu'il supprime des instanciations interdites obsolètes.

Passage au multi-variables par agent

Nous allons maintenant montrer que *DBS*, dans sa version multi-variables, est également complet. Pour cela, il suffit de montrer que les modifications dûes au passage de *DBS* mono-variable vers *DBS* multi-variables par agent conservent cette complétude.

DBS ne sauvegarde que les solutions locales différentes au niveau des variables d'interface. Un agent qui souhaite communiquer sa solution locale ne la communiquera pas entièrement à chaque agent (propriété 2). Lorsqu'un agent reçoit une demande de backtrack, l'instanciation des variables d'interface concernées est modifiée. Ces méthodes ne remettent pas en cause la complétude [Belaissaoui et Bouyakhf, 2004].

Afin de rechercher une solution locale consistante, chaque agent utilise la notion de VPI (Cf. déf. 24, section 3.3.2) qui ne remet pas en cause la complétude [Ezzahir, 2008]. *DBS* sauvegarde la cause de l'inconsistance de chaque solution locale. Grâce aux VPI, les solutions sont toujours testées, seule la méthode de vérification de consistance est modifiée. Cela ne remet pas non plus en cause la preuve de complétude. L'algorithme *DBS* est donc **complet**.

3.4.3 Complexité spatiale

Théorème 3 *Chaque agent de* DBS *nécessite une complexité spatiale en* $O(N.d^n)$.

Preuve 5 *L'espace mémoire nécessaire à un agent dépend de la taille de son contexte. La complexité spatiale est bornée par la structure nommée* $inconsAV$. *Cette structure peut contenir au maximum N triplets pour chacune des O(d.n) solutions locales. La complexité spatiale de chaque agent est donc* $O(N.d^n)$.

3.4.4 Complexité temporelle

Théorème 4 *Si nous notons pour un agent A,* e_A *le nombre de contraintes intra-agent,* n_A *le nombre de variables (dont la taille du domaine est inférieure à* D_A*) encapsulées dans A,* e_{inter} *le nombre de contraintes inter-agents alors la complexité temporelle de* DBS *sera dans le pire des cas :*

$$O(\sum_{A \in \mathscr{A}} e_A d_A^{n_A} + e_{inter} d^N)$$

Preuve 6 *Dans le pire des cas, la complexité temporelle est la somme du temps requis pour résoudre le CSP local et du temps requis pour la recherche globale. Nous considérons que les messages sont envoyés et reçus en un temps constant.*

La première étape consiste à trouver toutes les solutions de tous les CSP locaux. Pour un agent donné A, la complexité de cette première étape est $O(e_A d_A^{n_A})$. *La seconde étape consiste à effectuer la recherche globale. Chaque agent A peut disposer de* $O(d^{n_A})$ *solutions locales, et dans le pire des cas, nous devons vérifier les* e_{inter} *contraintes inter-agents pour chacune des* $O(\prod_{A \in \mathscr{A}} d_A^{n_A}) \in O(d^N)$ *combinaisons possibles de solutions locales.*

Rappelons que *DBS* est un algorithme de recherche asynchrone, donc une grande partie du travail peut être effectué en parallèle (et en particulier la recherche de solutions locales).

3.4.5 Filtrage des messages obsolètes

Nous allons expliquer, dans cette section, différentes techniques permettant de supprimer des messages obsolètes sans les traiter (c'est-à-dire sans utiliser les écouteurs « when receive $\langle OK? \rangle$ »

et « when receive $\langle BT \rangle$ »). De plus, nous verrons que ces propriétés ne remettent pas en cause la preuve de complétude. Soit $Self$ un agent générique, nous nommons BR_{A_i} la liste des messages reçus et en attente de traitement.

Nous présentons un exemple, sur le tableau 3.1, de la boîte de réception de l'agent $Self$. Ce dernier traite les messages dans leur ordre d'arrivée, c'est-à-dire, ici, du message 0 au message 7. Rappelons qu'un message de type $\langle OK?\rangle$ est de la forme $\langle OK?, (agent, sol, session)\rangle$ et qu'un message de type $\langle BT\rangle$ est de la forme $\langle BT, (agent, valeur, session), listeBT\rangle$.

TABLE 3.1 – Boîte de réception de l'agent $self$

Position	Message
0	$\langle OK?, (A_1, (X_0 = 1), 1)\rangle$
1	$\langle OK?, (A_2, (X_0 = 1), 1)\rangle$
2	$\langle BT, (self, (X_0 = 2), 1), \emptyset\rangle$
3	$\langle BT, (self, (X_0 = 3), 1), \emptyset\rangle$
4	$\langle OK?, (A_1, (X_0 = 5), 2)\rangle$
5	$\langle OK?, (A_1, (X_0 = 6), 3)\rangle$
6	$\langle OK?, (A_1, (X_0 = 2), 3)\rangle$
7	$\langle BT, (self, (X_0 = 4), 2), \emptyset\rangle$

Filtre 1 *Si A_i dispose, dans BR_{A_i}, de plusieurs messages $\langle OK?\rangle$ d'un agent A_k, seul le dernier message $\langle OK?\rangle$ émis par A_k doit être traité, les autres sont supprimés. Ce filtre, nommé F_1, n'élimine aucune solution au DisCSP.*

Preuve 7 *Comme la session de travail d'un agent ne peut pas diminuer, si un agent A_k envoie plusieurs messages $\langle OK?\rangle$ à un agent A_i, alors le dernier message $\langle OK?\rangle$, nommé m_{last}, de A_k dans BR_{A_i} aura un numéro de session supérieur ou égal à tous les autres messages $\langle OK?\rangle$ émis par A_k.*

Supposons que l'agent A_i dispose dans BR_{A_i} de plusieurs messages $\langle OK?\rangle$ de l'agent A_k. Le dernier message $\langle OK?\rangle$ reçu de A_k est le suivant : $m_{last} = \langle OK?, (A_k, v_k, s_k)\rangle$. Les autres messages $\langle OK?\rangle$ émis par A_k, de la forme $\langle OK?, (A_k, v_i, s_i)\rangle$, diffèrent de m_{last} par v_i et s_i. Quatre cas sont possibles :

1. *($v_i = v_k$ et $s_i = s_k$) : Impossible car un agent ne peut pas soumettre plusieurs fois la même valeur dans une même session.*
2. *($v_i = v_k$ et $s_i < s_k$) : Le contexte d'un message est défini par un numéro de session. Les messages contenant un numéro de session obsolète ne sont pas traités. Ici, si A_i traite le message $\langle OK?\rangle$ contenant s_k, alors les messages $\langle OK?\rangle$ contenant $s_i < s_k$ deviennent obsolètes et peuvent être supprimés.*
3. *($v_i \neq v_k$ et $s_i = s_k$) : Chaque agent teste les valeurs de son domaine dans un ordre défini à l'initialisation de l'algorithme. Cet ordre ne change pas durant la résolution. De ce fait, la valeur v_i de la session s_i est forcément une valeur obsolète à la valeur v_k de la même session[5]. Ces messages sont obsolètes par rapport à m_{last} et peuvent être supprimés. En effet, si A_k a changé sa valeur courante (v_i) pour v_k, c'est parce que A_k a reçu une demande de backtrack portant sur v_i.*
4. *($v_i \neq v_k$ et $s_i < s_k$) : Les messages contenant $s_i < s_k$ peuvent être supprimés. (Cf. deuxième cas).*

5. Ce traitement ne change pas si à chaque fermeture de session, A_i utilise une heuristique de réordonnancement des valeurs telles que celles citées dans [Lecoutre *et al.*, 2007].

En appliquant le filtre F_1, le contenu de BR_{self} devient celui du tableau 3.2.

TABLE 3.2 – Boîte de réception de l'agent $self$

Position	Message
0	$\langle OK?, (A_2, (X_0 = 1), 1)\rangle$
1	$\langle BT, (self, (X_0 = 2), 1), \emptyset\rangle$
2	$\langle BT, (self, (X_0 = 3), 1), \emptyset\rangle$
3	$\langle OK?, (A_1, (X_0 = 2), 3)\rangle$
4	$\langle BT, (self, (X_0 = 4), 2), \emptyset\rangle$

Filtre 2 *Si un agent A_i dispose, dans BR_{A_i}, de plusieurs messages de backtrack et d'au moins un message $\langle OK?\rangle$ alors les messages de backtrack peuvent être supprimés de BR_{A_i}. Ce filtre, nommé F_2, n'élimine aucune solution au DisCSP.*

Preuve 8 *Supposons que A_i traite en premier le message $\langle OK?\rangle$, la session de travail de A_i sera incrémentée. Les messages de backtrack présents dans BR_{A_i} sont forcément obsolètes car ils portent sur une valeur appartenant à une ancienne session de A_i. Etant obsolètes, ils sont supprimés.*

En appliquant les filtres $F_1 + F_2$ le contenu de BR_{self} contenant initialement 8 messages (Cf. tableau 3.1) devient celui du tableau 3.3.

TABLE 3.3 – Boîte de réception de l'agent $self$

Position	Message
0	$\langle OK?, (A_2, (X_0 = 1), 1)\rangle$
1	$\langle OK?, (A_1, (X_0 = 2), 3)\rangle$

Filtre 3 *Si un agent A_i a envoyé une demande de backtrack à un agent A_k appartenant à $agentView(A_i)$, alors tant que A_i n'a pas reçu un message $\langle OK?\rangle$ de A_k, il peut supprimer tous les messages de backtrack contenus dans BR_{A_i}. Ce filtre, nommé F_3, n'élimine aucune solution au DisCSP.*

Preuve 9 *Comme A_i a envoyé un message de backtrack à A_k, il va recevoir obligatoirement au bout d'un temps fini, un message $\langle OK?\rangle$, nommé m_k, de l'agent A_k. Si A_i attend que m_k lui parvienne avant de traiter les messages de backtrack présents dans BR_{A_i}, en appliquant la propriété 2, A_i peut supprimer les messages de backtrack reçus.*

Filtre 4 *Si l'agent A_i reçoit plusieurs messages $\langle OK?\rangle$, alors il traite en priorité ceux venant des agents de priorité plus élevée. Ce filtre, nommé F_4, n'élimine aucune solution au DisCSP.*

Preuve 10 *Aucun message n'est supprimé de BR_{A_i}. Comme* DBS *est complet, il fonctionne quel que soit l'ordre d'arrivée des messages (donc même dans l'ordre obtenu avec la propriété 4). Cependant, l'hypothèse suivante (Cf. section 3.2) doit être respectée : « Pour un couple d'agents donné, les messages sont reçus dans l'ordre dans lesquels ils ont été envoyés. »*

Nous avons prouvé que *DBS* est complet, correct et qu'il se termine. Après avoir donné la complexité spatiale de chaque agent ainsi que la complexité temporelle de *DBS*, nous avons exposé des propriétés permettant de supprimer des messages obsolètes sans remettre en cause la complétude.

3.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche permettant la résolution de DisCSP disposant de problèmes locaux complexes. Nous avons montré que notre algorithme, nommé *DBS*, a la particularité de ne pas utiliser de nogoods afin d'établir un contexte aux différents messages émis par les agents. Il utilise pour cela la notion de session. Nous avons vu que ces sessions sont des entiers permettant de connaître, pour chaque agent, l'état de la recherche globale. Ainsi chaque agent ajoute aux messages qu'il émet un numéro de session servant de contexte.

La première section a détaillé les différentes hypothèses et notations utilisées par *DBS*. Des hypothèses communes à la majorité des algorithmes de résolution de DisCSP ont été reprises par *DBS*. Puis nous avons exposé les structures de données permettant de définir à chaque instant le contexte des agents. Nous avons vu que celui-ci est composé, par exemple, d'une $agentView$ permettant de connaître les valeurs des variables des agents de priorité supérieure ainsi que de différentes structures permettant de connaître pour la session courante d'un agent les solutions locales qu'il avait déjà proposées.

La seconde section a été consacrée au fonctionnement de l'algorithme *DBS*. Nous avons commencé par décrire la méthode permettant le traitement de problèmes locaux complexes. Nous avons vu les différentes étapes suivies par chaque agent afin de construire la liste de solutions locales, le parcours efficace de ces solutions locales en tenant compte de l'$agentView$ et des contraintes inter-agents et la communication de ces solutions locales. Puis, nous avons détaillé l'ensemble des écouteurs, fonctions et procédures composant *DBS*. Enfin, nous avons illustré nos propos sur un exemple comportant trois agents.

La troisième et dernière section a porté sur les propriétés de l'algorithme *DBS*. Nous avons prouvé qu'il est correct, c'est-à-dire que lorsqu'il fournit une solution, celle-ci respecte bien la totalité des contraintes intra et inter-agents. Nous avons ensuite fourni la preuve de complétude en nous inspirant des preuves existantes des algorithmes ABT et *DDB*. Cette partie s'est achevée sur différents filtres permettant de réduire considérablement le nombre de messages présents dans les boîtes de réceptions de chaque agent et non encore traités. Pour chacun des quatre filtres présentés, nous avons prouvé qu'ils ne remettaient pas en cause la preuve de complétude de l'algorithme.

Dans le chapitre suivant, nous exposerons les résultats que nous avons obtenus avec *DBS*. Ce dernier sera ainsi comparé aux algorithmes présentés au chapitre 2. Nous évaluerons pour commencer la version multi-variables par agent puis nous traiterons le cas particulier où une unique variable est affectée à chaque agent. *DBS* sera donc comparé à des algorithmes mono-variable et multi-variables par agent.

4

Mise en œuvre et validation

Sommaire

4.1 Introduction

Nous avons présenté dans le chapitre précédent notre proposition, nommée *DBS*, permettant de résoudre des DisCSP mono-variable et multi-variables par agent.

Dans ce chapitre, nous allons nous intéresser à l'évaluation de *DBS*. Dans la seconde section, nous commencerons par observer l'impact des différents filtres permettant de supprimer des messages obsolètes dans les boîtes de réception des agents, tout en préservant la complétude. Nous observerons pour chaque filtre, ainsi que pour différentes combinaisons de filtres, les gains obtenus selon différents critères qui seront présentés.

Dans la troisième section de ce chapitre, nous évaluerons ensuite *DBS* dans le cas général où le problème local de chaque agent est complexe. Nous comparerons notre proposition aux algorithmes de résolution de DisCSP multi-variables par agent présentés en section 2.4, c'est-à-dire *Multi-ABT*, *Multi-AWC* et *AFC*. Afin de comparer *DBS* à ces algorithmes, nous utiliserons les deux principaux benchmarks de notre domaine : les DisCSP aléatoires et les problèmes de coloration de graphes distribués.

Dans la quatrième section, nous traiterons le cas particulier des DisCSP où le problème local de chaque agent est composé d'une unique variable. Pour cela, plutôt que d'utiliser un benchmark classique, nous nous baserons sur une application, à savoir le problème d'exploration multi-robots. Nous commencerons par présenter une façon de formaliser ce problème sous la forme d'un DisCSP. Ensuite, nous détaillerons les métriques que nous proposons pour évaluer notre proposition sur ce problème et nous présenterons les différents résultats obtenus. Pour ce

cas particulier mono-variable par agent, nous nous comparerons aux algorithmes *ABT* et *AWC* que nous avons introduit en section 2.3.

4.2 Évaluation des filtres des messages obsolètes

Dans cette section, nous allons observer l'impact des différents filtres de suppression des messages obsolètes sur l'algorithme *DBS*. Nous commençons par rappeler les quatre filtres présentés en section 3.4.5. Soit A_i un agent générique, nous notons BR_{A_i} sa boîte de réception qui contient les messages qu'il a reçu et qu'il n'a pas traité :

- F_1 : Si BR_{A_i} contient plusieurs messages $\langle OK? \rangle$ provenant d'un agent A_k, seul le dernier message $\langle OK? \rangle$ émis par A_k doit être traité, les autres sont ignorés et supprimés de BR_{A_i}.

- F_2 : Si BR_{A_i} contient plusieurs messages de backtrack et au moins un message $\langle OK? \rangle$ alors les messages de backtrack peuvent être supprimés de BR_{A_i}.

- F_3 : Si un agent A_i a envoyé un message de backtrack à un agent A_k appartenant à $agentView(A_i)$, alors, tant que A_i n'a pas reçu un message $\langle OK? \rangle$ de A_k, il peut supprimer tous les messages de backtrack contenus dans BR_{A_i}.

- F_4 : Si l'agent A_i reçoit plusieurs messages $\langle OK? \rangle$, alors il traite en priorité ceux venant des agents de priorité la plus élevée.

L'évaluation d'un algorithme de résolution de DisCSP est très difficile de par sa nature distribuée. En effet, afin d'obtenir une évaluation la plus juste possible, nous pensons qu'il faudrait utiliser un ordinateur par agent. Cependant, en pratique, les chercheurs du domaine des DisCSP effectuent leurs simulations sur un unique ordinateur. Pour cela, il existe trois façons de procéder :

- les agents sont simulés sans utiliser de plateforme multi-agents. Dans ce cas, l'utilisateur crée lui-même la structure permettant de gérer, en autre, les ressources propres des agents, les canaux de communication, les boîtes de réception.

- les agents sont simulés par une plateforme multi-agents de manière synchrone. Les agents sont activés à tour de rôle en utilisant une horloge commune basée sur la notion de cycles. Au cours d'un cycle, chaque agent lit tous les messages présents dans sa boîte de réception et, éventuellement, émet des messages qui pourront être lus par les autres agents au cycle suivant. Lorsqu'un agent A_i n'a plus de traitement à effectuer, c'est au tour de l'agent A_{i+1}. La plupart des algorithmes de résolution de DisCSP ont été évalués en utilisant le nombre de cycles requis pour résoudre le problème. Cependant, cette méthode a l'inconvénient d'imposer un synchronisme entre les agents ne représentant pas forcément les résultats pouvant être obtenus lors d'une évaluation avec un agent par ordinateur.

- les agents sont simulés par une plateforme multi-agents, telles que DIMA [Guessoum, 1998] ou JADE [Bellifemine *et al.*, 2000], permettant une simulation asynchrone des agents. Nous avons utilisé la plateforme JADE pour nos expérimentations. Celles-ci sont présentées dans cette section ainsi que dans la section 4.3. Cette plateforme, très utilisée par la communauté IAD, à l'avantage de gérer automatiquement l'asynchronisme entre les agents et per-

met un ordonnancement des agents non déterministe. De plus, JADE respecte les normes FIPA [FIPA, 2002]. Par contre, avec l'utilisation de cette plateforme, l'évaluation basée sur le nombre de cycles requis ne peut plus être utilisée car les agents ne sont plus activés à tour de rôle.

Afin d'évaluer les différents filtres de suppression de messages obsolètes, nous avons effectué différentes expérimentations sur un benchmark très utilisé lors de l'évaluation des DisCSP. Il s'agit d'un générateur de DisCSP aléatoires utilisant les paramètres suivants : $\langle m; n; d; \delta_{global}; t_{global}; \delta_{local}; t_{local}\rangle$ où :

- m : le nombre d'agents,
- n : le nombre de variables par agent,
- d : la taille du domaine de chaque variable,
- δ_{global} : la densité des contraintes inter-agents (compris entre 0 et 100 %),
- t_{global} : la dureté des contraintes inter-agents (compris entre 0 et 100 %),
- δ_{local} : la densité des contraintes intra-agent,
- t_{local} : la dureté des contraintes intra-agent.

Le tableau 4.1 présente les résultats obtenus par *DBS* pour la résolution de DisCSP générés aléatoirement avec les paramètres $\langle 15; 5; 5; 30\%; 30\%; 30\%; 30\%\rangle$. Nous avons choisi arbitrairement ces paramètres (permettant d'être dans la phase de transition des DisCSP où 50% des DisCSP générés sont inconsistants) car il nous est impossible de présenter des résultats en faisant varier l'ensemble des sept paramètres. Afin d'évaluer ces filtres, nous observons le temps maximal requis par un agent, le nombre de Non Concurrent Constraint Checks (NCCC) [Meisels *et al.*, 2002], le nombre total de messages échangés entre les agents et le nombre maximal de messages présents simultanément dans la boîte de réception d'un agent. Les valeurs qui figurent dans le tableau sont une moyenne sur 20 expérimentations. Les résultats que nous présentons ont été réalisés sur un ordinateur dual-core 2.8 Ghz avec 3 Gio de RAM et en utilisant la plateforme multi-agents JADE. Nous présentons une moyenne sur un grand nombre d'expérimentations car l'écart-type sur ce problème est important dû à la topologie du graphe représentant le problème [Prosser, 1996].

TABLE 4.1 – Filtrage des messages obsolètes dans *DBS*.

Filtres	Temps CPU maximal / agent (en s)	Non Concurent Constraint Check (en millions)	Nombre de messages échangés	Nombre max de messages dans les BR
aucun	83,4	154,9	22 967	5 668
F_1	4,3	7,9	3 991	43
F_2	44	73,9	23 061	3 132
F_3	71,2	115,5	18 781	4 196
F_4	63,1	66,4	17 547	3 272
$F_1 + F_2$	4,1	6,5	3 103	43
$F_1 + F_2 + F_3$	3,9	6,1	2 931	37
$F_1 + F_2 + F_3 + F_4$	3,4	6	2 895	52

Nous observons que le filtre F_1 est le plus efficace, suivi de F_4, F_2 puis F_3 que ce soit en termes de NCCC, temps CPU maximal et de messages échangés. De plus, la combinaison de ces filtres permet d'obtenir des résultats plus intéressants que ce soit pour le temps CPU maximal

utilisé, le nombre de NCCC et le nombre de messages échangés.

Nous observons que l'utilisation combinée des quatre filtres ($F_1 + F_2 + F_3 + F_4$) est très performante. Le temps CPU maximal utilisé par un agent pour résoudre le problème est de 83,4 secondes sans filtre et tombe à 3,4 secondes avec les quatre filtres. Le nombre de NCCC passe de 154,9 millions à 6 millions et le nombre de messages échangés passe de 22 967 à 2 895. Ici, sans filtre, le nombre maximal de messages en attente dans les boîtes de réception des agents est de 5 668. La combinaison de ces filtres fait tomber ce nombre à 52. Notons également que le filtre F_1, correspondant à la prise en compte des messages plus récents, impacte très fortement sur les résultats pour les différents critères.

Après avoir observé l'intérêt des filtres permettant de supprimer des messages obsolètes dans les boîtes de réception des agents, nous allons maintenant nous intéresser à l'évaluation de *DBS* dans le cas général multi-variables par agent.

4.3 Évaluation de DBS multi-variables par agent

Nous allons présenter, dans cette section, les résultats que nous avons obtenus lorsque nous avons évalué *DBS* en le comparant aux algorithmes multi-variables présentés en section 2.3. Nous avons décidé de comparer notre proposition à *Multi-ABT* car il s'agit de la généralisation à plusieurs variables par agent de *ABT* qui est l'algorithme de référence du domaine des DisCSP. *DBS* sera également comparé à *Multi-AWC* qui a la particularité d'avoir une priorité dynamique des variables affectées à chaque agent (la priorité des variables évolue au cours de la recherche contrairement à la priorité des agents qui reste inchangée). Enfin, *DBS* sera comparé à *AFC* dont la particularité est de combiner une recherche synchrone avec une propagation asynchrone.

Comparons donc, en termes d'efficacité, notre proposition (*DBS*) aux algorithmes *Multi-AWC* [Yokoo et Hirayama, 1998], *AFC* [Meisels et Zivan, 2007], *Multi-ABT* [Hirayama *et al.*, 2004] ainsi qu'à une version simplifiée de *DBS* nommée *Simple-DBS* :

Définition 25 *L'algorithme* Simple-DBS *reprend le principe de l'algorithme dbs sans l'utilisation des vpi (définition 24).*

Cette version simplifiée de *DBS* permettra d'observer l'apport de l'utilisation des VPI.

Ces quatre algorithmes seront comparés et évalués grâce à deux benchmarks. Nous commencerons par utiliser un benchmark de DisCSP aléatoires (section 4.3.1). Puis, nous détaillerons un benchmark pour des problèmes de coloration de graphes distribués ainsi que les résultats obtenus (section 4.3.2).

4.3.1 DisCSP aléatoires

Afin d'évaluer *DBS* dans le cas général où chaque agent dispose d'un problème local complexe, nous utilisons le générateur de DisCSP aléatoires présenté dans la section précédente lors de l'évaluation des filtres de suppression de messages obsolètes.

Nous présentons deux séries de tests (tableaux 4.2 et 4.3). Chaque résultat rapporté dans ces tableaux est une moyenne sur une série de 20 résolutions de DisCSP générés avec les mêmes paramètres. Ces tableaux présentent le nombre de messages échangés, le temps CPU ainsi que le nombre de contraintes vérifiées pour résoudre des DisCSP. Les résultats que nous présentons ont été réalisés sur un ordinateur dual-core 2.8 Ghz avec 3 Gio de RAM et en utilisant la plateforme multi-agents JADE [Bellifemine *et al.*, 2000].

TABLE 4.2 – Variation du nombre d'agents : $\langle n = 5; d = 3; \delta_{local} = 30\%; t_{local} = 30\%; \delta_{global} = 30\%; t_{global} = 30\% \rangle$.

m		*AFC*	*Multi-ABT*	*DBS*
2	Nombre de messages (unité)	2	4	4
	Temps CPU max (en ms)	35	4	4
	NCCC (unité)	1	1	2
3	Nombre de messages (unité)	5	58	47
	Temps CPU max (en ms)	45	9	15
	NCCC (unité)	4	8	7
4	Nombre de messages (unité)	memory	133	101
	Temps CPU max (en ms)	out	18	79
	NCCC (unité)		15	15
6	Nombre de messages (unité)	memory	14 356	8 425
	Temps CPU max (en ms)	out	1 202	3 009
	NCCC (unité)		3 914	1 213
8	Nombre de messages (unité)	memory	14 138	7 954
	Temps CPU max (en ms)	out	4 273	2 215
	NCCC (unité)		3 234	968
10	Nombre de messages (unité)	memory	13 321	8 109
	Temps CPU max (en ms)	out	657	658
	NCCC (unité)		3 097	739

Sur la première série d'expérimentations (tableau 4.2), nous faisons varier le nombre d'agents de 2 à 10. Le nombre de variables par agent (n) est fixé à 5 et d est fixé à 3. Le pourcentage de contraintes intra-agent et inter-agents est 30% et la dureté de ces contraintes est fixée à 30%. Nous pouvons observer qu'à partir de 4 agents, l'algorithme *AFC* atteint sa limite. En effet, lorsque le DisCSP contient 20 variables (5 variables par agent), *AFC* requiert un espace mémoire trop important pour l'ordinateur que nous avons utilisé pour nos expérimentations. Les algorithmes *Multi-ABT* et *DBS* résolvent ces DisCSP en des temps raisonnables (inférieurs à deux heures) tout en utilisant moins d'espace mémoire que *AFC*. Nous pouvons observer que pour des DisCSP comportant 10 agents, le nombre de messages échangés par *Multi-ABT* (13 321) est supérieur à celui de *DBS* (8 109). Le temps CPU est similaire pour ces deux algorithmes. Par contre, le nombre de NCCC est nettement inférieur pour *DBS* que pour *Multi-ABT*.

Sur la seconde série d'expérimentations (tableau 4.3), nous avons fait varier le nombre de variables par agent pour des DisCSP créés avec les paramètres suivants : $\langle m = 3; d = 3; \delta_{local} = 30\%; t_{local} = 30\%; \delta_{global} = 30\%; t_{global} = 30\% \rangle$. Nous observons qu'avec 2 variables par agent, *AFC*, *Multi-ABT* et *DBS* obtiennent des résultats similaires.

Lorsque le nombre de variables par agent vaut 6, les algorithmes échangent tous les trois une vingtaine de messages. Le temps CPU maximal utilisé par un agent pour résoudre ces DisCSP est largement supérieur pour *AFC* (702) alors qu'il n'est que de 5 pour *Multi-ABT* et de 78 pour *DBS*. De même le nombre de NCCC est important pour *AFC* (18 780) comparé aux algorithmes *Multi-ABT* (2 181) et *DBS* (10 356). A partir de 8 variables par agent, *AFC* ne peut résoudre les problèmes sans dépasser l'espace mémoire qui lui a été attribué. Comme pour le cas précédent, cette limite apparaît lorsque le nombre total de variables est d'environ 20. Lorsque le nombre de variables par agent vaut 10, *DBS* est plus performant que *Multi-ABT* en termes de messages échangés et de temps CPU. Ces deux algorithmes obtiennent un nombre de NCCC comparable (environ 13 000 000).

TABLE 4.3 – Variation du nombre de variables par agent : $\langle m = 3; d = 3; \delta_{local} = 30\%; t_{local} = 30\%; \delta_{global} = 30\%; t_{global} = 30\% \rangle$.

n		*AFC*	*Multi-ABT*	*DBS*
2	Nombre de messages (unité)	3	2	3
	Temps CPU max (en ms)	16	15	5
	NCCC (unité)	16	6	16
4	Nombre de messages (unité)	5	10	4
	Temps CPU max (en ms)	62	15	31
	NCCC (unité)	545	89	74
6	Nombre de messages (unité)	18	25	15
	Temps CPU max (en ms)	702	5	78
	NCCC (unité)	18 780	2 181	10 356
8	Nombre de messages (unité)	memory	198	738
	Temps CPU max (en ms)	out	31	686
	NCCC (unité)		83 928	785 191
10	Nombre de messages (unité)	memory	10 315	3 130
	Temps CPU max (en ms)	out	1 513	8 595
	NCCC (unité)		13 063 453	13 849 973

Nous observons, figure 4.3, que *Multi-ABT* est plus performant que *DBS* au niveau du temps CPU pour des DisCSP comportant une trentaine de variables. Pour des DisCSP d'une cinquantaine de variables (figure 4.2), les deux algorithmes sont équivalents en temps CPU. D'autres simulations, non présentées dans ce mémoire, confirment cette observation.

4.3.2 Coloration de graphes distribués

Le problème de coloration de graphes distribués est un problème souvent utilisé lors de la comparaison d'algorithmes de DisCSP. Le but est d'affecter des couleurs aux variables du problème de manière à ce qu'il n'y ait pas de variables reliées par une contrainte avec la même couleur. Toutes les contraintes du problème sont des contraintes d'inégalité que ce soit les contraintes intra-agent ou inter-agents. Afin de générer des instances de ce problème, un générateur de coloration de graphes distribués [Yokoo et Hirayama, 1998] peut être utilisé. Ce dernier dispose de quatre paramètres :

- Le nombre total de variables du problème (N),
- Le nombre de valeurs (couleurs) possibles pour chaque variable (d),
- Le nombre de contraintes (e),
- Le nombre d'agents (m).

Une instance de ce problème peut donc être représentée par un tuple $\langle N, d, e, m \rangle$. Cependant, nous devons préciser certains points de ce générateur qui est à l'origine un générateur de CSP centralisé. Selon les auteurs de [Yokoo et Hirayama, 1998], afin d'obtenir des problèmes difficiles, le nombre de contraintes e doit être égal à $2,7 \times N$ lorsque d vaut 3. Les N variables sont réparties équitablement entre les m agents et la distribution des contraintes entre les variables de différents agents doit respecter la règle suivante : les contraintes ne doivent pas être complètement générées aléatoirement afin d'obtenir des problèmes locaux complexes. Si ce n'était pas le cas, le nombre de C_{intra} serait très faible. Par exemple, avec 10 agents, chacun d'eux ayant 10 variables, il y aurait 270 contraintes ($2,7 \times 10 \times 10$) en tout et seulement 10% d'entre elles seraient des C_{intra}.

Comme dans [Yokoo et Hirayama, 1998], la moitié des contraintes seront des C_{intra} équitablement réparties entre les agents et l'autre moitié des C_{inter}.

Nous présentons deux séries de tests (tableaux 4.4 et 4.5). Chaque résultat rapporté dans ces tableaux est une moyenne sur une série de 50 résolutions de DisCSP générés avec les mêmes paramètres. Ces tableaux présentent le nombre de messages échangés, le temps CPU ainsi que le nombre de contraintes vérifiées pour résoudre des DisCSP. Lorsque le temps réel d'exécution (différent du temps CPU) est supérieur à deux heures, la résolution du problème est arrêtée.

TABLE 4.4 – Variation du nombre de variables par agent ($m = 15\,;\, d = 3\,;\, e = 2,7 \times n \times m$).

	Nb messages (milliers)			Temps total CPU (en s)			Contraintes vérifiées (milliers)		
n	*Multi ABT*	*Simple DBS*	*DBS*	*Multi ABT*	*Simple DBS*	*DBS*	*Multi ABT*	*Simple DBS*	*DBS*
5	26	12	13	7	3,4	3,7	61 221	391	32
7	32	11	11	20	4,2	4,1	253 550	802	34
9	62	10	11	87	4,9	4,5	1 271 448	1 622	44
11	431	21	24	7 354	18	14	1 433 406	13 913	124
13	-	47	48	> time out	75	43	-	73 669	324
15	-	47	61	> time out	221	147	-	184 174	631

Pour la première série de tests (tableau 4.4), les variables sont réparties équitablement sur les 15 agents et nous faisons varier le nombre de variables par agent de 5 à 15. d est fixé à 3. Nous pouvons observer que le nombre de messages échangés est quasiment identique que ce soit pour *DBS* ou *Simple-DBS*. Cela provient du fait que l'amélioration qu'apporte *DBS* par rapport à *Simple-DBS* n'améliore que la recherche de solutions consistantes avec l'$agentView$ (réalisée au niveau intra-agent sans envoi de messages). Le nombre de messages échangés est plus important pour *Multi-ABT* que pour *DBS*. Par exemple lorsque chaque agent dispose de 11 variables, il y a en moyenne 431 messages échangés avec *Multi-ABT* et seulement 21 avec *DBS*.

Le temps de résolution est meilleur pour *DBS* que pour *Simple-DBS*. *Multi-ABT* utilise plus de temps CPU que *DBS* et l'écart entre ces deux algorithmes s'amplifie lorsque le nombre de variables par agent augmente. Lorsque le nombre de variables par agent dépasse 11, *Multi-ABT* n'arrive plus à résoudre les problèmes en des temps raisonnables (temps supérieur à deux heures). Nous pouvons observer que le fait d'utiliser la méthode sauvegardant la cause de l'inconsistance de chaque solution combinée à l'utilisation des VPI (Cf. déf. 24, section 3.3.2) est très efficace en termes de contraintes vérifiées (car *Simple-DBS* obtient un temps CPU plus faible et un nombre de contraintes vérifiées plus faible que *DBS*). *Multi-AWC* n'est pas présenté dans ce tableau car il ne peut résoudre ces DisCSP en des temps raisonnables.

Le tableau 4.5 porte sur des DisCSP de petites tailles (comportant de 2 à 6 agents) pouvant être résolus par *Multi-AWC* et *AFC*. Nous avons réduit le nombre de contraintes de $2,7 \times n \times m$ à $2 \times n \times m$ car il est impossible de respecter la règle qui impose que la moitié des contraintes sont des C_{inter} et l'autre moitié des C_{intra} compte tenu du si petit nombre d'agents.

L'algorithme *DBS* est le plus efficace en nombre de messages échangés, en temps total CPU et en nombre de contraintes vérifiées, suivi de *Simple-DBS*, *Multi-ABT*, *AFC* et enfin *Multi-AWC*. Nous observons que *Multi-AWC* nécessite plus d'une heure trente de temps CPU pour résoudre des DisCSP répartis sur 4 agents (contenant chacun 5 variables). Cela provient du fait que chaque agent exécutant *Multi-AWC* crée autant d'agents virtuels que des variables locales. L'algorithme *AFC* ne peut résoudre des DisCSP contenant 5 agents et 5 variables par agent en des temps raisonnables. Ceci est notamment dû à la gestion des nogoods minimaux qui est très coûteuse.

TABLE 4.5 – Variation du nombre d'agents (n = 5 ; d = 3 ; e = 2 × n × m).

m		*Multi-AWC*	*AFC*	*Multi-ABT*	*Simple-DBS*	*DBS*
2	Nombre de messages (unité)	189	8	32	28	28
	Temps total CPU (en ms)	64	1 256	6	1	2
	Contraintes vérifiées (unité)	7 799	1 085	17 517	704	679
3	Nombre de messages (unité)	599	28	3 876	51	49
	Temps total CPU (en ms)	13 869	1 269	525	4	3
	Contraintes vérifiées (unité)	20 807	5 730	2 182 399	1 502	723
4	Nombre de messages (unité)	1 122	53	6 949	327	333
	Temps total CPU (en ms)	5 663 823	23 134	1 053	45	48
	Contraintes vérifiées (unité)	38 829	16 453	6 111 723	7 763	2 986
5	Nombre de messages (unité)	-	-	8 047	2 394	2 903
	Temps total CPU (en ms)	> time out	> time out	1 136	341	373
	Contraintes vérifiées (unité)	-	-	5 569 175	54 722	16 187
6	Nombre de messages (unité)	-	-	5 469	2 239	2 075
	Temps total CPU (en ms)	> time out	> time out	834	358	296
	Contraintes vérifiées (unité)	-	-	3 513 698	65 987	10 209

Nous allons maintenant nous intéresser à un cas particulier de DisCSP : le cas où chaque agent ne dispose que d'une seule variable. Pour cela, nous allons évaluer l'algorithme *DBS* sur un générateur de DisCSP aléatoire ainsi que sur un problème concret : le problème d'exploration multi-robots dans un environnement inconnu.

4.4 Cas particulier : DBS mono-variable par agent

À notre connaissance, tous les algorithmes de résolution de DisCSP multi-variables sont des algorithmes de résolution de DisCSP mono-variable qui ont été généralisés pour traiter des problèmes locaux complexes. Ces algorithmes ont donc tous été évalués et comparés à des algorithmes mono-variable tels que *ABT* et *AWC*. Pour cette raison, nous avons décidé de présenter les résultats obtenus par *DBS* dans ce cas particulier. Pour cela, nous commençons par présenter une évaluation de *DBS* sur des DisCSP générés aléatoirement (section 4.4.1). *DBS* sera ensuite évalué sur un problème concret d'exploration multi-robots (section 4.4.2) pouvant être formalisé sous la forme d'un DisCSP mono-variable par agent.

4.4.1 DisCSP aléatoires

Afin de générer des instances de DisCSP, nous utilisons le générateur de CSP proposé par C. Bessière [6] et nous affectons chaque variable à un agent. Les paramètres de ce générateur de CSP sont les suivants : $\langle N; d; \delta; t\rangle$ où N représente le nombre total de variables du DisCSP, d la taille du domaine de chaque variable , δ le degré de connectivité locale et t la dureté des contraintes locales.

Après avoir introduit le générateur de DisCSP, nous allons maintenant observer le comportement de *DBS* sur des instances générées. Trois algorithmes seront comparés : *ABT*, *DBS* et une version de *DBS*, nommée *DBS + filtres*, utilisant les filtres permettant de supprimer les messages obsolètes (Cf. section 4.2).

6. `http://www.lirmm.fr/~bessiere/generator.html` (accessible en 2012)

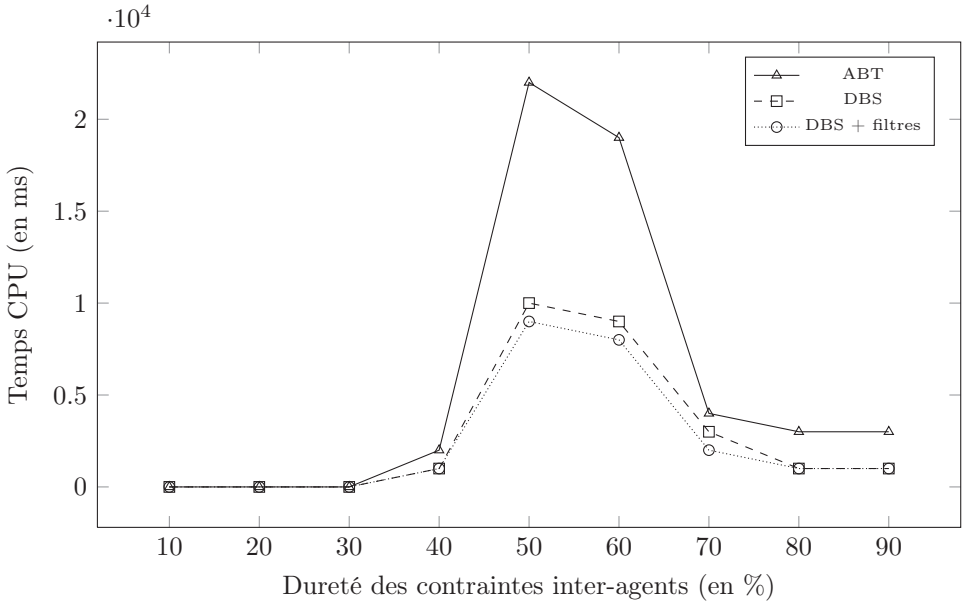

FIGURE 4.1 – DisCSP contraint avec 40% du nombre maximal de contraintes.

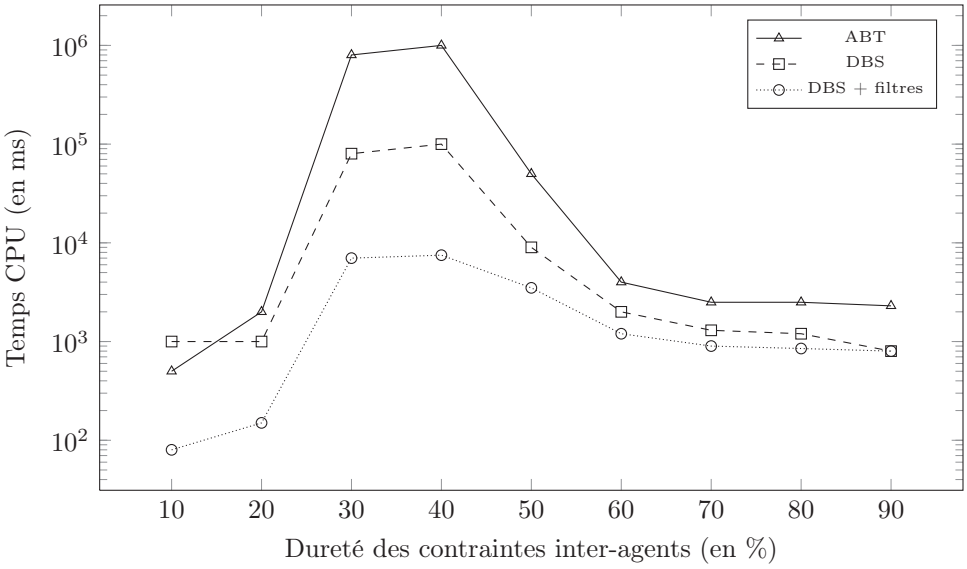

FIGURE 4.2 – DisCSP contraint avec 80% du nombre maximal de contraintes.

Nous pouvons observer, sur les figures 4.4.1 et 4.4.1, que *DBS* est plus rapide que *ABT* aussi bien pour des DisCSP très contraints que pour les DisCSP peu contraints. Pour les problèmes situés dans la phase de transition, *DBS* est beaucoup plus intéressant que *ABT* pour le temps CPU (la somme des temps CPU de chaque agent). De plus, l'utilisation des filtres de suppression de messages obsolètes permet une réduction importante du nombre de messages échangés et donc du temps CPU utilisé. Pour les DisCSP contraints avec 80% du nombre maximal de contraintes, durant la phase de transition, nous avons dû arrêter l'exécution de *ABT* après quatre heures de temps CPU et nous n'avons pas reporté ces résultats dans nos moyennes.

Le tableau 4.6 montre des résultats obtenus avec des DisCSP créés avec les paramètres sui-

TABLE 4.6 – $\langle N = 20; d = 10; \delta = 20\%; t = 60\% \rangle$.

Algorithme utilisé	temps CPU (en s)	Ecart-type (en s)	Nb contraintes vérifiées (en millions)	Nombre de messages échangés	Nombre max de messages dans les BR
ABT	29,3	104	2,7	19 650	1 340
DBS	26,7	74	2,8	236 076	1 554
DBS + filtres	14,8	60	1,3	1 015 105	9

TABLE 4.7 – $\langle N = 15; d = 10; \delta = 50\%; t = 50\% \rangle$.

Algorithme utilisé	temps CPU (en s)	Ecart-type (en s)	Nb contraintes vérifiées (en millions)	Nombre de messages échangés	Nombre max de messages dans les BR
ABT	428,2	751	24,1	109 700	15 471
DBS	20,5	13	2,6	170 137	18 823
DBS + filtres	9,3	6	0,8	44 915	10

vants : $\langle N; d; \delta; t \rangle = \langle 20; 10; 20\%; 60\% \rangle$. Nous observons que le nombre de messages échangés pour *ABT* est plus faible que pour *DBS*. *DBS* requiert moins de calculs que *ABT* pour traiter un message de backtrack : *ABT* calcule toutes les sous-parties inconsistantes de l'$agentView$ et envoie, pour chaque sous-ensemble, une demande de backtrack (très coûteux en calculs). Contrairement à *ABT*, *DBS* transmet un message de backtrack à l'agent fautif (l'agent ayant choisi une valeur provoquant une demande de backtrack) ou alors à un agent ayant une plus petite priorité que l'agent fautif (peu coûteux en calculs).

En utilisant un protocole similaire, d'autres expérimentations ont été faites (tableau 4.7) en utilisant les paramètres $\langle N; d; \delta; t \rangle = \langle 15; 10; 50\%; 50\% \rangle$. *ABT* résout des DisCSP en n'utilisant que 109 700 messages en 428 secondes. *DBS*, sans les filtres de suppression de messages obsolètes, résout les DisCSP en n'utilisant que 170 137 messages en 20 secondes. *DBS* utilisant les propriétés de suppression de messages obsolètes utilise 44 915 messages en 9 secondes. Nous avons observé que l'écart-type est très important pour tous les algorithmes.

Après voir détaillé les résultats obtenus par *DBS* sur un générateur classique de DisCSP, nous allons maintenant nous intéresser à l'évaluation de *DBS* sur un problème concret, à savoir le problème d'exploration multi-robots dans un environnement inconnu.

4.4.2 L'application multi-robots

Parmi les applications des systèmes multi-robots, l'exploration d'un environnement inconnu sous des contraintes de communication est considéré comme un problème difficile ([Rooker et Birk, 2007, Vazquez et Malcolm, 2004]). Les robots doivent maintenir, durant toute l'exploration, une connectivité avec le reste de la flotte de robots afin de pouvoir constamment envoyer des messages. En plus, les robots ne doivent pas se trouver à la même place afin d'éviter des problèmes de collision et ne pas explorer la même surface. Ils ne doivent donc ni être trop éloignés ni trop près les uns des autres. Une solution à ce problème peut être d'utiliser des mécanismes de coordination multi-agents en utilisant des techniques classiques telles que des algorithmes de bidding [Sheng *et al.*, 2006]. Il est également possible de modéliser ce problème en utilisant les DisCSP [Doniec *et al.*, 2009].

Dans cette application, nous considérons des robots avec des capacités de communication et d'observations (comme des lasers, sonars ou caméra). Ces deux capacités sont soumises à des limitations matérielles. Par exemple, deux robots ne peuvent pas communiquer s'ils ne sont pas présents dans l'intersection de leurs portées de communication. De la même manière, chaque robot dispose d'une portée d'observation limitant sa perception d'objets dans l'environnement.

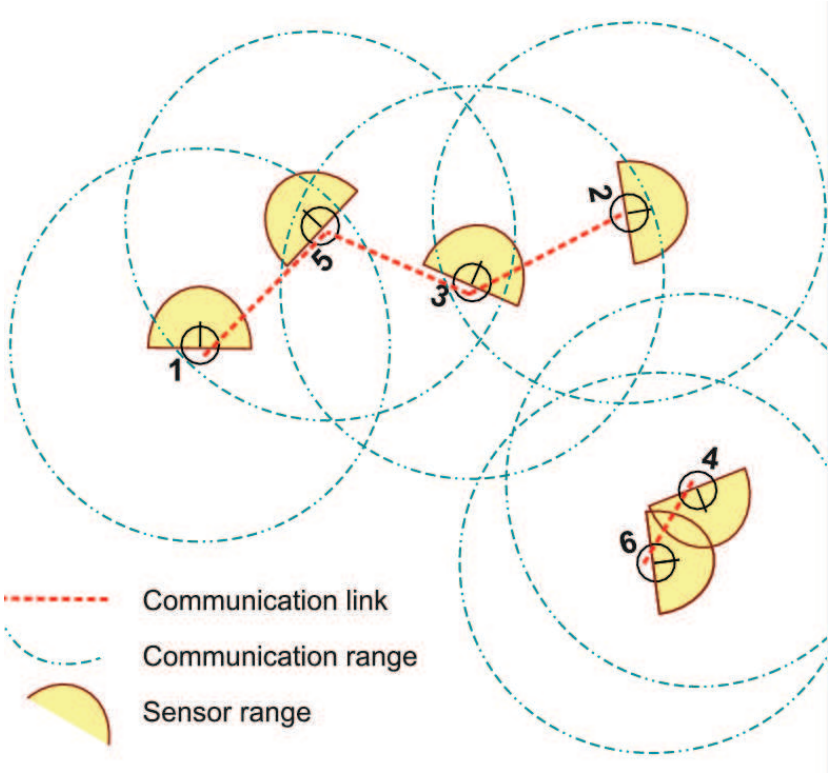

FIGURE 4.3 – Communication entre les agents.

Sur la figure 4.3, les robots 1 et 5 peuvent communiquer car il existe un lien de communication entre eux étant donné qu'ils se trouvent dans l'intersection de leurs portées de communication. En revanche, les robots 2 et 4 sont trop éloignés pour pouvoir s'échanger des données. Comme il existe un chemin de communication entre les robots 1 et 2, ils peuvent échanger des données (les robots 3 et 5 transmettent les données dans ce cas).

Afin d'effectuer une exploration efficace, les robots se déplacent vers la frontière entre la surface explorée et la surface inexplorée [Yamauchi, 1998]. De plus, les robots doivent (1) collaborer afin de se répartir sur la surface (ceci permet d'accélérer l'exploration et d'utiliser moins d'énergie) et (2) de rester en connexion avec les autres (ceci permet d'échanger des représentations partielles de l'environnement durant la résolution et de maintenir un lien de communication pour tout couple d'agents).

Ces deux derniers points peuvent être intégrés dans un schéma de coordination en tant que contraintes d'un DisCSP. Chaque robot est alors considéré comme un agent et dispose d'une unique variable à instancier. Cette variable représente sa future direction (c'est-à-dire la direction suivante à explorer). Le domaine de chaque variable est composé de 8 directions cardinales : $\{N, NE, E, SE, S, SO, O, NO\}$. Les assignations de chaque variable doivent satisfaire deux contraintes :

– la position future du robot ne doit pas briser la connectivité du réseau.

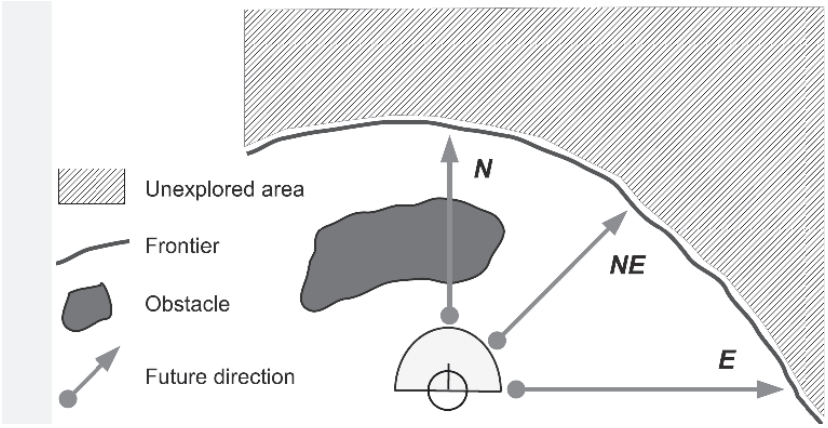

FIGURE 4.4 – Exploration vers la frontière surface explorée/surface inexplorée

– la position future du robot ne doit pas créer de chevauchement avec la portée d'exploration de ses voisins (cette contrainte pourrait éventuellement être relachée dans certains cas à conditions que cela ne provoque pas de collisions).

Sur la figure 4.3, le premier point n'est pas vérifié par les robots 1 et 5. Le second point n'est pas vérifié par les robots 4 et 6.

Le besoin de chaque robot de se déplacer près de la frontière à chaque mouvement peut être pris en compte grâce à un certain ordre des directions présentes dans les domaines. Cet ordre peut aussi prendre en compte les obstacles potentiels. Par exemple, à la figure 4.4, le robot essaye d'affecter sa variable avec la direction NO avant la direction O, avant la direction N. Le plus petit point vers la frontière est situé au Nord. Cependant, ce point ne peut pas être choisi car un obstacle est présent sur le chemin du robot. Les deux autres directions, NE et E, permettent toutes les deux d'accéder à un point de la frontière. Mais clairement, la frontière sera plus rapidement atteinte en suivant la direction NO. Cela signifie que, dans le domaine du robot, la direction NE précédera la direction E qui précédera la direction N .

Afin d'explorer un environnement inconnu, les robots doivent périodiquement résoudre le DisCSP présenté précédemment. La phase d'exploration consiste à répéter la séquence suivante :

1. chaque robot met à jour ses connaissances concernant sa connectivité [Le *et al.*, 2009],
2. chaque robot met à jour sa représentation de l'environnement avec la dernière position connue de ses voisins et met à jour la frontière environnement connu/environnement inconnu,
3. les contraintes du DisCSP sont calculées en se basant sur les connaissances obtenues aux points 1 et 2,
4. chaque robot ordonne les valeurs de son domaine en tenant compte de sa position vis à vis de la frontière et des obstacles,
5. chaque robot prend part à la résolution du DisCSP avant de trouver la direction vers laquelle il va se déplacer,
6. chaque robot se déplace dans la direction calculée à l'étape précédente.

Notons que le DisCSP ne peut être inconsistant qu'à la toute première résolution. En effet en cours de résolution, une solution peut toujours exister : une solution conduisant à obtenir l'état

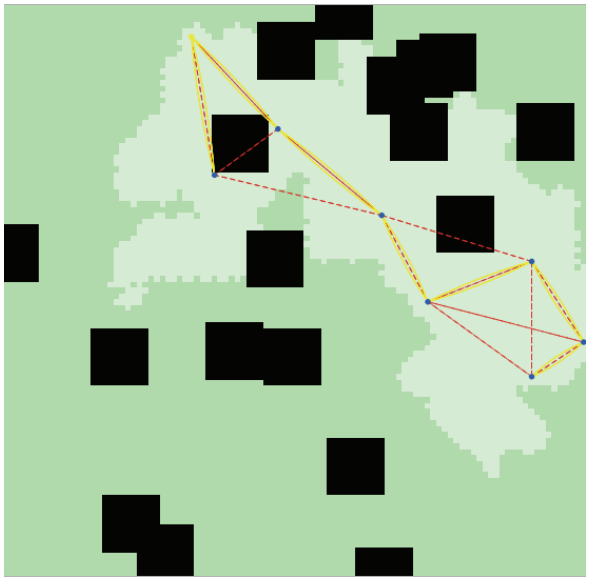

FIGURE 4.5 – Aperçu de l'application multi-robots.

précédent où les robots avancent d'un pas après avoir fait un virage à 180 degrés. L'exploration de l'environnement est terminée lorsqu'il n'y a plus de surface à explorer. Notons que n'importe quel algorithme de résolution de DisCSP mono-variable par agent peut être utilisé à l'étape 5.

La figure 4.5 présente un aperçu de l'application d'exploration multi-robots. Huit robots travaillent ensemble afin d'explorer l'environnement. L'environnement déjà exploré par les robots est représenté en gris clair. L'environnement inconnu est représenté en gris foncé et les zones noires représentent des obstacles disposés aléatoirement. Il existe un lien de communication uniquement lorsque les agents se situent à l'intersection de leurs portées de communication. L'exploration se termine lorsque l'environnement tout entier est en gris clair.

Depuis plusieurs années, de nombreux travaux portent sur le raisonnement par contraintes distribuées. Cependant, l'utilisation des DisCSP pour des problèmes multi-agents réels est loin d'être ordinaire. Très peu de publications portent sur des applications concrètes des DisCSP : *distributed timetabling problems* [Xiang et Zhang, 2008], *stable marriages problem* [Atkinson *et al.*, 2006], *sensorDisCSP* [Fernandez *et al.*, 2002]. La conséquence est que l'évaluation des algorithmes de DisCSP est basée uniquement sur des métriques « artificielles »– le nombre de cycles, le nombre de Non Concurrent Constraints Checks (NCCC [Meisels *et al.*, 2002]) et le nombre de messages échangés.

La résolution de problèmes concrets peut amener des problèmatiques qui ne sont pas considérées avec les benchmarks classiques (les problèmes de coloration de graphes [Omomowo *et al.*, 2008] ou encore les DisCSP aléatoires [Ezzahir *et al.*, 2009]). Pour cette raison, nous présentons, dans la section suivante, les métriques que nous utiliserons pour mesurer efficacement différents aspects de l'exploration multi-robots.

Lors de la comparaison d'algorithmes de résolution de DisCSP, la plupart des auteurs de la littérature ne s'intéressent qu'à une évaluation quantitative de l'efficacité en ne tenant compte que de quelques métriques tels que le nombre de messages échangés et le nombre de NCCC. Quelques travaux essayent d'évaluer ces algorithmes de manière qualitative ; spécialement pour des problèmes distribués de sécurité ; mais dans ces évaluations, l'intérêt est seulement porté sur le respect de la privacité. Nous montrons dans cette section que d'autres métriques [Monier *et al.*,

2010b, Monier *et al.*, 2011] peuvent être prise en compte pour une évaluation qualitative.

Dans la majorité des articles, les algorithmes de résolutions de DisCSP sont comparés entre eux (et généralement avec ABT comme point de comparaison) en utilisant les trois métriques suivantes :

- le nombre de cycles. Chaque agent dispose d'une horloge commune qui est incrémentée à chaque cycle de calculs. Un cycle, pour un agent, consiste à lire tous les messages contenus dans sa boîte de réception, de traiter et transmettre des messages. Les messages transmis durant le cycle t seront disponibles et pourront être traités au cycle $t + 1$.

- le nombre de contraintes vérifiées ou le nombre de Non-Concurrent Constraint Checks (NCCC) [Meisels *et al.*, 2002].

- le nombre total de messages transmis afin de résoudre le DisCSP.

Cependant, nous trouvons que l'utilisation seule de ces trois métriques n'est pas suffisante pour notre problème d'exploration multi-robots. En effet, si deux algorithmes obtiennent le même nombre de cycles et le même nombre de contraintes vérifiées/NCCC, il se peut qu'un algorithme soit plus efficace que le second. Un cycle peut prendre t secondes de temps CPU en moyenne pour le premier algorithme et $5t$ secondes pour le second.

Afin d'obtenir une évaluation la plus juste possible, les trois précédentes métriques peuvent être complétées par les trois suivantes :

- le temps total CPU.

- la distance totale parcourue par les agents. Cette métrique est spécifique à notre application. Cette métrique permet d'évaluer si un algorithme est efficace pour des agents situés et mobiles dans un environnement inconnu. Cependant, ce n'est pas parce qu'un algorithme obtient un faible nombre de cycles et un faible nombre de messages échangés qu'il est le plus efficace pour obtenir un déplacement cohérent des agents.

- le nombre de DisCSP résolus pour explorer la totalité de l'environnement.À chaque pas de la simulation, la flotte de robots résout des DisCSP afin de choisir leurs futures directions. Cette métrique permet de vérifier, comme nous le verrons par la suite, que lorsque le nombre d'agents augmente, le nombre total de DisCSP résolus diminue.

Ces métriques sont utilisées dans la section suivante pour évaluer les algorithmes ABT, AWC et DBS sur le problème d'exploration multi-robots.

Les agents doivent résoudre le problème d'exploration multi-robots présenté dans la section 4.4.2. Les robots se déplacent dans un environnement fermé modélisé par une grille. Avant de commencer la résolution du problème, les robots sont placés sur des positions pseudo-aléatoires en respectant la règle suivante : un agent doit avoir au plus deux voisins et un chemin doit pouvoir exister reliant tout couple d'agents. Les agents doivent maintenir ce chemin durant toute la résolution.

Notre application a été implémentée [7] en utilisant Netlogo [Tisue et Wilensky, 2004]. Netlogo

7. Les tests ont été effectués sur un ordinateur dual-core 2.8 Ghz avec 3 Gio de RAM.

est un environnement multi-agents programmable permettant de créer rapidement et efficacement des agents situés dans un monde en deux dimensions. Nous considérons un environnement fermé et modélisé par une grille de 100×100 cellules. Chaque cellule peut être vide, occupée par un agent, explorée ou inconnue. Différents paramètres peuvent être modifiés comme la portée de vue, la portée de communication. Différentes données sont sauvegardées à chaque simulation et sont présentées dans la section suivante.

Dans cette section, nous faisons varier différents paramètres de l'application multi-agents : le nombre de robots (8 par défaut), le pourcentage de l'environnement à explorer (100 % par défaut), la portée de communication des robots (30 unités par défaut) et le pourcentage d'obstacles présents dans l'environnement (0 par défaut).

À chaque fois qu'un paramètre est modifié, nous allons observer pour les figures 4.6, 4.8, 4.9 et 4.10, en abscisse, la distance parcourue (a), le nombre de NCCC (b), le nombre de cycles (c), le nombre de DisCSP résolus (d), le nombre de messages échangés (e) et le temps CPU total utilisé par les agents (f). Chaque point présenté dans ces figures représente une moyenne sur 10 explorations. À chaque étape de l'exploration, un DisCSP est résolu puis les robots se déplacent.

Variation du nombre d'agents

Nous pouvons observer (fig. 4.6-a) que la distance totale parcourue par les agents pour explorer l'environnement est significativement plus faible pour *ABT* et *DBS* (approximativement 6 200 pour 8 agents) que pour *AWC* (approximativement 8 000 pour 8 agents). Ceci est dû à la priorité des robots dans *AWC*. Dans *ABT* ou *DBS*, la priorité entre les robots est statique et est basée sur l'ordre lexicographique des identifiants des robots : robot 0 $\succ$ robot 1 $\succ$ robot 2.... Comme le robot 0 est l'agent de plus haute priorité, il dirige la résolution du DisCSP et tente d'imposer sa future direction favorite [8]. Clairement, l'agent 0 peut être considéré comme un meneur pour la flotte de robots durant toute l'exploration. Contrairement à *ABT* et *DBS*, la priorité des agents exécutant *AWC* est dynamique. En fait, à chaque fois qu'un agent envoie une demande de backtrack, sa priorité devient la plus grande parmi ses accointances. Cela signifie qu'entre deux résolutions successives de DisCSP, deux différents agents peuvent imposer leurs propres préférences (leur future direction préférée). En d'autres termes, le meneur de la flotte peut changer entre deux résolutions successives. Cela entraîne des discontinuités et des oscillations dans le déplacement de la flotte. C'est pour ces raisons que la coordination multi-agents est moins performante en utilisant *AWC*.

Le nombre de contraintes vérifiées (fig. 4.6-b) est plus important pour *ABT* (approximativement 50 000 pour 8 robots) que pour *DBS* (approximativement 22 000 pour 8 robots) qui est un peu plus élevé que pour *AWC*. *ABT* a le plus grand nombre de contraintes vérifiées dû au fait que lorsqu'un agent doit envoyer une demande de backtrack, cet agent doit calculer un nogood minimal [Yokoo, 2001]. La création d'un nogood, dans *ABT*, est très coûteuse contrairement à *DBS* dont la priorité est de traiter rapidement les messages.

Le nombre de cycles (fig. 4.6-c) requis par *ABT* et similaire à celui de *DBS* (approximativement 1 500 pour 8 robots). *AWC* a obtenu un nombre de cycles plus faible (approximativement 600 pour 8 robots). *AWC* est meilleur pour le nombre de cycles grâce à la priorité dynamique entre des agents qui permet de réduire l'influence d'un mauvais choix fait par un agent de priorité supérieure. La différence entre *ABT* et *AWC* est similaire aux résultats obtenus sur des benchmarks de DisCSP aléatoires [Yokoo, 2001].

8. Rappelons que chaque agent ordonne son domaine en se basant sur sa distance par rapport à la frontière et de la position des obstacles.

Le nombre de DisCSP résolus (fig. 4.6-d) pour une exploration diminue lorsque le nombre de robots augmente. Ce résultat confirme un point "normal" des systèmes multi-robots : plus il y a de robots, plus l'exploration est rapide.

Le nombre de messages échangés (fig. 4.6-e) est meilleur pour *DBS* que pour *ABT* qui est lui-même meilleur que pour *AWC*. Ce résultat est extrêmement intéressant pour notre application. En effet, de nombreux réseaux mobiles ad-hoc ne supportent pas une surcharge de communication lorsqu'un grand nombre de messages est échangé entre les agents. Clairement, pour des expérimentations multi-robots sur des robots physiques, *DBS* serait le meilleur choix.

Finalement, le temps total CPU (fig. 4.6-f) varie d'une expérience à une autre et dépend de la charge de l'ordinateur durant les expérimentations. Cependant, nous avons décidé de présenter les temps CPU afin d'obtenir une idée du temps requis. Nous pouvons observer que *DBS* est meilleur que *ABT* qui est meilleur que *AWC*. La différence entre *ABT* et *DBS* est similaire aux résultats obtenus avec un benchmarks de DisCSP aléatoires [Monier *et al.*, 2009a].

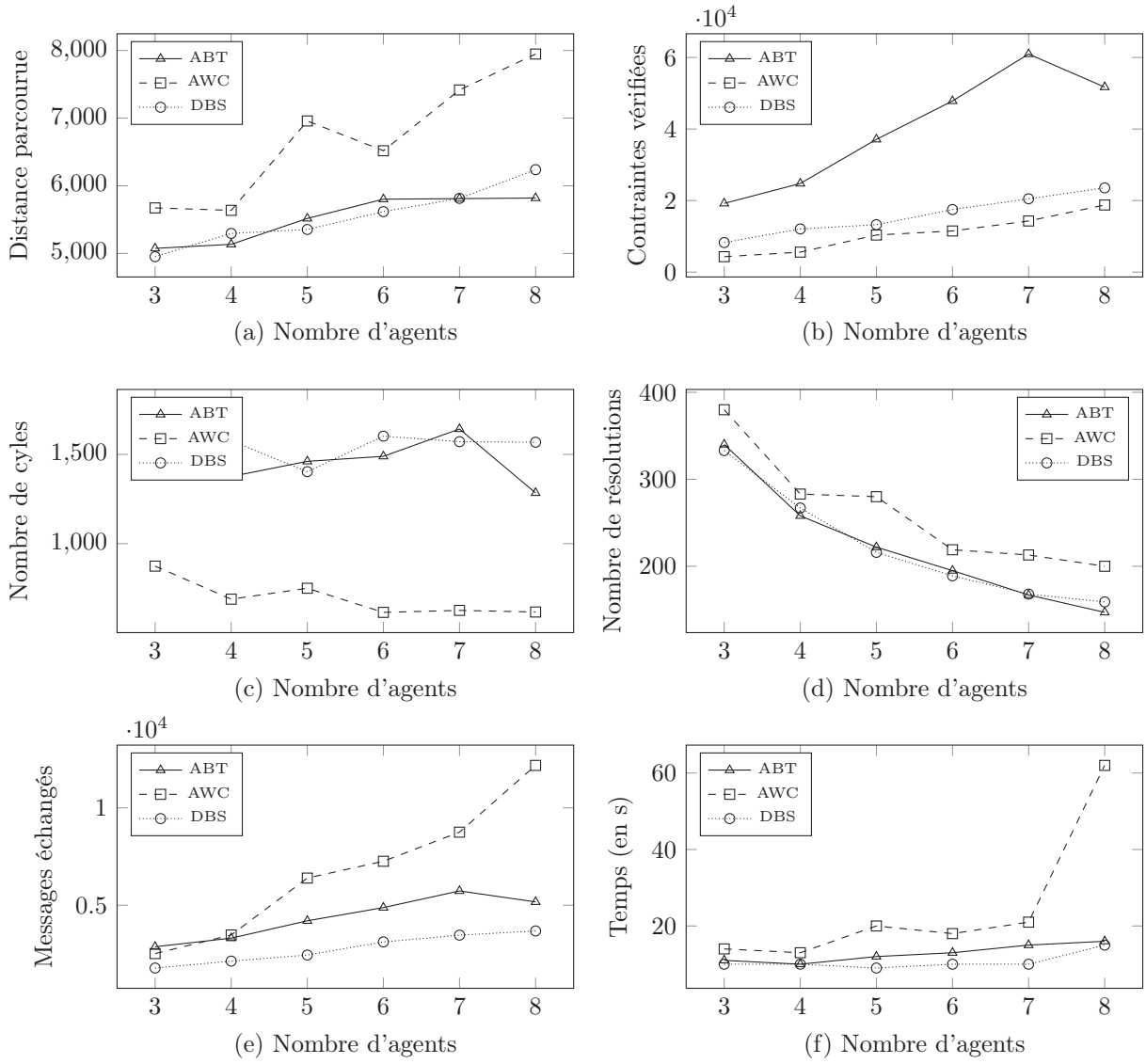

FIGURE 4.6 – Variation du nombre d'agents.

Surface explorée par les agents

Les figures 4.7-a, 4.7-b et 4.7-c montrent respectivement la surface explorée par l'agent 0, la surface minimum explorée par un robot et la surface maximum explorée par un robot pour un environnement de 100×100 cellules. Le nombre de robots, variant de 3 à 9 se situe en abscisse et le pourcentage de la surface explorée se situe en ordonnée.

Nous observons, à la figure 4.7-a qu'en utilisant *AWC*, le robot 0 explore le moins de surface quelque soit le nombre de robots. Par exemple, avec trois robots, le robot 0 explore $32,31\%$ de la surface avec *ABT*, $34,03\%$ de la surface avec *DBS* et seulement $28,18\%$ avec *AWC*.

Chaque robot se déplace vers la localité lui permettant d'explorer la plus grande surface inconnue tout en satisfaisant les contraintes de distance entre les agents de plus haute priorité. Comme l'agent 0 est l'agent le moins contraint avec *ABT* et *DBS* (car ce robot a la plus haute priorité [9]), il a plus de chance de se déplacer vers sa direction préférée que les agents de plus faible priorité. Donc, le robot 0 (utilisant les algorithmes *ABT* et *DBS*) explore davantage de surface que les autres agents de la flotte qui ont une priorité inférieure. Avec *AWC*, la priorité du robot 0 est dynamique et évolue au cours de l'exploration. Le robot 0 explore moins de surface à cause de la priorité qui est plus faible qu'avec *ABT* et *AWC*.

Nous observons dans la figure 4.7-b (respectivement sur la figure 4.7-c) que la plus petite surface explorée (respectivement la plus grande) est obtenue avec *AWC*.

En utilisant *ABT* et *DBS*, le robot 0 explore environ 1/ *nombre de robots* de la surface ($33,33\%$ s'il y a trois agents). Bien que le robot 0 ait la plus grande priorité, il n'explore pas plus que les autres robots. Ce résultat est très intéressant car il permet de montrer que la coordination basée sur des DisCSP assure une bonne distribution de travail entre les robots. Pour notre application, l'utilisation d'un ordre total entre les agents avec *ABT* et *DBS* n'a pas d'impact sur l'efficacité de la coordination. Peut-être que cette propriété pourrait être généralisée à d'autres applications multi-agents.

Pourcentage de la surface à explorer

Sur la figure 4.8, le pourcentage de la surface à explorer varie de 40% à 100%. Nous pouvons observer qu'en général, les courbes sont plus raides lorsque la surface à explorer varie de 40% à 60% que pour 60% à 80%. Cela signifie que moins d'efforts sont requis par les agents pour explorer l'environnement de 40% à 60% que pour 60% à 80%. Similairement, nous observons que la courbe augmente fortement dans les derniers pourcentages (proche de 100%). Cela est dû à l'exploration des dernières cellules qui sont éparpillées dans l'environnement. Pour les explorer, les agents doivent parcourir une plus grande distance pour un plus petit nombre de cellules donc le ratio (efforts d'exploration / gains obtenus en explorant les cellules) est plus faible. Nous pouvons observer que lorsque les agents explorent seulement 40% de l'environnement, les performances de *ABT*, *AWC* et *DBS* sont similaires. Lorsque les agents doivent explorer plus de 40% de l'environnement, *ABT* et *DBS* sont plus performants que *AWC*.

Variation de la portée wifi

Sur la figure 4.9, la portée de communication de chaque agent varie de 20 unités à 50 unités. Nous remarquons que plus la portée de communication augmente, plus les agents sont efficaces

9. Chaque robot doit respecter les contraintes avec les agents de plus haute priorité. Comme le robot 0 est l'agent de plus haute priorité, il n'a aucune contrainte à respecter.

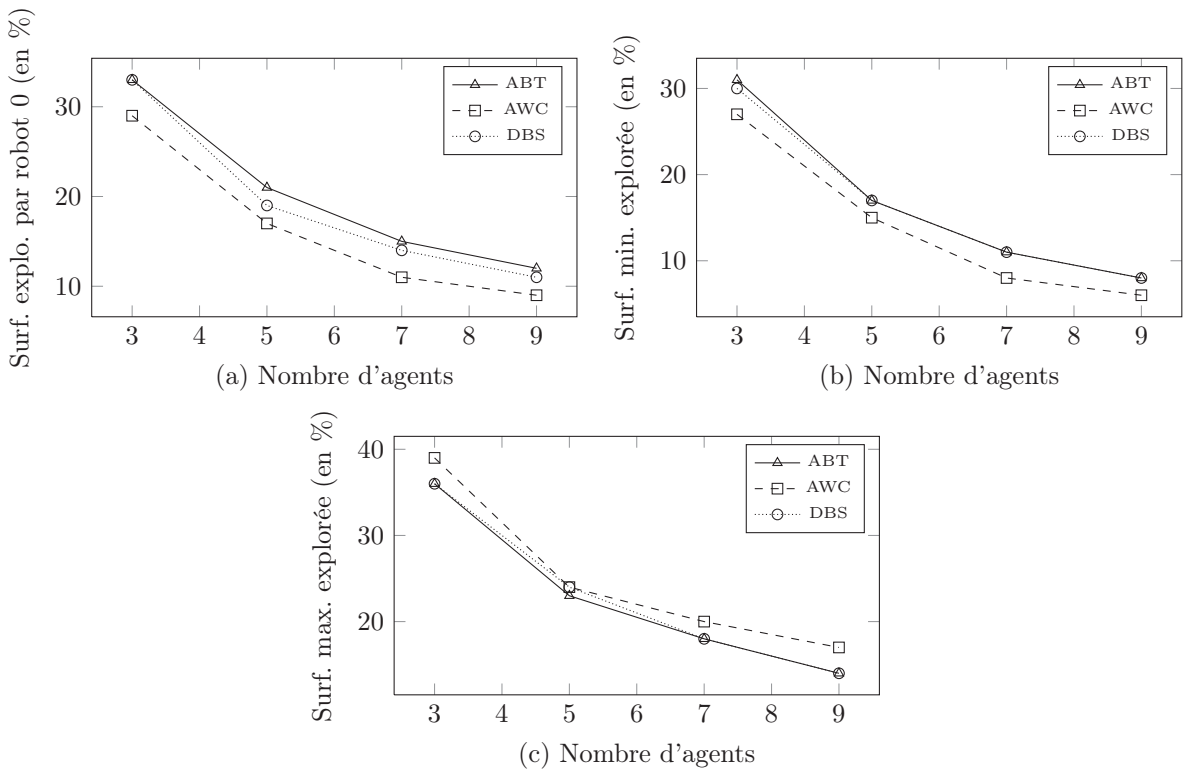

FIGURE 4.7 – (a) Surface explorée par le robot 0, (b) Surface minimale explorée par un robot, (c) Surface maximale explorée par un robot.

pour explorer l'environnement inconnu. Pour chacune des figures (a) à (f), avec l'algorithme *AWC*, les valeurs diminuent fortement de 20 à 30, modérément de 30 à 40 et peu de 40 à 50. Pour les algorithmes *ABT* et *DBS*, les tendances des courbes sont légèrement différentes : la courbe décroît de 20 à 30-40 puis stagne. Une portée de communication supérieure (> 50) améliore très peu les performances. Plus la portée de communication est faible, plus le DisCSP est difficile à résoudre. En effet, les robots peuvent se déplacer aussi loin qu'ils le veulent les uns des autres mais ne doivent pas être trop proche pour explorer la même surface. Donc, le nombre de solutions du DisCSP, respectant ces contraintes de distance, diminue fortement.

Variation de la proportion d'obstacles

Pour la figure 4.10, les robots explorent un environnement carré (de 100×100 cellules) contenant des obstacles représentés par des carrés (de 10×10 cellules). Nous avons fait varier le pourcentage d'obstacles présents dans l'environnement. Ces obstacles sont placés aléatoirement dans l'environnement. Nous remarquons que plus le nombre d'obstacles augmente dans l'environnement, moins les agents sont efficaces pour l'exploration.

Dans certaines simulations, les robots se sont retrouvés bloqués. En effet, plus le nombre d'obstacles présents dans l'environnement augmente, plus il y a de chances que des robots se trouvent de chaque côté d'un obstacle, incapable de déplacer sans casser la connectivité du réseau.

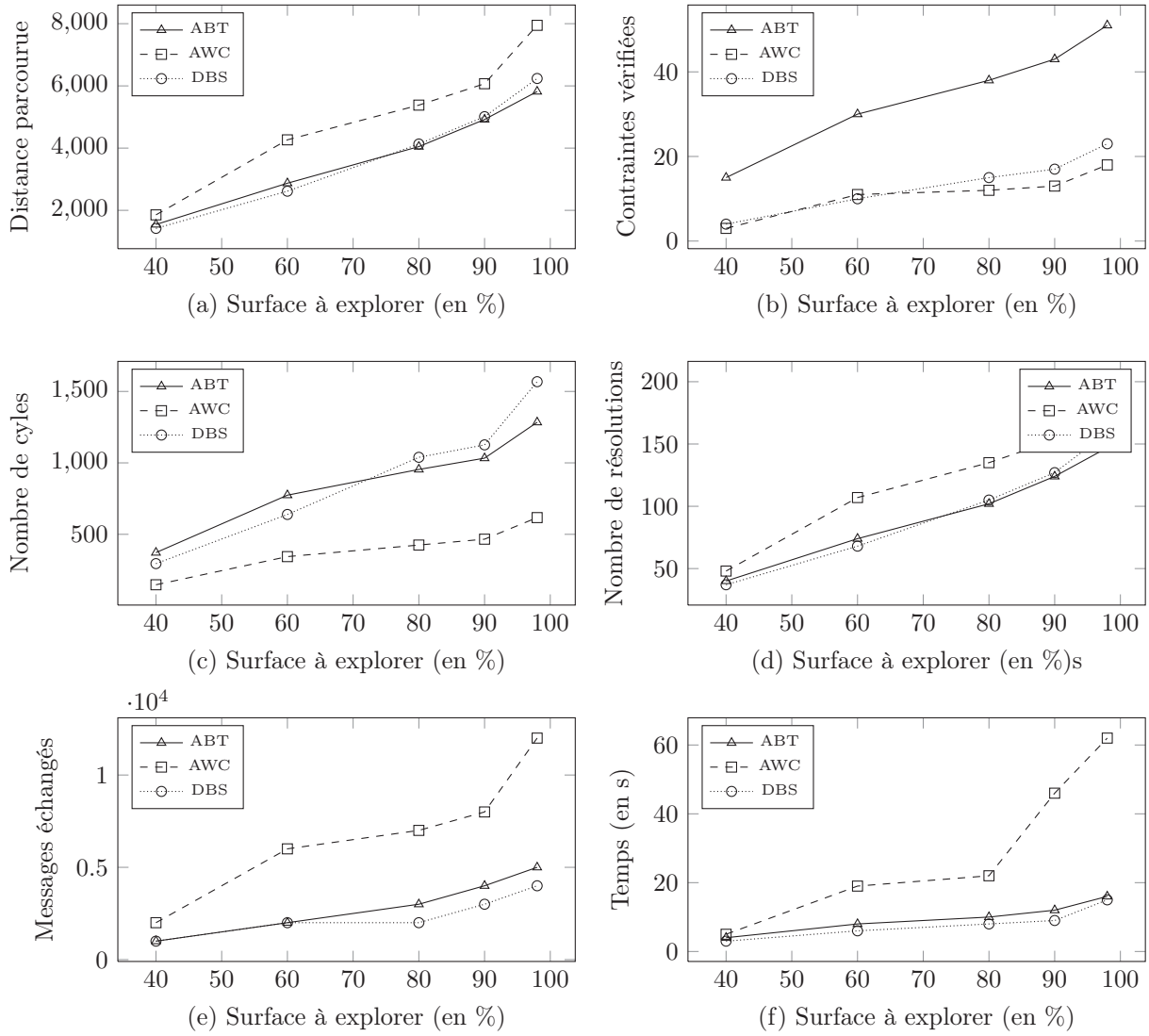

FIGURE 4.8 – Pourcentage de la surface à explorer.

Résumés des résultats obtenus

Sur la figure 4.6, *DBS* est meilleur en termes de distance parcourue, temps CPU et en nombre de résolutions de DisCSP et de messages échangés. *AWC* est meilleur pour le nombre de contraintes vérifiées et le nombre de cycles. Mais sur l'application proposée, nous avons montré (figures 4.6 et 4.7) que *AWC* n'est pas adapté car la priorité dynamique des agents pénalise la coordination multi-agents pour notre application.

La figure 4.8 montre que l'effort requis pour explorer un pourcentage de l'environnement n'est pas le même au début qu'à la fin de l'exploration. C'est dans l'intervalle [60%-80%] que le ratio (effort d'exploration / gains obtenus en explorant les cellules) est le meilleur. La figure 4.9 montre que plus la portée de communication est importante, moins cela nécessite d'efforts pour explorer l'environnement. La figure 4.10 montre que plus il y a d'obstacles, plus il est difficile pour les agents d'explorer l'environnement même s'il y a moins de cellules à explorer.

Pour une évaluation strictement quantitative, certains résultats obtenus et décrits dans cette

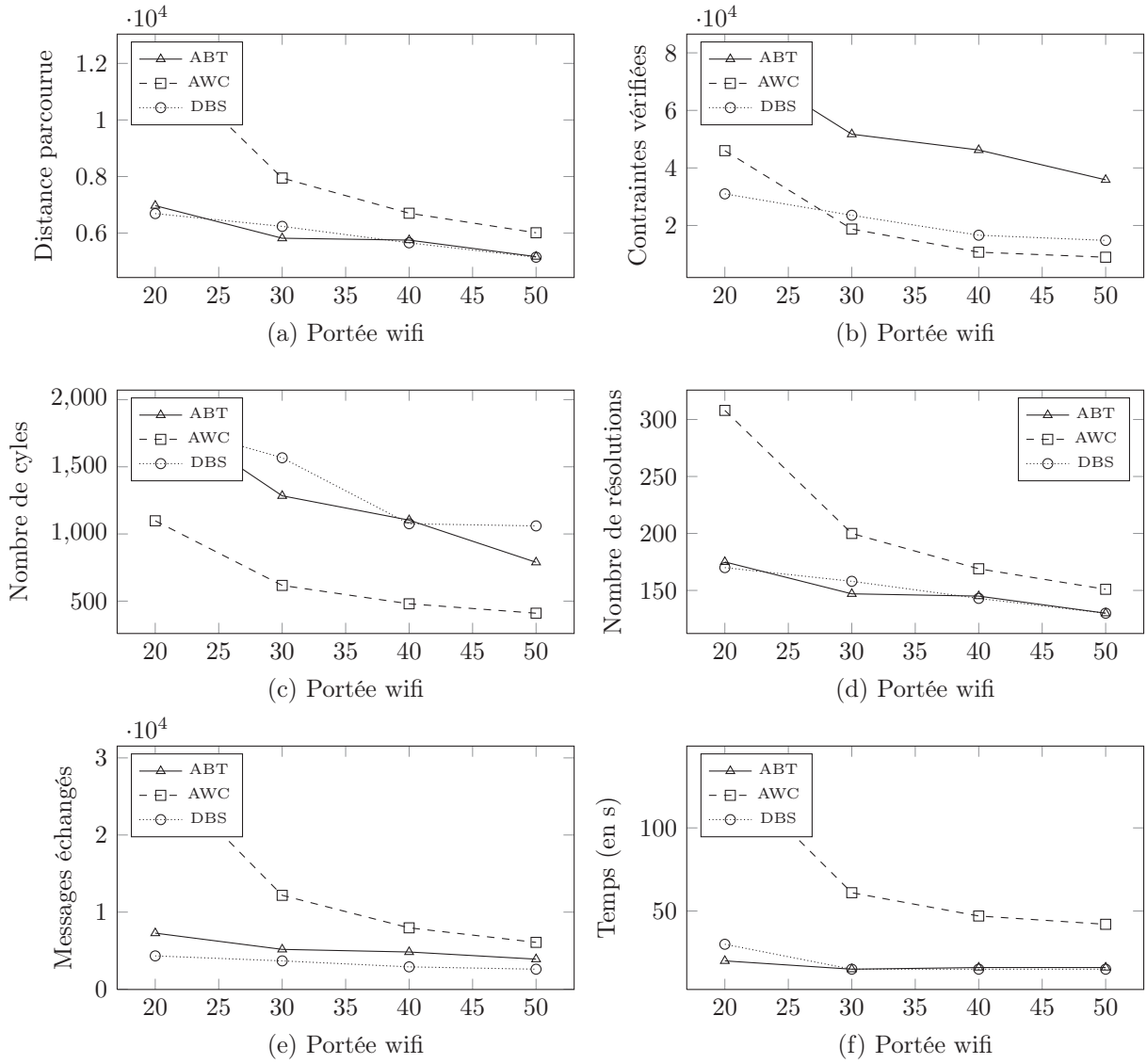

FIGURE 4.9 – Variation de la portée wifi.

section sont différents des résultats présentés dans la littérature et obtenus sur des DisCSP aléatoires en utilisant des métriques classiques. En général, *AWC* est connu pour être meilleur que *ABT* pour résoudre des DisCSP dans lesquels le nombre de solutions est important. Ce n'est pas le cas ici. De la même manière, dans [Monier *et al.*, 2009a], nous avons montré que *ABT* est meilleur que *DBS* en nombre de messages échangés. Seulement les résultats sur le nombre de cycles et sur le temps CPU sont similaires.

Finalement, considérant les métriques utilisées, *DBS* semble être la meilleure alternative parmi les trois algorithmes considérés.

4.5 Conclusion

Ce chapitre a été consacré à la mise en œuvre de l'algorithme *DBS* ainsi qu'à son évaluation. Nous avons évalué l'efficacité des filtres permettant la suppression des messages obsolètes dans les

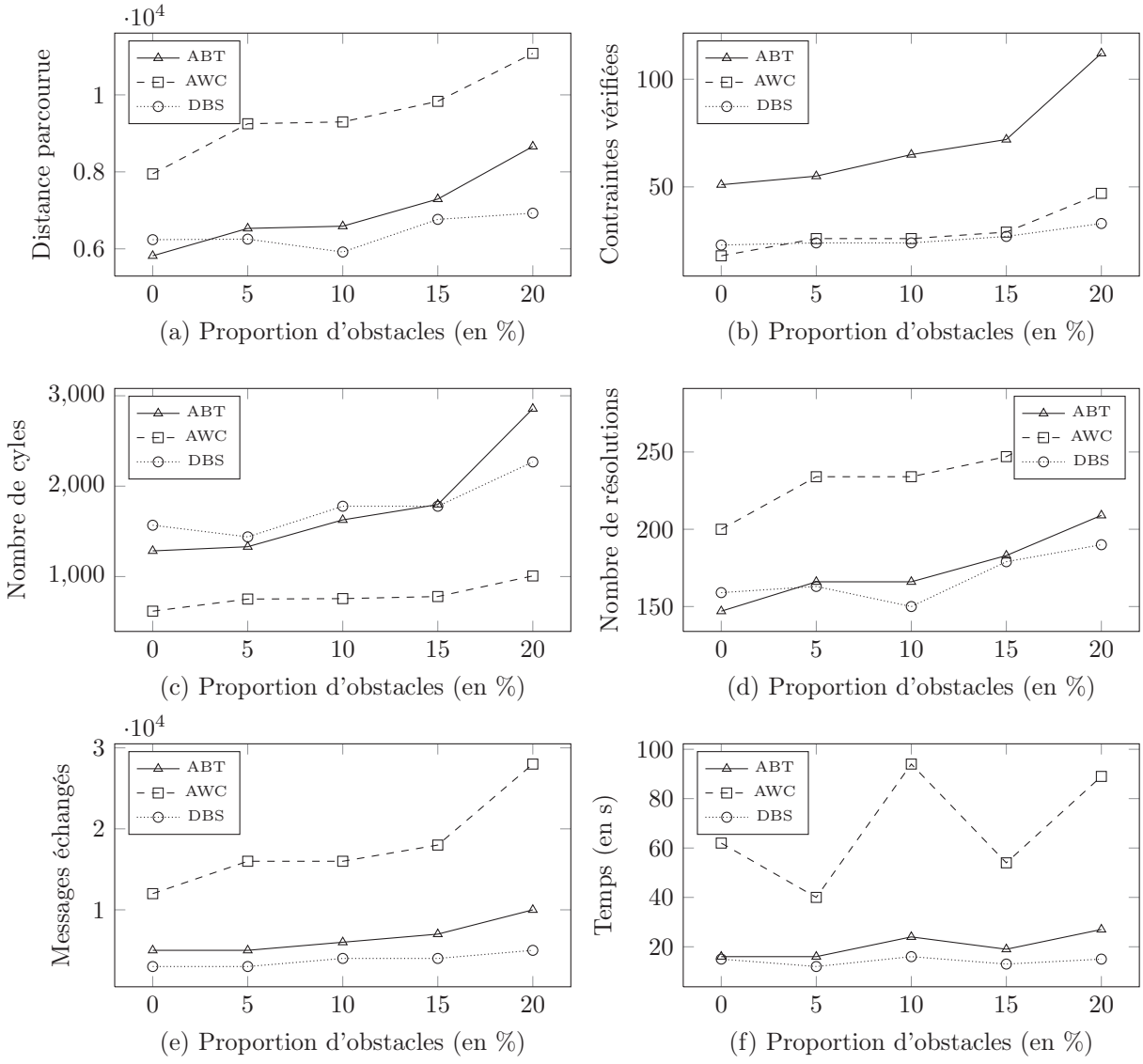

FIGURE 4.10 – Variation de la proportion d'obstacles présents dans l'environnement.

boîtes de réception des agents. Nous avons observé l'impact des filtres présentés en section 3.4.5 séparément. Nous avons constaté que le premier filtre, nommé F_1, permettant de ne traiter que le dernier message de type $\langle OK? \rangle$ reçu par chaque agent, était le plus efficace. Puis, nous avons vu que l'utilisation combinée de ces filtres permettait de supprimer davantage de messages et de réduire ainsi le temps global de résolution du DisCSP. De plus, ces filtres permettent de réduire considérablement le nombre de messages stockés et en attente de traitement dans les boîtes de réception des agents. En effet, sans l'utilisation de ces filtres, pour des DisCSP comportant une quinzaine d'agents, le nombre maximal de messages en attente de traitement pour un agent s'élevait à plus de 5000, ce qui est très coûteux à stocker. Grâce à l'utilisation de ces filtres, ce nombre est divisé par mille.

Nous avons évalué notre proposition, en section 4.3, dans le cas général où le problème local de chaque agent est composé de plusieurs variables et contraintes intra-agent. *DBS* a ainsi été comparé aux algorithmes *Multi-ABT*, *Multi-AWC* et *AFC* que nous avons préalablement

détaillé en section 2.4 en exposant leurs similarités et leurs différences. Nous avons expliqué que l'évaluation d'un algorithme de résolution de DisCSP est très difficile de par sa nature distribuée. En effet, afin d'obtenir la comparaison la plus juste possible, nous pensons que le mieux serait d'affecter un ordinateur par agent. Cependant, en pratique, lorsque le nombre d'agents est élevé, cela devient impossible et trop coûteux. De ce fait, l'évaluation s'effectue en général sur un unique ordinateur. Nous avons choisi, pour évaluer *DBS* multi-variables, d'utiliser une plateforme multi-agents nommée JADE permettant de gérer automatiquement l'asynchronisme entre les agents. Afin d'évaluer ces algorithmes, nous nous sommes servis des deux principaux benchmarks utilisés dans notre domaine : les DisCSP aléatoires et les problèmes de coloration de graphes distribués. Nous avons observé que sur ces deux benchmarks, l'algorithme *DBS* obtenait de meilleurs résultats que les algorithmes *Multi-ABT*, *Multi-AWC* et *AFC* que ce soit sur le nombre de messages échangés, le temps CPU et le nombre de contraintes vérifiées.

Dans la quatrième et dernière section, nous avons traité le cas particulier de DisCSP où le problème local de chaque agent est composé d'une unique variable. Pour cela, plutôt que d'utiliser un benchmark classique de DisCSP, nous nous sommes basés sur un problème réel étudié notamment en systèmes multi-agents, à savoir le problème d'exploration multi-agents. Nous avons commencé par détailler ce problème multi-robots et nous avons présenté une formalisation de ce problème avec les DisCSP. Nous avons vu que chaque robot pouvait être représenté par un agent disposant d'une unique variable. Une variable peut prendre comme valeur la future direction ($N, NE, E, SE, etc.$) du robot. Les contraintes inter-agents représentent des contraintes de distance entre les robots leur interdisant de se retrouver trop proches les uns des autres (afin d'éviter par exemple des problèmes de collision) ni trop éloignés (afin qu'il existe un chemin de communication pour tout couple d'agents permettant de s'échanger des messages). *DBS* a été comparé aux algorithmes *ABT* et *AWC* que nous avons introduit en section 2.3.

Nous avons utilisé une plateforme multi-agents nommée NetLogo ayant la particularité d'activer les agents à tour de rôle en utilisant la notion de cycles. L'évaluation a porté sur six métriques (le temps CPU, le nombre de NCCC, le nombre de DisCSP résolus, le nombre de messages échangés, la distance totale parcourue et le nombre de cycles). Nous avons vu que l'utilisation combinée de ces six métriques permettait d'obtenir une meilleur comparaison et nous avons montré que notre proposition était plus intéressante que les deux autres algorithmes. De plus, nous avons montré que l'utilisation d'une priorité dynamique entre les agents était un inconvénient pour cette application contrairement aux benchmarks classiques.

Nous allons maintenant conclure ce travail et proposer quelques perspectives théoriques et pratiques.

5

Conclusion générale et perspectives

5.1 Conclusion générale

Dans le travail présenté dans ce mémoire, nous avons proposé un nouveau mécanisme de coordination multi-agents basé sur la résolution de DisCSP. Le problème distribué à résoudre est réparti entre différents agents qui interagissent afin d'aboutir à l'emergence d'une solution globale à partir des solutions locales de chaque agent. Nous avons proposé un algorithme nommé *Distributed backtracking with Sessions* (*DBS*) permettant de résoudre des DisCSP contenant plusieurs variables d'interfaces par agent. Afin de gérer le multi-variables, le CSP local de chaque agent a été transformé de manière à ce que chaque agent ne dispose plus que d'une unique variable dont les valeurs possibles représentent les solutions locales du CSP. Le raisonnement intra-agent (recherche locale) s'effectue en parallèle du raisonnement inter-agents (recherche globale) et les différents agents composant le DisCSP agissent de manière asynchrone.

Dans le chapitre 1, nous avons fait un état de l'art sur les systèmes multi-agents, sur les problèmes de satisfaction de contraintes et sur les approches relatives aux CSP et aux SMA : les DisCSP. Pour commencer, nous avons énoncé des généralités sur les SMA. Nous avons décrit différents points caractérisant un SMA en nous basant sur l'approche Voyelle. Puis nous avons vu que le raisonnement intra-agent et inter-agents pouvait être basé sur un ensemble de relations entre différentes variables. La résolution du problème peut s'effectuer grâce à des agents interagissant afin d'obtenir une solution globale.

Nous nous sommes ensuite intéressés aux CSP. Après avoir donné les définitions de base, nous avons détaillé des algorithmes de filtrage. Nous avons vu que ces derniers permettent de transformer un CSP en un CSP équivalent disposant des mêmes solutions mais dont les domaines des variables ont été réduits. Ceci permet de restreindre la taille de l'arbre de recherche. Puis nous avons présenté divers algorithmes de résolution de CSP tels que BT, BJ et CDB. Nous avons détaillé brièvement différentes heuristiques permettant d'ordonnancer efficacement les valeurs et les variables. Enfin, le formalisme des DisCSP a été présenté. Nous avons vu différents algorithmes de filtrage distribué et nous avons présenté des heuristiques d'ordonnancement d'agents.

Dans le chapitre 2, nous nous sommes focalisés sur les DisCSP. Nous avons commencé par fournir des définitions nécessaires à la compréhension de ce domaine. Nous avons présenté des problèmes naturellement distribués pouvant être résolus grâce aux DisCSP. Puis nous avons remarqué que l'une des principales difficultés lors de la résolution des DisCSP est la gestion de l'asynchronisme entre les agents. Cela nécessite une contextualisation des messages échangés

entre les agents. Ensuite, les différentes méthodes permettant de détecter la terminaison de l'algorithme ont été décrites.

Nous avons présenté des algorithmes de résolution de DisCSP dans lesquels, historiquement, le problème local d'un agent ne contient qu'une unique variable. Nous avons détaillé *ABT* qui est le premier algorithme de résolution de DisCSP asynchrone. Puis nous avons expliqué le fonctionnement de l'algorithme *AWC* qui a la particularité d'utiliser un ordonnancement dynamique entre les agents. Nous avons ensuite abordé les algorithmes de résolution de DisCSP disposant de problèmes locaux complexes, c'est-à-dire composés de plusieurs variables et de plusieurs contraintes intra-agent. Nous avons observé que les généralisations de *ABT* et *AWC* avaient la particularité de créer autant d'agents virtuels que de variables dans le problème initial. Enfin, nous avons détaillé l'algorithme *AFC* qui combine une partie synchrone pour la recherche de solutions avec une partie asynchrone pour la propagation.

Nous avons ensuite détaillé les limites de ces algorithmes. Les agents n'inter-agissent pas complètement de manière asynchrone et peuvent créer des agents virtuels pour chacune de leurs variables locales, ce qui augmente considérablement la complexité du problème si les agents virtuels ne sont pas correctement gérés. Des créations de liens entre les agents apparaissent au cours de la recherche et des nogoods extrêmement coûteux sont utilisés par les agents. Nous avons ainsi vu l'intérêt d'un nouvel algorithme essayant de remédier à ces limites.

Dans le chapitre 3, nous avons présenté notre nouvelle approche permettant la résolution de DisCSP disposant de problèmes locaux complexes. Nous avons montré que notre algorithme, nommé *DBS*, a la particularité de ne pas utiliser de nogoods afin de contextualiser les messages échangés mais d'utiliser des sessions. Ces sessions sont des entiers qui renseignent, pour chaque agent, sur l'état de la recherche globale. Nous avons détaillé les hypothèses, les notations, les structures de données utilisées par *DBS*. Nous avons observé les différentes étapes suivies par chaque agent afin de construire sa liste de solutions locales, le parcours efficace de ces solutions locales en tenant compte de l'$agentView$ et des contraintes inter-agents et la communication de ces solutions locales. Puis, nous avons détaillé l'ensemble des procédures composants *DBS* que nous avons illustré par un exemple.

Les propriétés de l'algorithme ont été détaillées. Nous avons prouvé que *DBS* est correct et complet. Ensuite, nous avons décrit différents filtres permettant de réduire considérablement le nombre de messages présents dans les boîtes de réception de chaque agent et non encore traités. Nous avons prouvé que ces filtres ne remettaient en cause ni la correction, ni la complétude de l'algorithme.

Le chapitre 4 a été consacré à la mise en œuvre et l'évaluation de l'algorithme *DBS*. Avant de comparer notre proposition avec les principaux algorithmes de la littérature, nous avons évalué l'efficacité des filtres permettant la suppression des messages obsolètes dans les boîtes de réception des agents. Nous avons observé l'impact des filtres présentés au chapitre 3. Nous avons vu que ces filtres permettent de réduire considérablement le nombre de messages stockés en attente de traitement dans les boîtes de réception des agents.

Puis, nous avons évalué notre proposition dans le cas général où le problème local de chaque agent est composé de plusieurs variables et de plusieurs contraintes intra-agent. *DBS* a ainsi été comparé aux algorithmes *Multi-ABT*, *Multi-AWC* et *AFC*. Pour cela, nous nous sommes servis des deux principaux benchmarks utilisés dans notre domaine : les DisCSP aléatoires et les problèmes de coloration de graphes distribués. Nous avons observé que sur ces deux benchmarks, l'algorithme *DBS* obtient de meilleurs résultats que les algorithmes *Multi-ABT*, *Multi-AWC* et *AFC* aussi bien sur le nombre de messages échangés, que du temps CPU et du nombre de

contraintes vérifiées. Nous avons ensuite traité le cas particulier de DisCSP où le problème local de chaque agent est composé d'une unique variable.

L'évaluation de *DBS* a également été faite sur un problème réel d'exploration multi-robots dans un environnement inconnu. Nous avons commencé par modéliser avec une formalisation DisCSP. Chaque robot est représenté par un agent disposant d'une unique variable. Celle-ci peut prendre comme valeur la direction future $(N, NE, E, SE, \dots)$ du robot. Les contraintes inter-agents représentent des contraintes de distances entre les robots leur interdisant de se retrouver trop proches les uns des autres (afin d'éviter par exemple des problèmes de collisions) ni trop loin (afin qu'il existe un chemin de communication pour tout couple d'agents permettant de s'échanger des messages). *DBS* a été comparé aux algorithmes *ABT* et *AWC*.

L'évaluation a porté sur six métriques : le temps CPU, le nombre de NCCC, le nombre de DisCSP résolus, le nombre de messages échangés, la distance totale parcourue et le nombre de cycles. Nous avons vu que l'utilisation combinée de ces six métriques permet une meilleure comparaison et nous avons montré que *DBS* était plus intéressant que *ABT* et *AWC*. De plus, nous avons montré que l'utilisation d'une priorité dynamique entre les agents était un inconvénient pour cette application contrairement aux benchmarks classiques.

Afin d'améliorer notre proposition et de la valider sur des problèmes plus complexes, nous proposons différentes perspectives théoriques et pratiques.

5.2 Perspectives de recherches

La première partie de cette section est consacrée aux perspectives théoriques de l'algorithme *DBS*. *DBS* permet actuellement de résoudre des DisCSP appartenant à un système fermé où le nombre d'agents et le nombre de contraintes intra et inter-agents n'évoluent pas au cours de la résolution. Nous pensons que c'est l'une des limites de notre proposition que nous aimerions lever. Un autre point d'amélioration de *DBS* concerne la priorité des agents. En effet, celle-ci est actuellement statique et nous aimerions la rendre dynamique afin que la modification d'une valeur appartenant à un agent de priorité élevé et provoquant une inconsistance puisse s'effectuer plus rapidement. Cependant, nous verrons que différents obstacles se présentent si nous souhaitons conserver la complétude de notre algorithme. Nous verrons ensuite qu'il serait intéressant que *DBS* puisse résoudre des DisCSP avec contraintes n-aires.

La seconde partie de cette section porte sur des perspectives pratiques pour *DBS*. Les résultats présentés au chapitre 4 portent sur une simulation multi-robots. Nous aimerions savoir si les résultats se confirmeraient en utilisant des robots réels disposant chacun d'un processeur embarqué ainsi que des capacités de communication et d'observations qui lui sont propres. À défaut d'effectuer des tests sur des robots réels, il serait intéressant de pousser plus loin notre simulation en incluant davantage de problématiques pouvant survenir.

5.2.1 Perspectives théoriques

Nous détaillons, ici, plusieurs perspectives théoriques de nos travaux sur l'algorithme *DBS*. Pour chacune de ces perspectives, nous présentons son intérêt et les difficultés que nous risquons de rencontrer. De plus, il est indispensable que la complétude de *DBS* soit conservée et nous verrons donc l'éventuel impact de ces améliorations sur les propriétés de *DBS*.

Système ouvert

Un premier point d'amélioration serait que *DBS* puisse fonctionner dans un environnement ouvert, c'est-à-dire que l'algorithme puisse s'adapter, au cours de la recherche, à des ajouts et des suppressions de contraintes intra-agent, de contraintes inter-agents, de valeurs, de variables ou d'agents.

Ces ajouts ou retraits peuvent intervenir au cours de la résolution du problème et il faut qu'ils puissent être pris en compte sans remettre en cause la complétude de *DBS* et sans recommencer du début la recherche de solutions.

Afin de permettre la suppression d'une contrainte, il faudrait que la cause de l'inconsistance (par exemple à cause de la contrainte c_k) de chaque solution soit sauvegardée. Ainsi, si la contrainte c_k provoquant l'inconsistance d'une solution est supprimée, il serait alors possible de modifier le statut de cette solution à « *consistante* ». L'ajout d'une contrainte serait relativement facile à mettre en place. En effet, il faudrait uniquement demander aux agents impliqués par cette nouvelle contrainte de tester à nouveau leur solution courante pour rechercher une éventuelle inconsistance.

L'ajout d'une variable ou d'un agent ne remet pas en cause la complétude de l'algorithme. En effet, aucune contrainte du DisCSP ne peut porter sur ce nouvel agent ou cette nouvelle variable donc l'ensemble des solutions du DisCSP ne peut pas être modifié suite à ces ajouts. Cependant, la suppression d'un agent ou d'une variable nécessiterait davantage de précautions si l'on souhaite conserver la complétude de l'algorithme.

Priorité dynamique

L'une des limites de *DBS* porte sur la priorité statique des agents. En effet, la priorité des agents est calculée actuellement durant la phase d'initialisation. Puis, durant la phase de recherche de solutions, la priorité des agents reste inchangée. Nous pensons qu'une priorité dynamique des agents améliorerait les performances de *DBS* (à la fois au niveau du temps CPU, du nombres de messages échangés et du nombre de NCCC) pour de nombreux problèmes naturellement distribués.

La mise au point d'une priorité dynamique pour les agents nous paraît actuellement difficile à mettre en place dans *DBS*. En effet, l'utilisation unique des sessions comme contexte des messages rend difficile la mise en place d'une priorité dynamique. Dans des algorithmes tels que *AWC*, cela a été possible car le contexte d'un agent évolue en fonction des nogoods qu'il sauvegarde pour le reste de la résolution. Dans *DBS*, les agents sauvegardent une session, c'est-à-dire un entier, qui ne permet pas à lui seul d'obtenir une représentation précise de l'état du DisCSP si les priorités des agents venaient à être modifiées. Il serait éventuellement envisageable de mettre en place cette priorité dynamique en ajoutant à *DBS* une partie synchrone comme c'est le cas dans l'algorithme *AFC*. Cependant, nous préférerions conserver cet asynchronisme total entre les agents qui permet, selon nous, de réduire le temps où les agents se retrouvent en attente de traitements.

Afin d'obtenir une priorité dynamique des agents, une solution envisageable serait la suivante : si la recherche globale de solutions est trop longue, nous pourrions redémarrer la recherche globale en affectant une nouvelle priorité à chaque agent. Cette priorité tiendrait compte du nombre de messages de backtrack reçus et du nombre de contraintes vérifiées à la précédente résolution.

Contraintes n-aires

La majorité des algorithmes de DisCSP ne fonctionnent qu'avec des contraintes binaires. Lorsqu'il faut résoudre un DisCSP avec contraintes n-aires, les auteurs proposent d'utiliser au préalable un algorithme de transformation de DisCSP n-aires en un DisCSP binaire équivalent. Cependant cette transformation est extrêmement coûteuse et des variables et des agents sont ajoutés augmentant ainsi la complexité de la phase de recherche de solutions.

Actuellement, *DBS* ne gère que des contraintes inter-agents binaires. Lorsqu'une contrainte doit être vérifiée, c'est à l'agent de plus petite priorité contenant une variable sur laquelle porte la contrainte de vérifier cette contrainte. Nous pensons qu'il faut procéder de la même manière pour traiter le cas des contraintes n-aires. Ce serait donc uniquement à l'agent de plus petite priorité de vérifier la contrainte. Pour que cela soit possible, il faut que cet agent connaisse les valeurs des autres variables incluent dans cette contrainte. Pour cela, tout les agents de priorité supérieure à la sienne et inclus dans la contrainte doivent lui communiquer leurs valeurs via des messages ⟨OK?⟩. Notons qu'il est inutile que chaque agent connaisse la valeur des autres agents présents dans la contrainte vu que seul le plus petit agent vérifie la contrainte.

5.2.2 Perspectives pratiques

Nous détaillons dans cette section diverses pistes permettant d'évaluer de manière différente notre proposition. Nous commencerons par des pistes d'amélioration de l'application multi-robots que nous avons utilisée au chapitre 4. Puis nous verrons des perspectives portant sur la validation de *DBS* sur d'autres applications concrètes.

Amélioration de l'exploration multi-robots

Les résultats présentés au chapitre 4 concernant un problème d'exploration multi-robots ont été obtenus par simulation. Dans des conditions réelles où les agents sont des robots disposant d'un processeur embarqué et de capacités limitées de communication et d'observation, il se pourrait que les résultats obtenus différent des résultats obtenus en simulation. Nous pensons qu'il serait intéressant d'améliorer les points suivants de notre simulateur multi-robots :

- Lorsqu'un agent doit transmettre un message à un agent qui n'est pas à portée wifi, le message transite par d'autres agents qui servent de relais. Il faudrait donc tenir compte du nombre d'agents par lequel doit passer le message pour attribuer à ce message un certains coût (lui attribuer un temps d'emission par exemple). De plus, il serait intéressant, lorsque le nombre de robots devient important, d'utiliser des méthodes permettant de gérer efficacement l'acheminement des messages [Guizani *et al.*, 2011].
- Les agents évoluent dans nos simulations dans un monde en deux dimensions. Dans des conditions réelles, l'environnement serait en 3D et pourrait comporter des obstacles empêchant la communication wifi entre les agents. Il serait donc intéressant de pouvoir modéliser un tel monde contenant à la fois des obstacles laissant passer les communications wifi et des obstacles le bloquant. De plus, cet environnement pourrait être difficile pour les robots (en termes de déplacements) voire dynamique, il serait alors intéressant d'observer si la coopération entre les agents est toujours aussi efficace [Picard et Gleizes, 2005].
- Actuellement, le chemin de communication entre les agents ne doit jamais être brisé. Il serait intéressant d'observer le comportement des agents lorsque celui-ci se brise. Par exemple, nous pourrions obtenir le cas où la flotte d'agents serait coupée en deux. Chacun des deux groupes d'agents aurait alors sa propre vision de l'environnement. Les agents non reliés pourraient alors explorer des zones plus éloignées de la flotte d'agents initiale mais il se

pourrait aussi que des agents revisitent une zone déjà explorée par l'autre groupe. Lorsque ces deux groupes se retrouveraient à portée de communication, ils s'échangeraient alors leurs visions partielles de l'environnement. Nous pourrions donc faire des expérimentations en faisant varier le nombre d'agents par groupe.

Validation sur des applications

De nombreux articles traitent des problèmes d'emplois du temps distribués (Timetabling Problems $TTPs$). Les $TTPs$ regroupent différents types de problèmes réels dont :

- Distributed Meeting Sheduling [Brito et Meseguer, 2007, Brito, 2007]. Il s'agit d'un problème où le but est d'affecter des conférences à différents lieux. Chaque personne doit pouvoir se rendre aux conférences qu'elle a choisies (temps de trajet inclus).
- Distributed Stable Marriage Problem [Brito, 2007],
- Distributed Personnel Scheduling [Kaplansky et Meisels, 2007],
- Distributed University Timetabling Problem [Meisels et Kaplansky, 2002, Xiang et Zhang, 2008].

Nous aimerions évaluer *DBS* sur des problèmes d'emplois du temps universitaires distribués (Distributed University Timetabling Problem - $DisUTTP$). Il est possible de modéliser ce problème sous la forme d'un DisCSP de manière à ce que chaque enseignant soit un agent où alors que chaque Département/Institut soit un agent. Afin de résoudre un DisCSP dans lequel le problème local d'un agent est complexe, nous avons choisi d'attribuer à chaque agent un Département/Institut. De ce fait des algorithmes de résolutions de DisCSP pourront être utilisés.

Bibliographie

[Amilphastre, 1999] Amilphastre, J. (1999). Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes. Mémoire de D.E.A., Université Montpellier II.

[Armstrong et Durfee, 1997] Armstrong, A. et Durfee, E. (1997). Dynamic prioritization of complex agents in distributed constraint satisfaction problems. *In Proceedings of 15th IJCAI*, Nagoya, Aichi, Japan.

[Atkinson *et al.*, 2006] Atkinson, T., Bartak, R., Silaghi, M. C., Tuleu, E. et Zanker, M. (2006). Private and Efficient Stable Marriages (Matching) - a DisCSP Benchmark. *In Workshop on Distributed Constraint Satisfaction, ECAI.*

[Belaissaoui et Bouyakhf, 2004] Belaissaoui, M. et Bouyakhf, E. (2004). The Optimal distributed intelligent BackTracking. *RIST*, 14:55–77.

[Bellifemine *et al.*, 2000] Bellifemine, F., Giovani, C., Tiziana, T. et Rimassa, G. (2000). Jade programmer's guide. Rapport technique.

[Benelallam *et al.*, 2008] Benelallam, I., Belaissaoui, M., Ezzahir, R. et Bouyakhf, E. (2008). Dynamic Branch-and-Bound Distribué. *In quatrièmes Journées Francophones de Programmation par Contraintes (JFPC)*, Nantes, France.

[Bessière, 1991] Bessière, C. (1991). Arc-consistency in dynamic constraint satisfaction problems. *In AAAI-94*, pages 221–226.

[Bessière, 1994] Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190.

[Bessière *et al.*, 1999] Bessière, C., Freuder, E. et Régin, J. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148.

[Bessière *et al.*, 2001] Bessière, C., Maestre, A. et Meseguer, P. (2001). Distributed dynamic backtracking. *In* Silaghi, M., éditeur : *IJCAI'01 workshop on Distributed Constraint Reasoning*, pages 9–16, Seattle, Washington, USA.

[Bessière *et al.*, 2002] Bessière, C., Meseguer, P., Freuder, E. et Larrosa, J. (2002). On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 31:205–224.

[Bitner et Reingold, 1975] Bitner, J. R. et Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the ACM*, 18:651–656.

[Brito, 2007] Brito, I. (2007). *Distributed Constraint Satisfaction.* Thèse de doctorat, Institut d'Investigacio en Intelligencia Artificial, Bellaterra, Barcelona, Spain.

[Brito et Meseguer, 2007] Brito, I. et Meseguer, P. (2007). Distributed Meeting Scheduling. *Frontiers in Artificial Intelligence and Applications*, 163:38–45.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Robotics and Automation*, 2:14–23.

[Burke *et al.*, 2007] Burke, E. K., Mccollum, B., Meisels, A., Petrovic, S. et Quinqueton, R. (2007). A graph-Based Hyper-Heuristic for Educational Timetabling Problems. *European Journal of Operational Research*, 176:177–192.

[Chen et Beek, 2001] Chen, X. et Beek, P. V. (2001). Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81.

[Coltin *et al.*, 2010] Coltin, B., Liemhetcharat, S., Mericli, C., Tay, J. et Veloso, M. (2010). Multi-humanoid world modeling in standard platform robot soccer. *In Humanoid Robots*, Nashville.

[Dechter, 1990a] Dechter, R. (1990a). Enhancements schemes for constraint processing: backjumping, learning and cutest decomposition. *Artificial Intelligence*, 44:273–312.

[Dechter, 1990b] Dechter, R. (1990b). On the expresiveness of networks with hidden variables. *In AAAI*, pages 556–562, Boston, Massachusetts.

[Dechter, 2003] Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

[Dechter et Pearl, 1989] Dechter, R. et Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366.

[Demazeau, 1995] Demazeau, Y. (1995). From interactions to collective behaviour in agent-based systems. *In 1st European Conference on Cognitive Science*, pages 117–132.

[Doniec *et al.*, 2009] Doniec, A., Bouraqadi, N., Defoort, M., Le, V.-T. et Stinckwich, S. (2009). Distributed constraint reasoning applied to multi-robot exploration. *In 21st International Conference on Tools with Artificial Intelligence (ICTAI)*.

[Doniec *et al.*, 2008] Doniec, A., Mandiau, R., Piechowiak, S. et Espie, S. (2008). Anticipation based on constraint processing in a multi-agent context. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 17:339–361.

[Doniec *et al.*, 2005] Doniec, A., Piechowiak, S. et Mandiau, R. (2005). A DisCSP solving algorithm based on sessions. *In* Russell, I. et Markov, Z., éditeurs : *FLAIRS'05 : Recent advances in artificial intelligence: Proceedings of the eighteenth International Florida Artificial Intelligence Research Society Conference*, pages 666–670, Menlo Park, California.

[Dorigo, 1992] Dorigo, M. (1992). Optimization, learning and natural algorithms. Mémoire de D.E.A., Politecnico di Milano, Italie.

[Drogoul, 1993] Drogoul, A. (1993). De la simulation multi-agent à la résolution collective de problèmes. Une étude de l'émergence de structures d'organisation dans les systèmes multi-agents. Mémoire de D.E.A., Paris VI.

[Erman *et al.*, 1980] Erman, L. D., hayes Roth, F., Lesser, V. R. et Reddy, D. R. (1980). The Hearsay-II speech-understanding system : Integrating knowledge to resolve uncertainty. *Computing Surveys*, 2(12):213–253.

[Ezzahir, 2008] Ezzahir, R. (2008). *Traitement des problèmes de satisfaction et d'optimisation de contraintes distribués*. Thèse de doctorat, Université Mohammed V - Agdal, Maroc.

[Ezzahir *et al.*, 2008] Ezzahir, R., Bessière, C., Benelallam, I., Bouyakhf, E.-H. et Belaissaoui, M. (2008). Dynamic backtracking for distributed constraint optimization. *In ECAI'08*, pages 901–902, Patras, Greece.

[Ezzahir *et al.*, 2009] Ezzahir, R., Bessière, C., Wahbi, M., Benelallam, I. et Bouyakhf, E. H. (2009). Asynchronous Inter-Level Forward-Checking for DisCSPs. *In CP'09*, Lisbon, Portugal.

[Ferber, 1995] Ferber, J. (1995). *Les Systèmes Multi-agents, Vers une intelligence collective*. Paris, InterEditions.

[Fernandez *et al.*, 2002] Fernandez, C., Béjar, R., Krishnamachari, B. et Gomes, C. (2002). Communication and computation in distributed CSP algorithms. *In Lecture Notes in Computer Science*, pages 664–679.

[FIPA, 2002] FIPA (2002). Acl message structure specification. Rapport technique.

[Freuder, 1982] Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, 29:24–32.

[Geller et Veksler, 2005] Geller, F. et Veksler, M. (2005). Assumption-based pruning in conditional csp. *In Principles and Practice of Constraint Programming*, pages 241–255.

[Ginsberg, 1993] Ginsberg, M. (1993). Dynamic backtracking. *Artificial Intelligence Research*, 1:25–46.

[Guessoum, 1998] Guessoum, Z. (1998). Dima: Une plate-forme multi-agents en smalltalk. *Revue Objet*, 3:393–410.

[Guizani *et al.*, 2011] Guizani, B., el Ayeb, B. et Koukam, A. (2011). Hierarchical cluster-based link state routing protocol for large self-organizing networks. *In HPSR'11*, pages 203–208.

[Hamadi, 1999a] Hamadi, Y. (1999a). Optimal distributed arc-consistency. *In principles and practice of constraint programming*.

[Hamadi, 1999b] Hamadi, Y. (1999b). *Traitement des problèmes de satisfaction de contraintes distribués*. Thèse de doctorat, Université de Montpellier II, France.

[Hamadi *et al.*, 1998] Hamadi, Y., Bessière, C. et Quinqueton, J. (1998). Backtracking in distributed constraint networks. *In ECAI'98*, pages 219–223, Brighton, United Kingdom.

[Haralick et Elliot, 1980] Haralick, R.-M. et Elliot, G.-L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313.

[Hentenryck *et al.*, 1972] Hentenryck, P. V., Deville, Y. et Teng, C.-M. (1972). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321.

[Hirayama et Yokoo, 2000] Hirayama, K. et Yokoo, M. (2000). An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. *In ICMAS-2000*.

[Hirayama *et al.*, 2000] Hirayama, K., Yokoo, M. et Sycara, K. (2000). The phase transition in distributed constraint satisfaction problems: First results. *Principles and Practice of Constraint Programming*, 1894/2000:515–519.

[Hirayama *et al.*, 2004] Hirayama, K., Yokoo, M. et Sycara, K. (2004). An easy-hard-easy cost profile in distributed constraint satisfaction. *IPSJ Journal*, 45:2217–2225.

[Kaplansky et Meisels, 2007] Kaplansky, E. et Meisels, A. (2007). Distributed personnel scheduling – negotiation among scheduling agents. *Annals of Operations Research*, 155:227–255.

[Le *et al.*, 2009] Le, V. T., Bouraqadi, N., Moraru, V., Stinckwich, S. et Doniec, A. (2009). Making networked robot connectivity-aware. *In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Lecoutre *et al.*, 2007] Lecoutre, C., Sais, L. et Vion, J. (2007). Using SAT Encodings to Derive CSP Value Ordering Heuristics. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:169–186.

[Mackworth, 1977] Mackworth, A.-K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

[Meisels et Kaplansky, 2002] Meisels, A. et Kaplansky, E. (2002). Scheduling Agents - Distributed Timetabling Problems (DisTTP). *In 4th PATAT*.

[Meisels *et al.*, 2002] Meisels, A., Kaplansky, E., Razgon, I. et Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms. *In Workshop on Distributed Constraint Reasoning, in AAMAS-2002*.

[Meisels et Zivan, 2007] Meisels, A. et Zivan, R. (2007). Asynchronous Forward-checking for DisCSPs. *Constraints*, 12(1):131 – 150.

[Minton *et al.*, 1992] Minton, S., Johnston, M. D., Philips, A. B. et Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205.

[Mohr et Henderson, 1986] Mohr, R. et Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233.

[Monier *et al.*, 2010a] Monier, P., Belaissaoui, M., Piechowiak, S. et Mandiau, R. (2010a). Résolution de CSP Distribués avec problémes locaux complexes. *In RFIA'2010, Reconnaissance des Formes et Intelligence Artificielle, Actes du 17éme congrés francophone AFRIF-AFIA*.

[Monier *et al.*, 2010b] Monier, P., Doniec, A., Piechowiak, S. et Mandiau, R. (2010b). Metrics for the Evaluation of DisCSP: Some Experiments of multi-robot exploration. *In IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2010)*, pages 370–373, Toronto, Canada.

[Monier *et al.*, 2011] Monier, P., Doniec, A., Piechowiak, S. et Mandiau, R. (2011). Comparison of DCSP Algorithms: A Case Study for Multi-agent Exploration. *In 9th International Conference on Practical Applications of Agents and Multi-Agent Systems*.

[Monier *et al.*, 2009a] Monier, P., Piechowiak, S. et Mandiau, R. (2009a). A complete algorithm for DisCSP: Distributed Backtracking with Sessions (DBS). *In Second International Workshop on: Optimisation in Multi-Agent Systems (OptMas), Eigth Joint Conference on Autonomous and Multi-Agent Systems (AAMAS 2009)*, Budapest, Hungary.

[Monier *et al.*, 2009b] Monier, P., Piechowiak, S. et Mandiau, R. (2009b). Multi-Agent Reasoning based on Distributed CSP using Sessions : DBS. *In 7th International Conference on Practical Applications of Agents and Multi-Agent Systems*, Salamanca, Spain.

[Morin, 1977] Morin, E. (1977). *La méthode : la nature de la Nature. Le seuil*. Collection Points.

[Muscalagiu, 2005] Muscalagiu, I. (2005). The effect of flag introduction on the explosion of nogood values in the case of abt family. *Multi-Agent Systems and Applications IV*, 3690/2005: 286–295.

[Nguyen et Deville, 1995] Nguyen, T. et Deville, Y. (1995). A distributed arc-consistency algorithm. *In Science of Computer Programming*, pages 227–250.

[Omomowo *et al.*, 2008] Omomowo, B., Arana, I. et Ahriz, H. (2008). DynABT: Dynamic Asynchronous Backtracking for Dynamic DisCSPs. *In Artificial Intelligence: Methodology, Systems, and Applications*.

[Picard et Gleizes, 2005] Picard, G. et Gleizes, M.-P. (2005). Cooperative self-organization: Designing robust and adaptive robotic collectives. *In Third European Workshop on Multi-Agent Systems (EUMAS'05)*, Brussels, Belgium.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence ISSN 0824-7935*, 9:268–299.

[Prosser, 1996] Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artficial Intelligence*, 81:81–109.

[Rooker et Birk, 2007] Rooker, M. et Birk, A. (2007). Multi-robot exploration under the constraints of wireless networking. *In Control Engineering Practice*.

[Ros *et al.*, 2009] Ros, R., Arcos, J. L., de Mantaras, R. L. et Veloso, M. (2009). A case-based approach for coordinated action selection in robot soccer. *Artificial Intelligence*, 173: 1014–1039.

[Rosenschein, 1985] Rosenschein, J. S. (1985). Rational interaction : cooperation among intelligent agents. Mémoire de D.E.A., Stanford University.

[Russell et Norvig, 1995] Russell, S. et Norvig, P. (1995). *Artificial intelligence : A Modern Approach*. Prentice-Hall.

[Sabin et Freuder, 1996] Sabin, D. et Freuder, E. C. (1996). Configuration as composite constraint satisfaction. *In First Artificial Intelligence and Manufacturing Research Planning Workshop*.

[Schiex *et al.*, 1997] Schiex, T., Fargier, H. et Verfaillie, G. (1997). Problèmes de satisfaction de contraintes valuées. *Revue d'intelligence Artificielle (RIA)*, 11:339–373.

[Sheng *et al.*, 2006] Sheng, W., Yang, Q., Tan, J. et Xiang, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, 54:945–955.

[Sousa et Ramos, 1999] Sousa, P. et Ramos, C. (1999). A distributed architecture and negotiation protocol for scheduling in manufacturing systems. *Computers in Industry*, 38:103–113.

[Tisue et Wilensky, 2004] Tisue, S. et Wilensky, U. (2004). Netlogo: Design and implementation of a multi-agent modeling environment. *In Agent 2004 conference*.

[Tsuruta et Shintani, 2000] Tsuruta, T. et Shintani, T. (2000). Scheduling meetings using distributed valued constraint satisfaction algorithm. *In 14th European Conference on Artificial Intelligence (ECAI)*, pages 383–387, Berlin, Germany.

[Vazquez et Malcolm, 2004] Vazquez, J. et Malcolm, C. (2004). Distributed multirobot exploration maintaining a mobile network. *In 2nd international IEEE Conference on Intelligent Systems*.

[Veron et Aldanondo, 2000] Veron, M. et Aldanondo, M. (2000). Yet Another to CCSP for configuration problem. *In European Conference on Artificial Intelligence, Workshop on Configuration*, pages 59–62, Berlin, Allemagne.

[Vion, 2007] Vion, J. (2007). CSP4J: a Black-Box CSP Solving API for Java. *In Second International CSP Solver Competition*, pages 75–88, Nantes, France.

[Xiang et Zhang, 2008] Xiang, Y. et Zhang, W. (2008). Distributed university timetabling with multiply sectioned constraint networks. *In 21st International FLAIRS*.

[Yamauchi, 1998] Yamauchi, B. (1998). Frontier-based exploration using multiple robots. *In Proceedings of the Second International Conference on Autonomous Agents (Agent'98)*.

[Yokoo, 1995] Yokoo, M. (1995). Asynchronous weak-commitment search for solving distributed constraint satisfaction problem. *In Lecture Notes In Computer Science*, volume 976, pages 88–102.

[Yokoo, 2001] Yokoo, M. (2001). *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems*. Springer Verlag.

[Yokoo *et al.*, 1992] Yokoo, M., Durfee, E., Ishida, T. et Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. *In Conference on Distributed Computing Systems*.

[Yokoo *et al.*, 1998] Yokoo, M., Durfee, E., Ishida, T. et Kuwabara, K. (1998). The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685.

[Yokoo et Hirayama, 1996] Yokoo, M. et Hirayama, K. (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems. *In Second International Conference on Multiagent Systems (ICMAS-96)*, pages 401–408, Kyoto, Japan.

[Yokoo et Hirayama, 1998] Yokoo, M. et Hirayama, K. (1998). Distributed constraint satisfaction algorithms for complex local problems. *In Third International Conference on Multiagent Systems (ICMAS-98)*, pages 372–379, Paris, France.

[Yokoo *et al.*, 1990] Yokoo, M., Ishida, T. et Kuwabara, K. (1990). Distributed constraint satisfaction for dai problems. *In 10th International Workshop on Distributed Artificial Intelligence*.

Résumé

Le formalisme CSP (Problème de Satisfaction de Contraintes) permet de représenter de nombreux problèmes de manière simple et efficace. Cependant, une partie de ces problèmes ne peut être résolue de manière classique et centralisée. Les causes peuvent être diverses : temps de rapatriement des données prohibitif, sécurité des données non garantie, etc. Les CSP Distribués (DisCSP), domaine intersectant celui des SMA et des CSP, permettent de modéliser et de résoudre ces problèmes naturellement distribués. Les raisonnements intra-agent et inter-agents sont alors basés sur un ensemble de relations entre différentes variables. Les agents interagissent afin de construire une solution globale à partir des solutions locales.

Nous proposons, dans ce travail, un algorithme de résolution de DisCSP nommé *Distributed Backtracking with Sessions* (*DBS*) permettant de résoudre des DisCSP où chaque agent dispose d'un problème local complexe. *DBS* a la particularité de ne pas utiliser de nogoods comme la majorité des algorithmes de résolution de DisCSP mais d'utiliser à la place des sessions. Ces sessions sont des nombres permettant d'attribuer un contexte à chaque agent ainsi qu'à chaque message échangé durant la résolution du problème. Il s'agit d'un algorithme complet permettant l'utilisation de filtres sur les messages échangés sans remettre en cause la preuve de complétude. Notre proposition est évaluée, dans les cas mono-variable et multi-variables par agents, sur différents benchmarks classiques (les problèmes de coloration de graphes distribués et les DisCSP aléatoires) ainsi que sur un problème d'exploration en environnement inconnu.

Mots-clés: Système multi-agent, Agent, CSP, CSP distribué, Session, Contrainte

Abstract

The CSP formalism (Constraint Satisfaction Problem) can represent many problems in a simple and efficient way. However, some of these problems cannot be solved in a classical and centralized way. The causes can be multiple: prohibitive repatriation time, unsecured data and so on. Distributed CSP (DisCSP), domain intersecting MAS and CSP, are used to model and to solve these problems. The intra-agent and inter-agent reasonning are so based on a set of relation between different variables. The agents interact in order to build a global solution from local solutions.

We propose, in this work, an algorithm for solving DisCSP named *Distributed Backtracking with Sessions* (*DBS*) which allows to solve DisCSP where each agent owns a complex local problem. *DBS* has the particularity to not use nogoods like the majority of algorithms for solving DisCSP but to use instead of sessions. These sessions are numbers which allow to assign a context to each agent and each message exchanged during the resolution of the problem. *DBS* is a complete algorithm which allows the use of filters on messages exchanged without affecting the proof of completeness. Our proposal is evaluated, for mono-variable and multi-variables per agents problems, on different classical benchmarks (distributed graph coloring problems and random DisCSP) and on an unknown environment exploration problem.

Keywords: Multi-agent system, Agent, CSP, Distributed CSP, Session, Constraint