République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université abderahmane mira -Béjaia-
Faculté des sciences exactes
Département informatique

MÉMOIRE DE MASTER RECHERCHE

Filière Informatique

Option :Intelligence Artificielle

Résolution des problèmes de satisfaction de contraintes distribués (DisCSP) par les systèmes muti-agents (SMA)

Réalisé par :
Mme. Benlala Wissam
Mme. Abbas Fatma

Encadré par :
Mr. Amroun Kamal

Membres de jury :

Mme. S.ALOUI : - Maître de Conférence A
Mr. M.MOKTEFI : - Maître Assistant A

Promotion : 2020/2021

Remerciement

Nous tenons tout d'abord à remercier **DIEU le tout puissant et miséricordieux**, qui nous a donné la force et la patience d'accomplir ce Modeste travail.

En second lieu, nous tenons à remercier notre encadreur Monsieur **AMROUN KAMAL** pour son assistance, disponibilité, orientations et conseils.

Nos vifs remerciements vont également aux membres du jury **Mme S.ALOUI** et **Mr M.MOKTEFI** qui nous ferons l'honneur de lire et d'evaluer ce travail.

Nous tenons aussi à remercier tous les **enseignants** qui ont contribué à notre formation.

Enfin, Nous tenons également à remercier l'équipe de **DENZER TECHNOLOGIE** d'avoir accepté de nous accueillir durant notre période de stage.

Dédicace

Je tiens en tout premier lieu à remercier mes très cher PARENTS pour leurs efforts fournit pour mon éducation et mon bien être.

A Mes chers frère et soeurs ,

A Mes Amis ,

A Ma binôme : merci d'avoir partager ce parcoure avec moi, Je te souhaite un avenir plein de réussite et de sérénité.

Wissam

Dédicace

A ma MAMAN pour ses sacrifices, tendresse et soutien tout au long de mes études.
A ma chère soeur, copine et binôme,
A tous ceux qui m'aiment.
Fatima

Sommaire

Liste des tableaux

Table des figures

Liste des abbreviations

ABT Asynchrones Backtracking
AC Consistance darc
AFC Asynchrone Forward Checking
AWC Asynchronous Weak-Commitment
BJ BackJumping
BTD Backtracking on Tree-Decomposition
BT BackTrack
CD-BJ Algorithme Conflict Directed BackJumping
ConBT Concurrent Backtracking
CPA Current Partial Assignment
CSA Concurrent Search Algorithms
CSOP Constraint Satisfaction Optimisation Problème
CSP Problème de satisfaction de contraintes
DAC Consistance darc directionnelle
DBT Distributed Synchronous Backtracking
DiBT Dynamic Backtracking
DisCP Problèmes de satisfaction de contraintes distribués
DPC Consistance de chemin directionnelle
FC Forward Checking
IAD Intelligence artificielle distribuée
IA Intelligence artificielle
MAC Maintaining Arc Consistency
NC Consistance de nud
P2P Architecture pair-a-pair
PC Consistance de chemin
RPC Consistance de chemin restreinte
SD Système distribué
SMA Système Multi-Agents
VCSP Valued Constraints Satisfaction Problème

Introduction générale

La programmation par contraintes est un domaine de l'informatique qui est basée sur la puissance de son Framework nommé Problème de Satisfaction de Contraintes (**CSP**). Un **CSP** est un cadre général qui peut formaliser de nombreux problèmes combinatoires du monde réel. Divers problèmes d'intelligence artificielle peuvent être naturellement modélisés comme des **CSP**.

Dans ce mémoire, nous présentons un formalisme appelé problème de satisfaction de contraintes distribué **(CSP distribué)**. Un **CSP distribué** est un problème de satisfaction de contraintes **(CSP)** dans lequel les variables et les contraintes sont réparties entre plusieurs **agents** automatisés. Un **CSP** est un problème pour trouver une affectation cohérente des valeurs aux variables. De même, divers problèmes d'application dans l'intelligence artificielle distribuée **IAD** qui sont soucieux de trouver une combinaison cohérente des actions d'agents peuvent être formaliser comme CSP Distribué.
Un problème de satisfaction de contraintes distribué est composé d'un groupe d'agents, où chaque agent a le contrôle de certains éléments d'information sur tout le problème, c'est-à-dire les variables et les contraintes. Chaque agent est propriétaire de sa contrainte locale. Les variables des différents agents sont liées par des contraintes.

Les Systèmes Multi-Agents **(SMA)** permettent de résoudre de nombreux problèmes naturellement distribués tels que les problèmes de N-Reines et coloriage, ect. Afin de résoudre ces problèmes, les agents interagissent pour faire émerger une solution globale à partir des solutions locales de chaque agent. Le formalisme des problèmes de satisfaction de contraintes distribués **(DisCSP)** permet de représenter de nombreux problèmes d'une manière simple et efficace. Chaque agent dispose d'un CSP local et les différents agents, reliés par des contraintes inter-agents, communiquent par envoi de messages pour résoudre le **(DisCSP)**.

Lors de la résolution des CSP Distribués (**DisCSP**), les agents échangent des messages sur les affectations de variables et les conflits de contraintes. Plusieurs algorithmes distribués pour résoudre les DisCSP ont été conçus au cours des deux dernières décennies. Ils peuvent être divisés en deux groupes principaux : **asynchrones** et **synchrones** algorithmes. La première catégorie est constituée d'algorithmes dans lesquels les agents attribuent des valeurs à leurs variables de manière synchrone et séquentielle. La deuxième catégorie est constituée des algorithmes dans lesquels le processus de proposition de valeurs aux variables et d'échange de ces propositions est effectué de manière asynchrone entre les agents. Dans la première catégorie, les agents n'ont pas à attendre les décisions des autres, alors qu'en général un seul agent a le privilège de prendre une décision dans les algorithmes synchrones.

L'objectif de notre travail est de proposer un algorithme **asynchrone** pour la résolution des problèmes de satisfaction de contraintes distribués (**DisCSP**).pour cela, nous avons comparé entres deux algorithmes à fin d'arriver à choisir l'algorithme adéquat.

Ce mémoire est structuré autour d'une introduction générale, cinq chapitres, conclusion générale et une bibliographie :

Dans le premier chapitre intitulé **"Problèmes de satisfaction de contraintes CSP"**, nous exposons le formalisme **CSP** et son extension. Nous concluons ce chapitre avec les méthodes de résolution des CSP.

Dans le deuxième chapitre intitulé **"Généralités sur les systèmes distribués"**, nous présentons différentes caractéristiques et objectifs d'un système distribué suivie de quelques définitions ainsi qu'une vue sur l'architecture distribuée. Nous concluons ce chapitre par une présentation de quelques algorithmes distribués, ainsi avantages et inconvénients des systèmes distribués.

Dans le troisième chapitre, intitulé **"Système multi-agents"** nous exposons brièvement les généralités sur les système multi-agents, puis nous présentons les caractéristiques des SMA et l'organisation dans les SMA. Nous concluons ce chapitre par présentation de certains avantages de l'utilisation des SMA.

Dans le quatrième chapitre intitulé **"Modélisation des CSP par les SMA"**, nous présentons les généralités sur les **DisCSP**. Nous concluons ce chapitre par la présentation des algorithmes de résolution des DisCSP.

Dans le cinquième et dernier chapitre intitulé **"Implémentation et réalisation"** Nous exposons les problèmes traités ainsi que l'environnement de travail et les outils utilisés, puis nous présentons l'exécution des algorithmes sur les problèmes traités .Nous concluons ce chapitre par les résultats obtenus et une évaluation.

Chapitre I

Problème de Satisfaction de Contrainte CSP

Introduction

Un grand nombre de problèmes issus de différents domaines de l'informatique peuvent être modélisés comme des problèmes de satisfaction de contraintes **(CSP)**.

Le formalisme **CSP**, tout en permettant de représenter très simplement des problèmes, est aussi puissant dans la mesure où il est possible de représenter un éventail très large de problèmes. En outre, les problèmes de satisfaction de contrainte sont des problèmes mathématiques où l'on cherche des états ou des objets satisfaisant un certain nombre de contraintes ou de critères.

Dans ce chapitre, Nous rappelons d'abord le formalisme CSP et les définitions de base. Ensuite, nous présentons les principales méthodes de résolution ainsi les techniques de simplification par filtrage et on finira avec Les méthodes de décomposition structurelles.

I.1 Formalisme CSP

Un problème de satisfaction de contraintes est défini par un ensemble de variables, associée chacune à un domaine discret de valeurs de taille finie et par un ensemble de contraintes qui mettent en relation les variables et définissent des combinaisons de valeurs compatibles.
Voici quelques définitions [46] :

Définition 1.1

Un problème de satisfaction de contraintes **(CSP)** est un quadruplet (X, D, C, R), où [46] :

— $X = \{X_1, ..., X_n\}$ est un ensemble de n variables

— $D = \{D_1, ..., D_n\}$ est un ensemble de n domaines D_i finis, Chaque domaine est associé à une variable X_i.

— $C = \{C_1, ..., C_m\}$ est un ensemble de m contraintes, Chaque contrainte c_i est définie par un ensemble de n_i variables $X = \{X_{i1}, ..., X_{ini}\} \exists X$.

— $R = \{R_1, .., R_m\}$ est un ensemble de m relations. Chaque relation R_i définit l'ensemble des n_i-uplets sur $D_{i1}...D_{ini}$ autorisés par la contrainte c_i.

Définition 1.2 (Contrainte)

Une contrainte est une relation logique (une propriété qui doit être vérifiée) entre différentes variables, chacune prenant ses valeurs dans un ensemble donné, appelé domaine. Une contrainte restreint les valeurs que peuvent prendre simultanément les variables, elle peut être :

— Implicite, définie par une expression logique. Par exemple : $C = X + Y < 4$

— Explicite, définie par une relation composée de toute les tuples autorisés. Par exemple : pour un domaine $D = 1, 2, 3$ de X et Y, la contrainte précédente est $R = (1, 1), (1, 2), (2, 1)$. Les variables sur lesquelles porte la contrainte C_i sont appelées Portée de la contrainte, noté $S(Ci)$.

Définition 1.3 (Arité)

L'arité d'une contrainte $C_i = X_{i1}, ..., X_{ini}$ est le nombre i_{ni} de variables sur lesquelles porte C_i . Un CSP est dit binaire si pour toute contrainte $C_i \in C$, l'arité de C_i est au plus de 2, sinon il est n-aire.

Définition 1.4 (Réseaux de contraintes)

Un réseau de contraintes associé à un CSP est un graphe (pour les CSP binaire) ou un hypergraphe (pour les CSP n-aire) dont les noeuds sont les variables et les arêtes sont les contraintes.

Définition 1.5 (Instanciation)

Etant donné un CSP, $P = (X, D, C, R)$ et soit $Y \subset X$ un sous ensemble de variables de X. On appelle instanciation de Y l'association de chaque variable y de Y a une valeur de $D(y)$ (avec $D(y)$ est le domaine de la variable y).

Définition 1.6 (Instanciation consistante)

Etant donné un CSP, $P = (X, D, C, R)$ et soit $Y \subset X$ un sous ensemble de variables de X. Une instanciation des variables de Y sur D est dite consistante si et seulement si elle satisfait toutes les contraintes portant sur ses variables.

Définition 1.7 (Solution d'un CSP)

On appelle une solution d'un CSP $P = (X, D, C, R)$, l'instanciation consistante des variables de X sur D. Une instance CSP est dite consistante si elle possède au moins une solution.

I.2 Extension des CSP

Dans les CSP classiques, la notion de contrainte est définie au sens strict du terme, dans laquelle une combinaison de valeurs peut être autorisée ou interdite, pour de nombreux problèmes réels, le concept de contrainte ne traduit en fait qu'une préférence pour certains tuples de valeurs, ou encore un cout de violation. Devant le constat de l'inadéquation de la modélisation de ce genre de relations de préférence par des contraintes dures, plusieurs extensions au cadre CSP ont été proposées.
Nous nous intéressons ici au cadre des problèmes de la modélisation dans lesquels entre en jeu la notion de coût, de préférence, etc. Des extensions ont été proposées :

- **Max-CSP** : qui consiste à trouver l'affectation totale qui viole le moins de contraintes possibles pour un problème sur contraint (qui n'a pas de solution).
- **VCSP** : (Valued Constraints Satisfaction Problem) est un problème d'optimisation, ou les CSP valués consistent à chercher l'affectation totale qui minimise la somme des poids des contraintes.
- **CSOP** : (Constraint Satisfaction Optimisation Problem) Il consiste à chercher la solution du CSP qui maximise une fonction objectif, lorsque les différentes solutions d'un problème sous contraint ne sont pas toutes équivalentes.

I.3 Méthodes de résolution

Il existe de nombreuses méthodes et techniques pour résoudre les instances de CSP. Ces méthodes et techniques peuvent être classées comme suit [46] [54] :

I.3.1 Techniques de simplification par filtrage

L'objectif de ces techniques, n'est pas de trouver la solution d'un CSP mais d'éviter l'exploration des régions de l'espace de recherche qui ne contiennent pas de solution.

I.3.1.1 Filtrage

Le filtrage a pour objectif de simplifier l'instance CSP à résoudre afin de réduire la taille de l'espace de recherche à explorer. Plusieurs techniques de vérification de consistance locale ou globale et de propagation de contraintes ont été développées, ces techniques permettent de réduire la taille des domaines des variables en enlevant certaines valeurs qui ne peuvent pas figurer dans une solution. La propriété de consistance est atteinte lorsqu'aucune valeur d'un domaine ou tuple d'une relation ne pourra être supprimé.

I.3.1.1.1 Consistance de noeud NC

Une instance CSP est consistante de noeud si pour toute variable X_i et pour toute valeur d de D_i, l'affectation partielle $X_i = d$ satisfait toutes les contraintes unaires du problème. Le principe de cette consistance a pour but de supprimer de chaque domaine D_i associé à la variable X_i, toute valeur qui viole une contrainte unaire.

I.3.1.1.2 Consistance d'arc AC

Un domaine $D(_i)$ de la variable X_i est arc consistant si et seulement si pour chaque valeur $a \in D_i$, et pour toute variable X_j telle qu'il existe une contrainte C_{ij} entre X_i et X_j alors il existe une valeur b de D_j telle que (a, b) satisfait la contrainte C_{ij}.
Un CSP binaire est arc consistant si tous les domaines des variables sont arc consistants. Le filtrage par consistance d'arc consiste à supprimer des valeurs des domaines des variables qui ne satisfont pas la propriété de consistance d'arc.

Le premier algorithme implémentant la consistance d'arc est AC1 (algorithme 2) proposé par **Mackworth** [50].

Algorithm 1 Réviser (C_k)

1: $C_k = (X_i, X_j)$
2: **for** tout $d_i \in D_i$ **do**
3: **if** $\nexists d_j \in D_j \mid (d_i, d_j) \in R_k$ **then**
4: supprimer d_i de D_i
5: $modification \longleftarrow True$
6: **end if**
7: **end for**
8: retourner *modification*

FIGURE I.1 – Algorithme Réviser (C_k) [46].

Algorithm 2 AC1

1: **repeat**
2: $modification \longleftarrow false$
3: **for** chaque contrainte C_k **do**
4: $modification \longleftarrow$ Réviser$(C_k) \vee modification$
5: **end for**
6: **until** *not modification*

FIGURE I.2 – Algorithme AC1 [46].

L'inconvénient de **AC1** est que si une valeur d'un domaine est supprimée alors toutes les contraintes sont révisées même si le domaine modifié n'a aucune influence sur la plus part de ces contraintes. Une amélioration a été proposée dans **AC3** par Mackworth [50], elle consiste à ne pas réappliquer la procédure **Réviser** à toutes les contraintes, mais uniquement à celles susceptibles d'être affectées par la suppression d'une valeur d'un domaine.
Pour cela, cet algorithme utilise une file pour mémoriser ces contraintes.

Algorithm 3 AC3

1: $L \longleftarrow (X_i, X_j), i \neq j$
2: **while** $L \neq \emptyset$ **do**
3: choisir et supprimer de L un couple (X_i, X_j)
4: **if** Réviser$((X_i, X_j))$ **then**
5: $L \longleftarrow L \cup \{(X_k, X_i)\}$ / $\exists$ une contrainte liant X_k et X_i
6: **end if**
7: **end while**

FIGURE I.3 – Algorithme AC3 [46].

I.3.1.1.3 Consistance d'arc directionnelle (DAC)

L'objectif de l'arc consistance directionnelle est d'affaiblir l'arc consistance de telle sorte à réviser un arc dans une seule direction. Un CSP est arc consistant directionnel pour un ordre donné sur les variables (X_1,..., X_n) si et seulement si toute variable X_i est arc consistante avec toute autre variable X_j (avec $i < j$). L'algorithme 4 de l'arc consistance directionnelle suppose que les variables du problème sont ordonnées.

En général, on applique cet algorithme directionnel de la racine aux feuilles.

Algorithm 4 DAC(G)

```
1: for j←|nœuds(G)| downto 1 do
2:    for chaque arc (i,j) dans G | i < j do
3:       Réviser(i,j)
4:    end for
5: end for
```

FIGURE I.4 – Algorithme DAC(G) [46].

I.3.1.1.4 Consistance de chemin PC

La consistance de chemin est une forme de consistance plus forte que l'arc consistance. Un CSP est chemin-consistant PC si toute affectation consistante sur deux variables peut être étendue de manière consistante à une troisième variable.
Soit un CSP $P = (X, D, C)$, une paire de variable (X_i ,X_j) est PC ssi $\forall X_k \in X$, $\forall$(a, b) $\in R_{ij}$, $\exists c \in D_k$ |$(a, c) \in R_ik$ et $(b, c) \in R_{jk}$. Un CSP est PC si et seulement si $\forall X_i$, $X_j \in X$, (X_i, X_j) est PC. cette définition est valable uniquement pour les CSPs binaires.

I.3.1.1.5 Consistance de chemin directionnelle DPC

Cette forme de consistance nécessite un ordre sur les variables. Un CSP est chemin consistant directionnel DPC relativement à un ordre (X_1,...,X_n) si $\forall(X_i,X_j)$ et $\forall(a, b) \in R_{ij}$, $\forall k, k > i, j, \exists c \in D_k$ telle que $(a, c) \in R_{ik}$ et $(b, c) \in R_{jk}$.

I.3.1.1.6 Consistance de chemin restreinte RPC

Cette forme de consistance a été introduite par Berlandier [14]. Son objectif est d'avoir une consistance plus forte que la consistance d'arc.
Soit un CSP $P = (X, D, C)$, $\forall X_i \in X, \forall a \in D_i \mid \forall X_j \in X$ tel que a a un support unique b dans D_j , $\forall X_k \in X$ telle que C_{ik}, $C_{jk} \in C, \exists c \in D_k$ telle que $(a, c) \in Rel(C_{ik}) \bigwedge$(b,c) $\in Rel(C_{jk})$.
Avec cette forme de consistance, la structure du problème n'est pas modifiée. En effet, RPC n'efface pas de paires de valeurs et donc aucune contrainte n'est ajoutée ou modifiée dans le CSP.

I.3.1.1.7 k-consistance

Les trois niveaux de consistance (NC, AC et PC) ont été généralisés. Un CSP P est k-consistant [33] si toute instanciation consistante de $k-1$ variables différentes peut être étendue en une instanciation consistante avec une autre variable supplémentaire. P est fortement k-consistant ssi P est j-consistant $\forall$j $\leq k$.

I.3.1.1.8 (i,j)-Consistance

La (i,j)-consistance [34] généralise la k-Consistance. un CSP P est (i,j)-consistant ssi toute instanciation consistante de i variables peut être étendue à j nouvelles variables. P est (i,j)-fortement consistant, ssi il est (i,j)-consistant pour tout $k \leq i$.

I.3.1.1.9 Consistance inverse

La (1,$k-1$)-consistance est appelée k-consistance inverse [35]. On supprime les valeurs qui ne peuvent pas être étendues de manière consistante à $k-1$ nouvelles variables.
Une forme particulière de consistance inverse est la consistance de voisinage inverse (NIC) [35] : elle consiste a supprimer les valeurs de la variable X_i ne pouvant pas être étendues de manière consistante à toutes les variables directement reliées à X_i.

I.3.1.2 Filtrage des CSPs n-aires

Dans cette sous-section, nous présentons l'arc consistance généralisée, l'algorithme **GAC**, ainsi la transformation des CSPs n-aires en CSPs binaires et leur résolution.

I.3.1.2.1 Arc Consistance généralisée GAC

Pour décrire cette forme de consistance, nous allons d'abords présenter la notion de **support**. Un **support** pour une contrainte C_i est un ensemble d'instanciations pour exactement l'ensemble des variables de C_i (Scope(C_i)) tel que C_i est satisfaite. Un support d'une contrainte C_i qui inclut $X_j = v$ est appelé **support** de $X_j = v$ dans C_i.

Une contrainte C_i est arc consistante généralisée **(GAC)** s'il existe des **supports** pour toutes les valeurs des domaines de toutes les variables de Scope(C_i).

I.3.1.2.2 Algorithme GAC 2001

GAC 2001 (algorithme 5) [16] est une généralisation de **AC 2001** pour le filtrage des CSPs n-aires en se basant sur la consistance d'arc généralisée. **GAC 2001** opère sur un CSP dont les tuples des relations sont supposés être totalement ordonnés et le dernier support de chaque couple (X_i , a) pour une contrainte X_j. il est sauvegardé dans une variable $Last((X_i, a), X_j)$ initialisée à NIL. Et lors de la recherche du support d'une affectation, on vérifie premièrement si la valeur sauvegardée dans la variable *Last* correspondante existe toujours. Si c'est le cas, l'affectation possède un support sinon la recherche commence à partir de la valeur suivante. Les précédentes valeurs ont déjà fait l'objet d'une vérification.

Algorithm 5 GAC 2001

```
1: for chaque variable X_i do
2:    for chaque valeur a de D_i do
3:       for chaque contrainte C_j do
4:          Last((X_i, a), C_j) = NIL
5:       end for
6:    end for
7: end for
   Q ← {(X_i, C_j) | C_j ∈ C et X_i ∈ Scope(C_j)}
8: while Q non vide do
9:    choisir (X_i, C_j) de Q
10:   Q = Q − {X_i, C_j}
11:   if REVISE 2001(X_i, C_j) then
12:      Q = Q ∪ {(X_k, C_m) | C_m ∈ C, X_i, X_k ∈ Scope(C_m) et k ≠ i et j ≠ m}
13:   end if
14: end while
   Procédure REVISE 2001(X_i, C_j)
15: supprime←False
16: for chaque a_i ∈ D_i do
17:   τ = Last((X_i, a), C_j)
18:   if ∃k | τ[X_jk] ∉ [D_jk] then
19:      τ = succ(τ, D_{X_i=a}^{Scope(C_j)})
20:      while τ ≠ NIL and τ ∉ rel(C_j) do
21:         τ = succ(τ, D_{X_i=a}^{Scope(C_j)})
22:      end while
23:      if τ ≠ NILL then
24:         Last((X_i, a), Cj) = τ
25:      else
26:         supprimer a de D_i
27:         supprime←True
28:      end if
29:   end if
30: end for
31: return supprime
```

FIGURE I.5 – Algorithme GAC 2001 [46].

I.3.1.2.3 Transformation de CSPs n-aires en CSPs binaires

Trois principales approches sont proposées pour cette transformation :

- **Représentation duale** : Cette transformation consiste à utiliser chaque contrainte du CSP original comme une variable dans le nouveau CSP binaire. Son domaine est l'ensemble des tuples permis par sa relation correspondante. Chaque couple de variables du nouveau CSP qui représentent deux contraintes partageant des variables dans le CSP original seront reliées par une contrainte dans le nouveau CSP binaire.

- **Représentation en variables cachées** [61] : elle consiste à introduire des variables supplémentaires, appelées h-variables, en plus des variables initiales du CSP. Il existe une variable duale v_i pour chaque contrainte c_i du CSP. Le domaine de chaque variable duale est l'ensemble des tuples de la relation de la contrainte correspondante.

- **Représentation double** [65] : elle combine la représentation duale et la représentation en variables cachées. Les variables du nouveau CSP sont les variables initiales du CSP non binaire, les variables duales et les h-variables.

I.3.1.2.4 Résolution directe des CSPs n-aires

Cette approche consiste à résoudre le CSP directement dans sa formulation d'origine, et cela nécessite l'extension et l'adaptation des techniques et méthodes développées pour les CSPs binaires.

I.3.2 Méthodes incomplètes

Les méthodes incomplètes considèrent l'espace de recherche dans sa totalité mais elles ne l'explorent qu'en partie en se dotant des combinaisons d'heuristiques pour choisir les zones d'exploration. Ces méthodes visent à donner un résultat acceptable en un temps raisonnable mais elles ne peuvent prouver l'inexistence de solution d'un problème sur contraint.

Autrement dit, elles abordent la résolution d'un CSP comme un problème d'optimisation combinatoire pour lequel il s'agit de calculer une affectation satisfaisant le plus grand nombre de contraintes, l'objectif final étant de les satisfaire toutes.

Les différentes approches possibles d'une résolution incomplètes appartiennent a deux grandes familles de métaheuristques, celle de la recherche locale avec ses algorithmes : recuit simulé (RS), la recherche Tabou, colonies de fourmis (AC,Ant Columns), et celle dites évolutionniste avec les algorithmes génétiques [65].

I.3.3 Méthodes énumératives complètes :

Ces algorithmes consistent à visiter l'ensemble de toutes les affectations possibles des variables et effectuent un parcours systématique de l'espace de recherche. L'algorithme de type Backtracking constitue la méthode de base la plus répandue pour une recherche systématique. Cet algorithme, beaucoup trop coûteux, a conduit à la recherche d'algorithmes intelligents les plus efficaces [8].

Definition 1.8

Un conflit est une situation telle que l'affectation d'une valeur v appartient D_i à la variable X_i viole une des contraintes C du problème.

Definition 1.9

Un **ensemble de conflits** est lorsque, pour une sous-séquence $a = (a, ..., a_n)$ consistante et une variable X_i non affectée, il n'existe pas de valeur dans D_x qui soit consistante avec a.
On dira donc que a est un ensemble de conflit de X_i. S'il n'existe aucun sous-ensemble de a étant en conflit avec X_i, alors a est un ensemble de conflit minimal de X_i.

I.3.3.1 Retour arrière chronologique (Backtrack BT)

Le retour arrière ou le retour sur trace, **backtracking** en anglais, est une famille d'algorithmes utilisé en programmation, notamment pour résoudre les CSPs. Dans ces algorithmes, la cohérence est définie comme la satisfaction de toutes les contraintes dont les variables sont toutes affectées (tester systématiquement l'ensemble des affectations potentielles du problème). Ils consistent sur un parcours en profondeur sur l'arbre de décision du problème. L'idée est de partir du noeud parent, descendre dans le premier noeud fils satisfaisant la contrainte. Ce dernier devient donc un noeud parent et nous parcourons ensuite ses noeuds fils sous le même principe.

Si aucune solution n'est trouvée, la méthode abandonne et revient légèrement en arrière au nud parent et nous descendons dans le noeud fils suivant afin de sortir du blocage, d'où le nom de retour sur trace. La solution est identifiée lorsque nous arrivons à un noeud qui satisfait la contrainte et qui n'a pas de noeud fils [2].

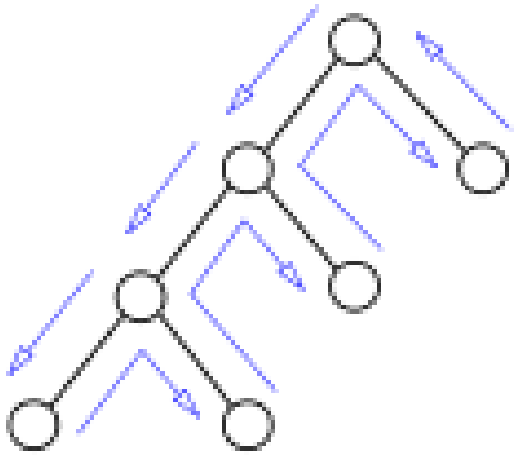

FIGURE I.6 – Arbre de recherche : Backtrack [6].

Algorithm 6 Backtrack

Input : un CSP $P = (X, D, C)$, et une affectation A. Initialement $A = \emptyset$.
Output : une solution du CSP si le CSP est consistant.

```
1: if A est non consistante then
2:   retourner False
3: else
4:   if A est une instanciation totale then
5:     retourner True
6:   else
7:     choisir une nouvelle variable X_i de X
8:     for Chaque valeur d_i ∈ D_i do
9:       if Backtrack(P = (X, D, C), A ∪ {X_i = d_i}) then
10:        retourner True
11:      end if
12:    end for
13:  end if
14: end if
```

FIGURE I.7 – Algorithme Backtrack [46].

I.3.3.2 Algorithmes avec retour arrière non chronologique

Selon Dechter et Frost [2002], le **BT** a comme principal problème de retomber sans cesse dans les mêmes situations d'inconsistance, le fait de faire des retours sur trace simples ou chronologiques, lors de la découverte d'une inconsistance, hors que la variable précédente peut ne pas être la cause de l'inconsistance. Par conséquent essayer de nouvelles valeurs pour cette variable conduira aux mêmes échecs. Pour remédier à ce problème, **le saut en arrière (Backjump)** a été proposé [18], permettant de réduire l'espace de recherche, et donc d'augmenter l'efficacité. Plusieurs algorithmes exploitant le retour-arrière non chronologique, nous citons :

I.3.3.2.1 Algorithme de BackJumping (BJ)

Cet algorithme est similaire à **BT**, sauf qu'il se comporte plus efficacement. Le BJ a comme but d'identifier la variable ayant causé l'inconsistance, si toute instanciation de la variable courante génère une inconsistance au lieu de revenir en arrière chronologiquement à la variable précédente, il fait un retour-arrière sur la variable la plus profonde qui a été vérifiée par rapport à la variable actuelle (Reliée avec cette dernière par une contrainte) [18], [38].

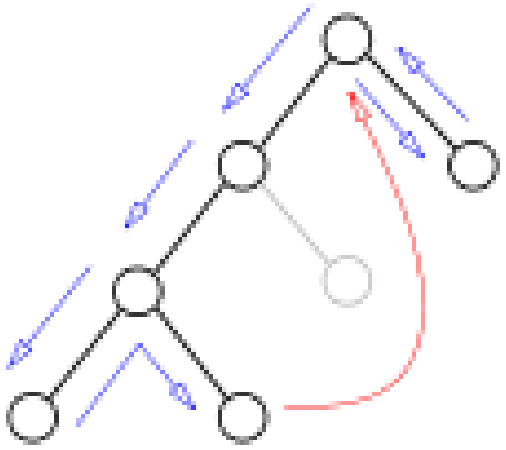

FIGURE I.8 – Arbre de recherche : Backjumping [6].

I.3.3.2.2 Graph-based Backjumping

Le saut arrière basé sur un graphique **(Graph-based Backjumping)** extrait les connaissances sur les ensembles de conflits possibles du graphe de contraintes exclusivement. Il exploite le graphe des contraintes. Chaque fois qu'une fin se produit et qu'une solution partielle ne peut pas être étendue à la variable suivante X_i (en cas d'échec d'extension cohérente sur une variable X_i), un retour-arrière est effectué vers la dernière instanciée dans le voisinage de X_i,X_j.
Si vous n'avez plus de valeurs (c'est-à-dire qu'il s'agit d'une impasse interne), un retour-arrière se produit à nouveau vers la variable la plus profonde appartenant au voisinage de X_j ou au voisinage de toute variable instanciée après X_j qui soit une cause possible de l'échec.

L'importance de cet algorithme est que l'étude avec des performances liées au graphe de contraintes conduit à des limites de complexité théorique des graphes et donc à des heuristiques basées sur des graphes visant à réduire ces limites. En particulier cet algorithme permet une limite de complexité en fonction de la profondeur de l'arbre couvrant la recherche en profondeur d'abord du graphe de contraintes [24].

I.3.3.2.3 Algorithme Conflict Directed BackJumping (CD-BJ)

Le saut arrière dirigé par un conflit **CD-BJ** [59] a un comportement de saut arrière plus sophistiqué que le **BJ**. C'est une amélioration de **BJ**. Cet algorithme Maintient pour chaque variable instanciée un ensemble de conflits, qui contient les variables passées qui ont échoué aux vérifications de cohérence avec son instanciation actuelle.
Pendant que BJ fait un simple retour chronologique, CD-BJ par contre, tente d'en faire plusieurs. Ceci est réalisé en maintenant pour chaque variable instanciée un ensemble des variables instanciées avant elles et qui sont en conflit avec elles pour au moins une de ses valeurs. Face à un échec, le retour-arrière se produit vers la dernière variable instanciée dans l'ensemble des conflits.

I.3.3.3 Heuristique d'ordre

Les expérimentations des algorithmes de résolution de CSP ont montré que l'ordre d'instanciation des variables ainsi que l'ordre des valeurs qu'elles peuvent prendre peuvent influer sur le temps de recherche d'une solution.
D'où, deux types d'heuristiques ont été proposés pour améliorer les techniques énumératives [8] :

I.3.3.3.1 Heuristiques sur l'ordre d'instanciation des variables

La majorité des heuristiques de choix des variables ont pour but de faire les choix les plus contraints d'abord pour arriver à une situation d'échec le plus tôt possible (first-fail). Ce qui a pour effet de réduire la taille des branches et ainsi d'accélérer la recherche. La taille de l'espace de recherche dépend de l'ordre d'affectation des variables et le choix d'un bon ordre dépend du problème traité. L'ordre d'instanciation des variables peut être statique ou dynamique.

L'ordre dynamique est défini au fur et à mesure de la résolution de CSP. Parmi ces heuristiques, nous citons : **dom+deg** [37] (consiste à choisir la variable ayant le nombre de valeurs minimum et en cas d'égalité choisir celle qui a le degré maximum), **dom/deg** [15] (où la prochaine variable à instancier est celle qui minimise le ratio entre la taille du domaine courant et le degré).
Dans le cas statique, par contre, l'ordre est prédéfini avant la résolution de CSP et se base généralement sur les propriétés structurelles du CSP : degré, tailles des domaines et satisfiabilité des contraintes. Parmi ces heuristiques, nous pouvons citer : **Maxdeg** [36] (les variables sont ordonnées selon un ordre décroissant sur les degrés), **MinCon** [36] (ordre selon le nombre de variables voisines instanciées).

I.3.3.3.2 Heuristiques sur l'ordre des valeurs à affecter en priorité aux variables

Min-conflict [37] est l'une des rares propositions d'heuristiques de choix des valeurs. Elle consiste à instancier la variable courante par la valeur qui a le nombre minimum de valeurs incompatibles avec les valeurs des variables non encore instanciées. Les heuristiques sur le choix des valeurs sont très lourdes à mettre en uvre ce qui les rend rarement utilisables.

I.3.3.4 Algorithme avec filtrage avant

Une amélioration possible pour l'algorithme du BackTrack **BT** est de le combiner avec les techniques de filtrage. Ces algorithmes consistent à exploiter le concept de consistance en avant, hors que, dans BT, la vérification de consistance s'effectue en fonction des variables déjà instanciées. Nous présentons deux algorithmes de ce type : Forward Checking **(FC)** et Maintaining Arc Consistency **(MAC)**.

I.3.3.4.1 Forward Checking (FC)

L'algorithme **FC** [40] est une amélioration de **BT** en applicquant un filtrage avant. Il vérifie la consistance entre la variable courante et celles qui ne sont pas encore instanciées dans son voisinage, en filtrant leurs domaines par la suppression des valeurs incompatibles avec la valeur courante. En effet, il ne construit que des affectations consistantes (succès) car les affectations inconsistantes sont éliminées par le filtrage anticipé. Il existe aussi d'autres algorithmes de forward checking FC (une généralisation de FC), au cas n-aire (nFC5, nFC4, nFC3, nFC2).

Algorithm 7 FC

Input : le quadruplet (A,NA,D,C), avec A est une affectation, initialement A $-$ {}, NA est l'ensemble des variables non encore instanciées, initialement $NA = X$, D est l'ensemble des domaines des variables et C est l'ensemble des contraintes.
Output : une solution du CSP si le problème est consistant, sinon le CSP est inconsistant.

```
1: if NA = {} then
2:    retourner A                    /* A est une solution */
3: else
4:    choisir X_i de NA
5:    repeat
6:       choisir une valeur v de D_i
7:       D_i = D_i - {v}
8:       if A ∪ {X_i, v} ne viole aucune contrainte then
9:          D' = Revise(NA - {X_i}, D, C, ⟨X_i, v⟩)
10:         if aucun domaine de D' n'est vide then
11:            result = FC(NA - {X_i}, A ∪ ⟨X_i, v⟩, D', C)
12:            if result ≠ NULL then
13:               retourner (result)
14:            end if
15:         end if
16:      end if
17:   until D_i = {}
18:   retourner(NULL)
19: end if
    Procedure Revise (W, D, C, a)
    a est une affectation d'une variable.
20: D' = D
21: for chaque variable X_j dans W do
22:    for chaque valeur v dans D'_j do
23:       if ⟨X_j, v⟩ est incompatible avec les contraintes de C then
24:          D'_j = D'_j - {v}
25:       end if
26:       retourner D'
27:    end for
28: end for
```

FIGURE I.9 – Algorithme FC [46].

I.3.3.4.2 Maintaining Arc Consistency (MAC)

Dans cet algorithme, la consistance d'arc est maintenue tout au long de la recherche. Elle est vérifiée pour toutes les variables non encore instanciées et non seulement celles qui sont en relation avec la variable courante comme c'est le cas avec le FC.

Le **MAC** propage cette vérification pour toutes les variables du CSP, que se soient celles qui sont en relation avec la variable instanciée, ou celles qui ne le sont pas, tout en garantissant un maintien de la consistance [63] .

Algorithm 8 MAC

Input : le quadruplet (A,NA,D,C), avec A est une affectation, initialement vide $A = \{\}$, NA est l'ensemble des variables non encore instanciées, initialement NA = X, D est l'ensemble des domaines et C celui des contraintes.
Output : *true* si le CSP est arc consistent, *false* sinon.

```
1: if NA = {} then
2:    retourner A /* A est une solution*/
3: else
4:    choisir une variable X_i de NA
5:    repeat
6:       choisir une valeur v de D_i
7:       D_i = D_i - {v}
8:       if A ∪ {X_i, v} est consistante then
9.          D' = Revise(NA - {X_i}, D, C, ⟨X_i, v⟩) /*Revise est similaire à celle de FC*/
10:         AC-x(NA - {X_i}, D', C) /*D' est filtré à l'aide d'un algorithme AC*/
11:         if aucun domaine de D' n'est vide then
12:            result- MAC(A ∪ {⟨X_i, v⟩}, NA - {X_i}, D', C)
13:            if result ≠ NULL then
14:               retourner (result)
15:            end if
16:         end if
17:      end if
18:   until D_i = {}
19:   retourner(NULL)
20: end if
```

FIGURE I.10 – Algorithme MAC [46].

I.3.3.5 Algorithmes Backtrack avec mémorisation

Le Backtrack intelligent a amélioré l'algorithme de base (BT) par une optimisation de la recherche en évitant les mêmes situations d'inconsistance. Les algorithmes avec mémorisation mémorisent des instanciations, appelées *nogoods*, qui ne peuvent pas être étendues en une solution. Cette mémorisation permet de ne pas les régénérer une autre fois et ainsi de ne pas explorer les mêmes sous-arbes.

Definition 1.10 (*Nogood*)

Un *Nogood* est une affectation partielle consistante qui ne peut pas s'étendre en une solution. Un *Nogood* minimal est tout *Nogood* non composé lui-même d'un *Nogood*.

Parmi les algorithmes qui adoptent la technique de mémorisation, nous pouvons citer :

— Nogood Recording (NR) [64].

— Learning Tree-Solve [11].

— Dynamic Backtracking (DBT) [62].

I.3.4 Méthodes de décomposition structurelles

Les méthodes de résolution des problèmes de satisfaction de contraintes, présentées jusqu'ici, se basent sur des techniques de recherche de type Backtrack dont la complexité temporelle et spatiale reste exponentielle. Cette exploitation des caractéristiques structurelles de l'instance traitée a donné naissance à une nouvelle approche de résolution qui est la résolution par décomposition. Cette approche exploite le fait que la traitabilité d'un CSP est liée aux caractéristiques topologiques de son graphe ou hypergraphe de contraintes.

Ce résultat théorique est formalisé par un théorème dû à Freuder [34], qui peut être résumé comme suit : "Si un réseau de contraintes est acyclique et arc consistant, il peut être résolu en appliquant un algorithme de degré polynomial (Backtrack free)". La résolution par décomposition est une technique qui analyse les CSPs et définit un schéma de décomposition avant la recherche d'une solution. Autrement dit, elle utilise la décomposition comme un prétraitement pour la résolution.

I.3.4.1 Principe des méthodes structurelles

Les méthodes de décomposition consistent à regrouper les sommets et les arêtes de l'hypergraphe d'un CSP en clusters de variables ou d'arêtes et d'organiser les clusters entre eux pour former un CSP dont la structure est un arbre. De plus, chaque méthode de décomposition propose une mesure de la cyclicité de CSP appelée largeur de la décomposition. Cette mesure garantit que la résolution n'est plus exponentielle en taille d'un problème mais en fonction de cette largeur. C'est pour cette raison qu'il est crucial de minimiser la largeur de la décomposition.

I.3.4.2 Quelques méthodes structurelles

Plusieurs méthodes de décomposition ont été proposées dans la littérature, parmi ces méthodes nous citons :

I.3.4.2.1 Méthode CCM Cycle Cutset

Le principe de cette méthode (proposee dans [25]) est basé sur la définition suivante :

Definition 1.11 (Cycle Cutset)

Etant donnée un graphe de contraintes, l'élimination de certaines variables après les avoir instanciées change la connectivité du graphe, et ensemble de variables est appelée coupe cycle ou Cycle Cutset Un ensemble cutset (coupe cycle) d'un graphe est un ensemble de sommets dont la suppression induit un graphe acyclique La largeur d'une décomposition Cycle Cutset est égale à la taille de l'ensemble couple cycle, et la largeur (width) d'une instance CSP est égale au nombre de sommets de l'ensemble coupe cycle le plus petit.
Cette technique est basée sur le fait que l'affectation des variables de l'ensemble coupe cycle rend le graphe de contraintes acyclique.

I.3.4.2.2 Méthode Hinge decomposition

La méthode Hinge decomposition [26] est une des premières méthodes à exploiter directement l'hypergraphe d'un CSP en se basant sur le concept des hinges, ce qui la rend applicable aussi pour les CSP binaires que pour les CSP n-aires.

Definition 1.12 (Hinge)

Soit (V, E) un hypergraphe et $H \subseteq E$ un ensemble contenant au moins deux arêtes. Soit $H1, ..., Hm$ les composantes connectées de $(E - H)$ par rapport à H. alors H est appelé **Hinge** si pour $i = 1$ à m, il existe une arête H_i dans H tel que $(\bigcup \mathrm{H}_i) \bigcap (\bigcup \mathrm{H}) \subseteq \mathrm{H}_i$ Et dans ce cas h_i est appelé séparateur de H_i. Un hinge est dit minimal s'il ne contient aucun hinge.

I.3.4.3.3 Méthode Tree-Clustering

La méthode tree clustering introduite par Rina Dechter [24] est l'une des méthodes de décomposition les plus intéressantes avant lapparition de l'hypertree decomposition. Elle se base sur le fait qu'un CSP est acyclique si et seulement si son graphe primal est à la fois chordal et conforme [47].

Definition 1.13 (Graphe conforme)

Le graphe primal d'un CSP est dit conforme si chaque clique maximale (composante complètement connectée maximale) correspond à une contrainte du CSP.

Definition 1.14 Graphe chordal)

Le graphe primal d'un CSP est dit chordal si chaque cycle de longueur supérieur à trois possède une corde i.e. une arête joignant deux noeuds non adjacents dans le cycle.

I.3.4.3.4 Méthode de décomposition arborescente TD (Tree décomposition)

La décomposition arborescente, en anglais : tree-decomposition, permet de définir une autre notion importante, la largeur arborescente ou largeur d'arbre (treewidth). Elle consiste en une décomposition d'un graphe en séparateurs (sous-ensembles de sommets dont la suppression rend le graphe non connexe), connectés dans un arbre.
Cette méthode a été proposée par Paul Seymour et Neil Robertson dans le cadre de leur théorie sur les mineurs d'un graphe. Elle est aussi connue en apprentissage automatique, où l'on parle d'arbre de jonction (un graphe de cliques construit de manière que le produit des fonctions de potentiels soit égal à la probabilité conjointe de l'ensemble des variables) [3].

I.3.4.3.5 Méthode BTD (Backtracking on Tree-Decomposition)

La méthode BTD est proposée par Jégou et Terrioux [43]. Elle exploite une décomposition arborescente TD (tree decomposition) pour résoudre des CSPs, en combinant l'efficacité pratique de l'énumération et les bornes de complexités issues de la TD du graphe de contraintes.

I.3.5 Méthodes hybrides

Les méthodes complètes apportent des garanties en ce qui concerne l'obtention dune solution optimale à un problème donné. L'inconvénient est que le temps de calcul augmente de façon exponentielle avec la taille des instances. Dans ce contexte, la garantie de l'optimalité est alors concédée vis-à-vis du temps de calcul.

Pour remédier à ce problème, une technique intéressante a été proposée consiste à combiner trois approches de résolution (énumérative, par filtrage et par décomposition) induit des algorithmes de résolution dites hybrides. Plusieurs algorithmes hybrides ont été proposés : algorithme qui fait l'hybridation entre un algorithme énumératif (BT) et Tree Décomposition ,BTD [43], hybridation d'une méthode énumérative et une méta-heuristique, etc.

Conclusion

Les problèmes de satisfaction de contraintes (CSP) sont au coeur de nombreuses applications en Intelligence Artificielle et en Recherche Opérationnelle. Un **CSP** est un formalisme à la fois simple et puissant permettant de trouver une valeur pour chaque variable afin de satisfaire l'ensemble des contraintes ou bien de satisfaire le maximum de contraintes pur résoudre un grand nombre de problèmes.

La recherche dans le domaine des CSP est motivée par la découverte de méthodes toujours plus efficaces en pratique pour les résoudre. Mais, comme les CSP font partie de la classe des problèmes combinatoires NP-Complets, les algorithmes connus pour les résoudre nécessitent un temps exponentiel en fonction du nombre de variables. La méthode la plus employée pour les résoudre est la recherche énumérative qui consiste à explorer systématiquement un arbre de recherche.

Dans ce chapitre, nous avons présenté le formalisme CSP et les déférentes méthodes de la résolution. Dans le prochain chapitre, nous detaillons les systèmes distribués.

Chapitre II

Systèmes Distribués

Introduction

Dans ce chapitre introductif, nous présentons les systèmes distribués, leurs architectures et quelques algorithmes utilisés dans ce domaine. Nous commençons d'abord par présenter quelques généralités relatives aux systèmes distribués.

II.1 Généralités sur les systèmes distribués

II.1.1 Définition

Un système distribué **(SD)** [55] est un ensemble d'ordinateurs (entités, sites) indépendants où chacun peut exécuter des tâches en concurrence (au même moment) avec les autres, interconnectées par un réseau de communication et qui apparaît à ses utilisateurs (qu'ils soient des personnes ou des programmes) comme un seul système cohérent, dans le but de résoudre en coopération une fonctionnalité applicative commune. Cet ensemble donne aux utilisateurs une vue unique des données du point de vue logique.

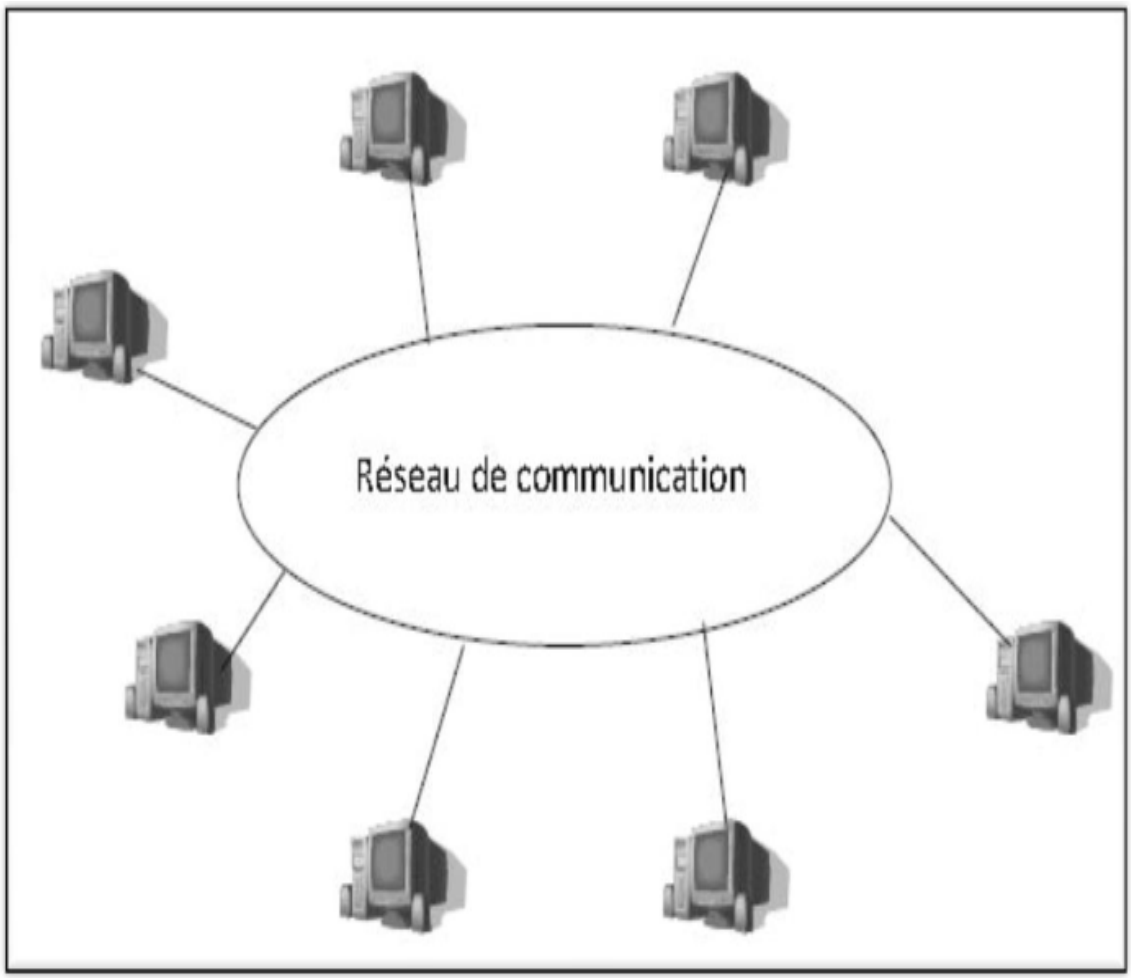

Figure II.1 – Architecture d'un système distribué [27].

II.1.2 Caractéristiques et Objectifs d'un système distribué

Un système est considéré comme distribué s'il présente les caractéristiques suivantes [7] :

II.1.2.1 Interopérabilité

Désigne la capacité à rendre deux systèmes quelconques compatibles (La compatibilité est la capacité qu'ont deux systèmes à communiquer sans ambiguïté.)
L'interopérabilité vise à réduire le problème de l'hétérogénéité (un problème de coopération dans le partage des ressources dans un système distribué) en la masquant par l'utilisation d'un protocole unique de communication.

II.1.2.2 Partage des ressources

L'objectif principal d'un système distribué est de faciliter l'accès et le partage des ressources distantes aux utilisateurs et les applications pour l'échange d'informations de manière contrôlée et efficace.

II.1.2.3 Ouverture

Désigne la possibilité d'ajout, de suppression, ou de modification des ressources et des services dans un système distribué.

— Un système distribué est ouvert et offre des services selon des règles standard décrit la syntaxe et la sémantique de ces services.

— Flexibilité (facilité d'utilisation et de configuration).

— Extensibilité (ajout de composants : les matériels et les logiciels, sans en affecter les autres).

II.1.2.4 Transparence de la distribution

La transparence permet de cacher aux utilisateurs les processus et les ressources dun système qui sont physiquement distribués sur plusieurs ordinateurs. Selon la norme (ISO, 1995) la transparence a plusieurs niveaux :

- Transparence d'accès : Cacher l'organisation logique des ressources (La façon dont une ressource est accessible).
- Réplication de ressources invisible.
- Transparence à la localisation des ressources.
- Relocalisation : Cacher le fait qu'une ressource peut changer d'emplacement en cours d'utilisation.

- Concurrence : Cacher qu'une ressource peut être partagé par plusieurs utilisateurs compétitifs. (Permet l'accès simultané à des ressources par plusieurs processus).
- Migration de ressources invisible Cacher qu'une ressource peut se déplacer vers un autre emplacement.
- Tolérance aux pannes : Ce qui permet à l'utilisateur de ne pas s'interrompre (ou même se rendre compte) à cause d'une panne d'une ressource.
- Transparence à la reconfiguration : Possibilité de reconfigurer le système invisible aux utilisateurs pour augmenter les performances en fonction de la charge.

II.1.2.5 Sécurité

Les ressources doivent être protégées contre des utilisations malveillantes.

II.1.2.6 Evolutivité ou mise à l'échelle (scalability)

Évolutivité peut être mesurée avec au moins trois dimensions. Un système distribué peut être :

- Evolutif par rapport à sa taille, ce qui facilite l'ajout des utilisateurs et de ressources au système. (Le nombre d'utilisateurs et/ou de processus).
- Géographiquement évolutif dans lequel les utilisateurs et les ressources peuvent se situer loin d'une part. (La distance maximale physique qui sépare les ressources du système).
- Administrativement évolutif, qui peut être facile à gérer même s'il couvre beaucoup d'organisations administratives indépendantes. (Le nombre de domaines administratifs).

II.1.3 Entités

Un système distribué est composées de trois éléments essentielles qui sont définies de la manière suivante :

- **Processuss** : Ils décrivent les comportements des programmes qui s'exécutent à l'intérieur du système,
- **Modèle de communication** : Il décrit le mode d'interaction ou de communication entre les processus,
- **Architecture d'interconnexion** : c'est la manière d'organisation des processus (la topologie).

II.1.3.1 Processus

Les processus sont des programmes en cours d'exécution. Ils communiquent en échangeant des messages par l'intermédiaire de canaux de communication, ou par l'accès à une variable commune dans une mémoire partagée. Un processus est considéré défaillant, s'il a un mauvais comportement ou s'écarte de ses spécifications initiales durant l'exécution d'une tâche, autrement, il est dit correct. Les défaillances sont classées en quatre catégories [58] :

II.1.3.1.1 Défaillance par arrêt définitif (Crash fault)

Le processus se comporte conformément à sa spécification jusqu'à ce qu'il subisse une défaillance franche, et à partir de celle-ci le processus cesse définitivement a toute activité.

II.1.3.1.2 Défaillance par omission (Omission fault)

Dans ce type de défaillance, un processus fautif peut omettre certaines tâches de sa spécification. Ces tâches sont liées principalement à l'envoi et la réception de messages. Dans le cas d'un processus, certains messages peuvent par exemple ne pas être envoyés. Le processus reprend ensuite son activité.

II.1.3.1.3 Défaillance temporelle (Timing fault)

Une défaillance temporelle, dite aussi de performance, signifie le fait que suite à une demande de service, le processus fournit la réponse soit trop tôt soit trop tard. De ce fait, le processus ne respecte pas les contraintes temporelles spécifiées à une tâche ou plusieurs tâches.

II.1.3.1.4 Défaillance arbitraire (Byzantine fault)

Ce type de défaillance est le plus général et le plus sévère parmi ceux déjà cités. Un processus peut avoir un comportement arbitraire pouvant aller de la panne franche aux comportements malveillants.

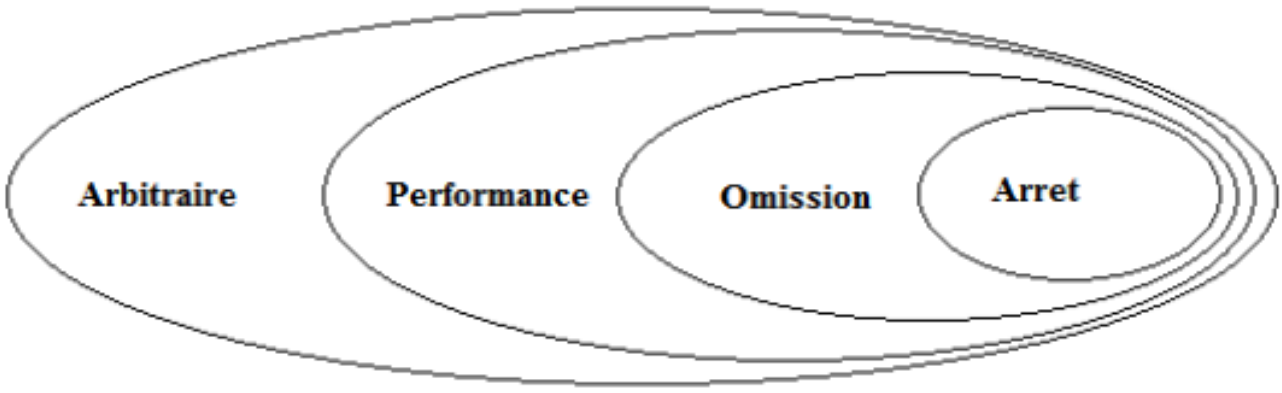

FIGURE II.2 – Classes des défaillances dans un système [28].

II.1.3.2 Mode de communication

La communication entre les processus d'un système distribué se fait de deux manières, soit par une mémoire partagée (shared memory). Soit par l'échange de messages (message passing).

II.1.3.2.1 Communication par mémoire partagée

Dans ce mode de communication [9], les processus communiquent entre eux grâce à une mémoire commune à travers laquelle ils peuvent échanger des données. L'accès à cette ressource commune est géré par deux primitives de lecture/écriture :

- **Lire** () : Permet de lire une donnée stockée dans une zone de la mémoire partagée.
- **Ecrire** () : Permet d'écrire une donnée dans une zone de la mémoire partagée.

II.1.3.2.2 Communication par échange de messages

Dans ce mode, la communication est effectuée grâce à un mécanisme d'échange de messages. Ce modèle est caractérisé par l'absence totale d'horloge globale et de mémoire commune. L'échange de messages est effectué par les deux primitives de communication suivantes :

- **Envoyer** (msg, p_j) : Permet d'envoyer le message msg au processus p_j.
- **Recevoir** (msg) : Permet de recevoir un message msg déjà envoyé par un processus donné.

Ce mode dispose aussi d'une autre primitive de communication destinée à la communication dans un groupe :

- **Diffuser** (msg) : Permet d'envoyer un message msg à tous les processus du système. l'échange de messages se fait via des canaux de communication suivant une organisation bien définie des processus. Les canaux de communication, à leurs tours, sont sujets à plusieurs types de défaillances :
  - La destruction du canal,
  - La perte de messages,
  - La duplication de messages,
  - La corruption de messages.

Suivant ces défaillances, les canaux de communication peuvent être classés en [30],[69] :

- **Canaux fiables** : Si un processus pi envoie un message msg à un processus p_j, alors, éventuellement,msg sera reçu par p_j. les canaux fiables sont implémentés en utilisant des techniques de retransmission de messages.
- **Canaux quasi-fiables** : Un processus correct pi envoie un message msg à un processus correct p_j , alors p_j finira par recevoir le message msg. Pratiquement, ce type de canaux est le plus recommandé pour la conception des algorithmes distribués car il suppose que l'émetteur et le récepteur soient corrects.

- **Canaux fiables FIFO** : Les canaux sont supposés fiables et en plus l'ordre de livraison de messages est celui de l'émission de ces messages entre deux processus.
- **Canaux fiables avec fautes de performances** : Ils supposent des canaux fiables avec la prise en compte des délais de transmission de messages.
- **Canaux avec des pertes équitables** : Si un processus p_i envoie à un processus correct p_j un message msg une infinité de fois, alors p_j reçoit msg une infinité de fois.
- **Canaux non fiables** : Des pertes de messages se produisent sans que l'on puisse faire la moindre hypothèse concernant leurs occurrences.

II.1.3.3 Topologie

La topologie d'un système distribué définit la manière d'organisation des processus. Elle fait référence à l'ensemble des processus qui la compose et leurs interconnexions. Il existe plusieurs types de topologies, qui sont classifiées en trois larges catégories :

— **Réseaux point-a-point (Unicast)** : Chaque processeur ne peut communiquer directement q'avec un certain nombre de processeurs, ses voisins,

— **Réseaux de diffusion (Broadcast)** : Où chaque processeur peut communiquer le même message simultanément à un certain nombre de récepteurs,

— **Réseaux mixtes** : Permettent la diffusion à un nombre limité de processus (multicast) [56].

II.1.4 Synchronie

La synchronie d'un modèle de système dépend des hypothèses temporelles faites sur le comportement des processus et des canaux de communication. Le synchronisme d'un système est défini en termes de bornes faites sur ces trois paramètres. Suivant ces bornes, on distingue plusieurs modèles de temps :

— Modèle synchrone,

— Modèle asynchrone,

— Modèle partiellement synchrone.

II.1.4.1 Modèle synchrone

Le système synchrone est un système caractérisé par des hypothèses temporelles sur la transmission de messages. L'approche synchrone consiste à supposer que [69] :

— Il existe une borne supérieure connue comme délai de transmission d'un message. Ce délai comprend le temps nécessaire de l'émission, la transmission et la réception du message.

— Chaque processus possède une horloge logique et la dérive de cette horloge par rapport au temps réel à une borne supérieure connue.

— Il existe une borne inférieure et une borne supérieure connues au temps nécessaire à un processus pour exécuter une instruction de son programme.

II.1.4.2 Modèle asynchrone

Dans un système asynchrone [49], il n'ya pas d'hypothèses temporelles sur les temps dexécution et de délivrance des messages, ce qu'il le rend le modèle le plus réaliste. Ce système modélise en particulier un système dont la charge n'est pas prédictible. C'est le cas de la plupart des systèmes réels utilisés actuellement. En effet, un système asynchrone est un système où il n'est pas possible de faire la différence entre un délai de la transmission d'un message et la défaillance de l'expéditeur de ce message.

II.1.4.3 Modèle partiellement synchrone

Le modèle partiellement synchrone ([67], [23],[21]) est un modèle intermédiaire entre le synchrone et l'asynchrone. Ce modèle affaiblit le modèle asynchrone par l'ajout de propriétés temporelles. La satisfaction de ces propriétés permet aux algorithmes basés sur ce modèle d'atteindre la terminaison.

En général, le système partiellement synchrone se comporte d'abord d'une manière asynchrone, ensuite, il se stabilise et commence à se comporter d'une manière synchrone. La figure suivante illustre les principales différences entre les trois modèles temporels :

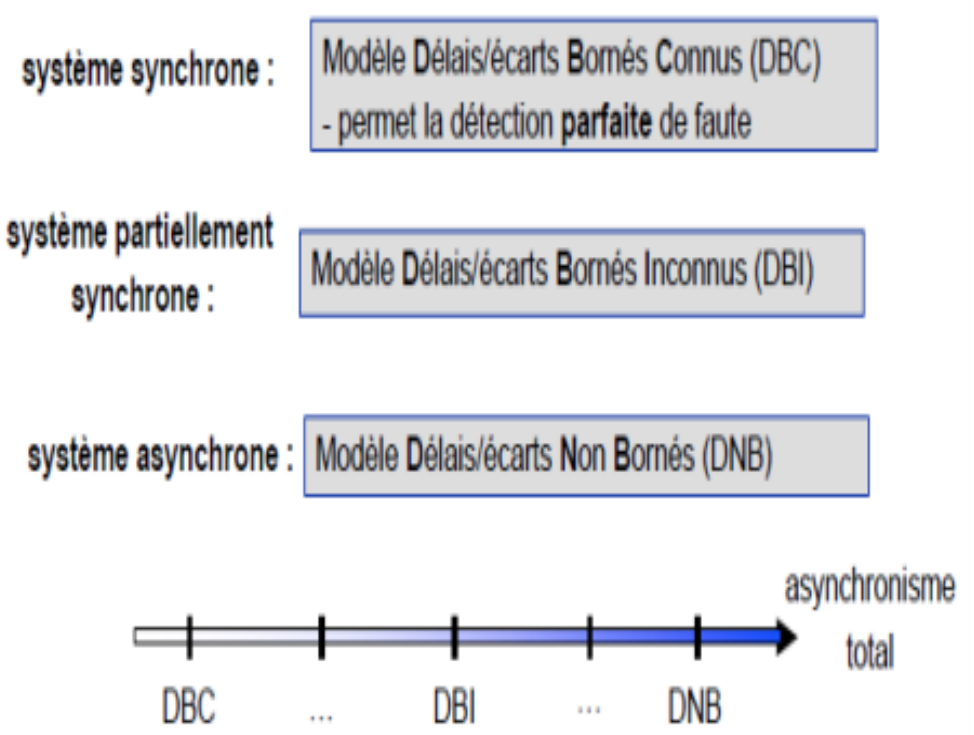

FIGURE II.3 – Modèles temporels [13].

II.2 Architecture distribuée

L'architecture d'un environnement informatique ou d'un réseau est distribuée lorsque toutes les ressources ne se trouvent pas au même endroit ou sur la même machine. Ce concept s'oppose à celui d'architecture centralisée dont une version est l'architecture client-serveur [57].

En effet, la programmation orientée objet a permis le développement des architectures distribuées en fournissant des bibliothèques de haut-niveau pour faire dialoguer des objets répartis sur des machines différentes entre eux. Les architectures distribuées reposent sur la possibilité d'utiliser des objets qui s'exécutent sur des machines réparties sur le réseau et communiquent par messages.

II.2.1 Types d'architecture distribuée

II.2.1.1 Architecture client-serveur

Le client server est avant tout un mode de dialogue entre deux processus. Le premier appelé client, demande l'exécution de services au second appelé serveur. Le serveur accomplit les services et envoi en retour des réponses.
En général, un serveur est capable de traiter les requêtes de plusieurs clients. Un serveur permet donc de partager des ressources entre plusieurs clients qui s'adressent à lui par des requêtes envoyées sous forme de messages. Pour l'architecture client-serveur, nous distinguons trois types d'acteurs principaux :

- **Client** : Un client est un processus demandant l'exécution d'une opération à un autre processus par l'envoi d'un message contenant le descriptif de l'opération à exécuter et attendant la réponse à cette opération par un message en retour.
  Le client est caractérisé par :
  - Il est actif le premier,
  - Il envoie des requêtes au serveur,
  - Il attend et reçoit les réponses du serveur.
- **Serveur** : On appelle serveur un processus accomplissant une opération sur demande d'un client et lui transmettant le résultat. Il est la partie de l'application qui offre un service, il reste à l'écoute des requêtes du client et répond au service demandé par lui-même. En effet, un serveur est généralement capable de servir plusieurs clients simultanément.
  Le serveur est caractérisé par :
  - Il est initialement passif,
  - Il est à l'écoute, prêt à répondre aux requêtes envoyées par des clients,
  - Dès qu'une requête lui parvient, il la traite et envoie une réponse.

- **Middleware** : Le middleware est l'ensemble des services logiciels qui assurent l'intermédiaire entre les applications et le transport de données dans le réseau afin de permettre les échanges des requêtes et des réponses entre client et serveur de manière transparente.

II.2.1.2 Architecture pair-à-pair (P2P)

Le pair-à-pair est un modèle de réseau informatique [32] proche du modèle client-serveur mais où chaque ordinateur connecté au réseau est susceptible de jouer tour à tour le rôle de client et celui de serveur. P2P est une architecture pouvant être centralisée ou décentralisée.
Le pair-à-pair peut servir au partage de fichiers en pair à pair, au calcul distribué ou à la communication entre noeuds ayant la même responsabilité dans le système. La particularité des architectures pair-à-pair réside dans le fait que les données peuvent être transférées directement entre deux postes connectés au réseau, sans transiter par un serveur central. Cela permet ainsi à chaque ordinateur d'être à la fois serveur de données et client des autres.

II.2.2 Avantages des architectures distribuées

- Augmentation des ressources : la distribution des traitements sur les ordinateurs d'un réseau augmente les ressources disponibles ;
- Répartition des données et des services : (cas de l'architecture 3-tiers à la base de la plupart des applications distribuées de commerce électronique permettant d'interroger et de mettre à jour des sources de données réparties).

II.3 Algorithmes distribués

II.3.1 Quelques problèmes soulevés par les systèmes distribués

La programmation des systèmes distribués doit être basée sur l'utilisation des algorithmes corrects, flexibles et efficaces car ils sont relativement différents des systèmes classiques. Le développement d'algorithmes distribués, algorithmes qui s'exécutent sur plus d'une machine ou processeur, est un métier assez différent de celui utilisé dans le développement d'algorithmes centralisés (classique ou Monoprocesseurs) sur trois points essentiels [68],[66] :

- **Manque de connaissance de l'état mondial**
  Dans un système distribué, il n'est pas possible de prendre une décision de contrôle basée sur l'état global car les entités (noeuds) d'un système distribué ont accès uniquement à leur propre état et non à l'état global de l'ensemble du système, contrairement au cas classique où nous avons une connaissance global de celui-ci.
  Une entité peut recevoir des informations sur l'état d'autres entités et baser ses décisions de contrôle sur ces informations par contre dans les systèmes centralisés, le fait que les informations reçues soient anciennes peut rendre les informations invalides car l'état de l'autre entité peut changer entre l'envoi des informations d'état et la décision basée sur celles-ci. Dans le cas distribué, nous pouvons donc réussir à accomplir des tâches en n'ayant qu'une connaissance partielle du système.

- **Absence du temps global sur le système**
  Pour certaines paires d'événements constituant l'exécution d'un algorithme distribué, il peut y avoir une raison de décider que l'un se produit avant l'autre, mais pour dautres paires, il est vrai qu'aucun des événements ne se produit avant l'autre. La relation temporale induite sur ces évènements n'est donc pas totale.
  Nous n'avons pas vraiment de contrôle permettant de savoir dans quel ordre les processus vont agir. Certains événements peuvent même avoir lieu de manière simultanée : Deux processus P1 et P2 peuvent commencer à accéder à la ressource, tandis que le début d'un processus ne précède temporellement le début de l'autre.

- **Non-déterminisme**
  En raison des différences possibles dans la vitesse d'exécution des composants du système, l'exécution d'un système distribué est généralement non déterministe : chaque demande doit être traitée immédiatement, et l'ordre de traitement est l'ordre dans lequel les demandes arrivent, mais comme les délais de transmission ne sont pas connus, les demandes peuvent arriver dans un ordre différent, et donc elles peuvent fournir des résultats différents.

Enfin, il n'existe pas d'horloge globale dans un système distribué : Chaque entité ne peut évaluer l'état d'avancement des autres entités que par le biais des communications qu'il a eu avec eux. L'étude des systèmes distribués consiste donc à étudier d'une part l'architecture de communication entre les différentes entités et d'autre part à concevoir des algorithmes distribués et à analyser leurs performances.

II.3.2 Concepts de base de l'algorithmique distribuée

Un algorithme distribué est basé sur un ensemble de concepts [45] tels que le jeton circulant, le calcul diffusant, l'estampillage, et Le transfert de connaissances.

- **Jeton circulant**
  Consiste à faire circuler un privilège (matérialisé par un message spécial ou par une configuration des variables d'état des processus) sur un ensemble dc processus. Sauf le processus qui détient le jeton peut réaliser une tache particulière correspondante en protocole exécuté (terminaison, exclusion mutuelle).
- **Calcul diffusant**
  Il s'agit d'envoyer une enquête a un ensemble de processus pour récupérer un résultat qui peut servir à une prise de décision lors de contrôle. Le calcul de diffusant est terminé lorsque le processus racine (père) reçoit toutes les réponses attendues par ses processus voisins.
- **Estampillage**
  L'estampille est la base de nombreux algorithmes distribués dans lesquels l'établissement d'un ordre entre les évènements est nécessaire. Chaque processus possède et gère une horloge 1ogique qui consiste en une valeur entière, croissante, initia1isée à zéro et qui s'incrémente après chaque émission ou réception de messages, Tout message émis est doté de la valeur de l'heure logique locale et de l'identité du site.

- **Transfert des connaissances**
  Cette technique consiste à acquérir des informations d'un ensemble de processus pour avoir une vue partielle ou globale sur l'état du système. Un processus qui désire avoir ces informations, diffuse un message a ses processus voisins, le processus récepteur diffuse ce message, qui est enrichi de ses connaissances locales, a ses processus voisins jusqu'à ce l'ensemble de processus soit atteint.

II.3.3 Quelques algorithmes distribués

Au cours des deux dernières décennies, de nombreuses familles d'algorithmes distribués ont été conçus, nous citons les principales [39] :

- **Algorithmes dits synchrones :** Avec SyncBT, il y'a une seule entité **(agent)** à la fois qui travaille. Le terme synchrone fait référence au fait qu'un agent ne peut pas choisir/modifier la valeur retenue pour sa variable de manière asynchrone, il doit attendre son tour.
- **Algorithmes dits asynchrones :** L'algorithme **Asynchronous Bactracking (ABT)** permet aux agents de travailler de manière simultanée. La façon d'y arriver est de permettre à chaque agent de changer à tout moment la valeur de sa variable (d'où son asynchronisme). Pour un agent donné, les agents qui apparaissent avant lui ont une priorité plus élevée, tandis que ceux qui apparaissent après lui ont une priorité inférieure.
- **Concurrent Search Algorithms (CSA) :** Comme pour SyncBT, les agents travaillent séquentiellement et de manière synchrone à la construction d'une solution. Par contre, le collectif travaille simultanément sur plusieurs solutions.

II.4 Avantages et inconvénients des systèmes distribués

II.4.1 Avantages

Les systèmes distribués présentent un certain nombre de points forts tels que [57] :

- **Utilisation et partage des ressources distantes :** Les matériels, logiciels et données peuvent être accédés à partir de n'importe quelle machine.
  - Robustesse : dans un système distribué, plusieurs machines travaillent ensemble pour fournir un service.
  - Extensibilité : l'extensibilité d'un système est une propriété qui lui permet de s'adapter facilement à une augmentation de sa charge.
  - Disponibilité : La disponibilité est la probabilité qu'un service soit disponible à un moment donné. La présence de plusieurs machines permet une conception conférant au service la possibilité de rester disponible, même si l'un des serveurs est indisponible.
- **Puissance :** la possibilité de combiner la puissance de plusieurs machines est très avantageuse, et l'ajout de nouvelles machines augmente la puissance globale du système.

- **Souplesse :** les systèmes distribués offrent une certaine souplesse, dans la mesure où ils peuvent faire face aux problèmes de montée en charge.
- **Tolérance aux pannes :** les systèmes distribués permettent d'être plus tolérants aux pannes, ils utilisent la technique de réplication des composants physiques et logiques qui les composent.
- **Hétérogénéité :** les composants physiques et logiques d'un système distribué sont généralement hétérogènes sur plusieurs niveaux : architectural, système d'exploitation, langages de programmation, etc.

II.4.2 Inconvénients

Certains systèmes distribués utilisent la notion de serveur central pour répondre à des besoins spécifiques. Dans ce cas, la panne de ce serveur peut avoir des conséquences sur le fonctionnement global du système, sauf si des techniques de tolérance aux pannes ont été mises en place. La communication représente l'un des problèmes fondamentaux des systèmes distribués, et il est d'ailleurs considéré comme étant la principale faille de ces systèmes.
Le dernier inconvénient est celui lié à la gestion et à l'administration. Etant donné qu'un système distribué regroupe différentes machines, celles-ci utilisent généralement des politiques de gestion et d'administration qui sont hétérogènes et parfois incompatibles.

Conclusion

Dans ce chapitre, nous avons présenté quelques notions générales sur les systèmes distribués à savoir leurs caractéristiques, les entités composantes de ces systèmes et la synchronie. Nous avons présenté aussi leurs architectures et quelques algorithmes distribués. Dans le prochain chapitre, nous détaillerons les systemes multi-agents.

Chapitre III

Systèmes Multi-Agents

Introduction

Un système multi-agents **(SMA)** représente un nouveau paradigme de programmation qui prend de plus en plus une place importante parmi les technologies de développement des systèmes complexes et distribués. Ce domaine est une branche de l'intelligence artificielle distribuée **(IAD)**.

Nous commençons ce chapitre par une généralité sur les **SMA**, notamment l'**IAD**, les agents et systèmes multi-agents, leurs propriétés et typologies. Ensuite, les caractéristiques et l'organisation dans les **SMA** ainsi leurs avantages.

III.1 Généralités sur les agents et systèmes multi agents

III.1.1 Intelligence Artificielle Distribuée (IAD)

La plupart des applications ou problèmes réels font intervenir des systèmes physiquement et fonctionnellement distribués.

Intelligence Artificielle Distribuée

=

IA : Modéliser le savoir des agents (compétence)

+

Distribution : Modéliser leurs interactions (organisation sociale).

L'intelligence artificielle distribuée **(IAD)** est une approche qui permet la résolution des problèmes complexes d'apprentissage, de planification et de prise de décision en proposant une distribution de l'expertise sur un ensemble de systèmes multi agents capables d'interagir en coopération dans un environnement commun, ce qui permet de résoudre les problèmes exigeant le traitement des ensembles de données très volumineux.

Les systèmes **IAD** sont constitués de noeuds de traitement d'apprentissage autonomes appelés agents, distribués. Ces noeuds peuvent agir indépendamment et des solutions partielles sont intégrées par communication entre les noeuds, souvent de manière asynchrone [60].

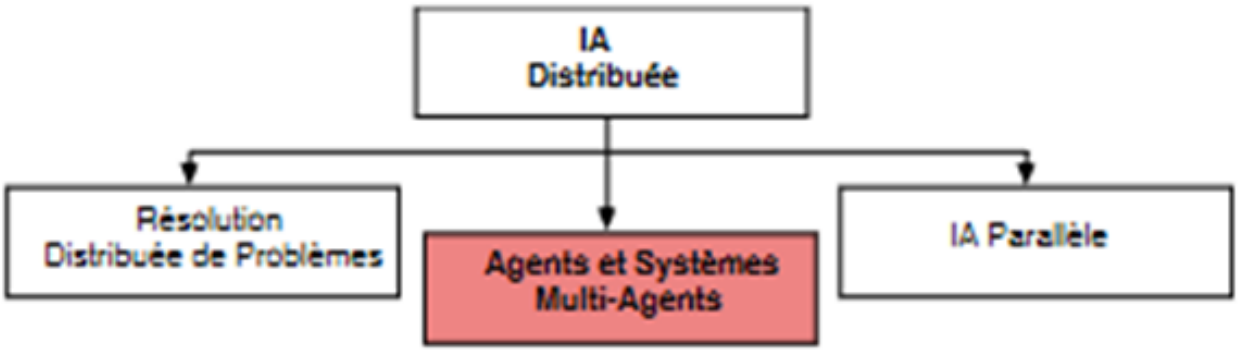

FIGURE III.1 – Branches de L'IAD [29].

III.1.2 Concept d'agent

III.1.2.1 Définition

Un agent est une entité physique (hard) ou logicielle (soft), indépendante qui peut prendre des décisions, qui perçoit son environnement extérieur et capable d'agir sur ce dernier et sur elle-même dans un objectif précis. Il fonctionne par la technologie de l'intelligence artificielle **(IA)**.

Cet agent peut être utilisé pour collecter de manière autonome des informations selon un horaire régulier et programmé, ou à la demande de l'utilisateur en temps réel. Il est également qualifié d'**intelligent** en raison de sa capacité à apprendre pendant le processus d'exécution des tâches [1].

Définition 3.1.1

On appelle un agent, une entité physique ou virtuelle [31] :

— Capable d'agir dans un environnement.
— Pouvant communiquer directement avec d'autres agents.
— Possédant des ressources propres à lui.
— Capable de percevoir (mais de manière limitée) son environnement.
— Disposant que d'une représentation partielle de cet environnement (et éventuellement aucune).
— Possédant des compétences et offrant des services.
— Pouvant se reproduire éventuellement.
— Ayant la tendance de satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

III.1.2.2 Propriétés

Les principales propriétés qui définissent un agent sont :

- **Autonomie :** capacité d'un agent à agir seul (sans une intervention directe d'humains ou d'autres agents), pour atteindre ses objectifs en ne subissant aucun contrôle sur son état interne, ni sur les actions qu'il réalise. On pourrait dire ainsi que le moteur dun agent, c'est lui-même. Et donc il n'est pas guidé par l'extérieur mais par ses tendances [31].
- **Flexibilité :** un agent est flexible signifie qu'il est :
  - Réactif : capacité de percevoir son environnement et de répondre à temps aux différents changements qui se produisent dans celui-ci.
  - Proactif : un agent n'agit pas seulement d'une manière réactive (en fonction de son environnement) mais il doit être aussi capables de montrer des comportements dirigés pour atteindre ses propres buts et objectifs et ceci en prenant des initiatives.
  - Social : capacité d'interagir avec les autres agents intelligents et humains grâce à des langages de communication, pour atteindre ses propres objectifs et aider les autres dans leurs activités.
- **Perception :** un agent doit percevoir directement son environnement par ses propres capteurs (recevoir les informations d'un environnement).
- **Adaptabilité :** capacité d'apprendre et de s'améliorer avec lexpérience.
- **Situation dans un environnement :** un agent agit sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement.
- **Communication :** La communication entre agents ce fait par le transfert d'informations par un agent vers un ou plusieurs agents. C'est l'un des modes principaux d'interaction existant entre les agents.
- **Action :** un agent est capable d'agir, et non pas seulement de raisonner. Cela repose sur le fait que les agents accomplissent des actions qui vont modifier l'environnement des agents et donc leurs prises de décision futures.
- **Croyance :** ce que l'agent connaît du monde (la représentation de son environnement, les autres agents et lui-même) [13].

III.1.3 Typologies des agents

Le choix du ou des types d'agents à utiliser dépend en fait du système multi-agents le plus pertinent pour le problème à résoudre. Ainsi, certains **SMA** n'utilisent qu'un seul type d'agents regroupés par objectif, d'autres plusieurs types correspondant à des rôles précis nécessaires à la résolution. Nous distinguons traditionnellement deux types d'agent : l'agent réactif et l'agent cognitif.

III.1.3.1 Agent réactif

Un agent réactif possède une représentation très simplifiée de son environnement. Il communique d'une façon indirecte via l'environnement, Il se caractérise par le manque de mémoire locale (il n'est pas capable de tenir compte de ses actions passées car il ne possède pas de moyen de mémorisation), l'absence de mécanismes de raisonnement et il n'est pas nécessaire que chaque agent soit personnellement intelligent pour parvenir à un comportement intelligent de l'ensemble (Les travaux sur ces agents s'intéressent plus à la modélisation d'une société d'agents qu'à l'agent lui-même).

Un agent réactif est constitué d'un ensemble de comportements, chaque comportement est une machine à états finis qui établit une relation entre une entrée sensorielle et une action en sortie (Perception/Action) Cependant, du fait, de leur nombre, ces agents réactifs peuvent résoudre des problèmes qualifiés de complexes. Les travaux sur ces agents s'intéressent plus à la modélisation d'une société d'agents qu'à l'agent lui-même [13], [44].

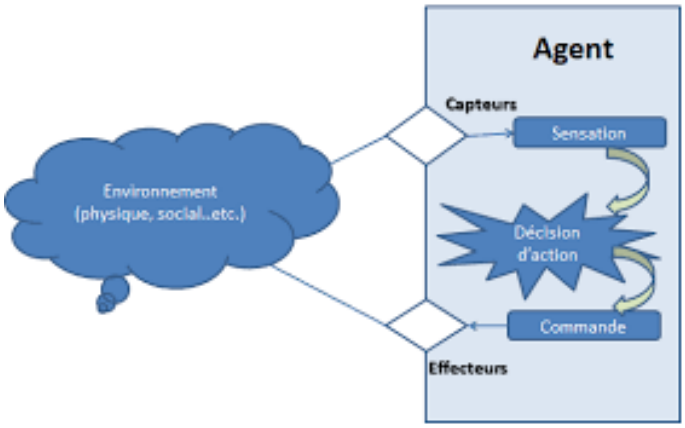

FIGURE III.2 – Cycle perception, action d'un agent réactif [13].

III.1.3.2 Agent cognitif

Contrairement à un agent réactif, un agent cognitif dispose d'une base de connaissances comprenant les diverses informations liées à son domaine d'expertise et à la gestion des interactions avec les autres agents et leur environnement. Il se caractérise par une représentation explicite de ce dernier et possède une architecture interne la plus complexe et suit un cycle Perception /Délibération/Action [71].

Un agent cognitif est capable de planifier son comportement, de mémoriser ses actions passées, de communiquer par envoi de messages ou via des langages d'interaction élaborés, de négocier, etc.

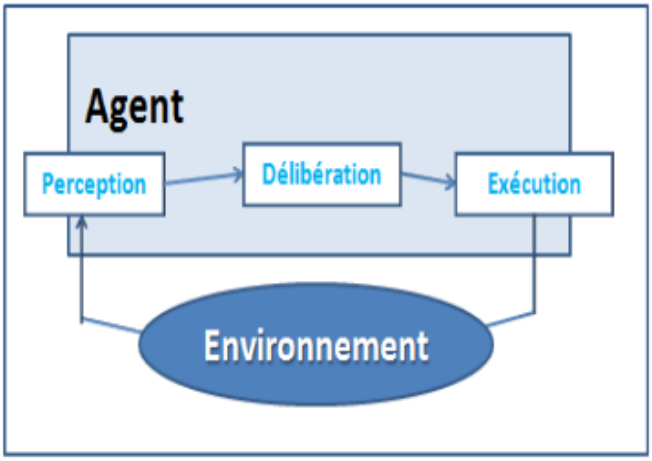

FIGURE III.3 – Cycle Perception Délibération Action d'un agent cognitif [10].

Les agents cognitifs sont généralement représentés par un **état mental** ayant les attitudes mentales de l'architecture BDI : Belief (Croyance), Desire (Désir), Intention (Intention).

- **Croyances :** correspondent aux informations dont dispose l'agent sur son environnement et sur d'autres agents qui existent dans le même environnement (peuvent être incorrectes, incomplètes ou incertaines).
- **Désirs :** correspondent aux états possibles de l'environnement que l'agent aimerait atteindre.
- **Intention :** les actions que l'agent a décidé d'accomplir pour satisfaire ses désirs.

III.1.3.3 Agent hybride

Les systèmes réactifs et les systèmes cognitifs peuvent ne pas convenir à la résolution de tous les problèmes. Leur différence se situe au niveau de leurs architectures internes et de la manière dont les informations perçues par ces agents sont traitées, et chacun de ces deux types de systèmes convient à certains types de problèmes et moins bien pour d'autres. Toutefois, il est maintenant possible de concevoir des systèmes hétérogènes comportant les deux types de comportements cognitif et réactif : les agents hybrides.

Un agent hybride est composé de modules qui gèrent indépendamment la partie réactive et la partie cognitive. Il possède des capacités cognitives et réactives et conjugue la rapidité de réponse des agents réactifs ainsi que les capacités de raisonnement des agents cognitifs. Cette approche est intéressante et semble apporter une solution adéquate pour modéliser les systèmes complexes dont l'environnement est dynamique, mais elle ne résout pas clairement le problème d'interaction entre les différents modules [19].

III.1.4 Concept des systèmes multi agents

III.1.4.1 Définition

Un système multi-agent est « un système composé d'un ensemble d'entités autonomes et intelligentes qui coordonnent leurs connaissances pour atteindre un objectif ou résoudre un problème (Weiss, 1999)».
C'est une branche de l'intelligence artificielle distribuée **(IAD)** qui repose sur le principe de faire interagir plusieurs agents entre eux pour réaliser un objectif global, tel que chaque agent détient une représentation partielle de son environnement, et possède des connaissances et des compétences propres qui lui permettent d'évoluer dans l'environnement commun de manière à satisfaire son but local.

L'interaction entre ces agents se présente sous forme de coopération, coordination, concurrence, compétition, négociation, etc [22].

Un système multi-agents peut être [13] :

- Ouvert : Les agents y entrent et sortent librement.
- Fermé : L'ensemble des agents constituant le système reste inchangé.
- Hétérogène : Les agents composants le système ont des modèles différents.
- Homogène : Tous les agents composants le système sont construits sur le même modèle d'organisation.

Définition 3.1.2

D'après Ferber [20] : « On peut définir un système multi-agents comme un système composé des éléments suivants :

- **un environnement**,
- **Un ensemble d'objets** situés, c'est-à-dire auxquels on peut associer une position dans l'environnement,
- **un ensemble d'agents**, qui sont des objets particuliers (les agents diffèrent des autres objets par leur capacité à agir sur eux-mêmes et sur l'environnement),
- **Un ensemble de relations**, ou de contraintes, qui unit des objets et/ou des agents entre eux,
- **Un ensemble d'opérations**, déterminant les façons dont les agents peuvent agir sur les objets,
- **Un ensemble d'opérateurs**, représentant la façon dont ces actions sont effectuées et leur effet sur l'environnement.»

III.1.4.2 Typologie des systèmes multi-agents

Comme il existe deux types principaux d'agents, nous comptons ainsi deux grandes classes de systèmes multi-agents en fonction du type des agents qui les composent, à savoir les systèmes multi-agents cognitifs et les systèmes multi-agents réactifs.

- **Systèmes multi-agents réactifs**
  Ces systèmes sont composés d'agents réactifs souvent en grand nombre. Dans un tel système, un comportement global complexe peut apparaître comme le résultat des interactions entre des composants simples en grande quantité. Ces systèmes se basent sur l'hypothèse qu'il est possible de produire des comportements collectifs intelligents complexes malgré la simplicité des comportements individuels.
- **Systèmes multi-agents cognitifs**
  Contrairement aux systèmes multi-agents réactifs, Les systèmes multi-agents cognitifs sont des systèmes constitués d'agents cognitifs généralement en faible nombre. Leur étude cherche à améliorer les comportements individuels des agents en s'intéressant à leur intelligence individuelle, leur mode cognitif et à la communication. Elle met l'accent sur l'agent et ses capacités.

III.2 Caractéristiques des SMA

Un système multi-agents possède plusieurs caractéristiques dont les principales sont [17] :

- **L'autonomie :** Dans les SMA, l'autonomie peut se définir en trois points essentiels l'existence propre et indépendance de l'agent, le maintien de sa viabilité en dehors de contrôle extérieur et la prise de décision en tenant compte uniquement de ses perceptions et de ses connaissances.
- **La distribution :** Dans un environnement multi-agents, la distribution signifie que plusieurs agents participent à la réalisation d'un objectif global en se partageant les connaissances, les traitements, les tâches et les ressources.
- **La décentralisation :** La décentralisation signifie la répartition du contrôle. Elle peut être dictée, entre autres, par des contraintes liées à la distribution physique du système ou par des limitations des capacités de décision globale.
- **La communication :** La communication permet aux agents d'échanger des informations et dassurer la cohérence du comportement global du système malgré la décentralisation. Il existe deux moyens de communication entre les agents (indirect et direct).
- **L'interaction :** Les agents interagissent au sein de l'environnement dans lequel ils évoluent. Ces interactions sont réalisées à travers la communication, et permettent ainsi d'exprimer des stratégies de coopération, collaboration, compétition et négociation entre les agents.

- **L'organisation :** Il existe de multiples relations complexes qui unissent les agents et qui peuvent porter sur les buts, plans, actions ou ressources, ces relations induisent des schémas globaux d'interactions entre les agents. Les organisations permettent donc de formaliser ces schémas et offrent un moyen de spécifier et de concevoir une structure du SMA.
- **L'adaptation :** L'adaptation est la capacité du système à modifier son comportement en cours de fonctionnement pour l'ajuster dans un milieu dynamique.
- **L'ouverture :** Les SMA sont constitués de plusieurs entités autonomes, hétérogènes, en interaction entre elles au sein d'environnements dynamiques. Les SMA sont ainsi caractérisés par l'ouverture qui se manifeste par l'évolution fonctionnelle du système. Cette évolution correspond à l'ajout, la modification ou la suppression dynamique d'entités du système.
- **L'émergence :** L'émergence est l'apparition progressive de comportements non spécifiés a priori au sein du système. En effet, la fonction globale du système est attendue à partir des spécifications locales de chacun des agents, elle n'est pas programmée à l'avance et elle apparaît comme résultat des interactions des agents entre eux.
- **La situation dans un environnement :** L'environnement d'un SMA est vu comme étant un espace partagé par l'ensemble des agents. C'est le lieu commun au quel les agents agissent et s'influencent les uns les autres. Les agents interagissent avec leur environnement par le biais des perceptions et des actions qu'ils peuvent effectuer sur lui.

III.3 Organisation dans les SMA

Dans un SMA, les agents ont une vision locale de leur environnement, ils sont amenés à coopérer pour atteindre l'objectif global du système. Il est donc nécessaire de définir une structure organisationnelle à l'intérieur du SMA afin d'établir :

— La communication entre les agents,

— L'Interaction entre agents,

— La négociation entre les agents.

III.3.1 Communication entre agents

Les agents communiquent entre eux pour pouvoir échanger des données, des informations et coordonner leurs activités. Ils peuvent interagir de deux manières, soit par signaux, via l'environnement, soit directe par envois de messages.

Il existe deux types de communication : les communications directes et indirectes [70].

- **La communication indirecte :** Le partage d'information dans cette démarche de communication passe généralement par l'environnement. Ainsi, pour assurer la communication, les agents doivent agir sur leur environnement. Ce mode de communication implique que les agents doivent être implémentés dans un environnement physique commun.

- **La communication directe :** La communication directe est une démarche de communication, où l'agent s'adresse individuellement aux autres par l'envoi de messages. Ce type de communication est basé sur les trois éléments suivants :

  — Le langage de communication : L'intérêt des langages de communication est de faciliter l'échange et l'interprétation des messages et l'interopérabilité entre les agents. Ces langages se focalisent essentiellement sur la manière de décrire exhaustivement des actes de communication d'un point de vue syntaxique et sémantique.

  — L'ontologie : est une spécification d'objets, de concepts et de relations dans un domaine d'intérêt. L'intérêt dune ontologie quand elle est partagée par des agents pour représenter leur connaissance est qu'ils ont les moyens de comprendre les « mots » utilisés dans une communication.

  — Les supports de communication : sont des mécanismes utilisés pour stocker et rechercher des messages.

III.3.2 Interaction entre agents

On appelle situation d'interaction, un ensemble de comportements résultant du regroupement d'agents qui doivent agir pour satisfaire leurs objectifs en tenant compte des contraintes provenant des ressources plus ou moins limitées dont ils disposent et de leurs compétences individuelles.

En général, les interactions sont mises en oeuvre par un transfert dinformations entre agents ou bien entre l'environnement et les agents, soit par perception, soit par communication. L'interaction peut être décomposée en trois phases non nécessairement séquentielles [41] :

— Réception d'informations,

— Raisonnement sur les autres agents à partir des informations acquises,

— Emission de messages ou plusieurs actions modifiant l'environnement.

III.3.2.1 Coopération

La coopération désigne que les agents doivent s'entraider pour atteindre leurs objectifs qui peuvent être éventuellement communs. On dira que plusieurs agents coopèrent ou encore qu'ils sont dans une situation de coopération si l'une des conditions suivantes est vérifiée :

— L'ajout d'un nouvel agent permet d'accroître différentiellement les performances du groupe,

— L'action des agents sert à éviter ou à résoudre des conflits potentiels ou actuels.

III.3.2.2 Coordination

La coordination est un processus dans lequel les agents se sont engagés en vue d'assurer une communauté d'agents individuels agissant avec cohérence et harmonie [74]. Les agents ont besoin de la coordination, pour être coordonnés de la même manière, puisqu'aucun agent ne possède une vue globale sur le système et parce qu'ils possèdent des capacités et expertises différentes, mais complémentaires.

La coordination comprend aussi l'allocation des tâches, qui consiste à affecter des responsabilités et des ressources nécessaires à la résolution de problèmes à un agent. Le créateur du système d'agents peut allouer toutes les tâches d'avance en engendrant ainsi une organisation de résolution des problèmes qui est non adaptable. Par contre, on peut avoir une allocation dynamique et flexible des tâches.

La planification fait également partie de la coordination. Pour un agent, elle constitue un processus de construction d'une séquence d'actions en tenant compte seulement des objectifs, des capacités et des contraintes environnementales. La planification a pour rôle d'éviter les conflits.

III.3.2.3 Négociation

La négociation est un processus de coordination et de résolution de conflits, son objectif est d'atteindre un accord final accepté par un groupe d'agents. Il existe deux types de négociation : les négociations compétitives et coopératives [74].

— **Négociation compétitive :** La négociation compétitive est valable dans la situation où des agents de différents intérêts tentent de faire un choix de groupe avec des alternatives bien définies.

— **Négociation coopérative :** La négociation coopérative est utilisée dans la situation où les agents ont un objectif unique global considéré pour le système.

III.3.3 Avantages des SMA

L'utilisation des Systèmes Multi-Agents **(SMA)** présente une série d'avantages [48], qui se résume comme suit :

- **Système dynamique :** Les agents sont structurés afin d'exercer une influence sur chacun pour faire évoluer le système dans son ensemble (système dynamique). On rencontre de nombreuses interactions entre agents telles que la coordination, la négociation et la coopération.
- **Robustesse et sûreté de fonctionnement :** La mise hors fonctionnement de quelques agents ne modifie pas sensiblement le comportement global du système.
- **Souplesse de l'outil informatique :** Qui permet de modifier le comportement des agents.
- **lexibilité et traitement des systèmes à grandes échelles :** Nous pouvons toujours augmenter le nombre d'agents pour traiter des systèmes de plus en plus grands, sans pour autant perturber le travail des agents existants.
- **Résolution distribuée de problèmes :** Il est possible de décomposer un problème en sous-parties et de résoudre chacune de façon indépendante pour aboutir à une solution stable.

Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur les agents et les systèmes multi-agents, tout en mettant l'accent sur l'organisation dans les SMA et les concepts d'environnements. Dans le chapitre suivant, nous allons modéliser les **CSP** par les **SMA**.

Chapitre IV

Modélisation des CSP par les SMA

Introduction

Dans le chapitre précédent, nous avons introduit les généralités de systèmes multi-agents. 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.

La première section de ce chapitre sera consacrée à la présentation des **DisCSP**. Nous verrons pour commencer, des définitions formelles portant sur les **DisCSP**, nous exposerons ainsi ses fonctionnement et défis. Puis, nous détaillerons les algorithmes de résolutions des **DisCSP**. Enfin, la conclusion .

IV.1 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 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. 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. Nous allons commencer, dans la section suivante, par énoncer des définitions sur les **DisCSP**.

IV.1.1 Définition des DisCSP

Définition 4.1 (DisCSP)

Un CSP Distribué est un tuple (X, D, C, A, F) où [73] :
X : est un ensemble $\{x_1, x_2..., x_n\}$ de variables .
D : est un ensemble de domaines finis $\{dx_1, dx_2..., dx_n\}$ tel que le domaine dx_i soit associé à la variable x_i.
C : est un ensemble $\{c_1, c_2..., c_n\}$ de contraintes.
$A = \{A_1, A_2..., A_p\}$ est un ensemble de p agents et $F : X \leftarrow A$ est une fonction qui associe chaque variable à un agent.

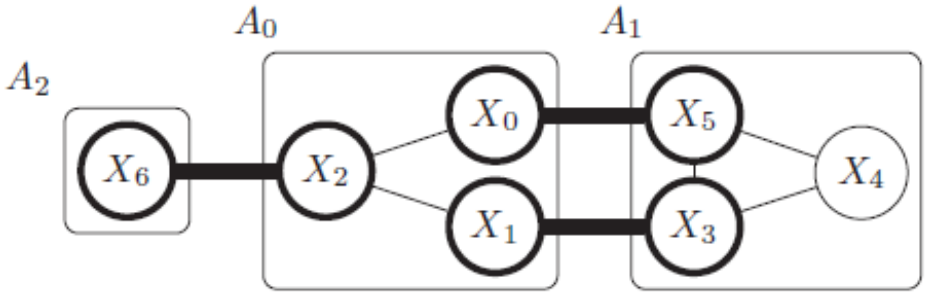

Figure IV.1 – Exemple de CSP Distribué [73].

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

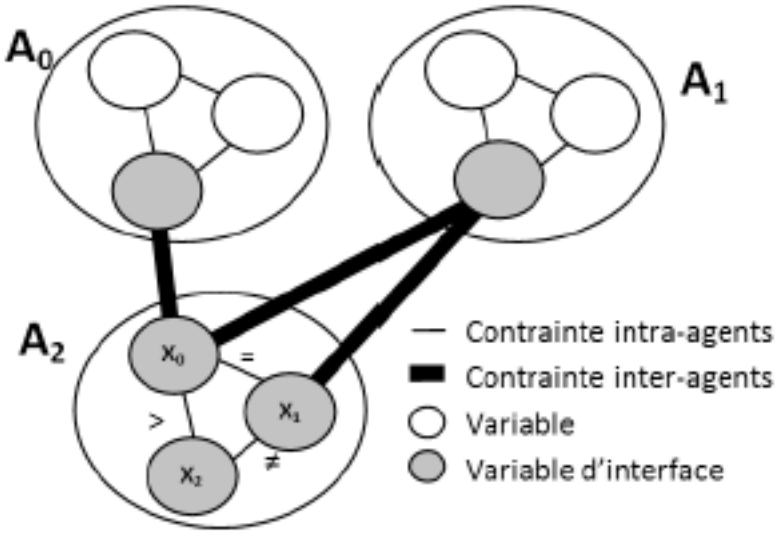

FIGURE IV.2 – DisCSP composé de trois agents [73].

Un DisCSP est présenté à la figure IV.2. Dans cet exemple, l'ensemble des agents est $A = \{A_0, A_1, A_2\}$. Concernant l'agent A_2, son CSP contient trois variables $X_{A2} = \{x_0, x_1, x_2\}$. et trois relations intra-agents, à savoir $(x_0 = x_1)$, $(x_0 > x_2)$ et $(x_1 \neq x_2)$. Les variables concernées par les C_{inter} sont appelées variables d'interfaces.

Définition 4.2 (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.

IV.1.2 Les défis des DisCSP

Les problèmes de satisfaction de contraintes distribués posent les mêmes problèmes que les CSP centralisés (explosion combinatoire, backtrack, filtrage). Mais de nouveaux défis spécifiques à la distribution apparaissent tels que la synchronisation des messages échangés, le coût des communications, la complexité de l'algorithme et également la confidentialité des données [73].

— **Synchronisation**

Les DisCSP sont sujets aux problèmes traditionnels des applications distribuées, l'ordre et les temps d'arrivée des messages ne sont pas garantis. Les algorithmes des **DisCSP** doivent donc tenir compte de ces problèmes et mettre en place des stratégies asynchrones.

— **Le coût des messages échangés**

Puisque les variables et les contraintes sont distribués entre différents agents, cette vérification nécessite souvent des échanges de messages. Ce coût de communication a un impact direct sur l'efficacité d'un algorithme. Par conséquence, un grand défi des algorithmes DisCSP est de réduire le nombre de communications entre les agents.

— **Problème de distribution**

Par définition des DisCSP, toutes les variables et toutes les contraintes sont distribuées sur un ensemble d'agents. Quelques éléments du problème sont publics et d'autres privés. Ceci complique la recherche d'une solution et réduit l'efficacité. Quand un agent n'a pas une vue globale sur le problème, il peut proposer une valeur à sa variable qui est incompatible avec la variable privée d'un autre agent.

L'incompatibilité des deux valeurs exige des communications supplémentaires afin que l'un des agents change la valeur instanciée à sa (ses) variable(s). Ces défis ont justifié le développement de nombreux algorithmes de résolution de **DisCSP**.

IV.2 Algorithmes de résolution des DisCSP

Nous nous intéressons dans cette section, aux techniques de résolution des CSP distribués et les différents travaux existant.

IV.2.1 La famille "ABT"

Dans les algorithmes de résolution de DisCSP, La plupart des recherches ont été focalisées sur un principe d'exploration complète et asynchrone de l'espace de recherche. Ces algorithmes font agir les agents en parallèle et leurs communication s'effectuent, si besoin est, pour maintenir la consistance des instanciations de variables qu'ils contrôlent. L'algorithme fondateur est appelé **"Asynchronous BackTracking" (ABT)** [73].

L'algorithme **ABT** effectue une exploration complète de l'espace de recherche.Cet algorithme est considéré comme une référence pour la conception de tout nouvel algorithme de résolution de DisCSP. L'algorithme ABT utilise un mode asynchrone pour la communication et procède par des envois de messages qui contiennent soit :

— une instanciation courante des variables de l'agent émetteur, ces messages sont notés "$ok?$",

— Une demande de BackTrack. Un tel message est appelé *nogood* et noté "ngd",

— Une demande d'ajout de lien qui permet de ne pas revisiter plusieurs fois la même partie de l'espace de recherche. Ces liens sont ajoutés de manière dynamique. Ce type de message "$add-link-request$", est noté "adl".

L'algorithme impose un ordre statique entre les agents selon un ordre de priorité. Cet ordre permet de déterminer le receveur d'un *nogood* (i.e. l'agent devant changer sa valeur à cause d'un BackTrack). Quand un agent choisit une valeur à sa variable il la communique à ses voisins successeurs dans le graphe de contrainte via un message "*ok*?". Le choix de cette valeur fait en sorte de ne pas violer les contraintes reliant l'agent à ses prédécesseurs dans le graphe de contraintes. Quand un agent reçoit un message "*ok*?" il vérifie que sa valeur ne rentre pas en conflit avec la nouvelle valeur reçue, sinon il change sa valeur afin de ne plus être en conflit et prévient ses successeurs du changement.
Si le domaine d'une de ses variables est vidé suite au dernier message reçu, l'agent envoie un *nogood* au plus proche prédécesseur qui est responsable de cet échec.

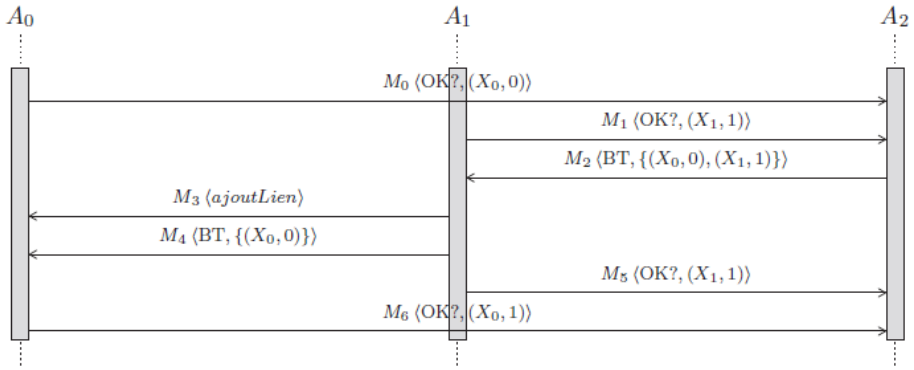

FIGURE IV.3 – Echange de messages entre les agents exécutant ABT [53].

Cette figure (IV.3) représente, 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. Lagent A_1 choisit la valeur 1 pour sa variable X_1 et la transmet grâce au message M_2. 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 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= $\{(X_0, 0), (X_1, 1)\}$ 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 :$\{(X_0, 1), (X_1, 1), (X_2, 0)\}$.

IV.2.2 Les différentes variantes de l'algorithme ABT

IV.2.2.1 Distributed Synchronous Backtracking (DBT)

L'algorithme synchrone backtracking distribué (DBT) constitue la méthode de base pour la résolution des DisCSP [73]. Il est basé sur un algorithme de backtracking simple.
Dans cet algorithme, chaque agent a exactement une variable. L'ordre des agents est défini de manière statique lors d'une phase de prétraitement en appliquant une simple heuristique pour définir l'ordre d'instanciation des variable. Chaque agent effectue un traitement synchrone, ce qui signifie ici qu'à un temps donné, un seul agent est actif.

Le principe est le même que celui du Backtracking centralisé, la seule différence réside dans la vérification des contraintes qui exige une communication avec les agents qui détiennent une partie du problème. Quand l'agent reçoit une instanciation partielle, il essaye d'étendre cette instanciation en lui ajoutant sa propre variable affectée d'une valeur de son domaine. Si c'est réussi, la nouvelle instanciation est transmise à l'agent suivant. Si aucune valeur n'est trouvée pour l'instanciation partielle reçue, alors l'agent renvoie un message de backtracking à l'agent émetteur.

Quand le dernier agent (dans l'ordre prédéfini au départ) a trouvé une affectation à sa propre variable, il annonce à tous les autres agents que la solution a été trouvée. Le défaut de cet algorithme est qu'il ne profite pas des avantages du parallélisme, et que l'ordre d'instanciation des variables est statique.

IV.2.2.2 Asynchronous Weak-Commitment (AWC)

C'est une extension de l'algorithme ABT, elle étend au cadre distribué la méthode séquentielle dite Weak Commitment search.
Le principe de cette méthode séquentielle consiste à réviser l'ordre d'instanciation des variables lors de chaque échec rencontré. Le même principe est utilisé dans le cadre distribué. Nous signalons que dans le cadre distribué, chaque changement local de priorité est diffusé à l'ensemble des agents, ce qui entraîne une augmentation des communications.
Ces diffusions entraînent une perte de localité pour les informations locales de chaque agent. De plus, deux agents du système peuvent échanger successivement leurs niveaux de priorité.

Cet algorithme enregistre les solutions partielles abandonnées comme de nouvelles contraintes, et évite de créer les mêmes solutions partielle qui ont été créée et abandonnée auparavant. Par conséquent, l'exhaustivité de l'algorithme (trouve toujours une solution s'il en existe une et se termine si aucune solution n'existe) est garanti.

IV.2.2.3 Dynamic Backtracking(DiBT)

C'est un algorithme calqué sur le modèle du Backtrack dynamique distribué. Il réalise des sauts dynamiques sur l'ensemble des agents en conflit. La recherche d'une solution nécessite la mise en place d'un ordre total entre les agents.

Cette phase est exécutée avant la phase de résolution. Il construit une hiérarchie d'agents en utilisant des critères heuristiques. Cette hié-rarchie induit un ordre partiel sur les agents, qui doit être complète pour former ensuite un ordre total et garantir la complétude.

Le principe de cette heuristique consiste à définir pour chaque agent, appelé $self$, $\lceil -$(self), qui désigne l'ensemble des agents partageant une contrainte avec self et qui sont classés plus haut dans la hiérarchie. Réciproquement, l'heuristique définit $\lceil +$(self), l'ensemble des voisins de $self$ classés plus bas dans la hiérarchie. Chacun des agents exécute l'algorithme **DiBT** et mémorise un ensemble de *nogoods*.
Le contexte de $self$ est l'ensemble des valeurs affectées (à sa connaissance),aux agents le précédant dans l'ordre. Il est toujours cohérent avec l'ensemble des *nogoods* mémorisés localement. Les agents échangent des affectations et des *nogoods*. Les affectations sont toujours acceptées, et le contexte mis à jour en conséquence. Un *nogood* est accepté s'il est cohérent avec le contexte de l'agent et sa propre instanciation. S'il est accepté, il pourra justifier le retrait de la valeur courante.

IV.2.2.4 Asynchrone Forward Checking (AFC)

C'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 dalgorithme de détection de la terminaison. Lorsqu'aucune solution n'existe, un agent le détecte et arrête la résolution du problème.

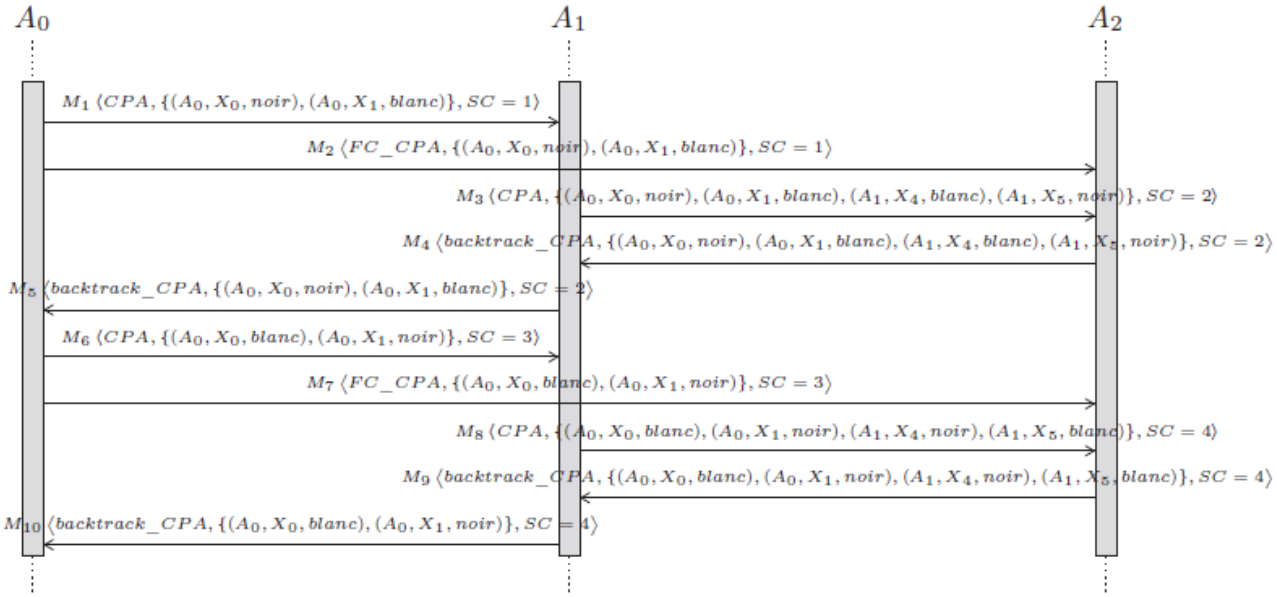

FIGURE IV.4 – Echange de messages entre les agents [52].

IV.2.2.5 Concurrent Backtracking (ConBT)

Dans ce travail, les auteurs [42] s'inspirent des travaux effectués dans le cadre d'un CSP centralisé, où la concurrence consiste à lancer en parallèle plusieurs solveurs sur le même problème. Avant tout démarrage, une phase de prétraitement est exécutée dans le but de définir l'ordre d'instanciation des variables ainsi qu'une phase de lancement des processus de recherche par l'agent le plus prioritaire. Autrement dit, par l'agent qui possède la première variable à instancier selon l'ordre prédéfini. Le nombre de processus lancés K doit être inférieur ou égal à la taille du domaine de la première variable à instancier. chaque processus a en charge l'exploration exhaustive de son sous espace.
La Figure IV.5 illustre cette exploration. Nous supposons que le problème est distribué entre trois agents A_1 ; A_2et A_3. Chaque agent possède une variable x_1 ; x_2 et x_3 respectivement, et l'ordre d'exécution est le suivant A_1 ; A_2 et A_3, donc l'agent A_1 crée deux processus de recherche p_1 et p_2. Le premier processus explore l'espace engendré par l'instanciation X_0 = a, alors que le second explore le sous espace restant.

Le problème dans cette version que les agents plus prioritaires dans l'ordre d'instanciation de leurs variables sont plus chargés par rapport aux autres (moins prioritaires). Le second problème est de ne pas exploiter les avantages de parallélisme ni de la coopération entre les différents processus de recherche. Ces inconvénients influent sur les performances de l'algorithme. La simulation des résultats de cet algorithme montrent l'efficacité de **ConBT** par rapport ABT.

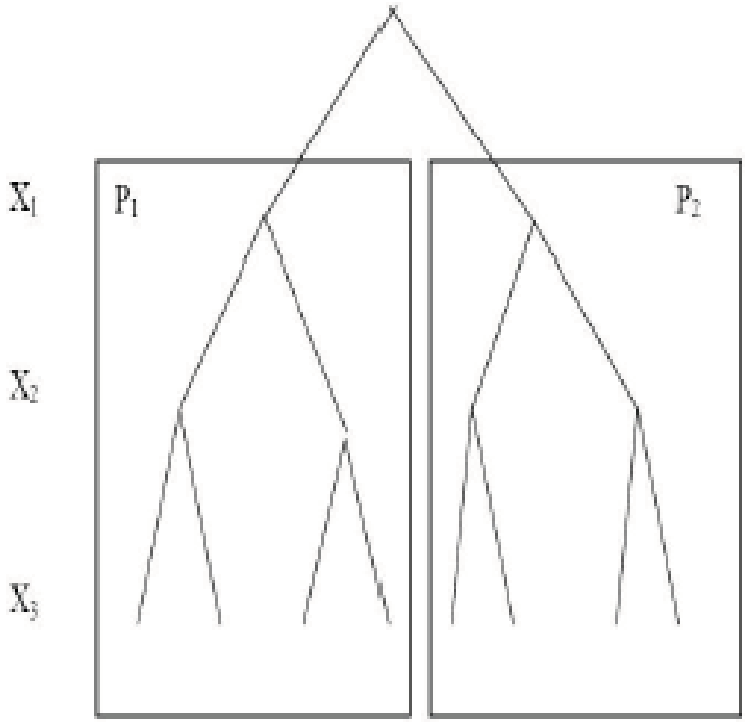

FIGURE IV.5 – Concurrent Backtracking [42].

Conclusion

Dans ce chapitre, nous avons présenté des méthodes de résolutions des problèmes de satisfaction contraintes distribués **DisCSP**. 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 eux.

Dans le chapitre suivant, nous exposerons les résultats d'application de deux algorithmes Distribués sur quelques problèmes qui peuvent être modélisés et résolus par le paradigme **DisCSP**.

Chapitre V

Implémentation et Réalisation

Introduction

Un problème de satisfaction de contraintes distribué **(Distributed CSP)** est un CSP dans lequel les variables et les contraintes sont réparties entre plusieurs agents automatisés et divers problèmes d'application dans l'intelligence artificielle Distribuée peuvent être formalisée en tant que CSP distribués.

Au cours de ce chapitre, Nous présentons quelques exemples de problèmes qui peuvent être modélisés et résolu par le paradigme DisCSP, les algorithmes de résolutions en montrant les moyens avec les idées adoptées, Enfin, les résultats obtenus.

V.1 Problèmes traités

Nous décrivons ici quelques exemples de problèmes qui peuvent être modélisés et résolus par le paradigme DisCSP.

V.1.1 N-reines

Le problème des N**-reines**, inspiré du jeu d'échec, est un problème combinatoire qui peut être formalisé et résolu par un problème de satisfaction de contraintes. Il consiste à positionner N reines dans un échiquier de dimension $N \times N$ de telle sorte qu'aucune reine ne puisse atteindre une autre : deux reines ne sont jamais sur la même ligne, la même colonne ou la même diagonale.

Classiquement, ce problème est modélisé en utilisant le formalisme des CSP dont le but est de trouver une configuration qui satisfait les conditions données (contraintes), où les variables à affecter sont les coordonnées des reines, les domaines de valeurs sont des ensembles de taille N et les contraintes impliquent que les reines ne peuvent être sur la même ligne, colonne ou diagonale. Ici, **les agents** sont les reines.

La figure V.1 représente une solution possible de la resolution de N-reines (N=4)

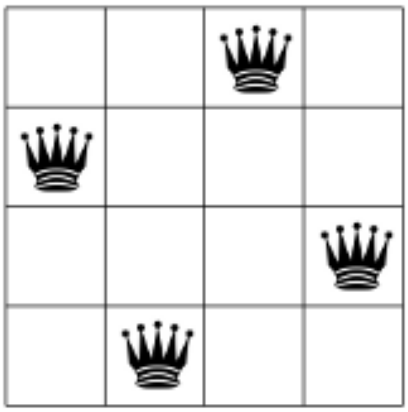

FIGURE V.1 – Résolution de problème de 4 reines [12].

V.1.2 Coloriage de graphe

Un autre exemple de problème typique est le problème de coloration de graphe. La coloration du graphe est l'un des problèmes les plus combinatoires étudiés en intelligence artificielle.
Le but de ce problème consiste à colorer tous les noeuds d'un graphe par des couleurs de sorte que deux sommets quelconques reliés par un arc (adjacents) soient coloriées avec deux couleurs différentes.

Le problème de coloration de graphes est simplement formalisé comme un CSP. D'où les noeuds du graphe sont les variables à colorier et les couleurs possibles de chaque noeud/variable forment son domaine. Il existe une contrainte entre chaque paire de variables/noeuds adjacents qui interdit à ces variables d'avoir la même couleur. La figure V.2 représente une solution possible pour le problème de coloriage de graphe avec 6 noeuds.

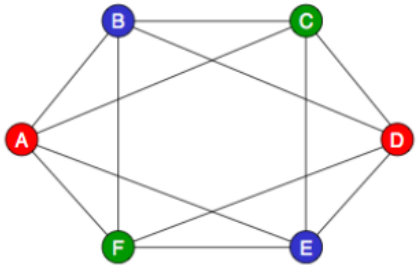

FIGURE V.2 – Résolution de coloriage de graphe [73].

V.2 Environnement de travail

Afin de bien mener ce projet, nous avons utilisé un ensemble de matériels dont les principales caractéristiques sont les suivantes :

— **Processeur** : Intel(R) Core(TM) i5-4310U CPU @ 2.00GHz 2.60GHz.
— **RAM** : 4.00 GO.
— **Système dexploitation** : Windows 10 64 bits, processeur x64.

V.3 Outils utilisés

V.3.1 Langages de programmation : Python

Python est le langage de programmation open source, multiplateformes et orienté objet le plus employé par les informaticiens. C'est un Langage principalement utilisé pour le machine Learning [1] et la data science [2], l'intelligence artificielle (génétique, neurones), python a fortement contribué à l'essor du big data [3].

FIGURE V.3 – Logo Python [5].

Grâce à ses nombreuses bibliothèques telles Panda, Bokeh, Numpy, Scipy, Scrapy, Matpotlib, Scikit-Learn ou encore TensorFlow, Python offre une grande flexibilité dans les tâches à effectuer et une grande compatibilité quelle que soit la plateforme utilisée.

les principales caractéristiques de Python sont :

- **Python est extensible** : comme Tcl ou Guile, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- **Python est dynamiquement typé** : Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- **La bibliothèque standard** de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standards (fichiers, pipes, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui **continue à évoluer**, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.

1. Machine Learning est une technologie d'intelligence artificielle permettant aux ordinateurs dapprendre sans avoir été programmés explicitement à cet effet.

2. data science est un domaine interdisciplinaire qui utilise des méthodes, des processus, des algorithmes et des systèmes scientifiques pour extraire des connaissances et des idées de nombreuses données structurelles et non structurées.

3. des données plus variées, arrivant dans des volumes de plus en plus importants et avec une vitesse plus élevée. Cette définition est également connue sous le nom des trois « V » Volume Vitesse et Variété

- **Version :** Python 3.7.

V.3.2 PyCharm

PyCharm est un environnement de développement intégré utilisé pour programmer en Python. Il permet l'analyse de code et contient un débogueur graphique. Il permet également la gestion des tests unitaires, l'intégration de logiciel de gestion de versions, et supporte le développement web avec Django.

Développé par l'entreprise tchèque JetBrains, c'est un logiciel multi-plateforme qui fonctionne sous Windows, Mac OS X et GNU/Linux. Il est décliné en édition professionnelle, diffusé sous licence propriétaire, et en édition communautaire diffusé sous licence Apache.

FIGURE V.4 – Logo Pycham [4].

V.3.3 Librairies utilisées

- **Matplotlib** : Est une bibliothèque du langage de programmation Python destinée à tracer et visualiser des données sous formes graphiques. Elle peut être combinée avec les bibliothèques python de calcul scientifique NumPy et SciPy [4]. Matplotlib est distribuée librement et gratuitement sous une licence de style BSD4 [5]. Sa version stable actuelle (la 2.0.1 en 2017) est compatible avec la version 3 de Python.

- **NumPy :** Numpy est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux. Plus précisément, cette bibliothèque logicielle libre et open source fournit de multiples fonctions permettant notamment de créer directement un tableau depuis un fichier ou au contraire de sauvegarder un tableau dans un fichier, et manipuler des vecteurs, matrices et polynômes.

4. est un projet visant à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique. Scipy utilise les tableaux et matrices du module NumPy.

5. La licence BSD (Berkeley Software Distribution License) est une licence libre utilisée pour la distribution de logiciels. Elle permet de réutiliser tout ou une partie du logiciel sans restriction, qu'il soit intégré dans un logiciel libre ou propriétaire.

- **Ctypes :** Ctypes est une bibliothèque d'appel à des fonctions externes en python. Elle fournit des types de données compatibles avec le langage C et permet d'appeler des fonctions depuis des DLL ou des bibliothèques partagées, rendant ainsi possible l'interfaçage de ces bibliothèques avec du pur code Python.
- **Random :** Random est un module **Python** regroupant plusieurs fonctions permettant de travailler avec des valeurs aléatoires. La distribution des nombres aléatoires est réalisée par le générateur de nombres pseudo-aléatoires Mersenne Twister, l'un des générateurs les plus testés et utilisés dans le monde informatique.

V.3.4 Algorithmes utilisés

Notre approche repose sur deux algorithmes distribués **"Asynchronous BackTracking" (ABT)** et **"Asynchronous Weak-Commitment (AWC)"** (définis et détaillés déja dans le chapitre précédent, sous-section IV.2.1, IV.2.2.2 respectivement).

ALGORITHM 9: Algorithme ABT (1ère partie)

```
   procedure ABT_kernel()
1. myValue←empty;
2. end←false;
3. CheckAgentView();
4. while (¬ end) do
5.     msg←getMsg();
6.     switch (msg.type) do
7.         ok? : ProcessInfo(msg);
8.         ngd : ResolveConflict(msg);
9.         stp : end ← true;
   procedure CheckAgentView(msg)
10. if (¬ consistent(myValue,myAgentView) then
11.     myValue ← ChooseValue();
12.     if (myValue) then
13.         foreach (child ∈ Γ+ (x_i)) do
14.             sendMsg :ok?(child,myValue);
15.     else
16.         Backtrack();
   procedure ProcessInfo(msg)
17. Update(myAgentView,msg.Assig);
18. CheckAgentView();
   procedure ResolveConflict(msg)
19. if (Coherent(msg.Nogood, Γ− (x_i))) then
20.     foreach (assig ∈ lhs(msg.Nogood) Γ− (x_i)) do
21.         Update(myAgentView, assig)
22.     add(msg.Nogood,myNogoodStore);
23.     myValue ← empty;
24.     CheckAgentView();
25. else
26.     if (msg.sender ∈ Γ+ (x_i) ∧ Coherent(msg.Nogood, x_i) then
27.         sendMsg.ok?(msg.sender,myValue)
```

FIGURE V.5 – Algorithme ABT (Partie 1) [72].

ALGORITHM 10: Algorithme ABT (2ème partie)

```
    procedure Backtrack()
25. newNogood ← solve(myNogoodStore);
26. if (newNogood = ∅) then
27.     end ← true;
28.     sendMsg.stp(system);
29. else
30.     sendMsg.ngd(newNogood);
31.     Update(myAgentVew, rhs(newNogood)←unknown);
32.     CheckAgentView();
    function ChooseValue()
33. foreach (v_i ∈ d_i not eliminated by myNogoodStore) do
34.     if (consistent(v_i, myAgentView) then
35.         return v_i;
36.     else
37.         add(x_j = v_j ⇒ x_i ≠ v_i, myNogoodStore);
38.         return empty;
    procedure Update(myAgentView, newAssig)
39. add(newAssig,myAgentView);
40. foreach (ng ∈ myNogoodStore) do
41.     if (Coherent(lhs(ng), myAgentView)) then
42.         remove(ng,myNogoodStore);
    function Coherent(nogood, agents)
43. foreach (var ∈ nogood ∪ agents) do
44.     if (nogood[var] ≠ myAgentView[var]) then
45.         return false;
46.     return true;
```

FIGURE V.6 – Algorithme ABT (Partie 2) [72].

```
procedure weak-commitment(left, partial-solution)
  when all variables in left satisfy all constraints do
      terminate the algorithm, current value assignment is a solution; end do
  (x_i, d) ← a variable and value pair in left that does not satisfy some constraint;
  values ← the list of x_i's values that are consistent with partial-solution;
  if values is an empty list;
    if partial-solution is an empty list
      then terminate the algorithm since there exists no solution;
      else record partial-solution as a new constraint (nogood);
        remove each element of partial-solution and add to left;
        call weak-commitment(left, partial-solution); end if;
    else value ← the value within values that minimizes
        the number of constraint violations with left;
      remove (x_i, d) from left;
      add (x_i, value) to partial-solution;
      call weak-commitment(left, partial-solution); end if;
```

FIGURE V.7 – Algorithme AWC [72].

V.4 Exemple d'xécution des algorithmes sur les problèmes traités

Nous illustrons l'exécution de l'algorithme en utilisant une version distribuée du problème des N-reines décrit dans sous-section V.1.1 (cas de $N = 4$). Il existe quatre agents, chacun correspondant à une reine dans chaque rangée. Le but des agents est de trouver des positions sur un échiquier 4×4 afin que les reines ne menace les uns les autres.

V.4.1 Asynchronous backtracking

Les valeurs initiales sont présentées sur la figure V.8 (a) (Il faut noter que la trace de l'exécution de l'algorithme peut varier de manière significative selon le retard des messages). Les agents communiquent ces valeurs entre eux et l'ordre de priorité est déterminé par l'ordre alphabétique des identifiants. Les agents sauf x_1 changent leurs valeurs, de tel sorte que la nouvelle valeur est cohérente avec son $agentView$ (Figure V.8 (b)), donc l'agent x_2 change sa valeur à 3, ce qui est cohérent avec la valeur de x_1. Puisque x_2 change de valeur, la valeur de x_3 n'est plus cohérente avec la nouvelle valeur, l'agent x_3 change sa valeur en 4, ce qui est cohérent avec la valeur de x_1 et x_2.

Et comme il n'y a pas de valeur cohérente pour l'agent x_4, il envoie un message *nogood* à x_3 et modifie sa valeur afin que la valeur soit cohérente avec son $agentView$, à l'exception de la valeur de x_3. Notez que x_3 ignorera ce message *nogood* car il a changé de valeur avant de recevoir ce message. Les agents envoient d'accord ? messages à d'autres agents. Alors, x_3 ne satisfait pas les contraintes avec x_1 etx_2, et il n'y a pas de valeur cohérente, tandis que les valeurs des autres agents sont cohérentes avec leur point de $agentView$. Par conséquent, x_3 envoie un $agentView$ message à x_2. Après avoir reçu ce $agentView$ message, x_2 change sa valeur en 4 (Figure V.8 (c)).

Ensuite, x_3 change sa valeur en 2. Il n'y a pas de valeur cohérente pour l'agent x_4, il envoie un message *nogood* à x_3, et modifie sa valeur afin que la valeur soit cohérente avec son $agentView$, à l'exception de la valeur de x_3 (Figure V.8 (d)). Encore une fois, ce message *nogood* est ignoré. Il n'y a pas de valeur cohérente pour l'agent x_4, il envoie un message *nogood* à x_3.

Après avoir reçu ce message, x_3 n'a pas d'autre valeur cohérente, donc x_3 envoie un message *nogood* à x_2. Après avoir reçu ce message, x_2 n'a pas non plus d'autre valeur cohérente, donc x_2 envoie un message *nogood* à x_1. Ensuite, x_1 change sa valeur en 2 (Figure 8 (e)). Ensuite,x_3 change sa valeur à 1. Il n'y a pas de valeur cohérente pour l'agent x_4, il envoie un message *nogood* à x_3 et modifie son value afin que la valeur soit cohérente avec son $agentView$, à l'exception de la valeur de x_3. Encore une fois, ce *nogood* message est ignoré et une solution est trouvée (Figure V.8 (f)) .

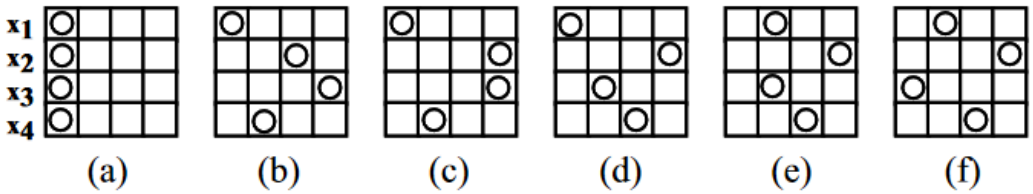

FIGURE V.8 – Exemple d'execution de ABT [51].

V.4.2 Asynchronous weak-commitment search AWC

Les valeurs initiales sont indiquées sur la figure V.9 (a). Les valeurs de priorité initiales sont 0. Puisque elles sont égales, l'ordre de priorité est déterminé par l'ordre alphabétique des identifiants.Par conséquent, seule la valeur de x_4 n'est pas cohérente avec son $agentView$. Comme il n'y a pas de valeur cohérente, l'agent x_4 envoie des messages *nogood* et incrémente sa valeur de priorité. Dans ce cas, la valeur minimisant le nombre de violations de contraintes est de 3, puisqu'il entre en conflit avec x_3 uniquement.
Donc, x_4 sélectionne 3 et envoie *ok*? messages à d'autres agents (figure V.9 (b)). Ensuite, x_3 essaie de changer sa valeur. Puisqu'il n'y a pas de valeur cohérente, l'agent x_3 envoie des messages *nogood* et incrémente sa valeur de priorité. Dans ce cas, la valeur qui minimise le nombre de violations de contraintes est 1 ou 2.

Dans cet exemple, x_3 sélectionne 1 et envoie *ok*? messages à d'autres agents (Figure V.9 (c)). Après cela, x_1 change sa valeur à 2, et une solution est obtenue (Figure V.9 (d)).

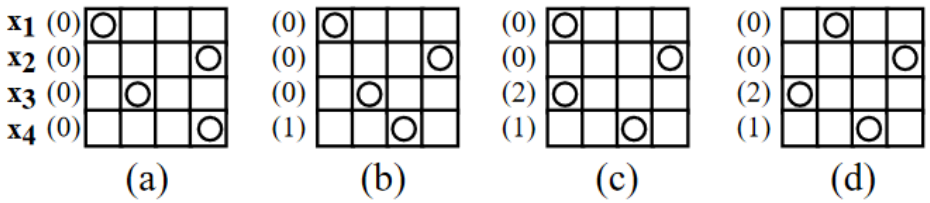

FIGURE V.9 – Exemple d'execution de AWC [51].

V.5 Résultas obtenus

Dans cette section, nous comparons les algorithmes de recherches ABT et AWC pour résoudre les CSP distribués définis dans la section V.1 (problèmes N-reines et coloriage des graphes) où chaque agent maintient sa propre horloge. Le temps d'un agent est incrémenté d'une unité de temps simulée lorsqu'il effectue toujours un cycle de calcul.. Nous évaluons l'efficacité de ces algorithmes en analysant les performances en termes de nombre de cycles nécessaires pour résoudre les problèmes.
Un cycle correspond à une série d'actions d'agent dans lesquelles un agent reconnaît l'état de son environnement (les attributions de valeur d'autres agents), puis décide de sa réponse à cet état, et communique ses décisions.

Nous avons d'abord appliqué ces deux algorithmes au problème distribué n-reines (n entre 10 et 100). Pour chaque n, nous avons généré 100 problèmes, dont chacun avait différentes valeurs initiales générées aléatoirement, et fait la moyenne des résultats pour ces problèmes.
Pour chaque problème, afin de mener les expériences dans un montant raisonnable du temps, nous fixons la limite du nombre de cycles à 100, et terminons l'algorithme si cette limite a été dépassée. Nous montrons la moyenne des essais réussis et le rapport (ratio) de problèmes terminés avec succès au nombre total de problèmes dans le tableau V.1.

n	asynchronous backtracking		asynchronous weak-commitment	
	rapport	cycles	rapport	cycles
10	100%	105.4	100%	41.5
50	50%	662.7	100%	59.1
100	14%	931.4	100%	50.8

Table V.1 – Comparaison entre ABT et AWC (n-reines).

Ensuite, nous avons appliqués les algorithmes sur le problème de coloration de graphe distribué. Ceci est une coloration graphique dans lequel chaque noeud correspond à un agent. Nous montrons les résultats de l'évaluation, où le nombre de variables/agents $n = 60$, 90, et 120, et le nombre de contraintes $m = n \times 2$, et le nombre de couleurs possibles est 3. Nous avons généré 10 problèmes, et pour chaque problème, 10 essais avec des valeurs initiales différentes ont été effectués.
Comme dans le problème distribué n-reines, les valeurs initiales ont été fixées de manière aléatoire. Les résultats sont résumés dans le tableau V.2.

n	asynchronous backtracking		asynchronous weak-commitment	
	rapport	cycles	rapport	cycles
60	13%	917.4	100%	59.4
90	0%	_	100%	70.1
120	0%	_	100%	106.4

Table V.2 – Comparaison entre ABT et AWC (Coloriage de graphe).

V.6 Evaluation

Nous pouvons voir les faits suivants à partir des résultats précédents :

- L'algorithme Asynchronous weak-commitment search AWC peut résoudre des problèmes qui ne peuvent pas être résolus dans un temps de calcul raisonnable par les algorithmes de backtracking asynchrones ABT.
- Dans l'algorithme ABT, l'ordre de priorité des agents est déterminé, un agent essaie de trouver une valeur satisfaisant les contraintes avec les variables des agents les plus prioritaires. Lorsqu'un agent définit une variable, il s'engage fortement sur la valeur sélectionnée qui ne sera pas modifié à moins qu'une recherche exhaustive ne soit effectuée par des agents de priorité inférieure.

Cet inconvénient est commun à tous les algorithmes de BT. En revanche, dans le cas de AWC, lorsqu'un agent ne peut pas trouver une valeur cohérente avec les agents de priorité plus élevée, l'ordre de priorité est modifié de sorte que l'agent aura la plus haute priorité. En conséquence, lorsqu'un agent commet une erreur de valeur sélectionnée, la priorité d'un autre agent devient plus élevée ainsi l'agent qui a fait l'erreur ne commettra pas la mauvaise décision, et la valeur sélectionnée sera modifié.

- Nous pouvons supposer que l'ordre de priorité représente une hiérarchie d'autorité d'agent : l'ordre de priorité de la prise de décision. Si cette hiérarchie est statique, les erreurs de jugement (sélections de valeurs incorrectes) des agents ayant une priorité plus élevée sont fatales à tous les agents.
  D'autre part, lorsque l'ordre de priorité est modifié les valeurs nominales et variables sont sélectionnées de manière coopérative, les erreurs de jugement d'agents spécifiques n'ont pas d'effets fatals, car les mauvaises décisions sont éliminées et seules les bonnes décisions restent.
- Lorsque l'ordre de priorité est statique, l'efficacité de l'algorithme est fortement dépend de la sélection des valeurs initiales, et la distribution des cycles est assez grande. En revanche, dans le asynchrone weak-commitment, les valeurs initiales sont moins critiques, et une solution peut être trouvée même si les valeurs initiales sont loin de la solution finale, puisque les valeurs des variables se rapprochent progressivement de la solution finale.

Conclusion

Dans ce chapitre, nous avons présenté deux algorithmes de base pour résoudre les CSP distribués, qui peuvent être améliorés, notamment au niveau de l'efficacité.

Le premier est nommé backtracking asynchrone ABT, dans lequel les agents agissent d'une manière asynchrone et en fonction de leurs connaissances locales sans aucun contrôle global, tandis que la complétude de l'algorithme est garantie. le deuxième est nommé Asynchronous weak-commit search AWC, qui permet de réviser une mauvaise décision sans recherche exhaustive, et cela sur une version distribuée des CSP (problème des n-reines et le coloriage de graphe).

Nous avons ainsi présenté une série de résultats expérimentaux afin de comparer l'efficacité de ces algorithmes qui montrent que l'algorithme AWC est le plus efficace puisqu'il peut résoudre des problèmes qui ne peuvent pas être résolus dans un temps de calcul raisonnable par des algorithmes de backtracking asynchrones ABT.

Conclusion générale

Dans ce mémoire, nous avons abordé **le problème de satisfaction de contraintes distribués (DisCSP)**. Au meilleur de notre connaissance, jusqu'à présent, les algorithmes DisCSP ont été principalement étudiés sur repères de CSP classiques (tels que N-Reines, Coloriage de Graphe, etc), formulés d'une manière distribuée.

Une fois que nous avons défini le formalisme de problème de satisfaction de contraintes distribué **(DisCSP)** et présenté quelques exemples de problèmes combinatoires qui peuvent être modélisés comme des **(DisCSP)**, nous avons formellement illustré la coordination **multi-agents**, et présenté quelques algorithmes dédiés à la résolution des problèmes de satisfaction de contraintes distribués. Ces algorithmes peuvent être améliorés, notamment au niveau de l'efficacité.

Nous avons testé deux algorithmes**DisCSP**, asynchrone backtracking **(ABT)** et asynchronous weak commitment search **(AWC)**.Nous avons constaté que l'AWC fonctionne bien mieux que l'ABT sur les instances satisfiables. Dans algorithme AWC, les agents agissent d'une manière asynchrone et à la fois sur la base de leurs connaissances locales sans aucun contrôle global, tout en garantissant l'exhaustivité de l'algorithme. Cet algorithme peut réviser une mauvaise décision sans recherche exhaustive en modifiant l'ordre de priorité des agents dynamiquement. Les résultats expérimentaux indiquent que cet algorithme peut résoudre des problèmes de satifaction de contraintes distribués, qui ne peuvent pas être résolu par un algorithme de backtracking asynchrone dans un montant raisonnable de temps. Ces résultats impliquent qu'une organisation d'agents flexible est plus performante qu'une organisation statique et rigide.

Dans l'algorithme de asynchrone backtracking, l'ordre de priorité des agents est déterminé, et un agent essaie de trouver une valeur satisfaisant les contraintes avec les variables des agents les plus prioritaires. Lorsqu'un agent définit une variable, l'agent s'engage fortement sur la valeur sélectionnée, c'est-à-dire que la valeur sélectionnée ne sera pas modifié à moins qu'une recherche exhaustive ne soit effectuée par des agents de priorité inférieure.

Par conséquent, dans les problèmes à grande échelle, une seule erreur de sélection de valeur devient fatale car faire une recherche aussi exhaustive est pratiquement impossible. Cet inconvénient est commun à tous les algorithmes de backtracking. En revanche, dans le cas asynchrone weak-commitment search, lorsqu'un agent ne peut pas trouver une valeur cohérente avec les agents de priorité plus élevée, l'ordre de priorité est modifié de sorte que l'agent a la plus haute priorité. En conséquence, lorsqu'un agent commet une erreur dans le choix de la valeur, la priorité d'un autre agent devient plus élevée, ainsi l'agent qui a commit l'erreur ne prendra pas la mauvaise décision, et la valeur sélectionnée sera modifié.

Nous avons montré dans l'évaluation des résultats obtenus que l'algorithme asynchrone weak-commitment search **AWC** peut résoudre des problèmes de satisfaction de contraintes distribués, que l'algorithme asynchrone backtracking **ABT** ne parvient pas à résoudre dans un délai raisonnable de temps. Nous pouvons supposer que l'ordre de priorité représente une hiérarchie d'autorité d'agent, c'est-à-dire l'ordre de priorité de la prise de décision.

Lorsque l'ordre de priorité est statique, l'efficacité de l'algorithme dépend de la sélection des valeurs initiales, et la distribution des cycles est assez grande, dans l'algorithme asynchrone weak-commitment search **AWC**, les valeurs initiales sont moins critiques, et une solution peut être trouvée même si les valeurs initiales sont loin de la solution finale, puisque les valeurs des variables se rapprochent progressivement de la solution finale.

Par conséquent, ces résultats impliquent qu'une organisation d'agents flexible, dans laquelle l'ordre hiérarchique est modifié dynamiquement, fonctionne mieux qu'une organisation dans laquelle l'ordre hiérarchique est statique et rigide.

Bibliographie

[1] Agent intelligent : définition et exemples.

[2] Algorithme de Résolution de sudoku.

[3] Décomposition arborescente Wikipédia.

[4] PyCharm Wikipédia.

[5] Python-Logo-PNG-Image.

[6] Retour sur trace non chronologique Wikipédia.

[7] Farid ABBACHE. *UNE APPROCHE DE DISTRIBUTION ET DORDONNANCEMENT FIABLE*. PhD thesis, Université de Batna 2, 2018.

[8] Ait Amokhtar Abdelmalek, Zineb Habbache, et al. *Exploitation de l'hypertree decomposition pour la resolution des problemes de satisfaction de contraintes*. PhD thesis, Université de Béjaia, 2008.

[9] AbrahamsonK. On achieving consensus using shared memory.In. In *7th ACM Symposium on Principles of Distributed Computing*, pages 291–302,, 1988.

[10] NACHET Bakhta. *Modèle multi-agent pour la conception de systèmes daide à la décision collective*. PhD thesis, Thèse Doctorat, Université Oran 1, 2013-2014, 2014.

[11] Roberto J Bayardo and Daniel P Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI/IAAI, Vol. 1*, pages 298–304, 1996.

[12] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete mathematics*, 309(1) :1–31, 2009.

[13] Houssein Ben-Ameur. Enchères multi-objets pour la négociation automatique et le commerce électronique. 2001.

[14] Pierre Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings the 11th Conference on Artificial Intelligence for Applications*, pages 32–37. IEEE, 1995.

[15] Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 61–75. Springer, 1996.

[16] Christian Bessiere, Jean-Charles Régin, Roland HC Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[17] Olivier Boissier, Sylvain Gitton, and Pierre Glize. Caractéristiques des systèmes et des applications, 2004.

[18] Simon Boivin. *Résolution d'un problème de satisfaction de contraintes pour l'ordonnancement d'une chaîne d'assemblage automobile /.* Université du Québec à Chicoutimi, Chicoutimi, 2005.

[19] Imane Boussebough. Les systèmes multi-agents dynamiquement adaptables. *Doctoral dissertation, Thèse de doctorat*, 2011.

[20] Jean-Pierre Briot, Samir Aknine, Zahia Guessoum, Jacques Malenfant, Olivier Marin, Jean-François Perrot, and Pierre Sens. Multi-agent systems and fault-tolerance : State of the art elements. *Deliverable, FTCAT Project, EuroControl INO CARE III Programme, Brétigny-sur-Orge, France*, 67, 2007.

[21] Dwork C., Lynch N.A., and Stockmeyer L. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2) :288–323,, 1988.

[22] Brahim Chaib-Draa, Imed Jarras, and Bernard Moulin. Systèmes multi-agents : principes généraux et applications. *Edition Hermès*, 242 :1030–1044, 2001.

[23] Dolev D., Dwork C., and Stockmeyer L. On the minimal synchronism needed for distributed consensus. In *ACM Symposium on Principles of Distributed Computing*, pages 77–97,, January 1987.

[24] Rina Dechter. Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3) :273–312, 1990.

[25] Rina Dechter and Judea Pearl. *The cycle-cutset method for improving search performance in AI applications.* University of California, Computer Science Department, 1986.

[26] Rina Dechter and Judea Pearl. Tree-clustering schemes for constraint-processing. In *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*, pages 150–154, 1988.

[27] Driss Djafri, Farouk Djafri, N Djebari, et al. *Implémentation dun protocole dauthentification et de partage de clés dans un système distribué. Cas détude.* PhD thesis, Universite de bejaia, 2016.

[28] E.P.Cortés P.Q. Duong and C. Collet. La tolérance aux fautes adaptable pour les systèmes à composants : application à un gestionnaire de données. In *Laboratoire LSR-IMAG.* Martin Hères, FRANCE, 2002.

[29] SMA-Bernard ESPINASSE. Intelligence artificielle distribuée (iad) & systèmes multi-agents (sma).

[30] Cristian F., Aghili H., Strong R., and Dolev D. Atomic broadcast, "from simple message-diffusion to byzantine agreement". In *Proceedings of the 15th International Symposium on FaultTolerant Computing Systems (FTCS-15*, 1985.

[31] Jacques Ferber. Les systmes multi agents : Vers une intelligence collective, intereditions, 1995.

[32] ff. Fichier PDF Cours systemes distribues S1 du M1.pdf.

[33] Eugene C Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11) :958–966, 1978.

[34] Eugene C Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM (JACM)*, 32(4) :755–761, 1985.

[35] Eugene C Freuder and Charles D Elfe. Neighborhood inverse consistency preprocessing. In *AAAI/IAAI, Vol. 1*, pages 202–208, 1996.

[36] Eugene C Freuder and Alan K Mackworth. *Constraint-based reasoning*, volume 58. MIT press, 1994.

[37] Daniel Frost, Rina Dechter, et al. Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578. Citeseer, 1995.

[38] John Gary Gaschnig. *Performance measurement and analysis of certain search algorithms.* Carnegie Mellon University, 1979.

[39] Anas Hanaf. *Algorithmes distribués de consensus de moyenne et leurs applications dans la détection des trous de couverture dans un réseau de capteurs.* PhD thesis, Université de Reims Champagne-Ardenne, 2016.

[40] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3) :263–313, 1980.

[41] Badiaa Hedjazi née Dellal. *Workflow de décision coopérative à base de système multi-agent cas de processus de décision spatial.* PhD thesis, Alger, Institut National de Formation en Informatique, 2006.

[42] Philippe Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *AAAI*, volume 93, pages 731–736, 1993.

[43] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75, 2003.

[44] N Kabachi. Intelligence artificielle distribuee et systemes multi-agents. *support de cours : extrait de mémoire de thèse de doctorat, partie état de lart*, page 33, 2008.

[45] E.T.U.D.E.E.T.I.M.P.L.E.M.E.N.T.A.T.I.O.N.D.E.S.A.L.G.O.R.I.T.H.M.E.S.D.E.C.O.N.T KADRI.BAYA. INTERBLOCAGE ET TERMINAISON, mémoire MAGISTER. In *Université des Sciences et de la Technologie HOUARI BOUMEDIENE.* June 1989.

[46] Amroun Kamal. *Décompositions dhypergraphes pour la résolution des problèmes de satisfaction de contraintes (CSP).* PhD thesis, Université Abderahmane MIRA de Bejaia, 2014.

[47] Peter Kissmann and Jörg Hoffmann. Bdd ordering heuristics for classical planning. *Journal of Artificial Intelligence Research*, 51 :779–804, 2014.

[48] Marjorie Le Bars. *Un Simulateur Multi-Agent pour l'Aide à la Décision d'un Collectif : Application à la Gestion d'une ressource Limitée Agro-environnementale.* PhD thesis, Université Paris Dauphine-Paris IX, 2003.

[49] Pease M., Shostak R., and Lamport L. The Byzantin generals problem.In Proc. *ACM Tran-sactions on Programming Languages and Systems*, 4, n 3 :382–401,, 1989.

[50] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1) :99–118, 1977.

[51] Roger Mailler. Improving asynchronous partial overlay. In *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 2, pages 9–16. IEEE, 2012.

[52] Pierre Monier. *DBS multi-variables pour des problèmes de coordination multi-agents.* PhD thesis, Valenciennes, 2012.

[53] Pierre Monier. *DBS multi-variables pour des problèmes de coordination multi-agents*. thesis, Valenciennes, March 2012. Publication Title : http ://www.theses.fr.

[54] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information sciences*, 7 :95–132, 1974.

[55] M Mosbah. Modèles et Approches Formels pour les Systèmes Distribués. page 8.

[56] H. Moumen. Evaluation de mécanisme de détection de défaillances dans un système répartie asynchrone. *Mémoire de magistère, université de Bejaia*, 2005.

[57] JT Mudikolele. Mise en œuvre d'un système distribué pour l'identification et le suivi du casier judiciaire, 2016.

[58] Urban P. and Schiper A. *Fault tolerance in Distributed Systems*. February 2000.

[59] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3) :268–299, 1993.

[60] Raphaël Richard. Intelligence Artificielle Distribuée.

[61] Francesca Rossi, Charles J Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *ECAI*, volume 90, pages 550–556. Citeseer, 1990.

[62] Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning : Proceedings of the Fifth International Conference (KR'96)*, volume 5, page 148. Morgan Kaufmann Pub, 1996.

[63] Daniel Sabin and Eugene C Freuder. Contradicting conventional wisdom in constraint satisfaction. In *International Workshop on Principles and Practice of Constraint Programming*, pages 10–20. Springer, 1994.

[64] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02) :187–207, 1994.

[65] Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *AAAI/IAAI*, pages 163–168, 1999.

[66] El Mehdi Stouti. *Conception et analyse de quelques algorithmes distribués probabilistes*. PhD thesis, Université Abdelmalek Essaâdi, 2016.

[67] Chandra T.D. and Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225–267„ 1996.

[68] Gerard Tel. Introduction to Distributed Algorithms. page 610.

[69] Hadzilacos V. and Toueg S. *Distributeds systems , chapter 5 : Fault Tolerant Broadcast and Related Problems*. Addison-Wesley, 1993.

[70] Jean-Baptiste Welcomme. *MASCODE : un système multi-agent adaptatif pour concevoir des produits complexes. Application à la conception préliminaire avion*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2008.

[71] Michael Wooldridge. Intelligent agents. *Multiagent systems*, 6, 1999.

[72] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction : A review. *Autonomous Agents and Multi-Agent Systems*, 3(2) :185–207, 2000.

[73] Makoto Yokoo, Toru Ishida, Edmund H Durfee, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *1992 12th International Conference on Distributed Computing System*, pages 614–615. IEEE Computer Society, 1992.

[74] Kamel Zidi. *Système interactif d'aide au déplacement multimodal (SIADM)*. PhD thesis, Ecole Centrale de Lille ; Université des Sciences et Technologie de Lille-Lille I, 2006.

Abstract

In this thesis, we present a formalism called a distributed constraint satisfaction problem ***(distributed CSP)*** *and algorithms for solving* ***distributed CSP****. A* ***DisCSP*** *is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents. Various application problems in Distributed Artificial Intelligence can be formalized as distributed CSP.*

We present algorithm called asynchronous backtracking that allows agents to act asynchronously and concurrently without any global control, while guaranteeing the completeness of the algorithm. Furthermore, we describe how this algorithm can be modified into a more efficient algorithm called asynchronous weak-commitment search, which can revise a bad decision without exhaustive search by changing the priority order of agents dynamically. The experimental results on various example problems show that the asynchronous weak-commitment search algorithm is by far more efficient than the asynchronous backtracking algorithm and can solve fairly large-scale problems.

Résumé

Dans ce mémoire, nous présentons un formalisme appelé problème de satisfaction de contraintes distribué (CSP distribué) et des algorithmes de résolution de CSP distribué. Un CSP distribué est un problème de satisfaction de contraintes dans lequel les variables et les contraintes sont distribuées entre plusieurs agents. Divers problèmes d'application dans l'intelligence artificielle distribuée peuvent être formalisés en tant que CSP distribué.

Nous présentons un algorithme appelée asynchrone backtracking qui permet aux agents d'agir de manière asynchrone et concurremment sans aucun contrôle global, tout en garantissant l'exhaustivité de l'algorithme. De plus, nous décrivons comment ce dernier peut être modifié en un algorithme plus efficace appelé asynchronous weak-commitment search, qui peut réviser une mauvaise décision sans recherche exhaustive en changeant l'ordre de priorité des agents de manière dynamique.

Les résultats expérimentaux sur divers exemples de problèmes montrent que l'algorithme asynchronous weak-commitment search est de loin plus efficace que l'algorithme de asynchrone backtracking et peut résoudre des problèmes à grande échelle.