ROYAUME DU MAROC
Université Mohammed V - RABAT -
جامعة محمد الخامس – الرباط –
Faculté des sciences
كلية العلوم
CENTRE D'ETUDES DOCTORALES - SCIENCES ET TECHNOLOGIES

N° d'ordre : 3265

THESE

En vue de l'obtention du : ***DOCTORAT***

Structure de Recherche : Laboratoire Informatique, Mathématique Appliquée, Intelligence Artificielle et Reconnaissance de Formes
Discipline : Informatique
Spécialité : Intelligence artificielle

Présentée et soutenue le : 26/11/2019 par :

Zakarya ERRAJI

Contribution to the development of multi-agent cooperation: Distributed CSP approach and application to solving classic problems

JURY

KASSOU Ismaïl	*PES, ENSIAS, Université Mohammed V de Rabat*	*Président*
BOUYAKHF El Houssine	*PES, Faculté des Sciences, Université Mohammed V de Rabat*	*Directeur de thèse*
HIMMI Mohammed Majid	*PES, Faculté des Sciences, Université Mohammed V de Rabat*	*Directeur de thèse*
BENELALLAM Imade	*PH, INSEA, Rabat*	*Co-directeur de thèse*
EZZAHIR Redouane	*PH, ENSA, Université Ibn Zohr de Agadir*	*Rapporteur / Examinateur*
BELLAFKIH Mostafa	*PES, INPT, Rabat*	*Rapporteur / Examinateur*
IDRISSI Abdellah	*PES, Faculté des Sciences, Université Mohammed V de Rabat*	*Rapporteur / Examinateur*
BENMILOUD Ibtissam	*PES, Ecole Nationale Supérieure des Mines, Rabat*	*Examinatrice*

Année Universitaire : 2019/2020

Dedication

Words alone cannot express the thanks I owe to my family for believing and loving me. They have always filled my life with generous love, and unconditional support and prayers. To them I dedicate this thesis. Last but not the least, the one above all of us, the omnipresent God, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up and throw in the towel, thank you so much Dear Lord. Finally, I would like to thank all my friends and everybody who was important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

Zakarya Erraji

Acknowledgements

The research work presented in this thesis has been performed in the LIMI-ARF Laboratory (Laboratoire d'Informatique Mathématiques appliquées Intelligence Artificielle et Reconnaissance de Formes) at Mohammed V University in Rabat, Faculty of Sciences, Morocco, during the years 2015-2019 under the supervision of Professor El Houssine Bouyakhf and Professor Mohammed Majid Himmi and under the co-supervision of Professor Imade Benelallam.
This dissertation would not have been possible without the support of many people and therefore it is almost unfeasible to thank everyone. In many cases, I do not even have the words to express all my gratitude.

First and foremost, I would like to express my thanks to Professor El Houssine BOUYAKHF (PES, Faculty of Sciences, Mohammed V University in Rabat), my first supervisor. He deserves my deepest gratitude for taking me into the group, for providing excellent facilities and the wonderful inspiring atmosphere. Furthermore, for his many suggestions, constant support during this research, and highly appreciated experience which led to many happy memories. I want to thank him especially for letting me wide autonomy while providing appropriate advice. I am indebted to him more than he knows.

I would also like to warmly thank Professor Mohammed Majid HIMMI (PES, Faculty of Sciences, Mohammed V University in Rabat) for accepting to be my second supervisor and for the help, the insights and the comments he gave. Also, I am glad he involved me as a temporary assistant Professor (Vacataire), this opportunity has provided me an excellent environment to teach and develop new pedagogical techniques.

It is with immense gratitude that I acknowledge all the support, advice, and

guidance of my co-supervisors, Professor Imade Benelallam (PH, INSEA, Rabat). It was a real pleasure to work with him. his truly scientist intuition has made him as a source of ideas and passions in science, which exceptionally inspire and enrich my growth as a student, a researcher, a Professor and a scientist want to be and hope to keep up our collaboration in the future.

I gratefully acknowledge Professor Ismaïl Kassou (PES, ENSIAS, Mohammed V University in Rabat) for honoring my jury by presiding it. I am thankful that in the midst of all his activities, he accepted to preside the jury of my dissertation.

I would like to thank Professor Abdellah Idrissi (PES, Faculty of Sciences, Mohammed V University in Rabat) for accepting to be a reporter of this thesis and for the precious comments.

In the same way, i wish to thank Professor. Mostafa Bellafkih (PES, INPT, Rabat) for accepting to be a reporter of this thesis. I am thankful that during all his activities, he accepted to review my thesis.

I would like to express my thanks also to Professor Redouane Ezzahir (PH, ENSA, Ibn Zohr University in Agadir) for his constructive comments on this thesis as reporter.

Finally, i would like to thank Professor Ibtissam Benmiloud (PES, Ecole Nationale Supérieure des Mines, Rabat) for the examination of the dissertation. Thanks a lot for being a jury and being part of my thesis defense.

Zakarya Erraji

Abstract

Constraint Programming (CP) is a general paradigm used to model and to solve complex combinatorial problems. Distributed Constraint Satisfaction Problems (DisCSP) belongs to Constraint CP and is a general framework for solving distributed problems. DisCSP have a wide range of applications in multi-agent coordination.

In this thesis, we extend the state of the art in solving the DisCSPs by proposing several algorithms and applications. Firstly, we propose a new DisCSP platform named JChoc DisSolver, a platform based on Choco Solver and JADE multi-Agent platform. Then, we propose a robotic dedicated version of JChoc DisSolver platform to solve the multi-robot problems. After that, we propose a new approach to model the intelligence and autonomous marketplace environment (sell and buy goods) this approach includes a new problem formulation and Distributed Constraint Reasoning protocol to deal with e-marketplace issues. After that, we used our platform named JChoc DisSolver to model, implement and illustrate a real-world use case application based on IoT techniques. Therefore, Constraint Programming techniques have been proved to be a very elegant paradigm to handle CIoT applications. Finally, The Improved Forward Checking Tree Algorithm (IFC) for Distributed Constraint Satisfaction Problems is described. The IFC is a new algorithm based on the Asynchronous Forward Checking Tree (AFC-tree) algorithm for the DisCSP problems. The experimental results show the efficiency of this algorithm by improving different metrics of the solving performances.

Keywords: *Artificial Intelligence, Constraint Programming, Multi-Agent Systems, Realistic use, Constraint Satisfaction Problem (CSP), Distributed CSP (DisCSP), Multi-robot Systems, Intelligent Marketplace, Agent Models and Architectures, Cognitive Internet of Things.*

Résumé

La programmation par contraintes (PPC) est un paradigme de l'intelligence artificielle qui vise la résolution de problèmes hautement combinatoires. Les problèmes de satisfaction de contraintes distribués (DisCSP) est un formalisme général de la PC développé pour la résolution des problèmes multi agents distribués. Le DisCSP a plusieurs applications dans les problèmes de coordination multi agents.

Dans cette thèse, nous présentons l'état de l'art de la résolution par les DisCSP en proposant plusieurs algorithmes et applications. Premièrement, nous proposons une nouvelle plateforme nommée JChoc DisSolver. Cette plateforme est basée sur le solveur de contraintes Choco et la plateforme multi agents JADE. Ensuite, nous proposons une version de JChoc DisSolver dédiée aux applications robotiques pour la résolution des problèmes multi robots suivie d'une nouvelle approche qui modélise le problème du marché électronique intelligent (vente et achat de biens). Nous avons utilisé aussi notre plateforme pour présenter notre vision pour la modélisation et la résolution des problèmes d'objets connectés et comment nous pouvons les rendre autonomes et intelligents à travers l'utilisation de la programmation par contraintes. Finalement, Nous détaillons notre nouvel algorithme nommé The Improved Forward Checking Tree Algorithm (IFC). Cet algorithme destiné à la résolution des DisCSP et IFC est basé sur l'algorithme the Asynchronous Forward Checking Tree (AFC-tree). Les résultats expérimentaux montrent l'efficacité du nouvel algorithme en améliorant la performance des différentes métriques de la résolution.

Mots clefs : *Intelligence Artificielle, Programmation par contraintes, Systèmes Multi Agents, Problèmes de satisfaction de contraintes (CSP), CSP Distribués (DisCSP), Système Multi robots, Marché électronique intelligent, Modélisation et architecture d'agent, Objets connectés cognitifs.*

Résumé étendu

La programmation par contraintes (PC) est un paradigme de l'intelligence artificielle qui vise la résolution de problèmes hautement combinatoires. Les problèmes de satisfaction de contraintes distribués (DisCSP) est un formalisme général de la PC développé pour la résolution des problèmes multi agents distribués. Le DisCSP a plusieurs applications dans les problèmes de coordination multi agents.

Dans cette thèse, nous présentons l'état de l'art de la résolution par le DisCSP en proposant plusieurs contributions d'algorithmes et d'applications. Nous commençons par un chapitre présentant l'état de l'art où nous définissons le formalisme des problèmes de satisfaction de contraintes centralisé et ensuite le formalisme des problèmes de satisfaction de contraintes distribué. Pour chaque formalisme, nous donnons les définitions nécessaires liées au formalisme et mais aussi aux travaux de notre thèse. Nous donnons aussi des exemples de formalisation et des méthodes de résolution.

Nous présentons ensuite notre plateforme JChoc pour la résolution des problèmes de satisfaction de contraintes distribués. C'est une plateforme multi agents basée sur les techniques du raisonnement par contraintes distribuées. JChoc est un open source développée grace à l'existence d'une autre plateforme multi agents 'JADE'. JChoc est présentée dans la littérature dans deux articles, le premier consacré à la version initiale de JChoc, et le deuxième introduit la nouvelle version de JChoc capable de s'adapter à tout type de changement ou de modification dans le problème distribué à traiter.

Dans le chapitre 4, nous nous intéressions à l'intégration de JChoc dans l'environnement robotique. Plusieurs problèmes décisionnels dans le domaine de la robotique sont de nature très complexe, np-complet, et sont formalisables par le formalisme des problèmes de satisfaction de contraintes distribué. Pour cela, nous avons réalisé une extension de JChoc en intégrant les outils de la robotique (exemple : le système d'exploitation robotique ROS). Afin de montrer la puissance de cet outil, nous avons traité un problème complexe de coopération multi-drones.

Dans les deux chapitres 5 et 6 qui suivent, nous nous sommes intéressés à deux domaines d'application de la programmation par contraintes, le marché électronique intelligent et les objets connectés cognitifs (CIOT). Pour le marché intelligent, nous proposons une modélisation et un protocole de résolution qu'on appelle 'ABT-Trader' qui permet, à travers la négociation, de satisfaire les contraintes à la fois des vendeurs et des acheteurs d'une façon automatique. Pour l'implémentation de l'algorithme et le test de l'approche, nous avons utilisé JChoc. Les résultats trouvés montrent l'importance de l'approche dans le cas pratique. Dans le domaine de l'IOT nous proposons une extension de JChoc avec changement de l'architecture pour l'adaptation. Dans ce chapitre 6 nous montrons la faisabilité de l'utilisation des techniques de la programmation par contraintes, pour la prise de la décision, dans le domaine des objets connectés.

Finalement, dans le chapitre 7 nous proposons un nouvel algorithme nommé IFC-tree pour la résolution des problèmes de satisfaction de contraintes distribués. L'algorithme IFC-tree est basé sur l'algorithme AFC-tree avec une amélioration des différentes métriques de résolution.

Mots clefs : *Intelligence Artificielle, Programmation par contraintes, Systèmes Multi Agents, Problèmes de satisfaction de contraintes (CSP), CSP Distribués (DisCSP), Système Multi robots, Marché électronique intelligent, Modélisation et architecture d'agent, Objets connectés cognitifs.*

Contents

List of Figures

CHAPTER

1 Introduction

Since the onset of real time electronic devices, mobiles, ubiquitous, and intelligent computing, new combinatorial problems have emerged in the Artificial Intelligence community such as: distributed resource management, distributed air traffic management, Distributed Sensor Network [Béjar et al., 2005], disaster rescue [Kitano et al., 1999] and distributed Meeting Scheduling Problems (SMP), for which it is not suitable to collect all data of problem in one site, to solve it by a centralized algorithm. The reasons are communication time and cost of translation of each sub-problem in a common format. In addition, to give a single agent all data of the problem can also be excluded for reasons of security and confidentiality. Therefore, some of the AI communities are motivated to take an interest in Distributed Constraint Reasoning (DCR), giving birth to other distributed formalism [Yokoo, 2001], whose work focused on developing techniques for modeling and solving distributed combinatorial problems with or without optimization criterion. Distributed Constraint Satisfaction Problems (DisCSP), Distributed Constraint Optimization Problems (DCOP) and Dynamic Distributed Constraints Satisfaction Problems provide a useful framework of multiagent systems for distributed and dynamic resolution of combinatorial problems [Yokoo and Hirayama, 1995a; Yokoo, 2001].
Nowadays, a number of companies, like ILOG and Dash Optimization, sell constraint programming toolkits, which are used by companies as diverse as Amazon.com, British Airways, Cisco, Ford, HP, SNCF, and Volvo.

Constraint programming is a declarative style of modeling combinatorial problems. The user identifies the decision variables, their possible domain of values, and specifies constraints over the allowed values. For instance, the constraint $X_1+X_2 \leq X_3$ specifies that any combination of values for variables X_1, X_2 and X_3 has to be such that the sum of X_1 and X_2 less than or equals X_3. Sophisticated AI search techniques like constraint propagation, which allow to prune irrelevant parts of the search tree,

and chronological backtracking can then be used to find solutions. In solving DisCSPs, agents exchange messages about the variable assignments and conflicts of constraints. Several distributed algorithms for solving DisCSPs have been designed in the last two decades.

A major motivation for research on distributed constraint satisfaction problem (DisCSP) is that it is an elegant model for many every day combinatorial problems that are distributed by nature. By the way, DisCSP is a general framework for solving various problems arising in Distributed Artificial Intelligence. Improving the efficiency of existing algorithms for solving DisCSP is a central key for research on DisCSPs. In this thesis, we extend the state of the art in solving the DisCSPs by proposing several algorithms. We believe that these algorithms are significant as they improve the current state-of-the-art in terms of runtime and number of exchanged messages experimentally.

The organization of this thesis is as follows. In chapter 2, we present the essential material to understand the technical presentation of this thesis. We present also the state of the art related to the constraint programming.

In chapter 3, we propose a new platform, called JChoc, supporting the dynamic aspect for DisCSPs. JChoc is an easy to use platform, based on an elegant Multi-agent communication sub-platform (e.i JADE). It deals with agents with local complex problems and allows a realistic use of agents on a real distributed and dynamic framework. This chapter is based on the research previously published in the following papers:

— Imade Benelallam, Zakarya Erraji, Ghizlaneg Elkhattabi, Jaouad Ait Haddou, El-Houssine Bouyakhf: “JChoc DisSolver - Bridging the Gap Between Simulation and Realistic Use”. ICAART (1) 2015: p66-p74. Lisbon, Portugal.

— Imade Benelallam, Zakarya Erraji, Ghizlane E. L. Khattabi, El-Houssine Bouyakhf: “Dynamic JChoc: A Distributed Constraints Reasoning Platform for Dynamically Changing Environments”. ICAART (Revised Selected Papers) 2015, LNAI, Springer, volume 9494 : p20-p36

In chapter 4, we prove that distributed constraint reasoning techniques can be utilized as a very elegant formalism for multi-robot decision making. First, we describe dynamic distributed constraint satisfaction formalism, the new platform architecture "RoboChoc" and specify how decision making can be controlled in multi-robots environment using dynamic communication protocols. Then we provide an example application that illustrates how our platform can be used to solve multi-robot problems using constraint programming techniques. Chapter 4 is based on the research previously published in the following paper:

— Zakarya Erraji, Mounia Janah, Imade Benelallam and El Houssine Bouyakhf: “Mobile Robotic JChoc DisSolver - A Distributed Constraints Reasoning Plat-

form for Mobile Multi-robot Problems". ICAART (1) 2016: p304-p310. Rome, Italy.

In chapter 5, we propose a new problem formulation and Distributed Constraint Reasoning protocol to deal with e-marketplace issues. More specifically, our attention is focused on constraint-based multi-agent approach offering a flexible and confidential multi-lateral negotiation mechanism, namely, ABT-Trader. The satisfaction of strict preferences offers the best negotiation to buy all the wanted products, otherwise, constraint and variable relaxations are adopted to look for feasible solutions. This approach has been implemented and tested on JChoc Platform using preliminary generated problems. The experimental results show that our approach is of practical interest: it proposes feasible time-tested solutions. Chapter 5 based on the the following paper:

— Zakarya Erraji, Amal Hakkou, BENAMRANE Amine, Imade Benelallam and El Houssine Bouyakhf: "A Distributed Constraint Reasoning Approach Towards Intelligent Marketplace Environment". In Constraint Modelling and Reformulation Workshop; CP'16. Toulouse, France.

In chapter 6, we present a new extension of Cognitive IoT (CIoT) based Constraint Programming frameworks. We introduce a dynamic and distributed Constraint Programming platform that covers explicitly several CIOT considerations (e.g. constraint acquisition, constraint reasoning, distributed communication protocols, etc.). We used our platform named JChoc DisSolver to model, implement and illustrate a real-world use case application based on IoT techniques. Therefore, Constraint Programming techniques have been proved to be a very elegant paradigm to handle CIoT applications. The experimental results are promising and meet our expectations concerning the handling of this category of applications.

Chapter 6 based on the the following paper:

— Zakarya Erraji, Amal Hakkou, BENAMRANE Amine, Imade Benelallam and El Houssine Bouyakhf: "A Distributed Constraint Reasoning Approach Towards Intelligent Marketplace Environment". In Constraint Modelling and Reformulation Workshop; CP'16. Toulouse, France.

In chapter 7, we propose a new algorithm based on the AFC-Tree algorithm for the DisCSP problems named Optimized Forward Checking Tree (OFC-Tree). The experimental results show the efficiency of our approach by improving different metrics of the solving performances.

Chapter 7 based on the the following paper:

— Zakarya Erraji, Imade Benelallam and El Houssine Bouyakhf: "Improved Forward Checking Tree Algorithm for Distributed Constraint Satisfaction Problems". In International Journal of Artificial Intelligence, 2019, volume 17, number 2.

In chapter 8, we conclude this thesis and give some perspectives to the future works.

CHAPTER

2 Background

Preamble

In this chapter, we present the essential material to understand the technical presentation of this thesis.

2.1 Constraint Programming

This section introduces the formalism of constraint programming (CP), which can represent many academic and industrial problems. More details can be found in the text books [Lecoutre, 2013; Rossi et al., 2006; Dechter, 2003].

Constraint programming is a declarative paradigm. The basic idea underlying CP is to model a combinatorial problem as a constraint network. That is, to specify a set of variables, a set of domain values, and a set of constraints. Each constraint is a rule that impose a limitation on the values that a variable, or a combination of variables, may be assigned. A solution of the constraint network is an assignment of variables to domain values that satisfies all constraints in the network. The Constraint Satisfaction Problem (CSP) is, therefore, the problem of determining whether a solution exists, finding one or all solutions, finding whether or not a partial instantiation can be extended to a full solution. The CSP is not known to admit polynomial running time algorithms to solve its instances; hence, CSP is NP-hard.

After this brief overview of CP, we first start with formal definition of the central concepts.

2.1.1 Basic Definitions

Constraint satisfaction problems include two important components, namely variables with associated domains and constraints.

Definition 2.1 (Variable and Domain)**.** ***Variables*** *are objects or items that can take on a variety of values. The set of possible values for a given variable is called its* ***domain****. In our context, the pair* (X,D) *is called the vocabulary, with* X *is a finite set* $\{x_1,...,x_n\}$ *of variables, and* $D=\{D(x_1),...,D(x_n)\}$ *are a finite subsets of* $\mathbb{Z}$ *named the domains.*

The second component of a constraint satisfaction problem is the set of constraints themselves.

Definition 2.2 (Constraint)**.** *Constraints are rules that impose a limitation on the values that a variable, or a combination of variables, may be assigned. Formally speaking, a constraint* c *is a pair* $(var(c),rel(c)$*, where* $var(c)$ *is a sequence of variables of* X*, called the constraint scope of* c*, and* $rel(c)$ *is a relation over* $D^{|var(c)|}$*, called the constraint relation of* c*. For each constraint* c*, the tuples of* $rel(c)$ *indicate the allowed combinations of simultaneous value assignments for the variables in* $var(c)$*. The arity of a constraint* c *is given by the size* $|var(c)|$ *of its scope.*

For the purpose of clarity, we limit ourselves to binary constraints; that is, constraints with a scope involving only two variables. In the follwing, we use c_{ij} to refer to the binary relation that specifies which pairs of values are allowed for the sequence $\langle x_i,x_j\rangle$. For instance, $\neq_{12}$ denotes the constraint specified on $\langle x_1,x_2\rangle$ with the relation "not equal".

A model that includes variables, their domains, and constraints is called a constraint network.

Definition 2.3 (Constraint Network)**.** *A constraint network over a given vocabulary* (X,D) *is a finite set* C *of constraints.*

In real problems, variables often represent components of the problem that can be classified in various types. For instance, in a school time-tabling problem, variables can represent teachers, students, rooms, courses, or time-slots. Such types are often known by the user.

Definition 2.4 (Variable Type)**.** *A* type T_i *is a subset of variables defined by the user as having a common property. A variable* x *is of type* T_i *iff* $x\in T_i$*. A scope* $var=(x_1,\ldots,x_k)$ *of variables is said to* belong *to a sequence of types* $s=(T_1,\ldots,T_k)$ *(denoted by* $var\in s$*) if and only if* $x_i\in T_i$ *for all* $i\leq k$*. Consider* $s=(T_1,T_2,\ldots,T_k)$ *and* $s'=(T'_1,T'_2,\ldots,T'_k)$ *two sequences of types. We say that* s' covers s *(denoted by* $s\sqsubseteq s'$*) iff* $T_i\subseteq T'_i$ *for all* $i\in 1..k$*. A relation* r holds *on a sequence of types* s *if and only if* $(var,r)\in C$ *for all* $var\in s$*. A sequence of types* s *is* maximal *with respect to a relation* r *if and only if* r *holds on* s *and there does not exist* s' *covering* s *on which* r *holds.*

When a variable is assigned a value from its domain, we say that the variable has been instantiated.

Definition 2.5 (Assignment/Example)**.** *Given a vocabulary* (X, D)*. Let* $Y = \{x_1, ..., x_k\}$ *be a subset of* X*. An assignment is a vector* $e_y = \{v_1, ..., v_k\}$ *in* $D^{|Y|}$*. This assignment is partial iff* $Y \neq X$*, complete otherwise (dented by* e*).* e_y *is rejected by a constraint* c *(i.e.,* $e_y \not\models c$*) iff* $var(c) \subseteq var(e_y)$ *and the projection* $e_y[var(c)]$ *of* e_y *on* $var(c)$ *is not in* c*. Otherwise we say that* e_y *is accepted by* c*.*

A solution of the constraint network is an assignment of variables to domain values that satisfies all constraints of the network.

Definition 2.6 (Solution of a Constraint Network)**.** *A solution of a Constraint Network* C *is an assignment of each variable of the* constraint network *to a value in its domain such that all the constraints are simultaneously satisfied. Formally speaking, a complete assignment* e *of* X *is a solution of* C *iff for all* $c \in C$*,* c *does not reject* e*. We denote by* $sol(C)$ *the set of solutions of* C*.*

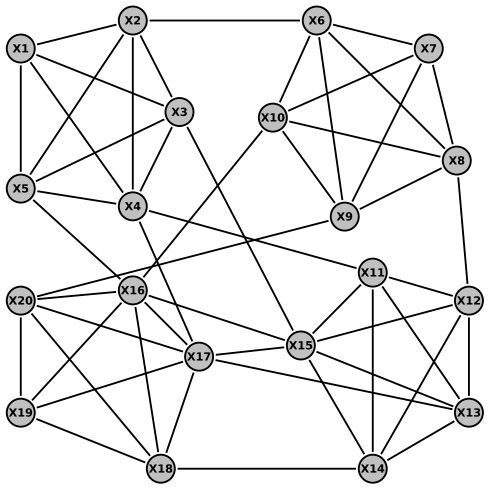


Figure 1 – An example of constraint graph

Graph concept is very useful in capturing the structure of a constraint problem. A constraint network can be represented by a graph called a primal constraint graph, where each vertex represents a variable and the arcs connect all vertices whose variables belong to a constraint scope (see Figure 1).

Definition 2.7 (Constraint Graph)**.** *A constraint graph can be associated with a constraint graph* $G = (V, E)$*, where vertices* V *represent variables* X*, and edges* E *represent constraints* C*.*

The number of variables on the scope of a constraint c_k is called the arity of the constraint c_k. Therefore, a constraint involving one variable (resp. two or n variables) is called unary (resp. *binary* or n-ary) constraint.

A constraint in $\mathcal{C}$ between two variables x_i and x_j is then denoted by c_{ij}. c_{ij} is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$).

Definition 2.8. *A binary constraint network can be associated with a constraint graph* $G = \{X_G, E_G\}$*, where vertices represent the variables of the problem (*$X_G = \mathcal{X}$*) and edges (*E_G*) represent the constraints.*

Definition 2.9. *Two variables are adjacent iff they share a constraint. Formally,* x_i *and* x_j *are adjacent iff* $c_{ij} \in \mathcal{C}$ *If* x_i *and* x_j *are adjacent we also say that* x_i *and* x_j *are neighbors.*

Definition 2.10. *Given a constraint graph* G*, an ordering* o *is a mapping from the variables (vertices of* G*) to the set* $\{1, \ldots, n\}$*.* $o(i)$ *is the ith variables in* o*.*

Solving a CSP is equivalent to finding an assignment of values to all variables in a way that all the constraints of the problem are satisfied.

We present in the following some typical examples of problems that can be intuitively modeled as constraint satisfaction problems. These examples range from academic problems to real-world problems.

2.1.2 Examples of Constraint Networks

In this section, we present some common examples of problems that can be intuitively modeled as constraint networks.

Queens Problem

The Description. The classic example used to illustrate a constraint satisfaction problem is the n-queens problem (see Figure 2). The problem is to place n queens on a $n \times n$ chessboard such that the placement of no queen constitutes an attack on any other.

The Model. One possible constraint network formulation of the problem is as follows. There is a variable for each column of the chessboard.
Variables: $X = \{x_1, \ldots, x_n\}$;
Domains: $D_i = \{1, \ldots, n\}$;
Constraints:

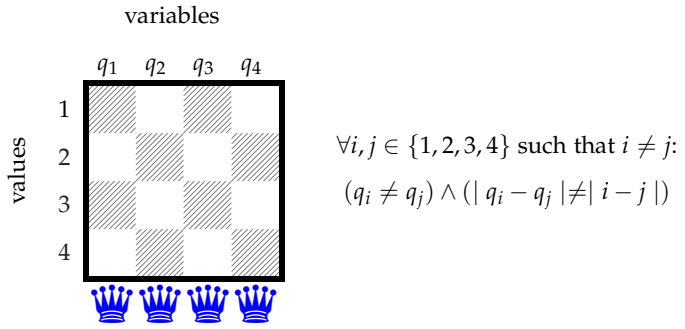
— $\forall i, j \in \{1, \ldots, n\}\ x_i \neq x_j$ (One queen each row.)

— $\forall i, j \in\in 1, \ldots, n\ |x_i - x_j| \neq |i - j|$ (One queen diagonally.)

Graph-coloring Problem

The Description. Another example of constraint network is the graph-coloring problem. In this problem, the goal is to color the nodes of a graph so that there is not a pair of linked nodes colored with the same color. Each node has a finite number of possible colors. This problem can be modeled as a constraint network by representing each node of the graph as a variable. The domain of each variable is

The 4 queens problem:



The 4 queens problem's solution:

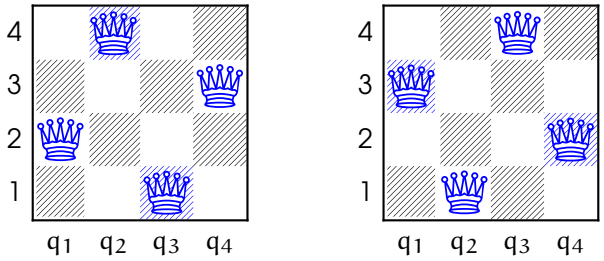


Figure 2 – An example of 4-Queens problem

defined by the possible colors that this variable can take. There exits a constraint between each pair of linked variables that forbids these variables to have the same color.

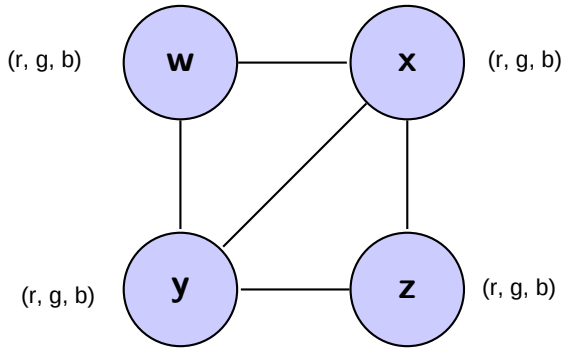


Figure 3 – An example of graph-coloring problem

The Model. The graph-coloring problem of Figure 3 is modeled as follows.
Variables: x, w, y, and z;

Domains: $D_i = \{red, green, blue\}$;
Constraints: adjacent regions must have different colors.

Sudoku Problem

The Description. Sudoku, originally called Number Place, is a logic-based combinatorial number-placement puzzle (Figure 4). The task is to fill a 9×9 grid with digits so that each column, each row, and each block contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.
The Model. The Sudoku puzzle can be modeled as follows.
Variables: $X = \{x_{11}, \ldots, x_{99}\}$;
Domains: $D = \{D_{ij} = \{1, \ldots, 9\} \mid \forall i, j \in 1, \ldots, 9\}$;
Constraints:

— $\forall i \in 1, \ldots, 9$ $\mathtt{AllDifferent}(x_{ij} \mid j \in 1, \ldots, 9)$;

— $\forall j \in 1, \ldots, 9$ $\mathtt{AllDifferent}(x_{ij} \mid i \in 1, \ldots, 9)$;

— $\forall i, j \in 1, \ldots, 9$ $\mathtt{AllDifferent}(x_{(3i+k)(3j+q)} \mid k, q \in 1, \ldots, 3)$.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 4 – An example of sudoku problem.

2.1.3 Solving Methods

Constraint satisfaction problems are solved using search techniques. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking Search

Backtracking is a recursive algorithm [Knuth, 1968; Rossi et al., 2006]. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it consecutively. For each value, the consistency of the partial assignment with the constraints is checked, and in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. In the literature, several variants of backtracking have been proposed. Backmarking [Gaschnig, 1977] improves the efficiency of checking consistency. Backjumping [Prosser, 1993] allows saving part of the search by backtracking "more than one variable" in some cases. Constraint learning [Dechter, 1990] infers and saves new constraints that can be later used to avoid exploring part of the tree search. Look-ahead [Haralick and Elliott, 1980] is also often used in backtracking to anticipate the effects of choosing a variable or a value, thus, sometimes determining in advance when a subproblem is satisfiable or unsatisfiable.

Propagation

Constraint propagation techniques are methods used to modify the structure of a constraint satisfaction problem [Bessiere, 2006]. More accurately, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraint propagation has various uses. First, it change a problem into equivalent one, which is usually simpler to solve. Second, it may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and for some certain kinds of problems. The most used form of local consistency are arc consistency, hyper-arc consistency, and path consistency. The most popular constraint propagation method is the $AC-3$ algorithm [Mackworth, 1977], which enforces arc consistency.

Local search

Local search methods [Hoos and Tsang, 2006] are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem has a solution. They work by iteratively improving a complete assignment over the variables. At each step, it changes the values of some variables, with the overall purpose of increasing the number of constraints satisfied by this assignment. based in that principle, the min-conflicts algorithm [Minton et al., 1992] is a local search algorithm dedicated for solving CSPs. In practice, local search appears to work well when these changes are also affected by random choices. Integration of backtracking search with local search has been developed, leading to hybrid algorithms [Wallace and Schimpf, 2002].

Forward Checking (FC)

The forward checking (FC) algorithm [Haralick and Elliott, 1979, 1980] is the simplest procedure of checking every new instantiation against the future (as yet uninstantiated) variables. The purpose of the forward checking is to propagate information from assigned to unassigned variables. Then, it is classified among those procedures performing a look-ahead.

FC is a recursive procedure that attempts to foresee the effects of choosing an assignment on the not yet assigned variables. Each time a variable is assigned, FC checks forward the effects of this assignment on the future variables domains. So, all values from the domains of future variables which are inconsistent with the assigned value (v_i) of the current variable (x_i) are removed. Future variables concerned by this filtering process are only those sharing a constraint with x_i, the current variable being instantiated. By the way, each domain of a future variable contains only consistent values with past variables (variables already instantiated). Hence, FC does not need to check consistency of new assignments against already instantiated ones as opposed to chronological backtracking. Therefore, FC is the easiest way to prevent assignments that guarantee later failure.

Unlike backtracking, forward checking algorithm enables to prevent assignments that guarantee later failure. This improves the performance of backtracking. However, forward checking reduces the domains of future variables checking only the constraints relating them to variable already instantiated. In addition to these constraints, one can checks also the constraints relating future variables to each other. By the way, domains of future variables may be reduced and further possible conflicts will be avoided.

2.2 Distributed constraint satisfaction problems (DisCSP)

A wide variety of problems in artificial intelligence are solved using the constraint satisfaction problem paradigm. However, there exist applications that are of a distributed nature. In this kind of applications the knowledge about the problem, that is, variables and constraints, may be logically or geographically distributed among physical distributed agents. This distribution is mainly due to privacy and/or security requirements: constraints or possible values may be strategic information that should not be revealed to other agents that can be seen as competitors. In addition, a distributed system provides fault tolerance, which means that if some agents disconnect, a solution might be available fot the connected part. Several applications in multi-agent coordination are of such kind. Examples of applications are sensor networks [Jung et al., 2001; Béjar et al., 2005], military unmanned aerial vehicles teams [Jung et al., 2001], distributed scheduling problems [Wallace and Freuder, 2002; Maheswaran et al., 2004], distributed resource allocation problems [Petcu and Faltings, 2004], log-based reconciliation [Chong and Hamadi, 2006], Distributed Vehicle Routing Problems [Léauté and Faltings, 2011], etc. Therefore, a distributed model allowing a decentralized solving process is more adequate to model

and solve such kind of problems. The *distributed constraint satisfaction problem* has such properties.

A *distributed constraint satisfaction problem* (DisCSP) is composed of a group of autonomous agents, where each agent has control of some elements of information about the whole problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints are satisfied. Hence, agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with constraints between agents [Yokoo et al., 1998; Yokoo and Hirayama, 2000]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them to check consistency of their proposed assignments against constraints that contain variables that belong to other agents.

2.2.1 Preliminaries

The *distributed constraint satisfaction problem* (DisCSP) is a constraint network where variables and constraints are distributed among multiple automated agents [Yokoo et al., 1998].

Definition 2.11. *A* DisCSP *(or a* distributed constraint network*) has been formalized as a tuple* $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$*, where:*

- $\mathcal{A} = \{A_1, \ldots, A_a\}$ *is a set of* a *agents.*
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ *is a set of* n *variables such that each variable* $myVar$ *is controlled by one agent in* $\mathcal{A}$*.*
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ *is a set of current domains, where* $D(x_i)$ *is a finite set of possible values for variable* $myVar$*.*
- $\mathcal{C} = \{C_1, \ldots, C_e\}$ *is a set of* e *constraints that specify the combinations of values allowed for the variables they involve.*

Values may be pruned from the domain of a variable. At any node, the set of possible values for variable $myVar$ is its current domain, $D(x_i)$. We introduce the particular notation of initial domains (or definition domains) $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$, that represents the set of domains before pruning any value (i.e., $\mathcal{D} \subseteq \mathcal{D}^0$).

In the following, we provide some material assumptions in context of DisCSPs. First, we assume a binary distributed constraint network where all constraints are binary constraints (they involve two variables). A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). For simplicity purposes, we consider a restricted version of DisCSPs where each agent controls exactly one variable ($a = n$). Thus, we use the terms agent and variable interchangeably and we identify the agent ID with its variable index. We assume also that each agent ($myAgent$) knows all constraints involving its variable and its neighbors with whom it shares these constraints. We also assume that only the agent who is assigned a variable has control on its value and knowledge of its

domain. In this thesis, we adopt the Yokoo [2000] model of communication between agents where it is assumed that:

— agents communicate by exchanging messages,

— the delay in delivering a message is random but finite and

— an agent can communicate with other agents if it knows their addresses.

Initially, each agent knows addresses of all its neighbors without excluding the possibility of getting addresses of other agents if it is necessary. However we discard the FIFO assumption on communication channels between agents. Hence, we assume that communication between two agents is not necessarily generalized FIFO (aka causal order) channels [Silaghi, 2006].

Almost all distributed algorithms designed for solving distributed CSPs require a total priority order on agents. The total order on agents is denoted by $\mathtt{myOrdering}$. In this thesis, we present two kind of distributed algorithms with regard the agents ordering: algorithms using a static order on agents and those performing a dynamic reordering of agents during search. For the first kind of algorithms and without loss any generality, we will assume that the total order on agents is the lexicographic ordering $[A_1, A_2, \ldots, A_n]$.

Definition 2.12. *An assignment for an agent* $\mathtt{myAgent} \in A$ *is a tuple* $(\mathtt{myVar}, \mathtt{myValue}, \mathtt{myTag})$*, where* $\mathtt{myValue}$ *is a value from the domain of* $\mathtt{myVar}$ *and* $\mathtt{myTag}$ *is the tag value. When comparing two assignments, the most up to date is the one with the greatest tag* $\mathtt{myTag}$*. Two sets of assignments* $\{(x_{i_1}, v_{i_1}, t_{i_1}), \ldots, (x_{i_k}, v_{i_k}, t_{i_k})\}$ *and* $\{(x_{j_1}, v_{j_1}, t_{j_1}), \ldots, (x_{j_q}, v_{j_q}, t_{j_q})\}$ *are compatible if every common variable is assigned the same value in both sets.*

Definition 2.13. *The agentview of an agent* $\mathtt{myAgent} \in A$ *is an array containing the most up to date assignments of higher priority agents.*

Based on the constraints of the problem, agents can infer inconsistent sets of assignments called nogoods.

Definition 2.14. *A nogood is a conjunction of individual assignments, which has been found inconsistent, either because the initial constraints or because searching all possible combinations.*

Example 2.1. *The following nogood* $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \ldots \wedge (x_k = v_k)]$ *means that assignments it contains are not simultaneously allowed because they cause an inconsistency.*

Definition 2.15. *A directed nogood ruling out value* v_k *from the initial domain of variable* x_k *is a clause of the form* $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k \neq v_k$*, meaning that the assignment* $x_k = v_k$ *is inconsistent with the assignments* $x_i = v_i, x_j = v_j, \ldots$*. When a nogood (*$\mathtt{ng}$*) is represented as an implication, the left hand side* $\mathtt{ng}$ *and the right hand side are defined from the position of* $\rightarrow$*.*

Example 2.2. *Often, a nogood is represented as an implication to form a directed nogood. There are clearly many different ways of representing a given nogood as an implication. For example,* $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \cdots \wedge (x_k = v_k)]$ *is logically equivalent to* $[(x_j = v_j) \wedge \cdots \wedge (x_k = v_k)] \rightarrow (x_i \neq v_i)$ *meaning that the assignment* $x_k = v_k$ *is inconsistent with the assignments* $x_i = v_i, x_j = v_j, \ldots$.

The current domain $D(myVar)$ of a variable $myVar$ contains all values from the initial domain $D^0(myVar)$ that are not ruled out by a nogood. When all values of a variable $myVar$ are ruled out by some nogoods ($D(myVar) = \varnothing$), these nogoods are resolved, producing a new nogood ($newNogood$). $newNogood$ is the conjunction of the left-hand sides of all these nogoods.

2.2.2 Examples of DisCSPs

A major motivation for research on Distributed Constraint Satisfaction Problems (DisCSPs) is that it is an elegant model for many every day combinatorial problems arising in Distributed Artificial Intelligence. Thus, DisCSPs have a wide range of applications in multi-agent coordination. Sensor networks [Jung et al., 2001; Béjar et al., 2005], distributed resource allocation problems [Prosser et al., 1992; Petcu and Faltings, 2004], distributed meeting scheduling [Wallace and Freuder, 2002; Maheswaran et al., 2004], log-based reconciliation [Chong and Hamadi, 2006] and military unmanned aerial vehicles teams [Jung et al., 2001] are non-exhaustive examples of real applications of a distributed nature that are successfully modeled and solved by the DisCSP framework. We present in the following some instances of these applications.

Distributed meeting scheduling problem (DisMSP)

The *Distributed Meeting Scheduling Problem* (DisMSP) presented above is a truly distributed benchmark where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [Wallace and Freuder, 2002; Meisels and Lavee, 2004]. The DisMSP consists of a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location. Each agent knows the set of the k among m meetings he/she must attend. It is assumed that each agent knows the traveling time between the locations where his/her meetings will be held. The traveling time between two meetings m_i and m_j is denoted by $TravellingTime(m_i, m_j)$. Solving the problem consists in satisfying the following constraints: (i) all agents attending a meeting must agree on when it will occur, (ii) an agent cannot attend two meetings at same time, (iii) an agent must have enough time to travel from the location where he/she is to the location where the next meeting will be held.

We encode the DisMSP in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. These k meetings are selected randomly among the m meetings . The domain of each variable contains the $d \times h$ slots where a meeting can be

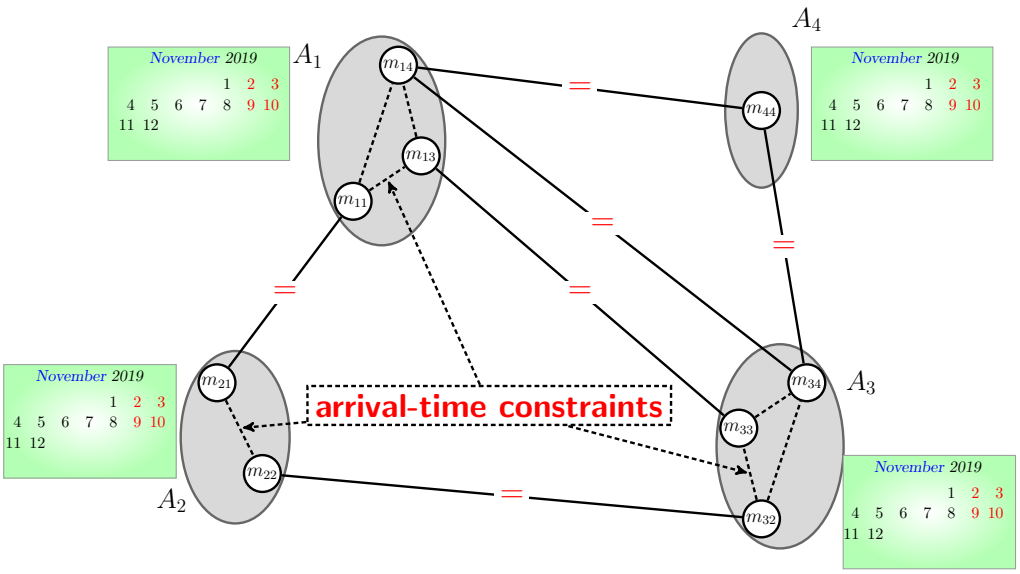


Figure 5 – The distributed meeting-scheduling problem modeled as DisCSP.

scheduled. A slot is one hour long, and there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. This equality constraint means that all agents attending a meeting must schedule it at the same slot (constraint (i)). There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. The arrival-time constraint between two variables m_i and m_j is $|m_i - m_j| - duration > TravellingTime(m_i, m_j)$, where $duration$ is the duration of every meeting. This arrival-time constraint allows us to express both constraints (ii) and (iii).

We illustrate in the encoding of the instance of the meeting scheduling problem shown in in the distributed constraint network formalism. This figure shows 4 agents where each agent has a personal private calendar and a set of meetings each taking place in a specified location. Thus,

— $\mathcal{A} = \{A_1, A_2, A_3, A_4\}$, each agent A_i corresponds to a real agent.

— For each agent $A_i \in \mathcal{A}$ there is a variable m_{ik}, for every meeting m_k that A_i attends, that is, $\mathcal{X} = \{m_{11}, m_{13}, m_{14}, m_{21}, m_{22}, m_{32}, m_{33}, m_{34}, m_{44}\}$.

— $\mathcal{D} = \{D(m_{kl}) \mid m_{kl} \in \mathcal{X}\}$ where,

  — $D(m_{11}) = D(m_{13}) = D(m_{14}) = \{s \mid s \text{ is a slot in } calendar(A_1)\}$.

  — $D(m_{21}) = D(m_{22}) = \{s \mid s \text{ is a slot in } calendar(A_2)\}$.

  — $D(m_{32}) = D(m_{33}) = D(m_{34}) = \{s \mid s \text{ is a slot in } calendar(A_3)\}$.

  — $D(m_{44}) = \{s \mid s \text{ is a slot in } calendar(A_4)\}$.

— For each agent A_i, there is a *private* arrival-time constraint (c^i_{kl}) between every pair of its local variables m_{ik}, m_{il}. For each two agents A_i, A_j that attend the same meeting m_k there is an equality inter-agent constraint (c^{ij}_k) between the variables m_{ik} and m_{jk}, corresponding to the meeting m_k on agent A_i, respectively A_j. Then, $\mathcal{C} = \{c^i_{kl}, c^{ij}_k\}$.

Distributed sensor network problem (sensorDCSP)

The *distributed sensor network problem* (SensorDisCSP) is a real real distributed resource allocation problem [Jung et al., 2001; Béjar et al., 2005]. This problem consists of a set of n stationary sensors, $\{s_1, \ldots, s_n\}$, and a set of m targets, $\{t_1, \ldots, t_m\}$, moving through their sensing range. The objective is to track each target by sensors. Thus, sensors have to to cooperate for tracking all targets. In order, for a target, to be tracked accurately, at least 3 sensors must concurrently turn on overlapping sectors. This allows the target's position to be triangulated. However, each sensor can track at most one target. Hence, a solution is an assignment of three distinct sensors to each target. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a target is visible. The compatibility constraint defines the compatibility among sensors (sensors with the sensing range of each other).

Figure 6 illustrates an instance of the SensorDCSP problem. This example includes 25 sensors (blue circular disks) placed on a grid of 5×5 and 5 targets (red squares) to be tracked. Figure 6 illustrates the visibility constraints, the set of sensors to which a target is visible. Two sensors are compatible if and only if they are in sensing range of each other.

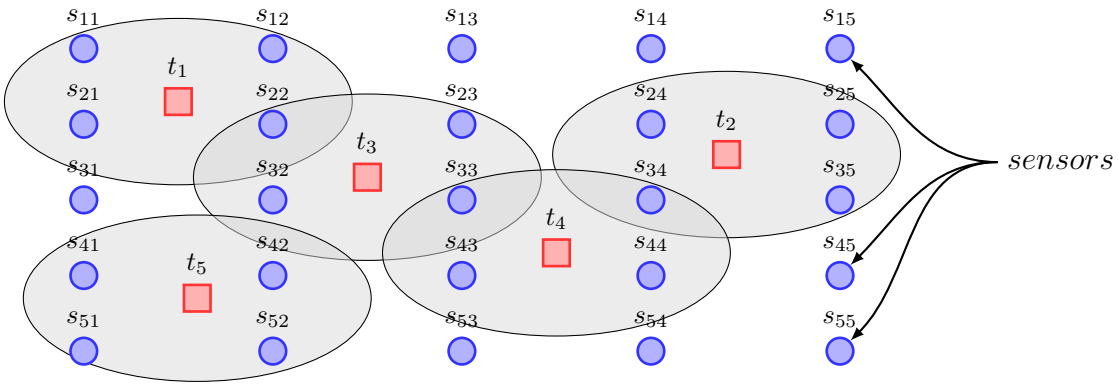


Figure 6 – An instance of the distributed sensor network problem.

The distributed sensor network problem (SensorDisCSP) was formalized in [Béjar et al., 2005] as follows:

— $S = \{s_1, \ldots, s_n\}$ is a set of n sensors;

— $T = \{t_1, \ldots, t_m\}$ is a set of m targets;

Each agent represents one target. There are three variables per agent, one for each sensor that we need to allocate to the corresponding target. The domain of each variable is the set of sensors that can detect the corresponding target. The intra-agent constraints between the variables of one agent (target) specify that the three sensors assigned to the target must be distinct and pairwise compatible. The inter-agent constraints between the variables of different agents specify that a given sensor can be selected by at most one agent.

2.3 Methods for solving distributed CSPs

A trivial method for solving distributed CSPs will be to gather all information about the problem (i.e., the variables, their domains, and their constraints) into a *leader* (i.e., system agent). Afterward the leader agent can solve the problem alone by a centralized constraint satisfaction algorithm. Such leader agent can be elected using a leader election algorithm. An example of leader election algorithm was presented in [Abu-Amara, 1988]. However, the cost of gathering all information about a problem can be a major obstacle of such approach. Moreover, for security/privacy reasons gathering the whole knowledge to a single agent is undesirable or impossible in some applications.

Several distributed algorithms for solving distributed constraints satisfaction problem (DisCSPs) have been developed in the last two decades. To mention only a few [Yokoo et al., 1992; Yokoo, 1995; Yokoo and Hirayama, 1995b; Hamadi et al., 1998; Yokoo et al., 1998; Bessiere et al., 2001; Meisels and Razgon, 2002; Brito and Meseguer, 2003; Meisels and Zivan, 2003; Brito and Meseguer, 2004; Bessiere et al., 2005; Silaghi and Faltings, 2005; Ezzahir et al., 2009]. Regarding the manner on with assignment are performed on these algorithms, they are clustered as synchronous, asynchronous or hybrid ones.

In synchronous search algorithms for solving DisCSPs, agents sequentially assign their variables. Thus, agents perform the assignment of their variable only when they hold a token. Thus, synchronous algorithms are based on notion of token, that is, the privileged of assigning the variable. The token is passed among agents in synchronous algorithms and then only the agent holding the token is activated while the rest of agents are waiting. Although synchronous algorithms do not exploit the parallelism inherent from the distributed system, their agents receive consistent information from each other.

In the asynchronous search algorithms, agents act concurrently and asynchronously without any global control. Hence, all agents are activated an then have the privileged of assigning their variables asynchronously. Asynchronous algorithms are executed autonomously by each agent in the distributed problem where agents do not need to wait for decisions of other agents. Thus, agents take advantage from the distributed formalism to enhance the degree of parallelism. However,

in asynchronous algorithms, the global assignment state at any particular agent is in general inconsistent.

2.3.1 Synchronous search algorithms on DisCSPs

Synchronous Backtracking (SBT) is the simplest search algorithm for solving DisCSPs [Yokoo, 2000]. SBT is a straightforward extension of the chronological Backtracking algorithm for centralized CSPs . SBT requires a total order on which agents will be instantiated. Following this ordering, agents perform assignments sequentially and synchronously. When an agent receives a partial solution from its predecessor, it assigns its variable a value satisfying constraints it knows. If it succeeds in finding such value, it extends the partial solution by adding its assignment on it and passes it on to its successor. When no value is possible for its variable, then it send a *backtracking* message to its predecessor. In SBT, only the agent holding the current partial assignment (CPA) performs an assignment or backtrack.

Zivan and Meisels [2003] proposed the *Synchronous Conflict-Based Backjumping* (SCBJ), a distributed version of the centralized (CBJ) algorithm [Prosser, 1993]. While SBT performs chronological backtracking, SCBJ performs backjumping. Each agent $myAgent$ keep the conflict set (CS_i). When a wipe-out occurs on its domain, a jump is performed to the closest conflict variable in CS_i. The *backjumping* message will contain CS_i. When an agent receives a backjumping message, it discards it current value, and updates its conflict set to be the union of its old conflict-set and the one received.

Extending SBT, [Meisels and Zivan, 2007] proposed the Asynchronous Forward-Checking (afc). Besides assigning variables sequentially as is done in SBT, agents in afc perform forward checking (FC [Haralick and Elliott, 1980]) asynchronously. The key here is that each time an agent succeeds to extend the current partial assignment (by assigning its variable), it sends the CPA to its successor and copies of this CPA to all agents whose assignments are not yet on the CPA. When an agent receives a copy of the CPA, it performs the forward checking phase. In the forward checking phase all inconsistent values with assignments on the received CPA are removed. The forward checking operation is performed asynchronously where comes the name of the algorithm. When an agent generates an empty domain as a result of a forward checking, it informs all agents with unassigned variables on the (inconsistent) CPA. Afterwards, one of these agents will receive the CPA and will backtrack. Thereby, only one backtrack can be generated for a given CPA. Meisels and Zivan have shown in [Meisels and Zivan, 2007] that afc is computationally more efficient than the Asynchronous Backtracking algorithm (ABT, see subsection 2.3.2).

The *Asynchronous Forward-Checking* (afc) incorporates the idea of the forward-checking (FC) algorithm for centralized csp [Haralick and Elliott, 1980]. However, agents perform the forward checking phase asynchronously [Meisels and Zivan, 2003, 2007], hence its name, Asynchronous Forward-Checking. As in synchronous backtracking, agents assign their variables only when they hold the current partial assignment (CPAmsg). The CPAmsg is a unique message (token) that is passed from

one agent to the next one in the ordering. The CPAmsg message carries the current partial assignment (CPA) that agents attempt to extend into a complete solution by assigning their variables on it. When an agent succeeds in assigning its variable on the CPA, it sends this CPA to its successor. Furthermore, copies of the CPA are sent to all agents whose assignments are not yet on the CPA. These agents perform the forward checking asynchronously in order to detect as early as possible inconsistent partial assignments. The forward-checking process is performed as follows. When an agent receives a CPA, it updates the domain of its variable, removing all values that are in conflict with assignments on the received CPA. Furthermore, the shortest CPA producing the inconsistency is stored as justification of the value deletion.

When an agent generates an empty domain as a result of a forward-checking, it initiates a backtrack process by sending nogood messages. nogood messages carry the shortest inconsistent partial assignment which caused the empty domain. nogood messages are sent to all agents with unassigned variables on the (inconsistent) CPA. When an agent receives the nogood message, it checks if the CPA carried in the received message is consistent with its agentview. If it is the case, the receiver stores the nogood, otherwise, the nogood is discarded. When an agent holding a nogood receives a CPA on a CPAmsg message from its predecessor, it sends this CPA back in a backcpamsg message. When multiple agents reject a given assignment by sending nogood messages, only the first agent that will receive a CPAmsg message from its predecessor and is holding a relevant nogood message will eventually backtrack. After receiving a new CPAmsg message, the nogood message becomes obsolete when the CPA it carries is no longer a subset of the received CPA.

The Asynchronous Forward-Checking (afc) is a synchronous search algorithm that processes only consistent partial assignments. These assignments are processed synchronously. In afc algorithm, the state of the search process is represented by a data structure called *Current Partial Assignment* (CPA).

Definition 2.16. *A current partial assignment, CPA, is an ordered set of assignments* $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. *Two CPAs are compatible if every common variable is assigned the same value in both sets.*

Each CPA is associated with a counter that is updated by each agent when it succeeds in assigning its variable on the CPA. This counter, called *Step-Counter* ({SC}), acts as a time-stamp for the CPA. In afc algorithm, each agent stores the current assignments state of its higher priority agents on the agentview. The agentview of an agent, say $myAgent \in \mathcal{A}$, has a form similar to a CPA and is initialized to the set of empty assignments $\{(x_j, Empty, 0) \mid x_j \prec x_i\}$.

2.3.2 Asynchronous search algorithms on DisCSPs

Several distributed asynchronous algorithms for solving DisCSPs have been developed, among which Asynchronous Backtracking (ABT) is the central one [Yokoo et al., 1992, 1998; Bessiere et al., 2005]. Agents in asynchronous algorithms do not

have to wait for decisions of others. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority agents. When an agent assigns a value to its variable, the selected value is sent to lower priority agents. When no value is possible for a variable, the inconsistency is reported to higher agents.

The first complete asynchronous search algorithm for solving DisCSPs is the *Asynchronous Backtracking* (ABT) [Yokoo et al., 1992; Yokoo and Hirayama, 2000; Bessiere et al., 2005]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Agents do not have to wait for decisions of others but they are subject to a total (priority) order. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors. When an agent assigns a value to its variable, the selected value is sent to lower priority neighbors. When no value is possible for a variable, the inconsistency is reported to higher agents in the form of a nogood. ABT computes a solution (or detects that no solution exists) in a finite time. To be complete, ABT requires a total ordering on agents. The total ordering on agents is static.

The required total ordering on agents in ABT provides a directed acyclic graph. Constraints are then directed according to the total order among agents. Hence, a directed link between each two constrained agents is established. ABT uses this structure between agent to perform the asynchronous search. Thus, the agent from which a link departs is the value-sending agent and the agent to which the link arrives is the constraint-evaluating agent. The pseudo-code executed by a generic agent $myAgent \in \mathcal{A}$ is presented in 3.

In ABT, each agent keeps some amount of local information about the global search, namely an agentview and a nogoodstore. A generic agent, say $myAgent$, stores in its agentview the most up to date values that it believes are assigned to its higher priority neighbors (connected to self by incoming links). $myAgent$ stores in its nogoodstore nogoods justifying values removal. Agents exchange the following types of messages (where $myAgent$ is the sender):

OKmsg: $myAgent$ informs a lower priority neighbor about its assignment.

ngdMsg: $myAgent$ informs a higher priority neighbor of a new nogood.

adlMsg: $myAgent$ requests a higher priority agent to set up a link.

stpMsg: The problem is insolvable because an empty nogood has been generated.

In order to be complete, ABT in its original version may request adding links between initially unrelated agents. Given the manner to how these links are set Bessiere et al. proposed 4 version of ABT that have been all proven to be complete [Bessiere et al., 2005]. They rediscover already existent algorithms like ABT [Yokoo et al., 1998], or DIBT [Hamadi et al., 1998].

ABT **(*Adding links during search*):** In ABT, presented above, new links between unrelated agents may be added during search. A link is requested by an agent when it receives a ngdMsg message containing unrelated agents in the or-

dering. New links are permanent. These links are used to remove obsolete information stored by a given agent.

ABT_{all} **(*Adding links as preprocessing*):** In ABT_{all}, all the potentially useful links are added during a preprocessing step. New links are permanent.

$ABT_{temp(k)}$ **(*Adding temporary links*):** In $ABT_{temp(k)}$, unrelated agents may be requested to add a link between them. However, the added links are temporary. This idea was firstly introduced in [Silaghi et al., 2001]. New links are kept only for a fixed number of messages (k). Hence, each added link is removed after exchanging k messages through it.

ABT_{not} **(*No links*):** ABT_{not} no more need links to be complete. To achieve its completeness, it has only to remove obsolete information in finite time. Thus, all nogoods that hypothetically could become obsolete are forgotten after each backtrack.

2.3.3 Conclusion

We have described in this chapter the basic issues of centralized constraint satisfaction problems (CSPs). After defining the CSP formalism and presenting some examples of academical and real combinatorial problems that can be modeled as CSP, we reported the main existent algorithms used for solving centralized CSPs. Next, we formally define the distributed constraint satisfaction problem (DisCSP) framework. Some examples of real world applications have been presented and then encoded in DisCSP. Finally, we provide the state of the art methods for solving DisCSPs.
In the next chapter we present the JChoc platform for solving the DisCSPs problems

CHAPTER

3

Dynamic *JChoc* : A distributed constraints reasoning platform for dynamically changing environments

Preamble

In Artificial Intelligence, a large number of problems (e.i. distributed resource management, distributed air traffic management, Distributed Sensor Network [Béjar et al., 2005]) can be modeled and solved as Distributed Constraint Satisfaction Problems (DisCSPs). As many real world problems change continuously and incessantly over time, some methods have been developed (e.g. DynABT), for solving problems which exhibit this dynamic behavior. Meanwhile, there was no available framework that helped users to develope intelligent multi-agent systems based on Dynamic and Distributed Constraints Reasoning (DCR) techniques.

In this chapter, we propose a new platform, called JChoc, supporting the dynamic aspect for DisCSPs. JChoc is an easy to use platform, based on an elegant Multi-agent communication sub-platform (e.i JADE). It deals with agents with local complex problems and allows a realistic use of agents on a real distributed and dynamic framework.
A real distributed problem is addressed to illustrate how the platform can be used to solve dynamically changing problems. However, the experimental results show the defectiveness of our platform.

3.1 INTRODUCTION

An agent must have a communication platform that allows the exchange of information or dialogue to coordinate their decision-making. This reliable communication tool allows agents to send and receive messages according to a given distributed protocol. However, various sophisticated solvers have been developed: DisChoco [Wahbi et al., 2011], Disolver [Hamadi, 2003], MELY [Galley], Frodo [Petcu, 2005]. Those solvers rely on several algorithms for solving DisCSP problems such as Asynchronous Backtracking (ABT [Yokoo et al., 1992], ABT Family [Bessière et al., 2005]), Asynchronous Forward Checking (AFC) [Meisels and Zivan, 2007] and Nogood-based Asynchronous Forward-Checking (AFC-ng) [Ezzahir et al., 2009]. Asynchronous Distributed Constraints Optimization (ADOPT) [Modi et al., 2005], Asynchronous Forward Bounding (AFB) [Gershman et al., 2009], Asynchronous Branch-and-Bound (BnB-ADOPT) [Yeoh et al., 2010] and Dynamic Backtracking for distributed constraint optimization (DyBop) [Ezzahir et al., 2008] were developed to solve DCOP problems. As well as the authors recognise that most of these tools are specially developed for simulation context. This fact can be clearly observed from its experimental setups. Given the difficulty that researchers are facing, they often make many simplifying assumptions (i.e. simple agent (one variable per agent), agents as multi-thread, single physical platform, communication via simulated perfect FIFO channels, etc.) about the underlying distributed problem, which might affect the predictions obtained from the simulation in non-trivial ways. Switching from the simulation to the actual development practice often leads to loss of accuracy. Hence, bridging the gap between simulation and actual development and deployment within distributed constraints solvers and include dynamic aspect are the motivations for presenting the different ideas discussed in the present chapter.

In this chapter we focus on the development of a Multi-agent platform for Distributed Constraint Reasoning and Dynamic Distributed Constraints Problems, namely JChoc DisSolver. This proposed platform allows non-expert user to address and solve easily, not only distributed constraint satisfaction problems, but also real Dynamic Distributed Constraint Satisfaction Problems.

This chapter is organized as follows. Section 2 presents a brief definition of Distributed Constraint Satisfaction Problem (DisCSP) and Dynamic and Distributed Constraint Satisfaction Problem (DDisCSP) with an example. In section 3, we present related work. Section 4 presents the global architecture of JChoc platform. In section 5, we show how use this platform in a distributed environment even if it changes dynamically. And finally, in section 6 we conclude the chapter by experiment this platform within a real Distributed and Dynamic Constraints Satisfaction Problems.

3.2 RELATED WORK

Recently, B. Lutati and al. [Lutati et al., 2014] have proposed a MAS platform, called AgentZero. This tool can be considered as a new addition to the available MAS tools in general and to the DCR research field in particular. The authors claim that AgentZero is generic and applicable to many domains, specifically introducing benefits for the DCR simulation domain. However, the platform has been designed only for simulation use and used only by researchers in Distributed Constraint Reasoning. So developing and setting computer software for real problems based on DCR is not simple and remains a difficult task for users in general.
In [Petcu, 2005] A. Petcu. Proposes a Framework for Open Distributed Optimization (FRODO). The framework is implemented in Java, and simulates a multiagent environment in a single Java virtual machine. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange. FRODO comes with several built in algorithms and a suite of problem generators for benchmarking.
The authors of [Sultanik et al., 2007] proposed a open-source tool for solving DCR, called DCOPolis. DCOPolis is an open-source framework designed to abstract algorithm implementation from the underlying platform (i.e. hardware, network, operating system). This allows a single implementation of an algorithm to be run in simulation (i.e. on top of the NS2 network simulator with AgentJ).

DCOPolis differs from existing DCR frameworks and simulators, however, it supports a novel type of simulation in which the runtime of any distributed algorithm can be accurately estimated on a single physical computer.
Researchers in DCR are concerned with developing new algorithms, and comparing their performance with existing algorithms. Therefore, in [Wahbi et al., 2011] the authors present an open source Java library , called DisChoco which aims at implementing DCR algorithms from an abstract model of agent . DisChoco allows to represent both DisCSPs and DCOPs, as opposed to other platforms. A single implementation of a DCR algorithm can run as simulation on a single machine.DisChoco is a elegant platform, but all the different issues of realistic uses and actual deployment have not been addressed.
Developing intelligent software applications based on DCR algorithms is a difficult task, because the programmer must explicitly juggle between many very different concerns, including centralized programming, distributed programming, asynchronous and concurrent management of distributed structures, communication concerns and others. In addition, there are very few open-source tools for solving DCR problems in a physically distributed environment. In this chapter we have been looking for a singular platform that would possess not only simulation qualities, but especially designed for realistic and actual deployment. JChoc platform is a new added value which allows bridging the gap between simulation and realistic use. To our knowledge, this is the first DCR platform respecting FIPA standards and specifications.

3.3 JCHOC PLATFORM

3.3.1 JChoc description

The best way to prove the effectiveness of a proposed distributed constraint reasoning algorithm, is to use it in a realistic multi-platform agent. This is how we can reduce the gap between theory and practice. JChoc is a distributed constraints multiagent platform proposed for solving combinatorial problems within a specific distributed environment. It can also be used to analyze and test the algorithms proposed by constraints programming community. This platform is presented in the form of programming environment (API) and applications to help different types of users. Hence, JChoc can be easily appropriated by two main actors:

- Developers to design and develop applications (e.i. client application, web application, mobile application, etc...) within distributed constraints programming based on JChoc API;
- Non-expert user to interact directly with applications based on distributed constraints programming.

This proposed platform has several advantages:

- A distributed constraints problem can be easily addressed and solved in a realistic environment by unsophisticated users;
- The performances of the proposed protocols (i.e. ABT, AFC, Adopt, etc...) can be actually tested and proved in a realistic communication channel (i.e. WLAN WPAN WMAN WWAN);
- It offers a modular software architecture which accepts extensions easily (i.e. security, confidentiality, cryptography, etc...);
- Thanks to the extensibility of JADE communication model [Bellifemine et al., 2007], JChoc allows the development of multiagent systems and applications consistent with Foundation for Intelligent Physical Agents (FIPA) [1] standards and specifications;
- Thanks to the the robustness of Choco platform [Jussien et al., 2008], complex agent (i.e. multiple variables per agent) can easily address and solve its local sub-problem and use solutions as a compiled domain.

This platform consists of several modules presented as services. The main constraint programming services offered are based Distributed Constraint Reasoning Protocols (DCRP) and Choco Solver (CS). Choco is a platform for research in centralized constraint programming and combinatorial optimization. This choice of Choco enabled us to benefit from the modules already implemented in it. In the next section, we will study the different elements of JChoc platform.

1. http://www.fipa.org/

3.3.2 JChoc Architecture

JChoc architecture is motivated by FIPA specifications, it allows the development of multiagent systems and applications conforming to MAS standards. It is implemented in JAVA and provides classes that implement and inherit from JADE and Choco platforms to define the behavior of specific agents. Figure 2 represents the main JChoc architectural elements. This platform has five main modules.

- DCRP ≪Distributed Constraint Reasoning Protocols≫ provides distributed constraints protocols as service. This element defines new types of messages and implements the behavior of the agent when receiving and sending a specific type of information (e.i. ABT, AFC, Adopt, etc...);
- CS ≪Choco Solver≫ provides the ability to address and resolve local CSP subproblem;
- DF ≪Director Facilitator≫ provides a service of "yellow pages" to the platform;
- ACC ≪Agent Communication Channel≫ manages the communication between agents;
- AMS ≪Agent Management System≫ oversees the registration of agents, their authentication, their access and the use of the system.

These five modules are activated at each time the platform is started.

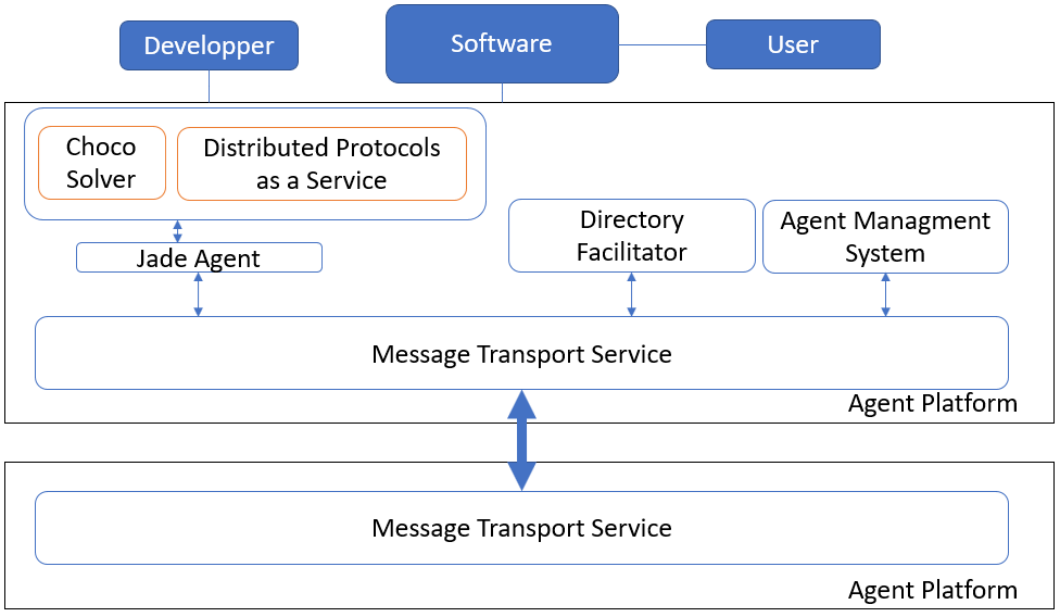


Figure 7 – The JChoc Architecture

The JADE agent is also a key player in our platform. Thanks to JADE an Agent Identifier (AID) identifies an agent uniquely.

JChoc uses extensively a sniffing tool for debugging, or simply documenting conversations between agents. The sniffer subscribes to AMS agent to be notified of all platform events and of all message exchanges between a set of specified agents. When the user decides to monitor an agent or a group of agents, every message directed to, or coming from, that agent/group is tracked and displayed in the sniffer GUI. The user can select and view the details of every individual message, save the message or serialize an entire conversation as a binary file.

3.4 USING DYNAMIC JCHOC

3.4.1 Using JChoc in distributed environment

In this section we present how to use the JChoc platform in real distributed environment. The MSP problem depicted in figure 1 is used to illustrate this proposed platform. Initially we generate a sub-problem for each agent involved in the global DisCSP problem, modeled by an expert as an XML file, which allows standardizing the syntactic structure of the sub-problems. A sub-problem containing only the information necessary for a single agent, so he can participate in solving the global problem in a real distributed environment.

Figure 20 shows an example of representation of the MSP sub-problem defined above in the XDisCSP format. Each variable has a unique ID, which is the concatenation of the ID of its owner agent and index of the variable in the agent. This is necessary when defining constraints (scope of constraints). For constraints, we used two types of constraints: TKC for Totally Known Constraint and PKC for Partially Known Constraint. Constraints can be defined in extension or as a Boolean function. Different types of constraints are predefined: equal to $eq(M_i,M_j)$, different from $ne(M_i,M_j)$, greater than or equal $ge(M_i,M_j)$, greater than $gt(M_i;M_j)$,etc. In this sub-problem there is 1 complex agent A_3 which controls exactly 3 variables. The domain of A_3 contain 14 values $D_3=\{1...14\}$. There are three constraints of Arrival time $ge(abs(sub(M_i,M_j))$: the first constraint is between $M_{3.2}$ and $M_{3.3}$ the second one is between $M_{3.3}$ and $M_{3.4}$ and the third is between $M_{3.2}$ and $M_{3.4}$, three constraints of equality $eq(M_i,M_j)$: between $M_{1.4}$ and $M_{3.4}$, between $M_{1.3}$ and $M_{3.3}$, between $M_{2.2}$ and $M_{3.2}$ after defining our sub-problem we can configure our solver.

Once the sub-problem is generated, we can test the functioning of the platform in a physically distributed environment. So we chose machines that simulate the different agents of the problem, and filed each sub-problem in a machine, before launching it.

Figure 21 shows how the master launches its communication interface listening on the network. We start with instantiate the dissolver object (line 7), This class models the distributed problem when JChoc is used to solve a problem in a real distributed environment. All information on distributed problem is encapsulated in this object (identities of agents, inter-agent constraints, protocol, etc.). Then, we define the

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
      <presentation name="MSP" type="DisCSP"
      model="Complex" constraintModel="TKC" format="XDisCSP 1.0" />
   <domains nbDomains="2">
      <domain name="D1" nbValues="7">1..7</domain>
   </domains>
   <variables nbVariables="3">
      <variable name="M3.2" id="1" domain="D1" description="M_2" />
      <variable name="M3.3" id="2" domain="D1" description="M_3" />
      <variable name="M3.4" id="1" domain="D1" description="M_4" />
   </variables>
   <constraints nbConstraints="3">
      <constraint model="TKC" name="C0" reference="ArrivalTime" scope="M3.2 M3.3 2" arity="2">
         <parameters>M3.2 M3.3 2</parameters>
      </constraint>
      <constraint model="TKC" name="C1" reference="ArrivalTime"
         scope="M3.3 M3.4 2" arity="2">
         <parameters>M3.3 M3.4 2</parameters>
      </constraint>
      <constraint model="TKC" name="C2" reference="ArrivalTime"
         scope="M3.2 M3.4 2" arity="2">
         <parameters>M3.2 M3.4 2</parameters>
      </constraint>
   </constraints>
   <predicates nbPredicates="2">
      <predicate name="ArrivalTime">
         <parameters>int Mi,int Mj,int cte</parameters>
         <expression>
            <functional>ge(abs(sub(Mi,Mj)), cte)</functional>
         </expression>
      </predicate>
      <predicate name="eq">
         <parameters>int Mi,int Mj</parameters>
         <expression>
            <functional>eq(Mi,Mj)</functional>
         </expression>
      </predicate>
   </predicates>
   <agents_neighbours>
      <agents_parent>
         <agent name="A1">
            <constraints nbConstraints="2">
               <constraint model="TKC" name="C0" reference="eq" scope="M1.4 M3.4" arity="2">
                  <parameters>M1.4 M3.4</parameters>
               </constraint>
               <constraint model="TKC" name="C1" reference="eq" scope="M1.3 M3.3" arity="2">
                  <parameters>M1.3 M3.3</parameters>
               </constraint>
            </constraints>

         </agent>
         <agent name="A2">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C0" reference="eq" scope="M2.2 M3.2" arity="2">
                  <parameters>M2.2 M3.2</parameters>
               </constraint>
            </constraints>
         </agent>
      </agents_parent>
      <agents_children>
         <agent name="A4" id="5" variable="M3.4" />
      </agents_children>
   </agents_neighbours>
</instance>
```

Figure 8 – Definition of DMS sub-problem in XDisCSP format.

type of master (line 8) (ABT in this case). Finally, we trigger the container and we launch the master (lines 10-11).

```
1  import JChoc.DisSolver;
2
3  public class Master
4  {
5     public static void main(String[] args)
6     {
7        DisSolver js = new DisSolver();
8        js.setType("MasterABT");
9        js.setGui(true);
10       js.setNumberOfAgents(4);
11       js.run();
12    }
13
14 }
```

Figure 9 – How the master launches its communication interface.

```
1  import JChoc.DisSolver;
2
3  public class Omar
4  {
5     public static void main(String[] args)
6     {
7        DisSolver js1 = new DisSolver();
8        js1.setType("AgentABT");
9        js1.addAgent("A1", "Problem1.xml",true,true);
10       js1.setContainer("192.168.1.8");
11       js1.run();
12    }
13 }
```

Figure 10 – how to implement and launch JChoc DisSolver in Omar agent (A1)

```
1  import JChoc.DisSolver;
2
3  public class Jean
4  {
5     public static void main(String[] args) {
6        DisSolver js1 = new DisSolver();
7        js1.setType("AgentABT");
8        js1.addAgent("A2", "Problem2.xml",true,true);
9        js1.setContainer("192.168.1.8");
10       js1.run();
11    }
12 }
```

Figure 11 – how to implement and launch JChoc DisSolver in Jean agent (A2)

Figures 10-13 show how to launch JChoc agents. We start with instantiate the DisSolver object (line 7), followed by the agent and distributed sub-problem declaration which specifies the resolution algorithm to be used (line 8-9). Next, the declara-

```
1  import JChoc.DisSolver;
2
3  public class Yun
4  {
5     public static void main(String[] args) {
6        DisSolver js1 = new DisSolver();
7        js1.setType("AgentABT");
8        js1.addAgent("A3", "Problem3.xml",true,true);
9        js1.setContainer("192.168.1.8");
10       js1.run();
11    }
12 }
```

Figure 12 – how to implement and launch JChoc DisSolver in Yun agent (A3)

```
1  import JChoc.DisSolver;
2
3  public class Mamadou
4  {
5     public static void main(String[] args) {
6        DisSolver js1 = new DisSolver();
7        js1.setType("AgentABT");
8        js1.addAgent("A4", "Problem4.xml",true,true);
9        js1.setContainer("192.168.1.8");
10       js1.run();
11    }
12 }
```

Figure 13 – how to implement and launch JChoc DisSolver in Mamadou agent (A4)

tion of the container containing the master with its IP address (line 10). Eventually, we launch the agent (line11).

The master waits for the confirmation of creation of all agents before ordering the start of the search. Thus, the problem can be solved using a specified implemented protocol (ABT for example).

3.4.2 Using JChoc in dynamic distributed environment

The use of JChoc platform in a dynamic environment is not very different to that in the case of distributed static problems. The difference is seen in the xml file that defines the sub-problem of each agent.

To see the platform's exploitation in the Dynamic case of Distributed Satisfaction Problems, we take a random example composed of five agents, each agent has one variable. Figure 14, shows a model of representation of a dynamic sub-problem of an agent that has two constraints with two other agents, 3 seconds after launching, one of the constraints is going to be removed, then after 4 seconds, another link with a third agent will be added.
In addition of the definition of variables, domains and constraints, we define the constraints that will be either added or removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
   <presentation name="MSP" type="DisCSP"
   model="Complex" constraintModel="TKC"
   format="XDisCSP 1.0" />
   <domains nbDomains="1">
      <domain name="D1" nbValues="20">1..20</domain>
   </domains>
   <variables nbVariables="1">
      <variable name="X4.1" id="1" domain="D1" description="X_4" />
   </variables>
   <predicates nbPredicates="0">
   </predicates>
   <constraints>
   </constraints>
   <agents_neighbours>
      <agents_parent>
         <agent name = "A1">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C1" reference="R1"
               scope="X1.1 X4.1" arity="2" change = "add_4">
                  <parameters>X1.1 X4.1</parameters>
               </constraint>
            </constraints>
         </agent>
         <agent name = "A2">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C1" reference="R2"
               scope="X2.1 X4.1" arity="2" change = "no">
                  <parameters>X2.1 X4.1</parameters>
               </constraint>
            </constraints>
         </agent>
         <agent name = "A3">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C1" reference="R3"
               scope="X3.1 X4.1" arity="2" change = "remove_3">
                  <parameters>X3.1 X4.1</parameters>
               </constraint>
            </constraints>
         </agent>
      </agents_parent>
      <agents_children>
         <agent name="A5" id="1" variable="X4.1" />
      </agents_children>
   </agents_neighbours>
   <relations>
      <relation name="R1" semantics="conflicts">2 8|8 20|5 7|9 20|17 14|
                     5 16|16 7|9 10|19 2|4 13|18 1|1 19|20 6|16 2|
                     5 18|11 14|10 2|19 19|19 18</relation>
      <relation name="R2" semantics="conflicts">6 16|14 2|19 15|20 2|8 2|
                     4 2|17 20|18 6|7 7|7 10|3 18|18 10|13 15|9 18|
                     14 16|19 6|13 18|3 14|14 20</relation>
      <relation name="R3" semantics="conflicts">16 12|20 6|8 17|17 5|4 18|
                     12 18|19 5|20 17|15 13|6 5|17 18|3 1|17 12|1 16|
                     2 1|8 5|13 3|17 10|6 20|7 9</relation>
      <relation name="R4" semantics="conflicts">|1 6|5 2|7 19|15 1|17 15|
                     13 7|2 5|8 5|1 14|2 16|4 14|12 14|9 19|3 4|
                     19 18|10 8|9 4|1 2</relation>
   </relations>
</instance>
```

Figure 14 – Definition of Dynamic sub-problem in XDisCSP format.

After the generation of the dynamic sub-problem, we can launch the resolution following the same approach as before, but instead to insert the name of an XML file

of a static sub-problem as argument, we insert the name of dynamic sub-problem XML.

3.5 EXPERIMENTAL RESULTS

3.5.1 Configuration example

To experiment the JChoc platform in a physically distributed environment, we chose five machines with features **2.93 GHz**, **CORE(TM) 2 duo** with **2 GB RAM** that simulate agents. These machines are connected via the **WLAN** of our laboratory. We also chose ABT algorithm to solve Meeting Scheduling problems (MSP). In figure 1 above, we depict an example of problem solved by this platform in a live distributed environment network. This figure illustrates an instance of MSP viewed as DisCSP where each agent has a personal private calendar and a set of meetings each taking place in a specified location. In that example, there are four agents, A_1, A_2, A_3 and A_4, and four meetings, m_1, m_2, m_3 and m_4. Each agent has its own calendar divided into **14** slots. The time required for traveling among places where meetings can be scheduled is **2 slots**.
We have intentionally limited the number of agents to 4 for this problem needs, but the number of the agents can be easily extended to **Ng4** for the neediest problems.
Figures 9 and 10 show the GUI of the sniffer agent at the start and the end of ABT resolution. The canvas provides a graphical representation of the messages exchanged between sniffed **ABTagents**, where each arrow represents a message and each color identifies a type of conversation. For example agent A_1 sends an **OK?** message to informs A_2 that he has done a new assignment $m_{1.1}$**:1** (line 5).

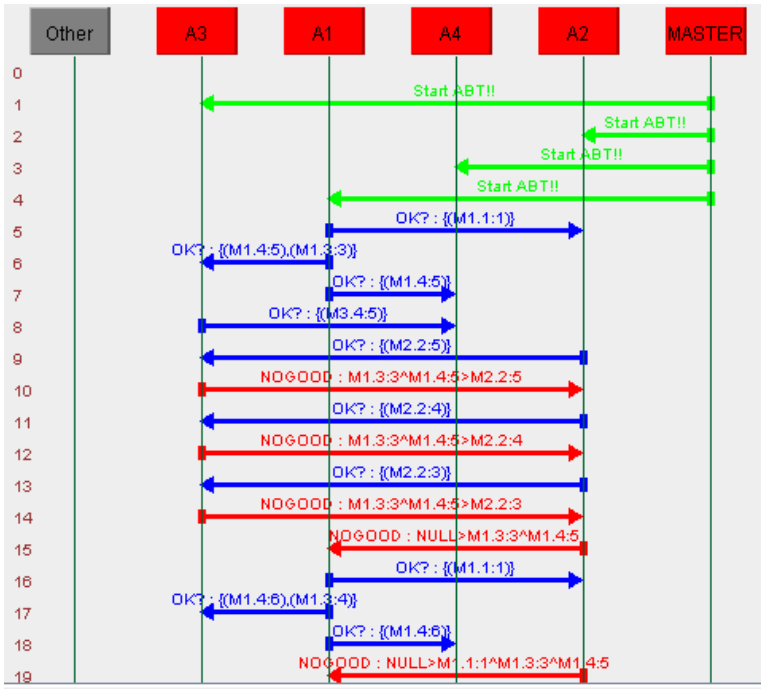


Figure 15 – The start on sniffer agent GUI.

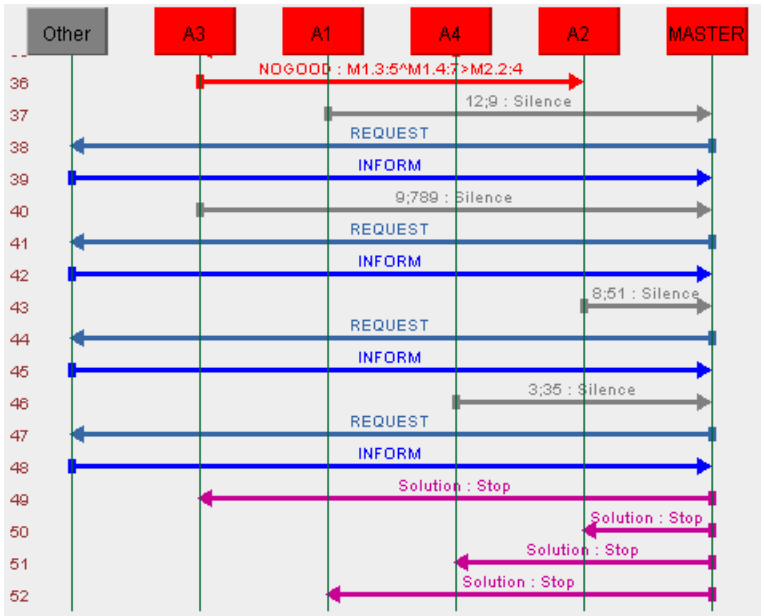


Figure 16 – The finish on sniffer agent GUI.

If no new consistent value is found (line 10), A_3 generates a new **nogood** $m_{1.3}$**:3**

$\wedge$ $m_{1.4}$**:5** $\Rightarrow$ $m_{2.2} \neq 5$ by the resolution of existing nogoods. Eventually, the system can stabilize in a state where each agent has a value and no constraint is violated. This state is a global solution and the network has reached quiescence, meaning that no message is traveling through it (lines 37, 40, 43, 46). Once the solution is found, the master should be advised to spread the stop order to all agents (lines 49-52).
A solution to this example is :
$A_1 \longrightarrow (m_{1.1} : 3;\ m_{1.3} : 7;\ m_{1.4} : 1)$, $A_2 \longrightarrow (m_{2.1} : 3;\ m_{2.2} : 5)$, $A_3 \longrightarrow (m_{3.2} : 5;\ m_{3.3} : 7;\ m_{3.4} : 1)$, $A_4 \longrightarrow (m_{4.4} : 1)$.

3.5.2 Platform scalability

The scalability of JChoc is the ability of the system, network, and process to handle a growing amount of work in a capable manner and its ability to be enlarged to accommodate that growth. In order to experiment our platform, we consider a large number of MSP instances. These Meeting Scheduling Problem are characterized by $< m, p, n, d, h, t, a >$, where m is the number of meetings, p is the number of participants, n is the number of inter-agent constraints d determines the number of days. Different time slots are available for each meeting, and h is the number of hours per day, t is a duration of the meeting and a is the percentage of availability for each participant. We present our results for the class $< m, p, n, 5, 10, 1, 90\% >$ and we vary three parameters : m, p, n (each agent has 2 meetings):

As shown in experimental results, in figure 7, the performance of our platform is measured in terms of network load (number of messages) and run-time execution. From these preliminary results we see that JChoc platform performs rapidly in small instances (#p $\in$ [4, 14]). The number of messages increases for #p $\in$ [15, 18] and reduces for #p > 18. This scalability behavior is due to complexity of MSP problems. When the instance is hard the problem can be solved rapidly.

3.5.3 Platform scalability in a dynamic changed environement

To compare the performance of the DDisCSPs with a platform that supports dynamic aspect and an other that doesn't. We made our experiments using ABT that can't solve such problem dynamically and resolve the problem when changes are available, and DynABT that can adapt changes and continuous problem's solving. We have introduced a rate change δ as a percentage of the total constraints in the problem (δ = 20%). In these experiments we generated problems randomly, with parameters $(a, i, n, d, p1, p2)$ using the platform generator, where: a is the number of agents = 20, i: the number of instances = 10, n: the number of variables = 20, $p1$: the density of constraints = 20%, and $p2$: the tightness of constraints with value 10% - 90% step 10%, the range of tightness 10% - 40% contains solvable problems, 50% contains both solvable and unsolvable problems, and 60% - 90% problems are unsolvable.

#p	#m	#n	#messages	Time (ms)
4	8	3	11	17070
5	10	5	11	17204
6	12	6	14	16144
7	14	7	14	17073
8	16	8	19	19180
9	18	9	24	20210
10	20	10	22	18294
11	22	11	32	20197
12	24	12	27	18516
13	26	15	30	20370
14	28	33	51	26073
15	30	35	105	31103
16	32	29	69	28914
17	34	33	175	38324
18	36	35	139	43172
19	38	38	141	37121
20	40	43	94	33457

Figure 17 – Performance of JChoc platform using ABT protocol on the Meeting Scheduling Problem (MSP).

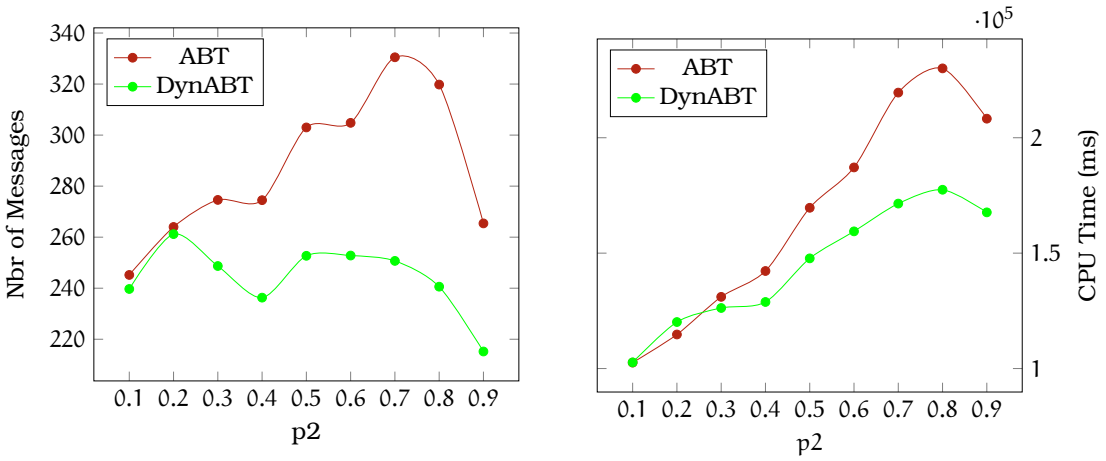


Figure 18 – ABT vs DynABT

The figure 18 shows the number of messages sent and CPU Time, measured for both ABT and DynABT implemented on JChoc platform and using our laboratory's wireless network, that allows the communication between Agents in the same environment and conditions. All results obtained show that DynABT significantly outperforms ABT in a dynamic changed environment. This comparison shows the

benefits of solving dynamic distributed problems in a real distributed changed environment with an algorithm that support dynamic aspect implemented in a suitable Platform. The platform is user friendly and lets users implement their Multi-agents applications for dynamic environment.

3.6 CONCLUSION

In this chapter, we have proposed a modular, reliable, deployable and distributed software architecture, called JChoc DisSolver, which can be used easily for several real dynamic combinatorial problems. The main purpose of our platform is to break down the barriers to building new and innovative applications. The possibility of combining the expressiveness of Choco, the extensibility of JADE and our powerful Dynamic Distributed Constraint Reasoning Add-On bring a strong added value in the development of innovative applications based on Constraints Programming paradigm. The JChoc platform presented in this chapter has been designed to support extensions: security, cryptography. In our experiments, We have implemented ABT protocol and solved the Meeting Scheduling problem (MSP) in a real distributed environment. In a dynamic environment, we have solved dynamic problems with DynABT, to show the benefits of our platform that supports the dynamic aspect. We found that, by using this platform, we can adopt easily any proposed protocol for solving distributed constraint problem even if environment changes dynamically. The next chapter is focusing on enhancing the platform, by adding other layers that will allow users to implement their multi-robots applications easily. The platform will allow the communication between robots in a dynamically changing environment.

CHAPTER

4

Mobile Robotic JChoc DisSolver: A Distributed Constraints Reasoning platform for mobile multi-robot problems

Preamble

Even if mobile robotics is a complex research area, in this chapter, we prove that distributed constraint reasoning techniques can be utilized as a very elegant formalism for multi-robot decision making. First, we describe dynamic distributed constraint satisfaction formalism, the new platform architecture "RoboChoc" and specify how decision making can be controlled in multi-robots environment using dynamic communication protocols. Then we provide an example application that illustrates how our platform can be used to solve multi-robot problems using constraint programming techniques.

Contents

4.1 INTRODUCTION

With increasing deployment of applications involving multiple robots and unmanned air vehicles, there is growing need for programming platform with realistic use context. Coordination between robots to solve a combinatorial problem, is one of the most complex research area in mobile robotics. In the last decade, as many approaches of real-time constraint handling have been proposed, constraint

programming has proved to be a stand-alone technology that can be used in several research fields. This chapter presents a new multi-robot platform named Mobile Robotic JChoc DisSolver (RoboChoc), based on constraint programming to solve multi-robot combinatorial problems in dynamically changing environments. The resolution of such problem is impacted by the tools that are used. The RoboChoc platform is based on a powerful constraint solver, named Choco [Jussien et al., 2008], a multi-agent platform JADE [Bellifemine et al., 2007] that insures best management and communication between agents, and a Robot Operation System (ROS) [Quigley et al., 2009] that provides libraries and tools to help software developers create robot applications. RoboChoc uses a ROS Layer that benefits from the ROS community contributions (e.g. navigation [Zaman et al., 2011], localization [Thomas and Ros, 2005], etc). Since a problem is modeled using constraint programming techniques, users can implement the application and solve it easily.

The chapter is organized as follows. Section 2 presents related works. Section 3 shows a brief definition of Distributed and Dynamic Constraint Satisfaction Problems (DisCSPs and DDisCSPs). Section 4 presents the RoboChoc platform architecture. Section 5 describes an example of application. Finally, Section 6 presents our conclusions and perspectives.

4.2 RELATED WORKS

Many research focus upon the application of constraint programming in the robotic field. ThingLab [Borning, 1981] proposed a software package that allows users to graphically design and simulate a system which can use constraints to define the properties of the system and guided simulation of circuits, mechanical linkages, and other geometries.

The research of D. K. Pai [Pai, 1989, 1991] describes a new framework for robot programming using constraint satisfaction techniques. Pai defined two special classes of CSP variable. Input variables were used to capture current state and output variables were defined such that their values can be taken from the solution and translated to vehicle actuator commands. For each iteration, the previous solutions were fed into the solver, and then the input variables were updated. A repair algorithm then fixed the solution given the changes and the constraints.

Zhang and Mackworth [Mackworth, 1997; Zhang and Mackworth, 1994, 2002] proposed a constraint-based system for the design of intelligent and hybrid agents. The hybrid agents are those which can interact continuously and discretely with their environments. A controller based on constraint satisfaction drives the system toward satisfying all constraints. During the task, system detects the dissatisfaction of any constraint, the controller can initiate a backtrack.

Other contributions were recently proposed. the researches of Richard S. Stansbury and Arvin Agah [Stansbury and Agah, 2012] presents a decision making framework that help robots to perform the best decision at any given moment. The selected task must satisfy all constraints. The framework models this decision as a Constraint Satisfaction Problems (CSPs). Where the contribution of Doru Panescu and Carlos Pascal [Panescu and Pascal, 2014] presents a multi-robot system planning based on a combination between a multiagent system and a constraint satisfaction problem approach. They make some simulation tests with a multi-robot system presented by four robots to solve an assembly goals. They use in these simulations the coloured Petri net models, specifically developed for a distributed constraint satisfaction algorithm.

To date, no researchers have proposed a constraint programming platform to practically implement multi-robot application. DisCSP has been used as a formalism for diagnosing coordination failures in multi-robot systems [Meir et al., 2006]. This work shows that, in general, a trade-off exists between the computational requirements of the algorithms, and their diagnosis results. In contrast, we use sophisticated APIs (JADE, ROS and Choco) to model the ideal coordination relationships between robots. We present results from experiments conducted in realistic usage conditions. In addition, previous contribution doesn't use standard and distributed platform, in contrast to our work.

4.3 PLATFORM ARCHITECTURE

4.3.1 RoboChoc description

RoboChoc is a JADE-based platform. This platform is a distributed constraint multi-robot system, proposed for mobile robotic applications. It can also be used to analyze and test algorithms proposed by constraints programming community. This platform is presented in the form of programming environment (API) and applications to help different types of robotic users. Hence, RoboChoc can be easily adopted by two main actors:

- Developers to design and develop multi-robot applications.
- Non-expert user can interact directly with applications based on distributed constraint programming.

This proposed platform has several advantages:

- A multi-robot problem modeled as distributed constraint problem, can be easily addressed and solved in a realistic environment by non-expert users;
- The performance of the proposed protocols (e.g. ABT, AFC, Adopt, etc) can be actually tested and proved in a realistic robot cordination, using communication channel (i.e. WLAN WPAN WMAN WWAN);

- It offers a modular software architecture which accepts extensions easily (e.g. security, confidentiality, cryptography, etc);
- Thanks to the extensibility of JADE communication model [Bellifemine et al., 2007], RoboChoc allows the development of multiagent systems and applications consistent with Foundation for Intelligent Physical Agents (FIPA) [1] standards and specifications;
- Thanks to the the robustness of Choco platform [Jussien et al., 2008], complex agent (i.e. multiple variables per agent) can easily address and solve its local sub-problem and use solutions as a compiled domain.
- Thanks to the the Robot Operation System ROS [Quigley et al., 2009], The distributed nature of ROS also fosters a large community of user-contributed packages that add a lot of value on top of the core ROS system.

This platform consists of several modules presented as services. The main constraint programming services offered are based Distributed Constraint Reasoning Protocols (DCRP) and Choco Solver (CS). Choco is a platform for research in centralized constraint programming and combinatorial optimization. This choice of Choco enabled us to benefit from the modules already implemented in it. In the next section, we will study the different elements of RoboChoc platform.

4.3.2 RoboChoc Architecture

RoboChoc architecture allows FIPA specifications. It is implemented in JAVA and provides classes that implement and inherit from JADE and Choco platforms to define the behavior of specific agents. Figure 19 represents the main RoboChoc architectural elements. This platform has five main modules.

- Distributed Constraint Reasoning Protocols unit: provides distributed constraints protocols as service. This element defines new types of messages and implements the behavior of the agent when receiving and sending a specific type of information (e.g. ABT, AFC, Adopt, etc...);
- Constraint Solver Unit: provides the ability to address and resolve local CSP sub-problem by using the Choco Solver;
- Services Management Unit: manages services in the platform. It's based on the Director Facilitator (DF) agent of JADE;
- Communication Management Unit: manages the communication in the platform. It's based on the Agent Communication Channel (ACC) of JADE;
- Authority Unit: oversees the registration of robots, their authentication, their access and the use of the system.
- ROS Layer Unit : provides a set of tools to apply a decision or to read the robot situation (e.g. scan sensor state, moving the robot, etc.).

These six modules are activated at each time the platform is started.

1. http://www.fipa.org/

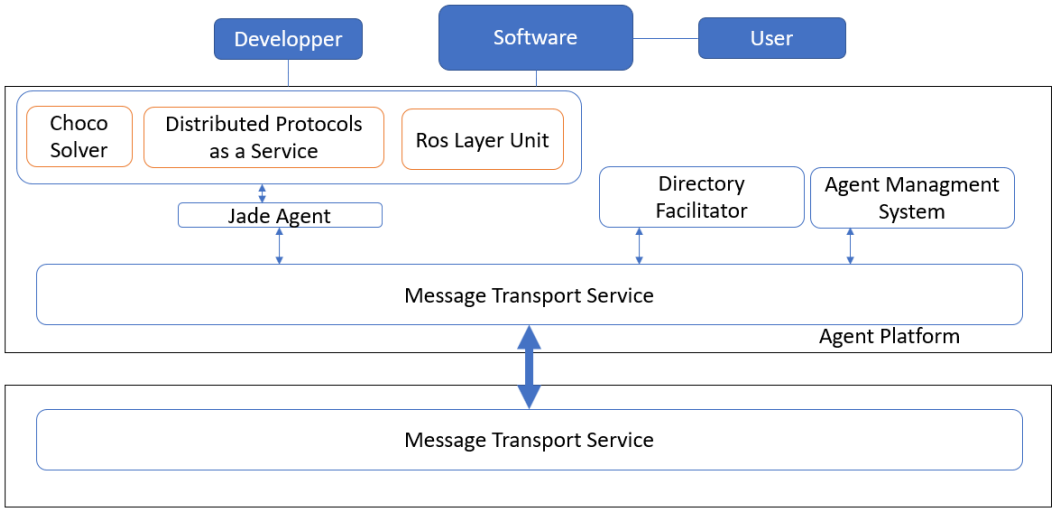


Figure 19 – RoboChoc Architecture

The JADE agent is also a key player in our platform. Thanks to JADE an Agent Identifier (AID) identifies an agent uniquely.

RoboChoc uses extensively a sniffing tool for debugging, or simply documenting conversations between robots. The sniffer subscribes to Authority Unit to be notified of all platform events and of all message exchanges between a set of specified robots. When the user decides to monitor a robot or a group of robots, every message directed to, or coming from, that robot/group is tracked and displayed in the sniffer GUI. The user can select and view the details of every individual message, save the message or serialize an entire conversation as a binary file.

The platform can be used with all robots that use ROS as an operating system, and no additional configuration is needed. The user has to choose the appropriate robots for its application.

4.4 EXAMPLE APPLICATION

4.4.1 Description of 3D N^2 queens problem:

The n-queens problem is a classical combinatorial problem that can be formalized and solved by constraint satisfaction problem. In the n-queens problem, the goal is to put n queens on an n×n chessboard so that none of them is able to attack (capture) any other. Two queens attack each other if they are located on the same row, column, or diagonal on the chessboard. This problem is called a constraint satisfaction problem because the purpose is to find a configuration that satisfies

the given conditions (constraints).

Three Dimensional Queens Problems is an extension of the well known n-queens problem in 3D dimensional space. The goal in this problem is to place N^2 queens in an $N \times N \times N$ cube such that none of them is able to attack any other. A queen can attack another if they exist in the same row, column, or diagonal formed by the division of the tree dimensional space. We mention that the space is divided into n^3 position which the n^2 queens will take place.

This problem was treated, modeled and solved in the following chapter, using linear programing: Three Dimensional Queens Problems [Allison et al., 1989]. In this research they found that there is no solution for the n less than 11, n=12, n=14. They found exactly for n=11, 528 solutions using an other formalism far from constraint programming.

In our work, we propose for the first time, a DisCSPs modeling for the same problem. The DisCSP that model the problem is formalized as a tuple (X, D, C, A, ψ) where:

— $X = \{ X_{1,1}, \ldots, X_{n,n}\}$ is a set of $n \times n$ variables.

— $D = \{D_{1,1}=\{1..n\}, \ldots, D_{n,n}=\{1..n\}\}$ is a set of domains, where $D_{i,j}$ is a finite set of possible values for the variable $X_{i,j}$.

— C = {
C1=$\{(X_{i,j1} \neq X_{i,j2}) and (|(X_{i,j1} - X_{i,j2}| \neq |j1 - j2|) \ / \ (i, j1, j2) \in [1, n]\}$ where $X_{i,j1}$ and $X_{i,j2}$ designate queens existing in the same row.

C2=$\{(X_{i1,j} \neq X_{i2,j}) and (|(X_{i1,j} - X_{i2,j}| \neq |i1 - i2|) \ / \ (i1, i2, j) \in [1, n]\}$ where $X_{i1,j}$ and $X_{i2,j}$ design queens existing in the same column.

C3={if $i1 \neq i2$ and $j1 \neq j2$ and $|i1 - i2| \neq |j1 - j2|$ then $X_{i1,j1} \neq X_{i2,j2}$ / $(i1, i2, j1, j2) \in [1, n]$}
} is the set of constraints.

— $A = \{ A_{1,1}, \ldots, A_{n,n}\}$ is a set of $n \times n$ agents.

— ψ is the function that assign the variable $X_{i,j}$ to the agent $A_{i,j}$.

In this modeling, each queen $Q_{i,j}$ is represented by a variable $X_{i,j}$

4.4.2 Using RoboChoc to solve the 3D N^2 queens problem

The problem can be solved by using RoboChoc. Hence, each agent is presented as a robot. We use XML files to describe the sub-problems for each robot. The figure 20 shows an example of the XML file used to describe the sub-problem for the robot

named Robot3.1.

Each variable has a unique ID, which is the ID of its owner robot. This is necessary when defining constraints (scope of constraints). For constraints, we used two types: TKC for Totally Known Constraint and PKC for Partially Known Constraint. Constraints can be defined in extension or as a Boolean function. Different types of constraints are predefined: equal to $eq(V_i, V_j)$, different from $ne(V_i, V_j)$, greater than or equal $ge(V_i, V_j)$, greater than $gt(V_i; V_j)$, etc. In this sub-problem the constraints presented in the DisCSPs modeling are defined by the predicate p1. In the tag topics we indicate the topics to be created and used by the ROS Layer Unit. these topics will be used to control the robot motion in this application. After defining our sub-problem we can configure our solver.

The problem can be solved using the ABT algorithm assuming that no perturbation will occur during the resolution, because we will use the Modular Open Robots Simulation Engine(Morse) ([Echeverria et al., 2011]) with ROS as a middle ware. The adequate type of robot that can represent queens for this application is the drone Quadrotor_Dynamic.

We solved the modeled problem with the Constraint Solver Unit in the platform, and we have found exactly the same results presented in the research of [Allison et al., 1989]. To simplify the problem in this application, we will use just n=4 as the figure 23 shows. For the reason that there is not any solution for n=4, the robots who can't belonging to the solution will be taken from the operational scene. Thus only robots that will satisfy all constraints will fly to take their positions. We can test the functioning of the platform in a physically distributed environment. So we chose machines that simulate the different robot of the problem, and filed each sub-problem in a machine, before launching it.
Figure 21 shows how the master launches its communication interface listening on the network. We begin by instantiating the AgentsContainer object (line 7), This class models the distributed problem when RoboChoc is used to solve a problem in a real distributed environment. All information on distributed problem is encapsulated in this object (identities of agents, inter-agent constraints, protocol, etc.). Then, we define the type of master (line 8) (ABT in this case). Finally, we trigger the container and we launch the master (lines 10).

Figures 22 shows how to launch robots as agents. We start with instantiate the AgentsContainer object (line 8), followed by the agent and distributed sub-problem declaration which specifies the resolution algorithm to be used (line 9-10). Next, the declaration of the container containing the master with its IP address (line 11). Eventually, we launch the agent (line12).

The master waits for the confirmation of creation all agents before ordering the start of the search. Thus, the problem can be solved using a specified implemented protocol (ABT for this example). Figures 23 and 24 show the initial and final state

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
      <presentation type="DisCSP"/>
   <domains nbDomains="1">
       <domain name="D1" nbValues="4">1..4</domain>
   </domains>
   <variables nbVariables="1">
      <variable name="V3.1" domain="D1" />
   </variables>
   <topics nbTopics="1">
      <topic name="/drone3.1/waypoint3.1" />
   </topics>
   <predicates nbPredicates="1">
      <predicate name="p1">
      <parameters>int X int Y int i int j</parameters>
      <expression>
       <functional>and(neq(X, Y), neq(abs(minus(X, Y)),abs(minus(i-j))))</functional>
      </expression>
     </predicate>
   </predicates>
   <agents_neighbours>
      <agents_parent>
         <agent name="Robot1.1">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C5" reference="p1" scope="V1.1 V3.1 1 3">
                  <parameters>V1.1 V3.1 1 3</parameters>
               </constraint>
            </constraints>
         </agent>
         <agent name="Robot2.1">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C5" reference="p1" scope="V2.1 V3.1 2 3">
                  <parameters>V2.1 V3.1 2 3</parameters>
               </constraint>
            </constraints>
         </agent>
         <agent name="Robot1.3">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C5" reference="neq" scope="V1.3 V3.1">
                  <parameters>V1.3 V3.1</parameters>
               </constraint>
            </constraints>
         </agent>
         <agent name="Robot2.2">
            <constraints nbConstraints="1">
               <constraint model="TKC" name="C5" reference="neq" scope="V2.2 V3.1">
                  <parameters>V2.2 V3.1</parameters>
               </constraint>
            </constraints>
         </agent>
      </agents_parent>
      <agents_children>
         <agent name="Robot4.1" variable="V3.1" />
         <agent name="Robot3.2" variable="V3.1" />
         <agent name="Robot3.3" variable="V3.1" />
         <agent name="Robot3.4" variable="V3.1" />
         <agent name="Robot4.2" variable="V3.1" />
      </agents_children>
   </agents_neighbours>
</instance>
```

Figure 20 – Example of a sub-problem for the robot3.1

of resolution of the problem. Between these two states, the solving phase was performed to satisfy all constraints.

```
1  import JChoc.DisSolver;
2
3  public class Master
4  {
5      public static void main(String[] args)
6      {
7          DisSolver js = new DisSolver();
8          js.setType("MasterABT");
9          js.setGui(true);
10         js.setNumberOfAgents(4);
11         js.run();
12     }
13
14 }
```

Figure 21 – How the master launches its communication interface.

```
1  package jchoc.run;
2  import jchoc.tools.AgentsContainer;
3
4  public class RunDrone
5  {
6      public static void main(String[] args) {
7
8          AgentsContainer agentContainer = new AgentsContainer();
9          agentContainer.setAgentType("AgentABT");
10         agentContainer.addAgent("Robot3.1","Robot3.1.xml",true);
11         agentContainer.setMainContainer("192.168.1.14");
12         agentContainer.run();
13           }
14 }
```

Figure 22 – Run drone.

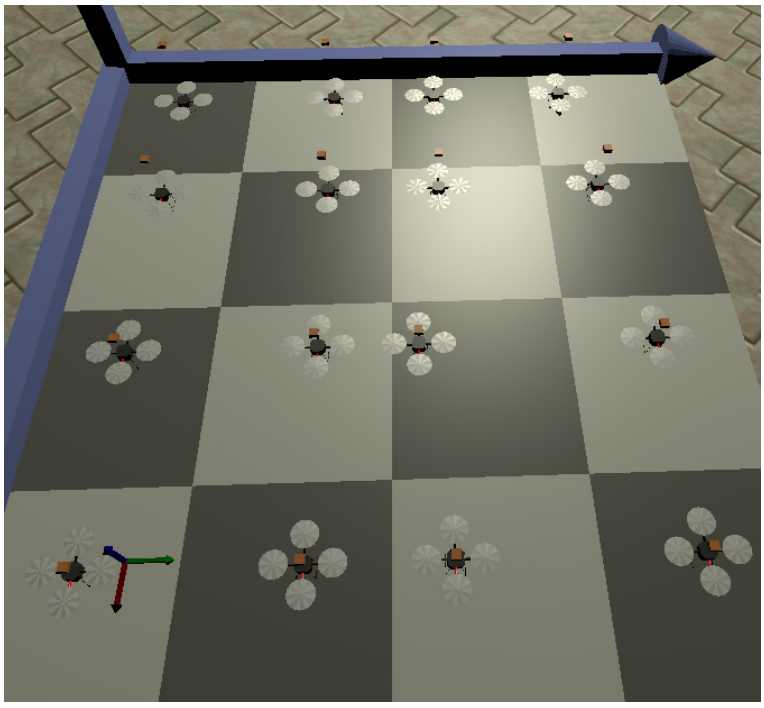


Figure 23 – Initial state.

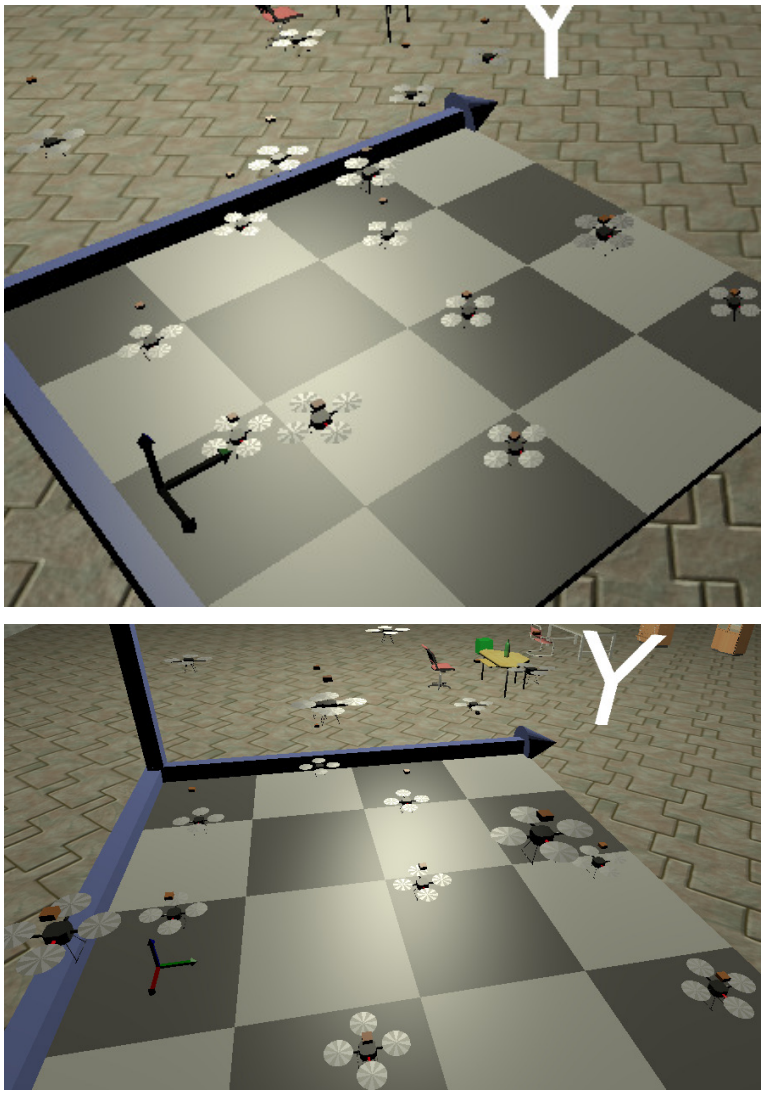


Figure 24 – Final state

Figures 25 shows the exchanged messages in the resolution phase sniffed by the Sniffer Agent in the platform. ABT algorithm is executed autonomously by each robot. Robots do not have to wait for decisions of others but they are subject to a total(priority) order. Each robot tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors. When an robot assigns a solution to its local problem, the selected value is sent to lower priority neighbors. When no solution is possible for a local problem, the inconsistency is reported to higher robots in the form of a *nogood*. ABT computes a solution (or detects that no solution exists) in a finite time. The total ordering on robots is static.

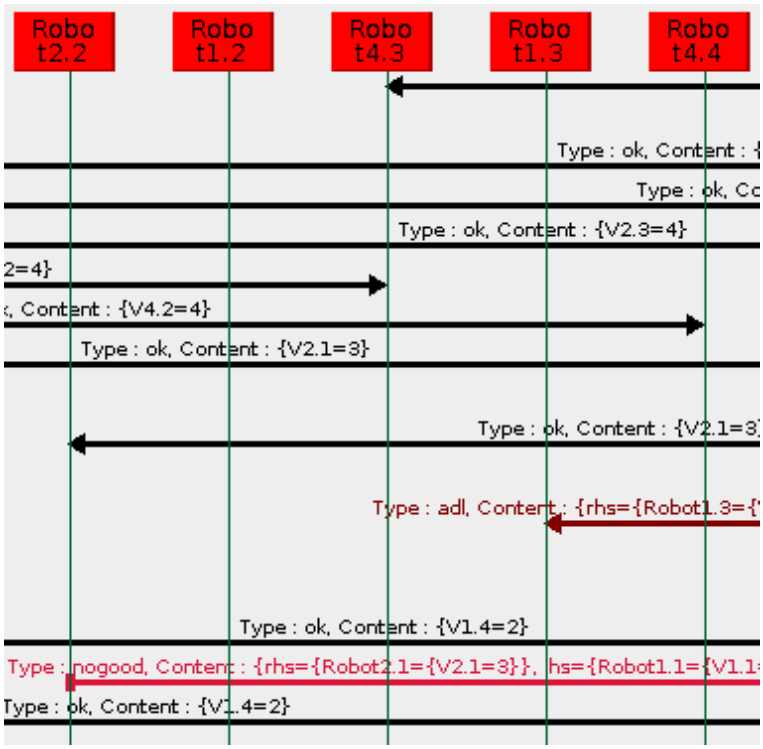


Figure 25 – Sniffed messages in the solving phase.

4.5 CONCLUSIONS

The goal of this work was to propose a Distributed Constraints Reasoning (DCR) platform, which can be easily used to solve robotic application in a realistic use.RoboChoc platform presented in this chapter has been designed to solve any multi-robot problems designed as a DisCSP and to support security and cryptography extensions (thanks to JADE API).

In this chapter we have modeled the 3D N^2 queens problems as a DisCSP problem. We have used the MORSE Simulator to run drone-robots, where each queen is presented by a drone. Each drone performs as an ABT agent that applies the decision made by the algorithm or reads his actual situation using the ROS Layer Unit, which is independent of the solving protocol. Users can contribute in the ROS Layer Unit by adding new ROS applications to simplify the use of new destined robotic task.

The next chapter is focusing on enhancing the platform to deal with e-marketplace issues by proposing a new problem formulation and Distributed Constraint Reasoning protocol.

CHAPTER

5

A use case of negotiation networks: smart e-marketplace platforms

Preamble

With the growing success of e-marketplace adoption, the need for new intelligent approaches to support both buying and selling goods or services becomes a fundamental requirement. In the recent past, several techniques have been proposed, in different research areas, allowing the simulation of a market with a set of sellers proposing products and buyers who have a list of interests. In this chapter, we propose a new problem formulation and Distributed Constraint Reasoning protocol to deal with e-marketplace issues. More specifically, our attention is focused on constraint-based multi-agent approach offering a flexible and confidential multi-lateral negotiation mechanism, namely, ABT-Trader. The satisfaction of strict preferences offers the best negotiation to buy all the wanted products, otherwise, constraint and variable relaxations are adopted to look for feasible solutions.

This approach has been implemented and tested on JChoc Platform using preliminary generated problems. The experimental results show that our approach is of practical interest: it proposes feasible time-tested solutions.

Contents

5.1 Introduction

Electronic marketplace is a place where customers and suppliers can meet, negotiate, make decision and transact as in a traditional marketplace. With the growing need of e-marketplace in our daily life, the need for new innovative techniques to support both customers and suppliers in buying and selling goods or services is increasing quickly.
In the recent past, several platforms [Eriksson et al., 1998] where people can look for a given good or service has changed the traditional ways of negotiating and doing business. These various forms of e-marketplaces are considered as revolutionary vectors of e-marketplace technology. However, most of them only serve a small part of the transaction process and need human intervention before, during, or after negotiation process. Nowadays, autonomous and intelligent multi-agent systems make the business processes in e-market more efficient and revolutionary [Ren et al., 2009] and [Hemaissia et al., 2007]. These new sophisticated systems may replace both buyers and sellers on decision making in order to reach a negotiated agreement. In particular, virtual sellers and buyers can be provided with a negotiation protocol that help them to specify constraints and then to look for the optimal/pseudo-optimal decision regarding these specified preferences (constraints). When the combinations of multi-lateral negotiation become complicated, and the constraints both on the customers and suppliers sides become vague and complex, intelligent negotiation gives rise to new challenges for developers of architecture and software technologies underlying e-marketplaces. Hence, intelligent agents should be initially created with their complete set of strategies and should be able to learn and make decision; rather than having an automation behaviour, they should have the ability to acquire experience from previous negotiations they've conducted. Although a completely uncontrolled e-marketplaces are still rather a vision than reality.
To achieve this challenge and remove these limitations, some projects have been initiated, such as SICS MarketSpace [Eriksson et al., 1998]. Afterwards, many negotiation mechanisms have been proposed in the literature [Fatima et al., 2007], [Lai et al., 2007], [Ren et al., 2009] and [Hemaissia et al., 2007]. The most important work has been done on negotiation process and the formulation model still unripe and is not applicable in realistic environments.
One of the most promising answer to the new challenges posed by arising e-marketplaces is "Constraint Programming" paradigm. Several papers have proposed Distributed Constraint Reasoning (DCR) as a paradigm for problem modeling and solving in framework of Multi-agent Systems. One of the main features of DCR approaches is its distributed nature in which a set of intelligent agents can each make local decisions and communicate in order to improve global decision problem. To the best of our knowledge, few works have been interested to the formulation of e-marketplace environment using Distributed Constraint Reasoning techniques [Parunak et al., 1998] [Rahwan et al., 2002].
In this chapter, we propose a new problem formulation and DCR protocol to deal

with e-marketplace issues. More specifically, our attention is focused on constraint-based multi-agent approach offering a dynamic and privacy multi-lateral negotiation mechanism, namely, ABT-Trader. Also, the framework is based on two resolution phases, the registration and building network phase, in which each agent builds its distributed constraint network, and the negotiation phase, in which the sellers and buyers agents negotiate using the distributed constraint based techniques reasoning. The satisfaction of strict preferences offers the best negotiation to buy all the wanted products, otherwise, constraint and variable relaxations are adopted to look for a feasible solution.
The remainder of this chapter is organized as follows. Section 2 presents the background of Distributed Constraint Reasoning. Section 3 describes the main problem of e-marketplace environment, and the important issues to fully automated negotiation. Section 4 addresses the distributed constraint reasoning model using privacy and relaxation concepts. Section 5 focus on constraint resolution protocol as a negotiation mechanism. Section 6 discusses preliminary experiment evaluation, and section 7 gives a brief overview of related works. Finally, section 8 provides a conclusion and areas for further research.

5.2 Preliminaries

In a dynamic environment a DisCSP may change over time, these changes could be due to addition and/or deletion (relaxation) of variables, values, constraints, or agents. The Dynamic and Distributed Constraint Satisfaction Problems (DDisCSPs) can be described as a five tuple (X, D, C, A, ψ, δ) where :

- A, X, D, C, and ψ remain as described in DisCSP

- δ is the change function which introduces modifications.

Many DDisCSPs approaches (e.i : DynABT [Omomowo et al., 2008]) are proposed to solve such type of problems, and can be easily implemented in this platform.

A solution to a DisCSP is an assignment of a value to every variable of the distributed constraint network, in such a way that every constraint is satisfied. Solutions to DisCSPs can be found by searching through the possible assignments of values to variables such as ABT algorithm [Bessière et al., 2008].

Regarding constraints privacy, we adopte the Partially Known Constraints (PKC) [Brito and Meseguer, 2003], when the scope $variables(C_{ij})$ of each constraint is known by every related agent, but the relation $relation(C_{ij})$ is partially known. The model is as follows. A constraint C_{ij} is only partially known by its related agents. From $C_{i(j)}$ with agent j as,

$$variables(C_{i(j)}) = \{x_i, (x_j)\} \qquad relation(C_{ij}) \subseteq relation(C_{i(j)})$$

Where x_j in $variables(C_{i(j)})$ means that agent i knows little about the other variable of the constraint. From constraint C_{ij}, agent j knows the constraint $C_{(i)j}$,

$$variables(C_{(i)j}) = \{(x_i), x_j\} \qquad relation(C_{ij}) \subseteq relation(C_{(i)j})$$

It is required that,

$$relation(C_{ij}) = relation(C_{i(j)}) \cap relation(C_{(i)j})$$

5.3 Problem statement

Electronic marketplace is a place where customers and suppliers can meet, negotiate, make decision and transact as in a traditional marketplace. According to the literature, the classification of multi-agent e-marketplaces are based on three characteristics:

— Character of negotiation: On either side - on the buyers' and on the sellers' side - one or more participants may be negotiating; multilateral negotiation or bilateral negotiation.

— Number of issues: This characteristic represents the number of negotiation issues. In the simplest case, the negotiation can be reached over one-dimensional issue (price). In more complicated cases, the negotiation can be reached over multi-dimensional issues (related to price, quality, terms and conditions, etc.).

— Level of preferences : the preferences regarding the negotiation issues may be crisp or fuzzy.

In this chapter, we focus on intelligent electronic marketplaces and furthermore, on the special case of multi-agent systems negotiation. We assume:

— A multilateral negotiation space;

— A multi-dimensional issues (for simplification assumption we consider only the price attribute and for the other attributes quality, terms and conditions, etc. can be considered in an extended formulation);

— A strict preference (constraint) model that can be relaxed (removed) during negotiation.

The problem can be described as bellow :

— An actor in the market can be whether seller or buyer.

— An actor can trade many products (i.e. goods or services).

— Each seller cannot sell a product less than his corresponding lowest price.

— Each buyer cannot buy a product more than his corresponding highest price. We assume that this assumption can we relaxed during resolution.

— Each buyer have his fixed budget that cannot exceeded.

— Each buyer can buy a limited number of products depending on his budget and the priority of products those he wants to buy.

— Each buyer can choose the best received offer for a product using different criterion (e.g. warranty, quality, delivery, price, etc). In our model, we assume that except the price all criterion are equi priority. Thus, the offer evaluation is based only on the price.

— Each agent preference is considered as private.

5.4 Distributed constraints reasoning model

The first contribution of this chapter is to design the problem already explained as a Distributed Constraint problem, to do this, we will translate the different components of the problem as a set of <Agents, Variables, Domains, Constraints> such that the model describes exactly our problem. The model can be done in various ways, therefore we propose the one that we see the clearest, the simplest and the most efficient.

5.4.1 Agents

$A = \{Seller_1, ..., Seller_p, Buyer_1, ..., Buyer_q\}$ is a set of p sellers and q buyers.

5.4.2 Variables

We assume that :

— $X_j^{Buyer_i}$ is the j^{th} product needed by the buyer i.

— $X_l^{Seller_k}$ is the l^{th} product offered by the seller k.

X is the set of n variables $X_j^{Actor_i}$ that correspond to the j^{th} product of the i^{th} actor.

A variable is the product that can be a service or a good.

5.4.3 Domains

Each product has a set of possible prices. The price is not always fixed or known. As a variable, the product in the seller side can take a value from a set of prices those present his domain. Each buyer has an idea about the possible prices for each product, Those prices present the domain of a product for a buyer. So, the domain can be whether an interval value or a singleton.

5.4.4 Constraints

In our model the set of the constraints is defined as bellow:

$$C = \{C_{intra}^{seller}, C_{inter}^{seller}, C_{intra}^{buyer}, C_{inter}^{buyer}\}$$

— We assume that there are no intra constraints for a seller agent, means that, he can sell any product independently, thus

$$C_{intra}^{seller} = \varnothing$$

— To preserve the seller confidentiality, the inter constraints for a seller are partially known constraints and buyers cannot see these constraints, says that the seller cannot sell a product with a price lower than the smallest value in his domain thus

$$C_{inter}^{seller} = \{C_{seller_i, buyer_j} = \{X_k^{Buyer_j} \geq min(D(X_k^{Seller_i}))\}$$
$$/ \text{ for each seller i offering the product k interesting the buyer j}\}$$

— The buyer has to respect his budget. For each buyer j we assume the intra constraint as bellow:

$$C_{intra}^{buyer} = \{\sum_{i=1}^{k} X_i^{buyer_j} \leq Budget(Buyer_j)$$
$$/ \text{ for each buyer j } \}$$

— For the same reasons of confidentiality (constraint's privacy), we assume the inter constraints those avoid the sellers to sell a product with a price lower than the smallest value in his domain, thus

$$C_{inter}^{buyer} = \{C_{buyer_i, seller_j} = \{X_k^{seller_j} \geq max(D(X_k^{Buyer_i}))\}$$
$$/ \text{ for the seller i offering the product k interesting the buyer j}\}$$

5.4.5 Agent priority

In our model, there is two levels of priority 1 and 2. Each buyer is in the level 2 of priority, and has a set of parents, where each parent is a seller. All sellers have priority 1. Sellers can sell products to different buyers in parallel, and the buyer can negotiate with different sellers in parallel too.

5.4.6 Value Ordering heuristics

While the negotiation process, sellers select the values from their domains in descending order. However, the buyers choose the lowest proposed offers (i.e. prices) sent by the sellers for each product.

5.4.7 Variable and constraint relaxation

In case of unsolved problem, a buyer could remove variables (i.e. products) according to a variable priority heuristic to look for possible products with highest priority, and those satisfy the budget constraint. In this case, the inter constraints of the buyer could be relaxed.

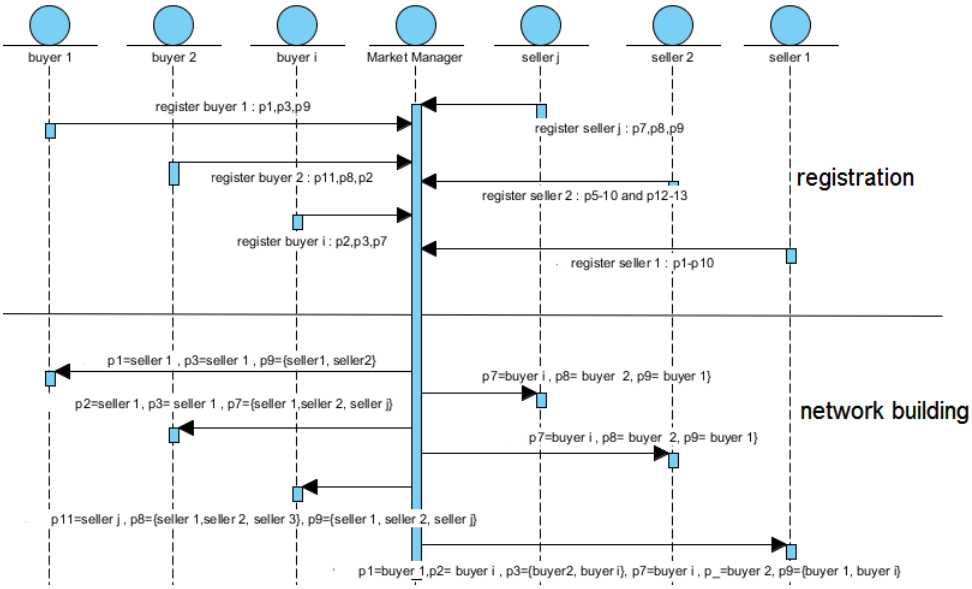


Figure 26 – Registration and building the DCSP network

5.5 Resolution protocol

In the resolution protocol we can observe two phases, the first phase is the building of the Distributed Constraint Satisfaction Problem network, and the second phase is the negotiation process. The registration and negotiation phases presented in the figures 26 and 27 show two types of agents: buyer and seller.

5.5.1 Registration process

The first phase of the solving process is divided into two parts: the registration and the network building. Every trader in the marketplace (seller or buyer) must be registered. This registration allows the manager of the marketplace to know the needs of each buyer, and the products proposed by each seller, thus, the building of the DisCSP network by linking between them.

For example, the buyer 1 registers himself to the marketplace and communicates his needs (e.g. product 1, product 3 and product 9), thereafter the manager sends to the buyer 1 a list of sellers for product 1, 3, and 9.

5.5.2 Negotiation process

The negotiation process starts when the marketplace manager sends for each buyer a list of sellers offering the needed products.

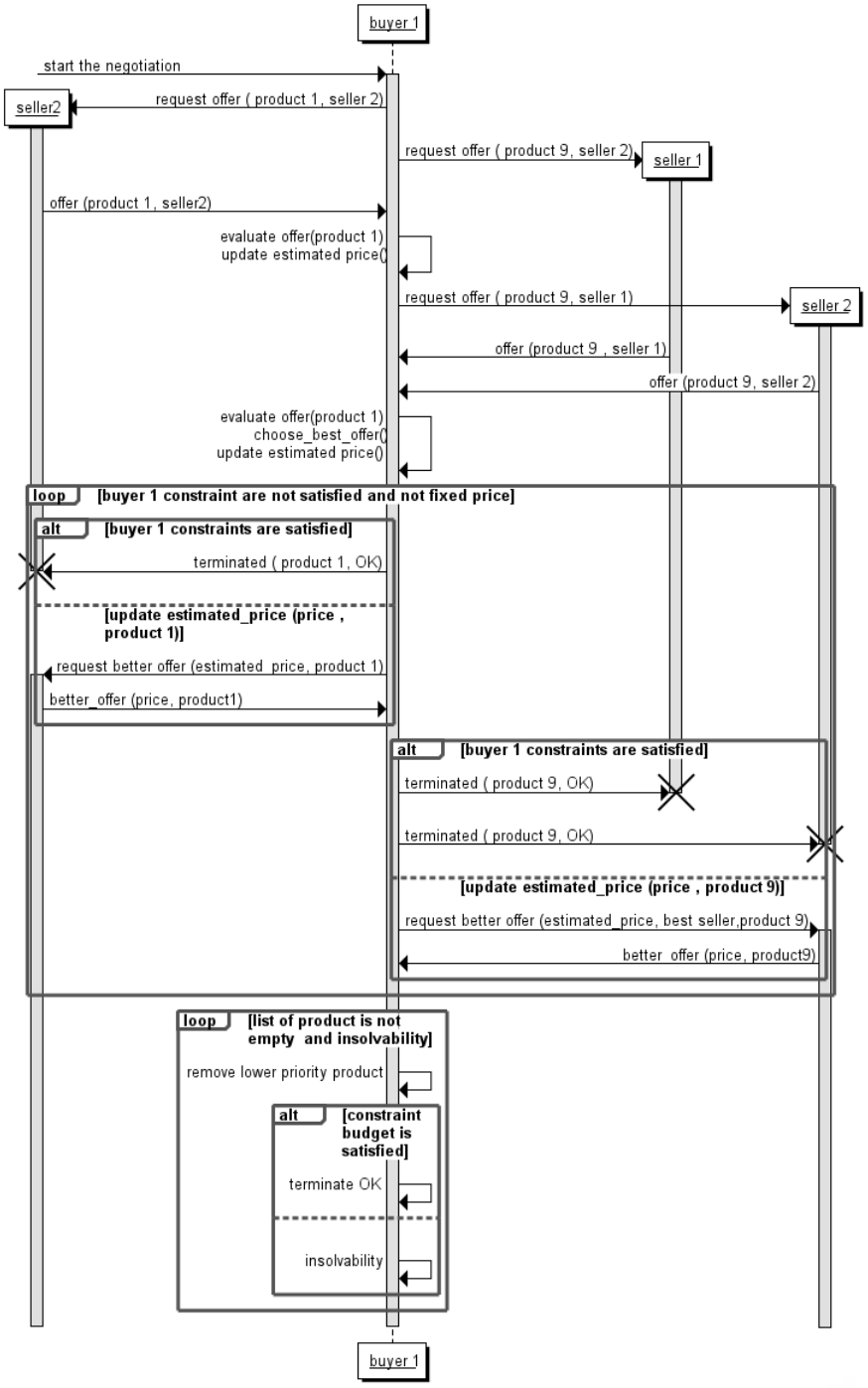


Figure 27 – Negotiation process

To simplify the explanation of our negotiation process let's consider a buyer 1 that wants to trade asynchronously with two sellers for 2 products (i.e. product 1 and 9) with a fixed budget b where price(product 1) + price(product 9)≤b.

The first step for the buyer 1 is to ask all received sellers to send their offers, then he chooses the best offer for each product. If all constraints are satisfied then the buyer terminates his negotiation. Otherwise, for each product, he send a message to the best seller to propose a better offer while he has not sent a fixed price, in the other case (i.e. the seller has already sent a fixed price for this product), he chooses another seller who proposes the best offer and who have not sent a fixed price to resend a better offer for this product. While the solving, if all sellers send their fixed price and the buyer cannot satisfy all his constraints then he tries to buy only the products those have a high priority for him and those satisfy his constraints.

5.5.3 Algorithm description

The ABT-Trader is based on ABT[Bessière et al., 2008] and used by two types of agents: buyer and seller. The pseudo code of ABT-Trader represented in Figures 1 and 2 says that the seller stores one nogood per removed value in the NogoodStore for each negotiation till the end of the resolution. All agents start the search by calling the procedure *Setup()* in which they initialize their nogoodStores, agentViews, finalAgentViews, products, etc. Then each agent checks his type (buyer or seller). If the agent is a buyer, in this case, he asks all available sellers of each product to be his parents and to negotiate with him about the price of this product, after that, he calls the main procedure named Buyer(), if not (i.e. the agent is a seller) he runs a loop where he waits for possible buyers, and when a buyer arrives (i.e. *sell?* message received), the agent seller creates a copy of all data (i.e. domains, nogoodStores, etc) and calls a new Seller() procedure to negotiate with this buyer about the product carried in the message *sell?* .
In the first time, the buyer waits for all sellers to propose their prices within a time out, then he tries to choose the best proposed values/prices and try to assign them. The buyer agent assign all values only when all constraints are satisfied, in this case, a solution is found. Otherwise, when there is a problem in this phase (i.e. the buyer cannot assign all variables), he tries to send a nogood per product to the seller who have not sent a finalOK? message, and who have proposed the best price for this product, after that, he waits with a time out for the new propositions, in this part, he does not wait for all new proposition to arrieve, he tries to find a solution when a new proposition is received. In case of all sellers have proposed their final prices for all products (i.e. the size of the agentView equal the size of FinalAgentView) and there are some not satisfied constraints, the buyer uses the priority between products to buy just products who have a high priority and those satisfy budget constraint only.
New "environments"

Algorithm 1 ABT_{Trader} Part1

```
 1: procedure Setup()
 2: initialize products, domains, myAgentViews, myFinalAgentView, nogoodStores,
    and type;
 3: Register with all products and type. type=buyer
 4: for each product ∈ products do
 5:     for each seller ∈ sellers(product) do
 6:         sendMsg : sell?(seller,product);
 7:         add seller to the list of sellers
 8:     end for
 9: end for
10: Buyer() true
11: msg ← getMsg() msg.type=sell?
12: Seller(msg.Sender, msg.product);
13: msg.isRead← false
14:
15: procedure Buyer()
16: myvalues ← enmpty ; end ← false; ¬end
17: msg ← getMsg()
18: switch (msg.type) do
19:    ok? : ProcessOk(msg);
20:    finalOk :ProcessFinalOk(msg);
21:
22: procedure Seller(buyer,product)
23: ChooseValue(buyer,product) ; end ← false; copy nogoodStores and domains;
        ¬end
24: msg ← getMsg();
25: switch (msg.type) do
26:    nogood : ResolveConflict(msg);
27:    stop : end ← true; delete current nogoodStores and domains;
28:
29: procedure ProcessOk(msg)
30: Update(myAgentView,msg.Assig);
31: CheckAgentView();
32:
33: procedure ProcessFinalOk(msg)
34: Update(myAgentView,msg.Assig);
35: Update(myFinalAgentView,msg.Assig);
36: CheckAgentView();
37:
38: procedure ResolveConflict(msg) The size of values those not elimunated by the
    current nogoodStore > 1
39: Add(msg.Nogood,myNogoodStore);
40: ChooseValue(buyer,product);
41:
```

Algorithm 2 ABT_{Trader} Part2

1: **procedure** ChooseValue(buyer,product)
2: get all values ∈ D(product) and not eliminated by the current goodStore values.size=1
3: sendMsg: finalOk(buyer, values.getvalue, product);
4: sendMsg: Ok?(buyer, values.getBiggestValue, product);
5: **procedure** CheckAgentView() AgentView.size ≤ sellers.size and waiting_time < time_out
6: Exite this Procedure;
7: Choose the best prices from the agentView and try to assign them to myValues; myvalue are not assigned
8: Choose the best price for each product and not in finalAgentView and send nogood with the my proposed price to its seller; No nogood sent
9: Try to assign variables or products who have the high priority and send msg stop to all sellers;
10: end ← true;
11: Send msg stop to all sellers;
12: end ← true;

While the negotiation, when the agent seller receives a nogood, he tests the size of values coherent with the current nogoodstore (the nogoodstores and domains are a copy of the initial data and will be removed in the end of the current negotiation), if the size is 1, he send a final price using the FinalOk message, if not, he decreases the chosen value and sends it using the message ok?. The agent seller starts the negotiation with choosing the biggest value and when he receives a nogood he decreases it. For the first time the buyer can propose the lowers prices found in his products domains, and the seller remove all possible values bigger then the proposed values from his domain while the size of value is greater than 1.

5.6 Experiment

5.6.1 Experimental platform

JChoc [Benelallam et al., 2015] is a JADE-based[Bellifemine et al., 2007] platform. This platform is a distributed constraint multi-agent system, It can also be used to analyze and test algorithms proposed by constraints programming community, and it allows the use of real communication channels. We have implemented the ABT-Trader in this platform, where all services are managed by the services management unit and the messages are transported between agents by the communication management unit.

5.6.2 Experimental Settings

We use XML files to describe the sub-problems for each agent (i.e. buyer or seller). The figure 28 shows an example for an agent seller and the figure 29 shows an example for an agent buyer before the network building. The inter constraints will be added during registration process. We have used in this experimentation two buyers named C1 and C2 and three sellers named F1, F2 and F3. The agent C1 wants to buy products X, Y, and Z. C1 has 115 as a budget, and has an estimated price about each product: D(X) = [100, 200], D(Y)=[150, 300], D(Z)=[30, 70]. In case of unsolved problem, he tries to buy the products X then Y and then Z(i.e X is more prioritized than Y, and Y is more prioritized than Z). The agent C2 wants to buy two products X and Z. C2 has 30 as a budget, and has an estimated price about each product: D(X) = [50, 100] and D(Y)=[10, 70]. Like C1 In case of unsolved problem, he tries to buy products X then Y (i.e X is more prioritized than Y). As a seller, F1 proposes two products X and Y, he can sell X from 25 to 50 and Y from 70 to 100. F2 proposes the three products X with a fixed price: 70, Y for 70 to 100, and Z for 40 to 60. Finally, F3 proposes X and Y as products, 80 to 120 for X, and 60 as a fixed price for Y.
We assume that there are no constraints between sellers or between buyers. And for a seller there are no intra constraints. Each agent was launched in a machine core i7 with 8Go of Ram, and they can communicate between them using the http protocol (e.i. Each machine is connected to internet).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <instance>
3  <presentation name="eMarketPlace" type="DisCSP" model="Complex" constraintModel="PKC" format="XDisCSP
       1.0" />
4  <domains nbDomains="2">
5          <domain name="Dx" nbValues="">25..50</domain>
6        <domain name="Dy" nbValues="">70..100</domain>
7  </domains>
8  <variables nbVariables="12">
9      <variable name="Xf1" id="1" domain="Dx" description="X" />
10    <variable name="Yf1" id="2" domain="Dy" description="Y" />
11 </variables>
12    <constraints nbConstraints="0">
13    </constraints>
14
15    <predicates nbPredicates="0">
16    </predicates>
17
18    <agents_neighbours>
19    </agents_neighbours>
20    <services service="Provider-X Provider-Y"></services>
21 </instance>
```

Figure 28 – seller F1 xml file

5.6.3 Experimental Results

The solving phase is stopped by the agent buyer when finishing all negotiations. The figure 30 shows the start and the end of the buyer C1 negotiations, where he

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
<presentation name="eMarketPlace" type="DisCSP" model="Complex" constraintModel="PKC" format="XDisCSP 1.0" />
   <domains nbDomains="1">
         <domain name="Dx" nbValues="">100..200</domain>
      <domain name="Dy" nbValues="">150..300</domain>
      <domain name="Dz" nbValues="">30..70</domain>
   </domains>
   <variables nbVariables="12">
      <variable name="Xc1" id="1" domain="Dx" description="X" />
      <variable name="Yc1" id="2" domain="Dy" description="Y" />
      <variable name="Zc1" id="3" domain="Dz" description="Z" />
   </variables>

   <constraints nbConstraints="3">
      <constraint name="C1" reference="sum" scope="Xc1 Yc1 Zc1 115" arity="4" budget="115"/>
      <constraint name="C2" reference="priority"    scope="X Y Z" arity="3"/>
   </constraints>

   <predicates nbPredicates="0">
   </predicates>

   <agents_neighbours>
   </agents_neighbours>

   <services service="Customer-X Customer-Y Customer-Z"></services>
</instance>
```

Figure 29 – buyer C1 xml file

```
started at: 2016-07-14 03:27:42
Providers found are: {X=[F1, F3, F2], Y=[F1, F3, F2], Z=[F2]}
Final price proposed from provider: F2, for product: X: 70
Price proposed from provider: F1, for product: X: 50
Final price proposed from provider: F3, for product: Y: 60
Price proposed from provider: F3, for product: X: 120
Price proposed from provider: F2, for product: Y: 100
Price proposed from provider: F1, for product: Y: 100
Price proposed from provider: F2, for product: Z: 60
All providers have proposed prices: {X={F1=50, F2=70, F3=120}, Y={F1=100, F2=100, F3=60}, Z={F2=60}}
min prices chosen are: {X={F1=50}, Y={F3=60}, Z={F2=60}}
All constraints were tested -> sum=170  b=115
-- can buy all services: false from{X={F1=50}, Y={F3=60}, Z={F2=60}}
Sent nogood to F1:  X:100
Sent nogood to F1:  Y:150
Sent nogood to F2:  Z:30
Final price proposed from provider: F2, for product: Z: 40
Price proposed from provider: F1, for product: Y: 100
Price proposed from provider: F1, for product: X: 50
.......................
.......................
Sent nogood to F2:  Y:150
Price proposed from provider: F2, for product: Y: 50
All providers have proposed prices: {X={F1=25, F2=70, F3=80}, Y={F1=70, F2=50, F3=60}, Z={F2=40}}
min prices chosen are: {X={F1=25}, Y={F2=50}, Z={F2=40}}
All constraints were tested -> sum=115  b=115
-- can buy all services: true from{X={F1=25}, Y={F2=50}, Z={F2=40}}
Stopped at: 2016-07-14 03:27:42
```

Figure 30 – C1 log sample

can buy all products: X from F1 for 25, Y from F2 for 50, and Z from F2 for 40, And the figure 31 shows the start and the end of the buyer C2 negotiations, as a result, he can buy only the product X from the seller F1 for 25. These preliminary results show the effectiveness of our approach, since the negotiation perform a relevant negotiated price. In a time tested performance.

```
started at: 2016-07-14 03:27:42
Providers found are: {X=[F1, F3, F2], Y=[F1, F3, F2]}
Price proposed from provider: F1, for product: X: 50
Final price proposed from provider: F2, for product: X: 70
Price proposed from provider: F3, for product: X: 120
Final price proposed from provider: F3, for product: Y: 60
Price proposed from provider: F2, for product: Y: 100
Price proposed from provider: F1, for product: Y: 100
All providers have proposed prices: {X={F1=50, F2=70, F3=120}, Y={F1=100, F2=100, F3=60}}
min prices chosen are: {X={F1=50}, Y={F3=60}}
All constraints were tested -> sum=110  b=30
-- can buy all services: false from{X={F1=50}, Y={F3=60}}
Sent nogood to F1:  X:50
Sent nogood to F1:  Y:10
Final price proposed from provider: F1, for product: Y: 70
.....................
.....................
Final price proposed from provider: F3, for product: X: 80
All providers have proposed prices: {X={F1=25, F2=70, F3=80}, Y={F1=70, F2=50, F3=60}}
min prices chosen are: {X={F1=25}, Y={F2=50}}
All constraints were tested -> sum=75  b=30
-- can buy all services: false from{X={F1=25}, Y={F2=50}}
--End of negotiation with final proposed prices:{X={F1=25, F2=70, F3=80}, Y={F1=70, F2=50, F3=60}}
Can't buy all services
Priority is:  X then Y
My budget is: 30 and what i will buy: {X={F1=25}} because the sum is: 25
I will buy : {X={F1=25}}
Stopped at: 2016-07-14 03:27:42
```

Figure 31 – C2 log sample

5.7 Related works

MarCon [Parunak et al., 1998](market-based constraints) aims to support a mix of human and artificial agents by offering a systematic method for applying markets to a wide array of problems.
A MarCon configuration is a network of alternating variables and constraints, each variable and each constraint is a separated agent. The initial network construction must have at least two constraint and one variable, in other definition, it must have buyer, a seller which are agents constraints and a variable which is the environment where calculation happens.
Although the platform uses the notion of agent, but it does not respect the literature of distributed constraint problems in the side as the constraints and variables are separated agents rather than having each agent with a set of variables, domains, constraints.
On the other hand, the converge toward the solution is not always valid because the shrinking of range price. However, if the participants price or assignment ranges do not intersect, the variable agent recommends human negotiation between the buyer and seller, so the platform became invalid.

In [Rahwan et al., 2002], authors present a DisCSP framework for one to many negotiation by means of conducting a number of concurrent coordinated one to one negotiation, and assume that this framework can be extends to be able to solve the Many-To Many negotiation problem. Also, the framework is based on two negotiation levels , the agent negotiation level in which each agent use the constraint based techniques reasoning, and the coordination level in which the coordinator agent evaluates how well subordinates agent has done, and issues the new instructions. However, negotiations are focused on one product with multi criterias, and the

DisCSP formulation is not presented.

Moreover, other several works have been condacted abroad Constraint Programming. The model in [Fatima et al., 2007] performs optimal negotiation process, but it works only in the environment with a fixed number of agents. The model in [Lai et al., 2007] is a multi-issues negotiations between two agents. However, issues of multi-lateral model still not taken into consideration.

5.8 Conclusion and future works

Intelligent multi-agent electronic marketplaces are promising, and they will play an essential role in e- commerce and e-business activities. With the growing success of e-marketplace adoption, the need for new intelligent approaches to support both buying and selling goods or services become inevitable. With these approaches, the discovery procedures of products, the issues negotiation, and the solution search must use a real communication challenge.
In this chapter, advances of distributed constraint reasoning approaches was operated. We used agent concept to describe the stakeholders to respect all their proprieties. We have proposed a new problem formulation and DCR protocol to deal with e-marketplace issues. More specifically, our attention was focused on constraint-based multi-agent approach offering a dynamic and privacy multi-lateral negotiation mechanism, namely, ABT-Trader.
Our approach has been implemented and tested using preliminary generated problems. The opening experimental results are promising and further investigations will be conducted. We have just explored the offers destined to satisfy the continuing expectations of market, however we are convinced that all those offers should be treated by our approach. Particularly, fuzzy and dynamic preferences, real case experimentations and learning process.

CHAPTER

6

A use case of cooperative networks: Cognitive IoT

Preamble

The Internet of Things (IoT) has the potential to impact on how we live and on how we work. Nowadays, the number of devices grows continuously, and the number of their constraints and the involved data evolve exponentially. As a result, the computational complexity of the underlying algorithms to these devices may become NP-Complete. In the last decade, as many approaches of realtime constraint handling have been proposed, Constraint Programming (CP) has been considered a standalone technology that can be used in several research areas.

In this chapter, we present a new extension named Cognitive IoT (CIoT) based on Constraint Programming frameworks. We introduce a dynamic and distributed Constraint Programming platform that covers explicitly several CIOT considerations (e.g. constraint acquisition, constraint reasoning, distributed communication protocols, etc.). We used our platform named JChoc DisSolver to model, implement and illustrate a real-world use case application based on IoT techniques. Therefore, Constraint Programming techniques have been proved to be a very elegant paradigm to handle CIoT applications. The experimental results are promising and meet our expectations concerning the handling of this category of applications.

Contents

6.1 Introduction

The IoT is a recent paradigm where the things are interconnected and equipped (example of amenities: operating systems, sensors, etc.) to enable easy interaction with a wide variety of devices. In fact, many objects nowadays provides different functionality to be combined in a single application and generate enormous amounts of data which have to be stored, processed and presented in an efficient and easily interpretable form. In the literature many surveys have been addressed the Internet of Things and its promising future. Debasis Bandyopadhyay et al [Bandyopadhyay and Sen, 2011] study the state-of-the-art of IoT and they present the key technological drivers, potential applications, challenges and future research areas in the domain of IoT, where the network discovery mechanisms, the networking, the communication, the architecture, and the software and algorithms are a key technologies involved in Internet of Things. Daniele Miorandi et al [Miorandi et al., 2012] present a survey of technologies, applications and research challenges for Internet-of-Things. That 'Internet of Things' Thing [Ashton et al., 2009], was the first article that talks about the Internet of Things. The Internet of Things (IoT), firstly coined by Kevin Ashton as the title of a presentation in 1999 [Bessiere et al., 2003], is a technological revolution that is bringing us into a new ubiquitous connectivity, computing, and communication era.

Two big issues rise with the IoT: the security and and the massive amount of data that will be produced by all these devices. In the constraint programming research field we found in one hand some algorithms (ABT 1ph and ABT 2ph [Brito et al., 2009], DisFC [Brito and Meseguer, 2003], etc.) proposed to keep confidentiality of constraints and/or values during the solving process and when exchanging messages, and in the other hand some research works (e.g. Opportunities and Challenges for Constraint Programming. [O'Sullivan, 2012]) proposed to represent the 'Big data' as an opportunity of Constraint Programming.

Is only connected enough? Current research on Internet of Things (IoT) mainly focuses on how to enable general objects to see [Wu et al., 2014], hear, and smell the physical world for themselves, and make them connected to share the observations. Beyond that, things should have the capability to learn, think, and understand both physical and social worlds by themselves. For that, new paradigm, named Cognitive Internet of Things (CIoT) to empower the current IoT with a 'brain' for high level intelligence, inspired by the effectiveness of human cognition, is welcome.

In this chapter we present both Constraint Programming and The Internet of Thing paradigms and some related works. After that, we provide the architecture of our new Constraint Programming based Platform for Cognitive Internet of Things.

Then, we illustrate the use of the platform on a real-world problem. Finally, we draw conclusions from the results obtained and we give guidelines for future work.

6.2 background

6.3 Contribution

6.3.1 The Architecture of Constraint based CIoT platform

Constraint based CIoT platform is an extension of the "Dynamic JChoc" Platform[Benelallam et al., 2015]. "Dynamic JChoc" Platform handles agents with local complex problems and allows a realistic use of agents on a real distributed and dynamic framework. In fact, many approaches have been implemented and tested on this platform [Zakarya Erraji and Bouyakhf., 2016]. So that, this platform is a distributed constraint multi-Thing system, proposed for IoT applications.

Constraint based CIoT platform is implemented in JAVA and provides classes that implement and inherit from JADE [Bellifemine et al., 2007] and Choco [Prud'homme et al., 2017] 4.0 platforms to define the behavior of each connected thing. Figure 32 represents the main Constraint based CIoT platform architectural elements. This platform has these main modules described as bellow:

— **Human demand, social behavior:** The Human can define the configuration of its problem as a constraint formalization using an xml/json file;

— **Service Provisioning:** Agent that manages services in the platform (e.g. resources as a service). Any thing in the platform will send a list of services that it offers to the "Service Provisioning" in the same time it can check the services offered by the other things in the platform;

— **Performance Evaluation:** Using Constraint Programming metrics. These metrics are usually the number of exchanged messages and the number of constraint checks. These information give the platform supervisor an overview of the problems solving performance;

— **Decision-Making:** Using Constraint Programming algorithms (e.g. MAC [Sabin and Freuder, 1994], LiveABT [Benamrane and Benelallam., 2017], AFC-ng [Ezzahir et al., 2009], etc.). The platform takes the problem and determines its type (i.e. satisfaction, optimization, centralized, distributed, etc.) then it chooses the best algorithm to solve the problem and informs the things to use this algorithm for the solving. In that time the thing will create a virtual agent to participate in the solving of the problem, this means, the thing can participate in the solving of multiple problems in the same time if possible depending of its resources (CPU, RAM, etc.);

— **Semantic Derivation and Knowledge Discovery:** Pattern discovery[De Raedt et al., 2008] and machine learning via CP[De Raedt et al., 2010];

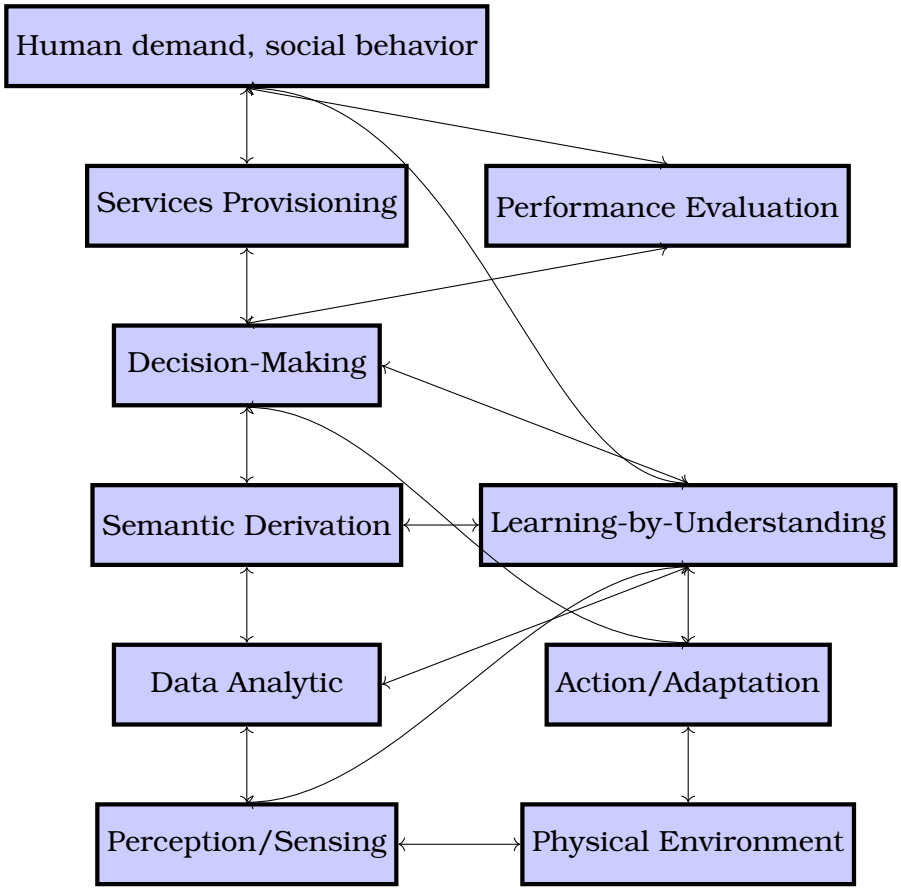


Figure 32 – Constraint based CIoT platform architecture

— **Data Analytic:** Constraint-based concept mining [Negrevergne and Guns, 2015; Aoga et al., 2016; Guns et al., 2017];

— **Perception/Sensing:** The values of Agent sensors are represented as unary constraint of equality;

— **Learning-by-Understanding:** Constraint acquisition [Daoudi et al., 2016], Nogood messages, etc. When facing a new problems, the thing can ask the human and by using of this answers, the thing can models the problems and proceed the the solving phase;

— **Action/Adaptation:** Dynamic CP framework algorithms (e.g. DynABT [Omomowo et al., 2008], LiveABT [Benamrane and Benelallam., 2017], minimal perturbation [Benelallam et al., 2016], etc.) ;

— **Physical Environment:** Sensors, ROS [Quigley et al., 2009], etc. The thing will be able to manage its own resources and/or share them with other things.

The figure 32 shows the Constraint based CIoT platform architecture inspired from the work of Qihui Wu et al [Wu et al., 2014]. In this architecture, there are

many consistent elements used to satisfy the user needs (i.e. solving its problem), generally, without any human intervention. In some cases, when the platform cannot find any solution and the user have not defined its preferences, it can ask him to get more information. The users define the whole problem and some information about its connected devices (connected things) then they save that in their xml files (i.e. input data). The platform uses the implemented algorithms to solve problems and to learn from its environment and to take decision about the devices (i.e. how to work, send data, receive data, ask for resources, etc.).
In brief, all these architecture elements are necessary to give the best working performance for the connected devices.

6.3.2 Simulation

There are many examples we can provide to explain how to use the Constraint based CIoT platform. As a matter of fact, IoT encompasses any connected device. Nowadays, we are surrounded with a world submerged of connected devices, for instance, connected watches, smart phones, connected and smart cars, connected fridges and many other useful connected things. In this section, we will present a direct application of our approach in the agriculture domain.

Leslie Lipper et al. [Lipper et al., 2014] say that Climate-smart agriculture (CSA) is an approach for transforming and reorienting agricultural systems to support food security under the new realities of climate change which disrupts food markets, posing population-wide risks to food supply. Threats can be reduced by increasing the adaptive capacity of farmers as well as increasing resilience and resource use efficiency in agricultural production systems.

In this example, we will use some IoT devices to control the soil and plant status and enhance the whole yield. Those devices communicate and act according to the data given by the sensors. This example is not for giving a complete solution to manage a farm. However, we explain from it how to use the platform; and we show also the benefit of Constraint Programming paradigm and its solving protocols.

We consider a field in the form shown in the figure 33. The field contains a water tank (WT) three sensors (i.e. S1, S2, S3) where each sensor measure the soil moisture. Whenever the value given by the sensor is low, the water demand is high. In addition, all these sensors are connected to a water tank that has a max value and a min value of the volume of water it contains. The sensors must have the same value in the solution. When the quantity of water in the tank is under the min value, the tank send automatically an order to the farm master to fill water.

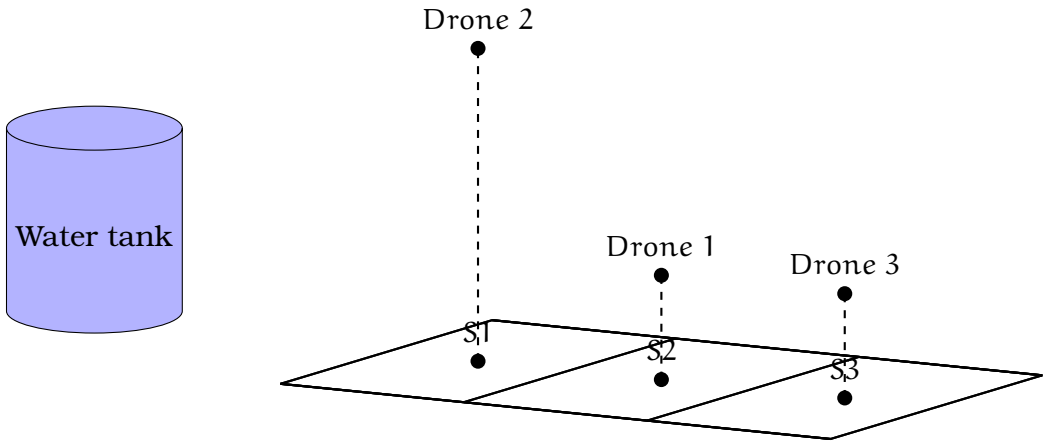


Figure 33 – Field with a water tank and three soil moisture sensors

This problem can be expressed as a Distributed Constraints Satisfaction Problem (DisCSP). The DisCSP is described by the five elements (X, D, C, A, ψ) where:

— X = $\{X_{S1d}, X_{S1c}, X_{S1f}, X_{S2d}, X_{S2c}, X_{S2f}, X_{S3d}, X_{S3c}, X_{S3f}, X_{WsumDemand}, X_{Wc}\}$ is the set of variables

— D = {

  — $D(X_{Sif}) = \{40, 41, \ldots, 50\}$;. the recommended moisture value for the soil in this example.

  — $D(X_{WsumDemand}) = \{1, 2, 3, \ldots, 100\}$;. the sum of sensor demand possible values.

  — $D(X_{Sid}) = \{1, 2, 3, \ldots, 100\}$; i = 1,2,3. the sensor demand possible values.

  — $D(X_{S1c}) = \{10\}$; the moisture value read by the sensor S1 at time : t_i.

  — $D(X_{S2c}) = \{23\}$; the moisture value read by the sensor S2 at time : t_i.

  — $D(X_{S3c}) = \{41\}$; the moisture value read by the sensor S3 at time : t_i.

  — $D(X_{Wc}) = \{50\}$; the water tank current water level.

  } is the set of domains

— C = {

  — C1 : All Equal(X_{Sif}) The moisture of the soil must be the same for all sensors.

  — C2 : $X_{Wc} - X_{WsumDemand} \geq 1$ where $X_{WsumDemand} = sum(X_{Sid})$ The water demand have to respect the quality of water in the water tank.

  } is the set of constraints

— A={S1, S2, S3, W} is the set of agent

— ψ: $X_{Si*} \rightarrow$ Si And $X_{Wf} \rightarrow$ W / (Si and W) ∈ A and $X_{Si*} \in$ X / is a function that maps each variable to its agent

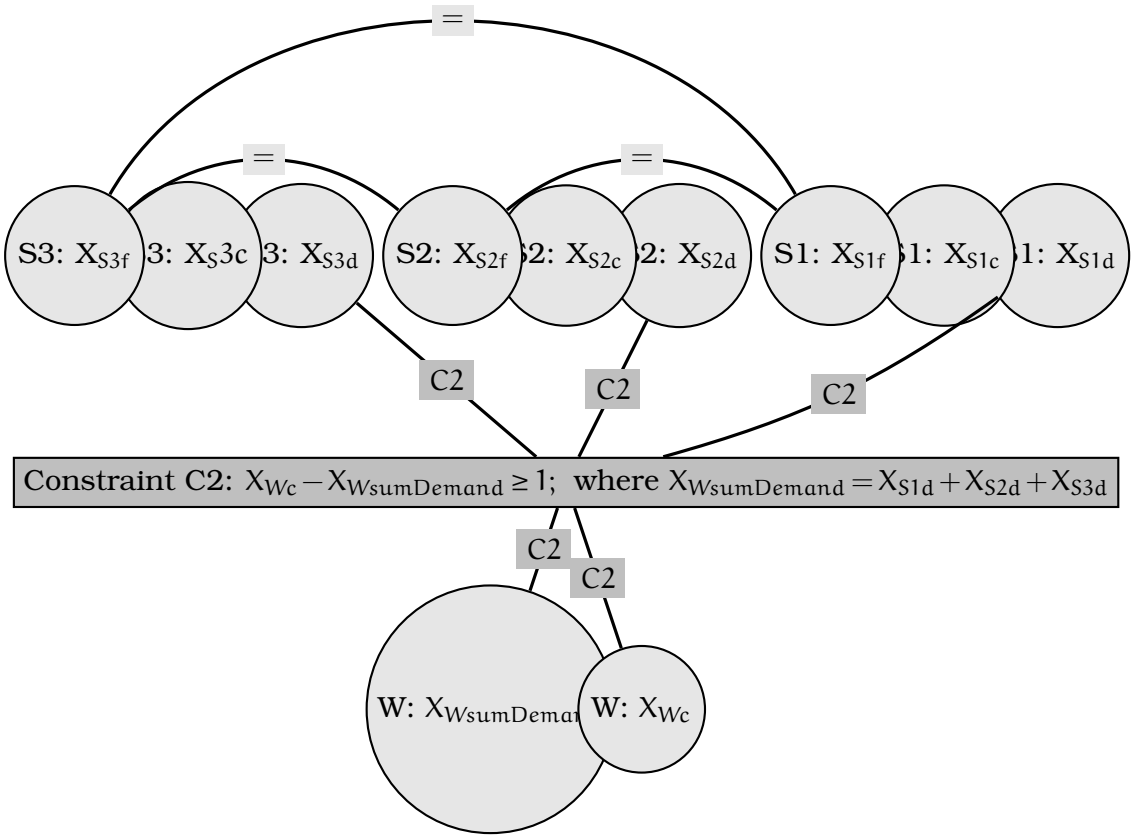


Figure 34 – The constraint network of the mission 1 problem

All initial data and values are saved in advance in an XML file presented in the listing 6.1 where we can find the different elements detailed. The given example is a DisCSP problem and we have chosen the ABT algorithm [Bessière et al., 2008] for solving the problem. ABT is a solving protocol that will lead each thing/agent to take the correct decision respecting the constraints of the whole problem. ABT uses a global priority order to manage agents. The id of each agent (XML file 6.1) can be considered as a global priority order.

Algorithm 3 The ABT algorithm

```
1: procedure ABT
2:     v_i ← Empty; t_i ← 0;
3:     end ← False;
4:     CheckAgentView;
5:     while ¬end do
6:         msg ← getMsg();
7:         switch msg.type do
8:             case ok?
9:                 ProcessInfo(msg)
10:            case ngd
11:                ResolveConflict(msg)
12:            case adl
13:                AddLink(msg)
14:            case stp
15:                end ← True
16:    end while
17: end procedure
18: procedure CheckAgentView
19:    if (¬isConsistent(v_i, AgentView)) then
20:        v_i ← ChooseValue() ;
21:        if (v_i ≠ Empty) then
22:            for each child ∈ Γ+(x_i) do
23:                sendMsg : ok?( myAssig⟨myVar, v_i, t_i⟩) to A_k
24:        end for
25:        else
26:            Backtrack();
27:        end if
28:    end if
29: end procedure
30: procedure ProcessInfo(msg)
31:    UpdateAgentView(msg.Assig);
32:    CheckAgentView();
33: end procedure
34: procedure UpdateAgentView(newAssig)
35:    if (newAssig.tag > AgentView[j].tag) then
36:        AgentView[j] ← newAssig;
37:    end if
38:    for each ng ∈ myNogoodStore do
39:        if ¬Compatible(lhs(ng),AgentView) then
40:            remove(ng,myNogoodStore);
41:        end if
42:    end for
43: end procedure
44: procedure ResolveConflict(msg)
45:    if ¬Compatible(msg.Nogood, AgentView) then
46:        CheckAddLink(msg.Nogood);
47:        add(msg.Nogood, myNogoodStore);
48:        CheckAgentView();
```

```
49:     else
50:         if Compatible(msg.Nogood, AgentView) then
51:             send : ok?(myAssig) to msg.Sender
52:         end if
53:     end if
54: end procedure
55: procedure BACKTRACK
56:     newNogood ← solve(myNogoodStore)
57:     if newNogood = Empty then
58:         end ← True;
59:         sendMsg : stp(system);
60:     else
61:         sendMsg : ngd(newNogood) to A_j
62:         UpdateAgentView x_j ← Empty
63:         CheckAgentView();
64:     end if
65: end procedure
66: procedure CHOOSEVALUE
67:     for each v ∈ myDom do
68:         if isConsistent(v, AgentView) then
69:             return(v);
70:         else
71:             store the best nogood for v;
72:         end if
73:     end for
74:     return Empty;
75: end procedure
76: procedure ADDLINK(msg)
77:     add(msg.Sender, Γ+(x_i));
78:     if v_i ≠ msg.Assig.Value then
79:         sendMsg : ok?(myAssig) to msg.Sender;
80:     end if
81: end procedure
82: procedure CHECKADDLINK(nogood)
83:     for each x_j ∈ lhs(nogood) \ Γ−(x_i) do
84:         add(x_j = v_j, AgentView);
85:         Γ−(x_i) ← Γ−(x_i) ∪ {x_j};
86:         sendMsg : adl(x_j = v_j) to A_j ;
87:     end for
88: end procedure
```

LISTING 6.1 – IoT xml file problem sample

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
  <presentation name="Custom" type="DisCSP" benchmark="RandomDisCSP" model="Simple" format="XDisCSP 1.0" />
  <agents nbAgents="4">
    <agent name="S1" id="1" description="sensor measure the soil moisture" />
    <agent name="S2" id="2" description="sensor measure the soil moisture" />
    <agent name="S3" id="3" description="sensor measure the soil moisture" />
    <agent name="WT" id="4" description="The water tank" />
  </agents>
  <domains nbDomains="6">
    <domain name="D1" type="Integer" nbValues="11">40..50</domain>
   <domain name="D2" type="Integer" nbValues="100">1..100</domain>
   <domain name="D3" type="Integer" nbValues="1">10</domain>
   <domain name="D4" type="Integer" nbValues="1">23</domain>
   <domain name="D5" type="Integer" nbValues="1">41</domain>
   <domain name="D6" type="Integer" nbValues="1">50</domain>
  </domains>
  <variables nbVariables="4">
    <variable agent="S1" name="X_S1d" id="1" domain="D2" description="Current level" />
   <variable agent="S1" name="X_S1c" id="2" domain="D3" description="Final level" />
   <variable agent="S1" name="X_S1f" id="3" domain="D1" description="Demand" />

   <variable agent="S2" name="X_S2d" id="1" domain="D2" description="Current level" />
   <variable agent="S2" name="X_S2c" id="2" domain="D4" description="Final level" />
   <variable agent="S2" name="X_S2f" id="3" domain="D1" description="Demand" />

   <variable agent="S3" name="X_S3d" id="1" domain="D2" description="Current level" />
   <variable agent="S3" name="X_S3c" id="2" domain="D5" description="Final level" />
   <variable agent="S3" name="X_S3f" id="3" domain="D1" description="Demand" />

   <variable agent="WT" name="X_Wc" id="1" domain="D2" description="Current level" />
   <variable agent="WT" name="X_WsumDemand" id="2" domain="D6" description="Water Demand" />
  </variables>
  <predicates nbPredicates="2">
    <predicate name="P1">
      <parameters>int X int Y int</parameters>
      <expression>
          <functional>eq(X, Y)</functional>
      </expression>
    </predicate>
    <predicate name="P2">
      <parameters>int X int Y int cte</parameters>
      <expression>
          <functional>geq(sub(X,Y),cte)</functional>
      </expression>
    </predicate>
  </predicates>
  <constraints nbConstraints="2">
    <constraint name="C1" arity="2" scope="X_S1f X_S2f" reference="P1"/>
   <constraint name="C2" arity="2" scope="X_S1f X_S3f" reference="P1"/>
    <constraint name="C3" arity="2" scope="X_S2f X_S3f" reference="P1"/>


    <constraint name="C4" arity="3" scope="X_Wc X_WsumDemand 1" reference="P2"/>
  </constraints>
  <relations nbRelations="0">
  </relations>
  <GuiPresentation type="DisCSP" benchmark="Custom" name="instance2" nbAgents="4"/>
</instance>
```

The exchanged messages and the final solution are showed in the figure 35. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Agents do not have to wait for decisions of others but they are subject to a total (priority) order. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors. When an agent assigns a value to its variable, the selected value is sent to lower priority

neighbors. When no value is possible for a variable, the inconsistency is reported to higher agents in the form of a Nogood. ABT computes a solution (or detects that no solution exists) in a finite time. To be complete, requires a total ordering on agents. The total ordering on agents is static.

In ABT, a generic agent, say myAgent, stores in its agentView the most up to date values that it believes are assigned to its higher priority neighbors (connected to self by incoming links). myAgent stores in its nogoodStore the nogoods justifying values removal. Agents exchange the following types of messages (where $myAgent$ is the sender):

ok?: $myAgent$ informs a lower priority neighbor about its assignment.

ngd: $myAgent$ informs a higher priority neighbor of a new nogood.

adl: $myAgent$ requests a higher priority agent to set up a link.

stp: The problem is insolvable because an empty nogood has been generated.

In the main procedure ABT(), each agent selects a value and informs other agents . Then, a loop receives and processes each message. The $checkAgentView()$ procedure checks if the current value (v_i) is consistent with the AgentView. If v_i is inconsistent with assignments of higher priority neighbors, $myAgent$ tries to select a consistent value. In this process, some values from $Domain(v_i)$ may appear as inconsistent. Thus, nogoods justifying their removal are added to the nogoodStore of $myAgent$. When two nogoods are possible for the same value, $myAgent$ selects the best with the *Highest Possible Lowest Variable* heuristic. If a consistent value exist, it is returned and then assigned to x_i. Then, $myAgent$ notifies its new assignment to all agents in $\Gamma^+(x_i)$ through ok? messages. Otherwise, $myAgent$ has to backtrack.

Whenever it receives an ok? message, $myAgent$ processes it by calling procedure ProcessInfo(). The agentView of $myAgent$ is updated only if the received message contains an assignment more up to date than that already stored for the sender and all nogoods becomes non compatible with the agentView of $myAgent$ are removed. Then, a consistent value for $myAgent$ is searched after the change in the agentView.

When every value of $myAgent$ is forbidden by the nogoodStore, procedure Backtrack() is called. In procedure Backtrack(), $myAgent$ resolves its nogoods, deriving a ($newNogood$). If the $newNogood$ is empty, the problem has no solution. $myAgent$ sends the stp message to the agent system and terminates the execution. Otherwise, the new nogood is sent in a ngd message to the agent, say A_j, owning the variable appearing in its rhs . Then, the assignment of x_j is deleted from the agentView . Finally, a new consistent value is selected.

Whenever $myAgent$ receives a ngd message, procedure ResolveConflict() is called. The nogood included in the ngdmsg message is accepted only if its lhs is compatible with assignments on the agentView of $myAgent$. Therefore, $myAgent$ calls procedure CheckAddLink(nogood). In procedure the CheckAddLink(nogood), the assignments in the received nogood for variables not directly linked with $myAgent$ are taken to update the agentView and a request for a new link is sent to the agent owning this

variable. Next, the nogood is stored, acting as justification for removing the value on its rhs. Then, a new consistent value for myAgent is searched if the current value was removed by the received nogood. If the nogood is not accepted, it is obsolete. Then, if the value of x_i was correct in the received nogood, myAgent re-sends its assignment to sender by an ok? message.

When a link request is received, myAgent calls procedure AddLink(msg). Then, the sender is included in
Gamma$^+(x_i)$. Afterwards, myAgent sends its assignment through an ok? message to the sender if its value is different than that included in the received msg.

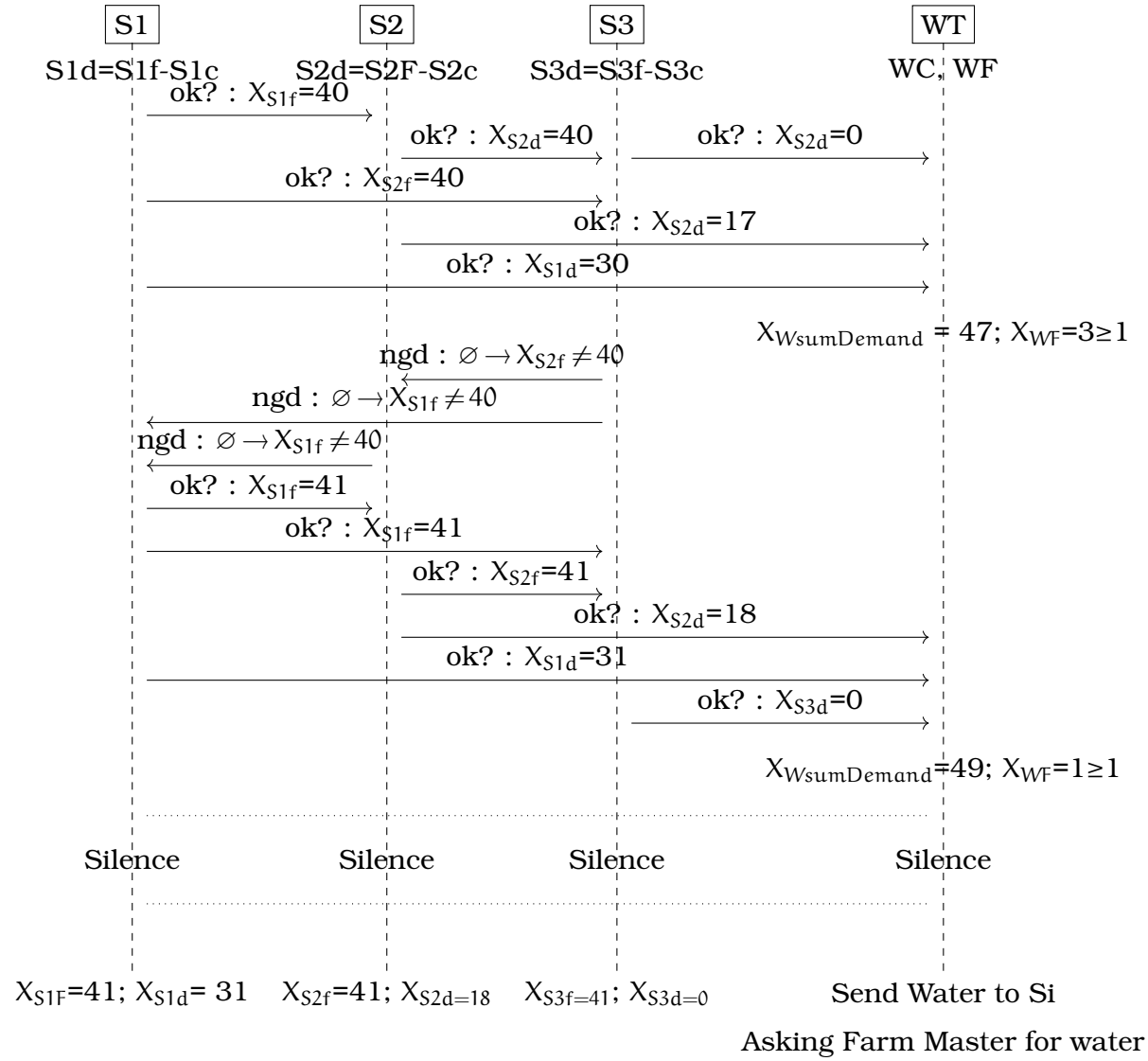


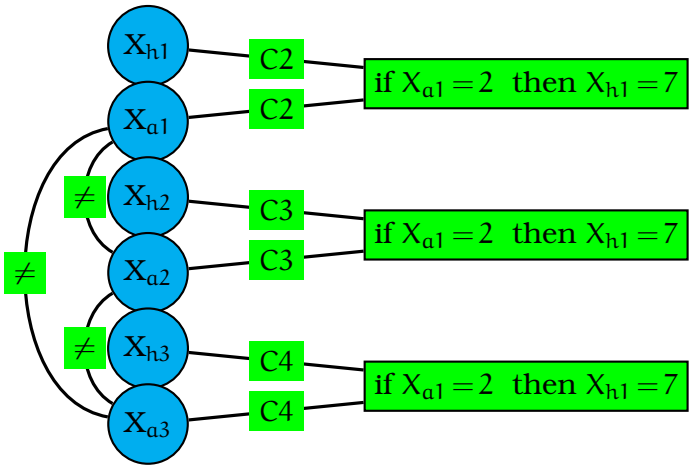
Figure 35 – Solving process with ABT algorithm

A second mission can be defined to survey the field with 3 drones. Each drone is considered as an autonomous agent that have to control a part of the field. As shown in the figure 33 the field contains three areas 1, 2 and 3. Let us consider a scenario where each drone controls two variables: The altitude position X_{hi} and number of the area to survey represented by X_{ai}. This scenario can be represented by the DisCSP described by the five elements (X, D, C, A, ψ) where:

— X = $\{X_{h1}, X_{h2}, X_{h3}, X_{h3}, X_{a1}, X_{a2}, X_{a3}\}$ is the set of variables

— D = {
  — $D(X_{h1}) = \{1,\ldots,10\}$;
  — $D(X_{h2}) = \{1,\ldots,5\}$;
  — $D(X_{h3}) = \{1,\ldots,10\}$;
  — $D(X_{a1}) = \{1,\ldots,3\}$;
  — $D(X_{a2}) = \{1,2\}$;
  — $D(X_{a3}) = \{1,\ldots,3\}$;
  } is the set of domains
— C = {
  — C1 : AllDiff(X_{ai}) / i = 1,2,3
  — C2 : if $X_{a1} = 2$ then $X_{h1} = 7$
  — C3 : if $X_{a2} = 2$ then $X_{h2} = 7$
  — C4 : if $X_{a3} = 2$ then $X_{h3} = 7$
  } is the set of constraints
— A={D1, D2, D3} is the set of agent (i.e. drones)
— ψ: $X_{ai} \rightarrow$ Di and $X_{di} \rightarrow$ Di / Di $\in$ A and $(X_{Si*}, X_{Si*}) \in$ X / is a function that maps each variable to its agent

Figure 36 – The constraint network of the mission 2 problem



In this second example we assume that the drone D2 cannot reach the area number 3 (definition domain) in the field because of the battery autonomy. And also, we have added two constraints, the first one is for allowing each drone to control one area and the second one is related to the areas 2 that must be surveyed on an altitude equals to 7.

The problem can be defined in the XML like in the listing 6.1, and, we can use the ABT algorithm to solve it. A solution found by the ABT algorithm is below:

— Drone D1 : $X_{h1} = 7$ and $X_{a1} = 2$

— Drone D2 : $X_{h2} = 1$ and $X_{a2} = 1$

— Drone D3 : $X_{h3} = 1$ and $X_{a3} = 3$

As interpretation, the drone D1 surveys the area 2 on an altitude equals to 7, the drone D2 surveys the area 1 on an altitude equals to 1, and finally the drone D3 surveys the area 3 on an altitude equals to 1.

6.4 Conclusion

In this chapter, we have used the Constraint Programming paradigm to handle Cognitive Internet of Things problems. The fusion of these two concepts has been done using a new platform namely Constraint based "CIoT" platform. We have presented through a simple example how can the platform be useful to solve many real-world problems without any human intervention. The Things communicate using the platform and collaborate to solve the problem (described by the human or learned by the things themselves) by the use of the constraint programming algorithms.

The next chapter is focusing on enhancing the solving performances in DisCSP by proposing a new Algorithm.

CHAPTER

7 Improved Forward Checking Tree Algorithm for Distributed Constraint Satisfaction Problems

Preamble

In this chapter, we propose a new algorithm based on the Asynchronous Forward Checking Tree (AFC-tree) algorithm for the DisCSP problems named Improved Forward Checking Tree (IFC-tree). The experimental results show the efficiency of this algorithm by improving different metrics of the solving performances.

Contents

7.1 Introduction

Any binary DisCSP problem can be presented as a graph of constraints where the vertices are the variables and the edges are the constraints. The AFC-tree algorithm uses a pseudo-tree arrangement of the constraint graph where the agents are ordered in a pseudo-tree such that agents in different branches of the three are not connected by any constraint.
The AFC-tree algorithm is one of the most powerful algorithm proposed to solve the

DisCSPs. By using the peudo-tree arrangement of the constraint graph, the AFC-tree runs simultaneous search processes in disjoint problem subtrees and exploits the parallelism inherent in the problem. The root agent starts the search then send its value each connected agent (linked by a constraint in the peudo-tree). Each agent will try to find a compatible value and sends this value to its connected children. If this agent cannot find any compatible value it will build a nogood message to ask the responsible agent to change its value. In the pseudo-tree, when the recipient of the value is a leaf it sends a message to its parent to inform it that it has a compatible value; then the parents will then wait for the other children to confirm the compatibility of their values. After that, this agent will do the same and inform its parent in the same way. In this time if one leaf/agent in the pseudo tree, cannot find a value, it sends a nogood message to the responsible agent. All the messages arrived to this agent from the agents of this sub pseudo-tree are now obsolete and cannot be used.
The produced obsolete messages in the build accept step cause an unnecessary computation effort (and also a communication load). This computation effort becomes very important when the graph density of the problem is high. By reducing the number of these obsoleted messages, in a real-world application (e.g Multi-robot decision making using constraint programming [Erraji et al., 2016]), we can save more battery capacity for robots/mobiles and also save the communication cost.
In order to exploit this fact and maximize the performances of the solving process we have proposed a new modified version named Improved Forward Checking Tree Algorithm for DisCSPs.

7.2 Related works

Asynchronous backtracking [Yokoo et al., 1992; Bessière et al., 2005] is the pioneer asynchronous complete algorithm that has been proposed for solving DisCSPs. It's an autonomous algorithm integrated in each agent participating in the solving process. Synchronous Backtracking (SBT) [Zivan and Meisels, 2003] is a synchronous search algorithm, where only the agent holding the Current Partial Assignment (CPA) performs an assignment or backtracks. ABT-hyb is proposed by Brito and Mesenguer [Brito and Meseguer, 2004] to add synchronization points in ABT on purpose to avoid some redundancy in messages after performing backtracks. In fact, this optimization make ABT-hyb more efficient than ABT in terms of computation effort and communication load. In [Meisels and Zivan, 2007], Meisels and Zivan proposed the Asynchronous Forward Checking (AFC) where agents perform forward checking asynchronously but backtrack synchronously. The paper [Nguyen et al., 2004] presents a distributed version of the centralized Back Jumping algorithm, called the Dynamic Distributed Back Jumping (DDBJ) algorithm. The advantage is twofold: DDBJ inherits the strength of synchronous algorithms that enables it to easily combine with a powerful dynamic ordering of variables and values, while maintining some level of autonomy for the agents. In the first

work published in [Ezzahir et al., 2009], Ezzahir Redouane et al. proposed two new asynchronous algorithms for solving DisCSPs. The first algorithm, AFC-ng, is a nogood-based version of Asynchronous Forward Checking. The second algorithm, Asynchronous Inter-Level Forward-Checking (AILFC), is based on the AFC-ng algorithm and is performed on a pseudo-tree ordering of the constraint graph. And later, in [Wahbi et al., 2013], the authors have proposed a new version of AFC-ng which is a nogood-based version of Asynchronous Forward Checking, performed on a pseudo-tree ordering of the constraint graph. Note that, besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks going from different agents to different destinations. AFC-tree runs simultaneous search processes in disjoint problem subtrees and exploits the parallelism inherent in the problem.

7.3 Description of the AFC-tree algorithm

The AFC-Tree algorithm uses the pseudo-tree ordering [Freuder and Quinn, 1985] and performs multiple AFC-ng to solve the DisCSP. Each Agent in the AFC-Tree algorithm have a priority number according to the pseudo-tree ordering. The most prioritized agent is the root of the pseudo-tree, so that the priority decreases when moving from root to leaf.

In the pseudo-tree, the children of an agent are its descendants connected to it through tree edges. The descendants of an agent are the agents belonging to the subtree rooted at this agent and the linked descendants are only the descendants constrained with it in the subtree. The ancestors of an agent are the agents forming the path from this agent to the root agent. A branch in the pseudo-tree is a path from the root agent to a leaf agent (a leaf agent is an agents with no children). The parent of an agent is the agent ancestor connected to it through a tree edge.

The search of the solution begins when the root agent generates a CPA and assigns its value. Then, it sends it to its connected descendants in the tree. After that, every child which has received the CPA, performs AFC-ng on the sub-problem restricted to its ancestors. In the AFC-Tree, the asynchronism is enhanced exploiting the potential optimization of using parallel exploration in solving DisCSPs.

The detection of the global solution in the AFC-tree is as follows. Whenever a leaf agent assigns its value successfully, it sends a message named accept to its parent. This message contains the CPA that was received from the parent incremented by the value-assignment of the leaf node. When a non-leaf agent A_i received accept messages from all its children that are all compatible with each other, all compatible with A_i's agentView and with A_i's value, the A_i makes a conjunction of all received accept as well as its value in a single message accept. The solution is detected when the A_i is the root agent; then it reports a solution and proceed to stop solving

process, otherwise, A_i sends the accept message to its parent.

An assignment is a tuple containing the variable, the chosen value and a tag value to indicated the most up to date value of the variable. The most up to date value is the one with the greatest tag. The CPA or the Current Partial Assignment is an ordered list of assignments. Two CPAs are Compatible if they contain the same values for any variable value. A Timestamp for a CPA is an ordered list of tags of variables. The timestamp is used to compare two CPA, where the strongest CPA is the one associated with the lexicographically greater timestamp. The AgentView is the structure where the agent stores the most up to date assignments received from higher priority agents in the pseudo-tree.

The AFC-tree is based on AFC-ng [Wahbi et al., 2013]. We present in the Algorithm 4 only the procedures that are new or different from those of AFC-ng. In *InitAgentView*, the *AgentView* of A_i is initialized to the set *ancestors*(A_i) and t_j is set to 0 for each *agent* (x_j) in *ancestors*(A_i) (line 20). The new data structure storing the received accept messages is initialized to the empty set (line 21). The CPA is a data structure where the agents assign their variables one by one.In *SendCPA(CPA)*, instead of sending copies of the CPA to all neighbors not yet instantiated on it, A_i sends copies of the CPA to its linked descendants (*linkedDescendants*(A_i), line 25). When the set *linkedDescendants*(A_i) is empty (i.e., A_i has no descendant, it's a leaf), A_i calls the procedure *SolutionDetection* to build and send an accept message. In *CheckAssign(sender)*, A_i assigns its value if the received CPA comes from its parent (line 32) (i.e., if the sender is the parent of A_i).

In *ProcessAccept(msg)*, when A_i receives an accept message from its child for the first time, or the CPA contained in the received accept message is stronger than that one received previously (The strongest one is the one associated with the lexicographically greater timestamp), A_i stores the content of this message (lines 36-37) and calls the *SolutionDetection* procedure (line 38). In *SolutionDetection*, if A_i is a leaf (line 42), it sends an accept message to its parent. The accept message sent by A_i contains its *AgentView* incremented by its own assignment (lines 42-43). If A_i is not a leaf, it calls function *BuildAccept* to build an accept partial solution PA (line 45). If the returned partial solution PA is not empty and A_i is the root, PA is a solution of the problem. Then, A_i reports a solution and sets the end flag to true to stop the search (line 47). Otherwise, A_i sends an accept message containing PA to its parent (line 48).
In *BuildAccept*, if an accept partial solution is reached. A_i generates a partial solution PA incrementing its *AgentView* with its assignment (line 54). Next, A_i loops over the set of accept messages received from its children. If at least one child has never sent an accept message or the accept message is incompatible with PA, then the partial solution has not yet been reached and the function returns empty (line 56). Otherwise, the partial solution PA is incremented by the accept message of each child (from the A_i children) (line 57). Finally, the accept partial solution is returned (line 60).

Algorithm 4 The AFC-tree algorithm

```
1: procedure AFC-TREE()
2:     InitAgentView();
3:     end ← false; AgentView.Consistent ← true;
4:     if A_i = root then Assign();
5:     end if
6:     while (¬end) do
7:         msg ← getMsg();
8:         switch (msg.type) do
9:             case cpa
10:                ProcessCPA(msg);
11:            case ngd
12:                ProcessNogood(msg);
13:            case stp
14:                end ← true;
15:            case accept
16:                ProcessAccept(msg);
17:    end while
18: end procedure
19: procedure INITAGENTVIEW()
20:    foreach ( A_j ∈ ancestors(A_i)) do AgentView[j] ← {(x_j,empty,0)};
21:    foreach ( child ∈ children(A_i)) do Accept[child] ← ∅;
22: end procedure
23: procedure SENDCPA(CPA)
24:    if children((A_i) ≠ ∅) then
25:        for each (desc ∈ linkedDescendants(A_i)) do
26:            sendMsg : cpa <CPA> to desc
27:        end for
28:    else SolutionDetection();
29:    end if
30: end procedure
31: procedure CHECKASSIGN(sender)
32:    if (parent(A_i) = sender) then Assign();
33:    end if
34: end procedure
35: procedure PROCESSACCEPT(msg)
36:    if (msg.CPA stronger than Accept[msg.Sender]) then
37:        Accept[msg.Sender] ← msg.CPA;
38:        SolutionDetection();
39:    end if
40: end procedure
```

```
41: procedure SolutionDetection()
42:     if (children(A_i) = ∅) then
43:         sendMsg : accept<AgentView ∪ {x_i, v_i, t_i)}> to parent(A_i);
44:     else
45:         PA ← BuildAccept();
46:         if (PA ≠ ∅) then
47:             if (A_i = root) then Report SOLUTION; end ← true;
48:             else sendMsg : accept <PA> to parent(A_i);
49:             end if
50:         end if
51:     end if
52: end procedure
53: function BuildAccept()
54:     PA ← AgentView ∪ {x_i, v_i, t_i)};
55:     for each (child ∈ children(x_i)) do
56:         if (Accept[child] = ∅ ∨ ¬Compatible(PA,Accept[child])) then return ∅;
57:         else PA ← PA ∪ Accept[child];
58:         end if
59:     end for
60:     return PA;
61: end function
```

All the messages presented in the AFC-tree algorithm have to be processed. In the first hand, the agent receiver of the message processes the data using its CPU and does not know if the message stills valid or not. In the second hand, both agents (i.e. the receiver and the sender of the message) have to pay the cost to ensure the communication (i.e. the number of exchanged messages between the agents). In the section "Our Contribution", we propose a new algorithm to solve DisCSP ; and in the Experimental Evaluation Section, we show the obtained result by comparing the AFC-tree algorithm and the proposed algorithm.

7.4 Our contribution

The AFC-tree algorithm is one of the most powerful algorithms for solving the DisCSPs. Although, there are too many obsolete messages produced in the build accept step. When an agent/leaf cannot find a compatible value, it sends a nogood message to the responsible agent; and in this case all build accept messages received by the responsible agent for the nogood are now obsoletes. This communication load (and also a computation effort) becomes very important when the graph density of the problem is high and the pseudo-tree arrangement contains a bigger number of branches and levels (example : figure 37).

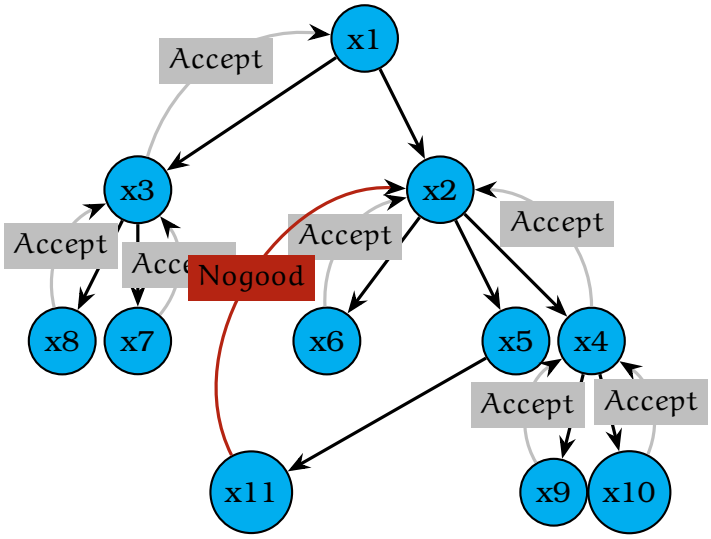


Figure 37 – Example of the obsolete messages in the AFC-tree

To avoid this loss (i.e. The obsolet messages) and save the computation effort, we cannot just add an add-on algorithm to detect the quiescence [Matocha and Camp, 1998; Silaghi et al., 2000] but we have to modify the whole algorithm (because, the build accept step is part of the algorithm). In this contribution, the root agent in the IFC-tree will communicate with all the leaves in the pseudo tree to receive their status and then to detect the solution. The figure 38 shows how our algorithm will treat the same situation illustrated in the figure 37.
In the algorithm 5 we present the pseudo code of the IFC-tree algorithm. The algorithm begins with an initialization of the list and the leaves awaiting confirmation of their status, a timeout that will be calculated based on the pseudo-tree and the problem's data (line 1). The root begins the solution search (line 4). When an agent receives a CPA message, it proceeds to process the message (line 13); and if the agent is a leaf in the pseudo-tree it informs the root that is searching for a solution (lines 10 and 11). When the root receives a searching message, it removes the sender from the leaves they found a solution (line 19). And when it receives waiting message, it adds the sender leaf, the leaves those found a solution (line 21), then it checks the list. If it is equal to initial leaves list, it stops the search process and sends a stop message all the agents in the tree. All the agents begin with the initialization of their agentviews to be prepared to receive the CPA messages (lines 2 and 28). After receiving a CPA message and if it can choose a new value compatible with the agentview, it sends a new CPA message to all its connected descendants (line 33). But if this agent is a leaf, it waits for a news CPA (with a timeout) after that it sends a waiting message to the root agent (line 48).

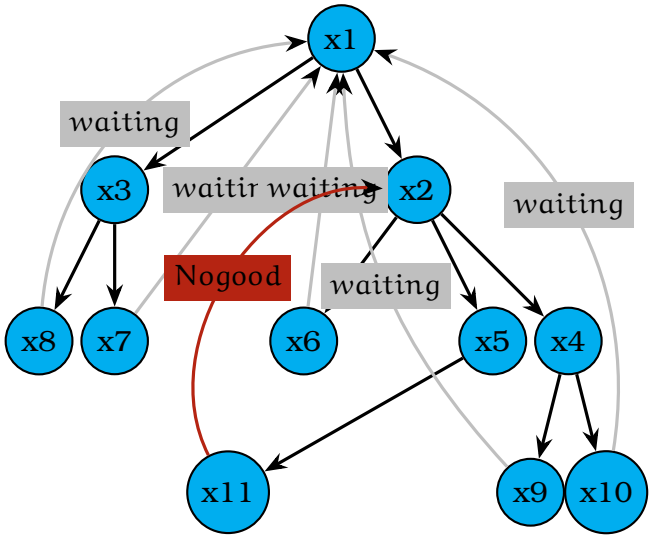


Figure 38 – Example of the use of IFC-tree to solve the problem presented in Figure 37

Algorithm 5 The IFC-tree algorithm

```
1: procedure IFC-TREE()
2:     InitAgentView(); leaves_solution ← empty, time_out ← value, time_waiting ← 0;
3:     end ← false; AgentView.Consistent ← true;
4:     if A_i = root then Assign();
5:     end if
6:     while (¬end) do                          ▷ Process any message at its arrival
7:         msg ← getMsg();
8:         switch (msg.type) do
9:             case cpa
10:                if (time_waiting ≥ time_out) and (children(A_i) = ∅) then
11:                    sendMsg : searching < A_i.id > to root
12:                end if
13:                ProcessCPA(msg);
14:            case ngd
15:                ProcessNogood(msg);
16:            case stp
17:                end ← true;
18:            case searching          ▷ only the root agent will receive this message
19:                remove msg.sender from leaves_solution list;
20:            case waiting            ▷ only the root agent will receive this message
21:                add msg.sender from leaves_solution list;
22:                if leaves_solution list constains all leaves agents then
23:                    sendMsg : stp to all agents in the pseudo-tree
24:                end if
25:    end while
26: end procedure
27: procedure INITAGENTVIEW()
28:    foreach ( A_j ∈ ancestors(A_i)) do AgentView[j] ← {(x_j,empty,0)};
29: end procedure
30: procedure SENDCPA(CPA)▷ used to forward the CPA to each linked descendant
31:    if children((A_i) ≠ ∅) then
32:        for each (desc ∈ linkedDescendants(A_i)) do
33:            sendMsg : cpa <CPA> to desc
34:        end for
35:    else SolutionDetection();
36:    end if
37: end procedure
38: procedure CHECKASSIGN(sender)    ▷ Used to assign the agent value in the CPA
39:    if (parent(A_i) = sender) then Assign();
40:    end if
41: end procedure
```

```
42: procedure SolutionDetection()      ▷ Only a leaf agent can use this procedure
43:     if (children(A_i) = ∅) then
44:         do
45:             Wait_for_time_out(time_out);
46:         while time_waiting ≤ time_out or no msg is received
47:         if time_waiting ≥ time_out then
48:             sendMsg : waiting < A_i.id > to root;
49:         end if
50:     end if
51: end procedure
```

The AFC-tree and the IFC-tree algorithms stop when finding the solution of the problem. Each agent has a limited number of values in its domain. In both IFC-tree and AFC-tree, agents forward only consistent partial assignments (CPAs). Hence, leaf agents receive only consistent CPAs. Thus, for AFC-tree, leaf agents only send to their parents accept messages holding consistent assignments, and for IFC-tree, leaf agents only send waiting message when they have consistent assignments. As a result, for the AFC-tree, the accept message A_i sends to its own parent contains a consistent partial solution and the root reports a solution and stops the search only when it can build itself such an accept message. For the IFC-tree, the root reports a solution and stops the search only when all leaf agents hold consistent partial solutions. We deduce that both the AFC-tree and IFC-tree agents cannot fall into an infinite loop. which ensures the termination of AFC-tree and IFC-tree algorithms.

7.5 Experimental Evaluation

To evaluate our contribution we have used the JChoc DisSolver [Benelallam et al., 2015] platform. JChoc DisSolver is a distributed constraint programming platform, supporting the dynamic aspect for DisCSPs. JChoc is an easy-to-use platform, based on an elegant Multi-agent communication sub-platform (e.i JADE [Bellifemine et al., 2007]). It handles agents with local complex problems and allows a realistic use of agents on a real distributed and dynamic framework.

We have tested AFC-ng and AFC-tree to confirm the result showed in the original paper. Then, we have compared AFC-tree with the new version IFC-tree on two types of problems : Randoms and n-queen problems; the first one deals with the random problems on which we can randomly test both algorithms on different densities (i.e. the network connectivity : the ratio of existing binary constraints,) and different constraint tightness. In all these problems, each agent controls exactly one variable. We have chosen two values of density.

For these experimetal évaluation, the random probems are défined as follows

— P1 = 20% : case of a not highly connected constraint network

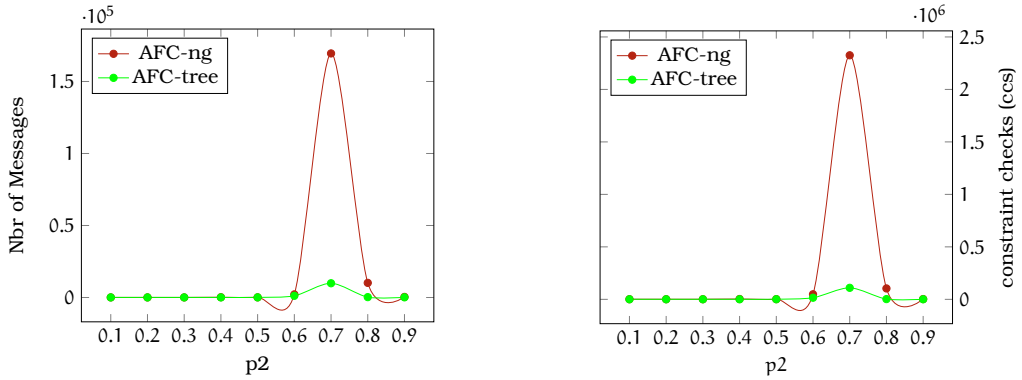


Figure 39 – Random problems: AFC-ng vs AFC-tree in p1 = 20%

— P1 = 70% : case of a highly connected constraint network
— For each P1; we vary the tightness P2 from 10
— 20 agents per instance
— Each variable has 10 values

For Each value of density, we have varied the tightness from 10% to 90% by steps of 10%. In all instances, we have 20 agents per problem, each agent controls one variable and each variable has 10 values in its domains. In the IFC-tree algorithm, we use a time_out value, which is calculated by the equation 7.1.

$$time\_out = m(t_{msg} + c * d * t_{cst}) = 20(0.01 + 1 * 10 * 0.018) = 3.8s \quad (7.1)$$

Where, in the equation 7.1, m is the max depth in the peudo-tree representation for all instances of the problem, t_{msg} is the duration to deliver a message in our platform, c is the number of constraints links a leaf and its parent, d is the size of its domain and t_{cst} is the max duration that Choco Solver [Prud'homme et al., 2017] takes to test one constraint from these problems. The result are shown in figures 39,40,41,42.In these figures, we confirmed the results presented in [Wahbi et al., 2013]. The results show also that IFC-tree is better or equal to the AFC-tree algorithm.

The second type of problems is the n-queen problems. We have also compared AFC-tree and IFC-tree on the n-queen problem to verify the behaviors of both algorithms when the network is fully connected, and the result are shown in the figure 43. The results confirm the efficiency of our contribution.

By observing all the results obtained, we must say that, like the AFC-tree algorithm, the IFC-tree algorithm reaches the potential speed-up expected from the parallel exploration. Moreover, the efficiency is better in terms of computational effort (number of constraint checks) and communication load (number of exchanged messages). IFC-tree is always better than or equivalent to AFC-Tree.

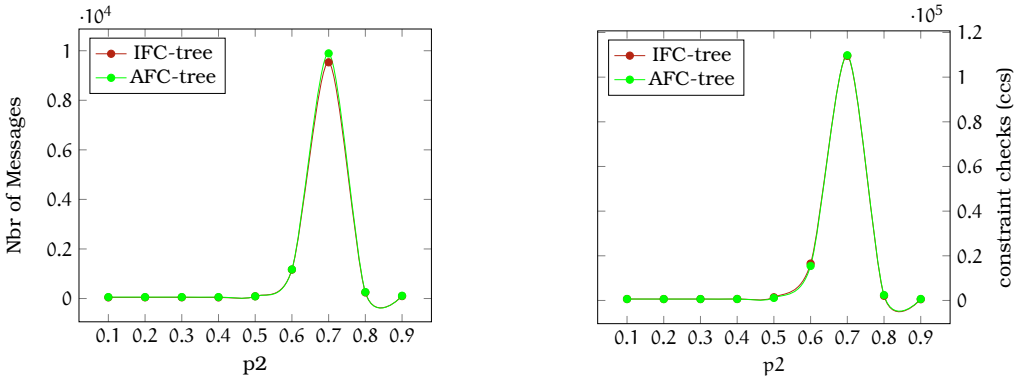


Figure 40 – Random problems: AFC-tree vs IFC-tree in P1=20%

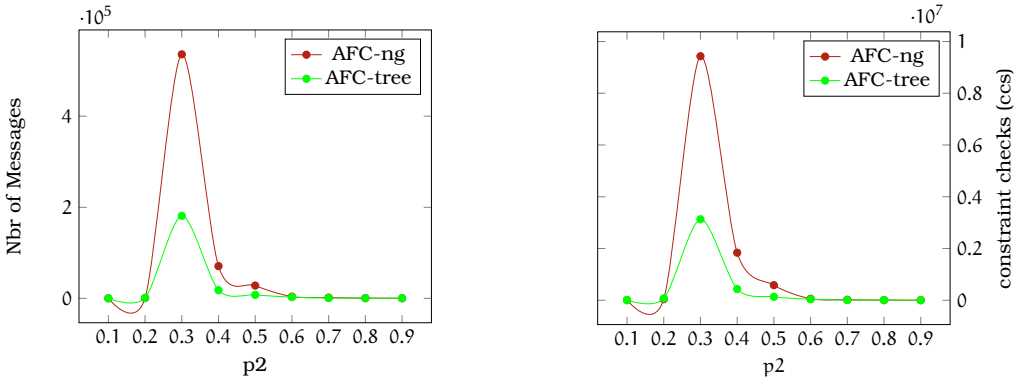


Figure 41 – Random problems: AFC-ng vs AFC-tree p1=70%

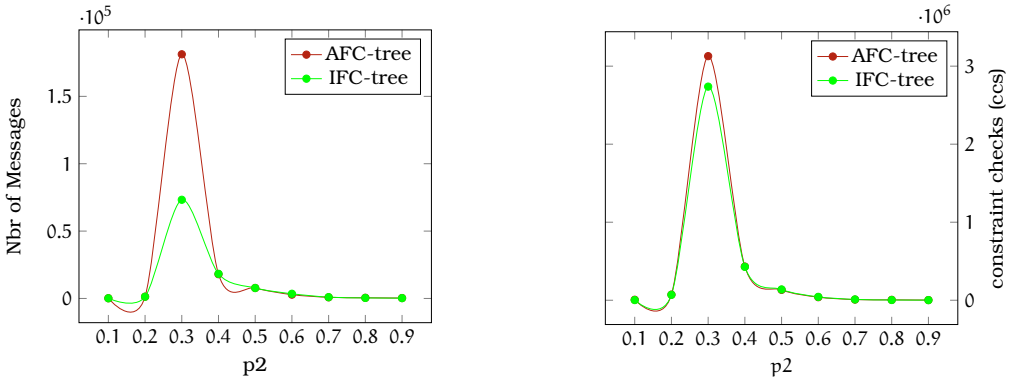


Figure 42 – Random problems: AFC-tree vs IFC-tree p1=70%

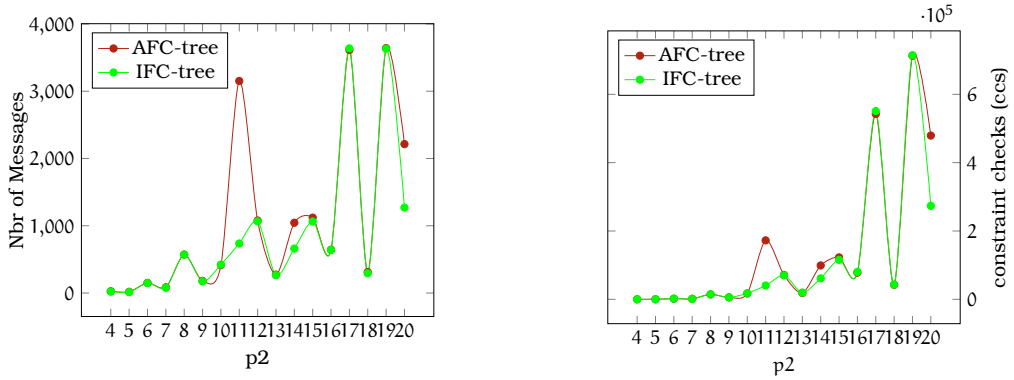


Figure 43 – N-queen problem: AFC-tree vs IFC-tree

7.6 Conclusion

In this chapter, we have proposed a new algorithm named Improved Forward Checking Tree Algorithm for DisCSPs. First, we have compared AFC-tree algorithm and AFC-ng using JChoc DisSolver platform and then we have compared the IFC-tree with the AFC-tree algorithm. The experimental results obtained allow us to say that IFC-tree is the most efficient algorithm for solving DisCSPs. The IFC-tree, like the AFC tree, performs simultaneous AFC-ng processes on each branch of the pseudo-tree to exploit the parallelism inherent in the problem. In addition, IFC-tree saves the computation effort by avoiding the unnecessary sending of obsolete messages to confirm the compatibility of the chosen values (from leaf to root). In the IFC-tree, the root can detect directly if all the leaves have finished the search process.

We will focus our future activities in this area on the use of IFC-tree algorithm in cognitive IoT [Erraji et al., 2017] by proposing a new version that supports both dynamic and optimization aspects.

CHAPTER

8 Conclusions and perspectives

We have addressed in this thesis the Distributed Constraint Satisfaction Problem (DisCSP) framework. We have proposed several applications and algorithms; and have provided a complete evaluation of the efficiency of our contributions against the existing approaches in literature. Our experimental results show that these contributions improve the current state of the art.

Once we defined the context, the centralized constraint satisfaction problem framework (CSP) and presented some examples of problems, we mentioned some existing algorithms used for solving centralized CSPs. Next, we formally defined the distributed constraint satisfaction problem framework. We illustrated how some instances of real-world applications in multi-agent coordination can be formulated and solved in DisCSP without delivering their personal information to a centralized agent.

Various contributions have been proposed in this thesis. Our first contribution is the JChoc DisSolver platform for distributed constraint reasoning. In many real-world distributed problems, each agent has its own part of the problem and must communicate with the other agents to find a simple or optimized solution for the global problem. In Constraint Programming, the agents use the same protocol (i.e algorithm) to communicated and to coordinate to solve the problem. To implement and test the efficiency of these protocols, Constraint Programming researchers can now easily use JChoc DisSolver where we have provided many tools for managing the messages, the data structures, etc. JChoc DisSolver is a JADE-based platform which implements all those basic FIPA (Foundation for Intelligent Physical Agents) specifications that provide the normative framework within which FIPA agents can exist, operate and communicate.

After proposing the JChoc DisSolver platform, we have addressed the robotic environment. Several decisional problems in the robotic field are complex/np-complete by nature and can be solved by the frameworks of Constraint Programming. For this, we have decided to propose an extension of JChoc DisSolver. Several Robots (e.g. PR2, TurtleBot 1/2/3, Nireo One, Nao, etc.) use the Robotic Operating System (ROS) that provides the services we would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. All these robots can now use RoboChoc patform, the new version of JChoc DisSolver platform, to solve decisional complex problems those can be formalized by Constraint Programming frameworks. This contribution has been released by adding a new layer to JChoc DisSolver platform to allow the communication between the solving protocol and the robot hardware.

Later, we have addressed the competitive networks and the cooperative networks. As a use case for the competitive networks, we have proposed a new approach for the smart e-marketplace. In this contribution, we have proposed a formalization of the marketplace using the DisCSP framework. This problem must be solved in a competitive context. And for this, we have modified the ABT protocol and proposed a new protocol named ABT-Trader to make the resolution of the problem possible. In this approach, we have two type of agents: Sellers and Buyers. The integration of this new approach in a real e-marketplace platform is possible by adapting the formalization of the problem and adding all the product and their specifications to the model. The cooperative networks field has been addressed using an Internet of Things (IoT) use case. In this use case, we have formalized and solved the problem of the smart farm resources management. We have proposed a global Artificial Intelligence architecture to handle many real IoT problems. The results of our approaches are very promising and show the efficiency of the propositions.

The last contribution of this thesis is proposed to enhance the state of the art in solving the DisCSP problems. Many algorithms have been proposed to solve the DisCSP. In our exploitation of the DisCSP state of the art, we have found that the AFC-tree algorithm is one of the most powerful solving protocols. We have studied and implemented the AFC-tree algorithm in JChoc DisSolver; then we tested its performances. In the solution detection phase of the AFC-tree, many messages become obsoletes and useless. To overcome this issue and improve the algorithm, we have changed the solution detection phase by proposing new procedures. We have proposed a new algorithm for Solving the DisCSP named IFC-tree algorithm. We have implemented, tested the IFC-tree algorithm on JChoc DisSolver platform. The experiment results show that we have improved the different resolution metrics.

Perspectives

In this thesis we have addressed principally the distributed CSP framework. For Dynamic JChoc platform, we can add more solving protocols from the other Constraints Programming frameworks. Our goal is also to propose an easy graphical interface to help non-experts to model and solve their distributed problems.

The IFC-tree algorithm can be improved in two directions: the solution detection and the peudo-tree representation. The time-out used to detect the silence in the network and to announce the global solution can be calculated using the pseudo-tree levels and data from parents' agents (number of values in the domain, constraints tightness, etc.) for each leaf agent. We believe that, the conversion process of the problem graph to a pseudo-tree can use heuristics to produce the best peudo-tree that will optimize the communication between agents to find the solution using less effort.

Bibliography

Hosame H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37:449–453, April 1988. Cited page 30.

Lloyd Allison, CN Yee, and M McGaughey. Three-dimensional queens problems. Monash University, Department of Computer Science, 1989. Cited pages 55 and 56.

John OR Aoga, Tias Guns, and Pierre Schaus. An efficient algorithm for mining frequent sequence with constraint programming. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 315–330. Springer, 2016. Cited page 79.

Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009. Cited page 77.

Debasis Bandyopadhyay and Jaydip Sen. Internet of things: Applications and challenges in technology and standardization. *Wireless personal communications*, 58 (1):49–69, 2011. Cited page 77.

Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman, and Magda Valls. Sensor networks and distributed csp: communication, computation and complexity. *Artificial Intelligence*, 161(1-2):117–147, 2005. Cited pages 14, 24, 27, 29, and 35.

Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007. Cited pages 38, 51, 53, 71, 78, and 100.

Yosra Acodad El Houssine Bouyakhf Benamrane, Amine and Imade Benelallam. Liveabt: A real-time repairing protocol for incremental and dynamic discsps. *International Journal of Artificial Intelligence*, 15(1), 2017. Cited pages 78 and 79.

Imade Benelallam, Zakarya Erraji, Ghizlane EL Khattabi, and El Houssine Bouyakhf. Dynamic jchoc: A distributed constraints reasoning platform for dynamically changing environments. In *International Conference on Agents and Artificial Intelligence*, pages 20–36. Springer, 2015. Cited pages 71, 78, and 100.

Imade Benelallam, El Houssine Bouyakhf, et al. A commentary on "hybrid search for minimal perturbation in dynamic csps". *Constraints*, 21(2):349–354, 2016. Cited page 79.

Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. 2006. doi: 10.1016/S1574-6526(06)80007-6. URL http://dx.doi.org/10.1016/S1574-6526(06)80007-6. Cited page 23.

Christian Bessiere, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In *Proceeding of Workshop on Distributed Constraint Reasoning, IJCAI'01*, Seattle, Washington, USA, August 4 2001. Cited page 30.

Christian Bessiere, Ismel Brito, Arnold Maestre, and Pedro Meseguer. The asynchronous backtracking family. *LIRMM-CNRS,, Montpellier, France March*, 2003, 2003. Cited page 77.

Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161:7–24, 2005. Cited pages 30, 32, and 33.

Christian Bessière, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161(1-2):7–24, 2005. Cited pages 36 and 92.

Christian Bessière, Arnold Maestre, and P Messeguer. La famille abt. In *Journées Nationales sur la Résolution Pratique de Problemes NP-Complets*, pages 57–67. Springer, 2008. Cited pages 63, 69, and 82.

Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. volume 3, pages 353–387. ACM, 1981. Cited page 51.

Ismel Brito and Pedro Meseguer. Distributed forward checking. In *International Conference on Principles and Practice of Constraint Programming*, pages 801–806. Springer, 2003. Cited pages 30, 63, and 77.

Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for discsp. In *CP*, page 791, 2004. Cited pages 30 and 92.

Ismel Brito, Amnon Meisels, Pedro Meseguer, and Roie Zivan. Distributed constraint satisfaction with partially known constraints. *Constraints*, 14(2):199–234, 2009. Cited page 77.

Yek Loong Chong and Youssef Hamadi. Distributed log-based reconciliation. In *Proceedings of the 17th European Conference on Artificial Intelligence*, ECAI'06, pages 108–112, 2006. Cited pages 24 and 27.

Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El Houssine Bouyakhf. Constraint acquisition using recommendation queries. In *IJCAI: International Joint Conference on Artificial Intelligence*, pages 720–726, 2016. Cited page 79.

L De Raedt, T Guns, and S Nijssen. Constraint programming for itemset mining in proc. In *ACM SIGKDD*, pages 204–212, 2008. Cited page 78.

Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for data mining and machine learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. Cited page 78.

Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990. doi: 10.1016/0004-3702(90)90046-3. URL http://dx.doi.org/10.1016/0004-3702(90)90046-3. Cited page 23.

Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003. ISBN 978-1-55860-890-0. URL http://www.elsevier.com/wps/find/bookdescription.agents/678024/description. Cited page 17.

G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular openrobots simulation engine: Morse. In *Proceedings of the IEEE ICRA*. 2011. Cited page 56.

Joakim Eriksson, Niclas Finne, and Sverker Janson. To each and everyone an agent: augmenting web-based commerce with agents. In *Proceedings of the International Workshop on Intelligent Agents on the Internet and Web, Fourth World Congress on Expert Systems*. Citeseer, 1998. Cited page 62.

Zakarya Erraji, Mounia Janah, Imade Benelallam, and El Houssine Bouyakhf. Mobile robotic jchoc dissolver - a distributed constraints reasoning platform for mobile multi-robot problems. In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART,*, pages 304–310. INSTICC, Rome: Italy, SciTePress, 2016. ISBN 978-989-758-172-4. doi: 10.5220/0005833003040310. Cited page 92.

Zakarya Erraji, Imade Benelallam, and El Houssine Bouyakhf. Towards a constraint programming approach for cognitive iot. In *Workshop on Progress Towards the Holy Grail*. CP2017, Melbourne: Australia, 2017. Cited page 103.

Redouane Ezzahir, Christian Bessiere, Imade Benelallam, Houssine Bouyakhf, and Mustapha Belaissaoui. Dynamic backtracking for distributed constraint optimization. In *ECAI*, volume 8, pages 901–902, 2008. Cited page 36.

Redouane Ezzahir, Christian Bessiere, Mohamed Wahbi, Imade Benelallam, and El Houssine Bouyakhf. Asynchronous inter-level forward-checking for discsps. In *International Conference on Principles and Practice of Constraint Programming*, pages 304–318. Springer, 2009. Cited pages 30, 36, 78, and 93.

Shaheen S Fatima, Michael Wooldridge, and Nicholas R Jennings. Approximate and online multi-issue negotiation. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 156. ACM, 2007. Cited pages 62 and 75.

Eugene C Freuder and Michael J Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *IJCAI*, volume 85, pages 1076–1078, 1985. Cited page 93.

M Galley. Distributed constraint programming platform using sjavap (2000). Cited page 36.

John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, August 1977*, page 457, 1977. Cited page 23.

Amir Gershman, Amnon Meisels, and Roie Zivan. Asynchronous forward bounding for distributed cops. *Journal of Artificial Intelligence Research*, 34:61–88, 2009. Cited page 36.

Tias Guns, Anton Dries, Siegfried Nijssen, Guido Tack, and Luc De Raedt. Miningzinc: A declarative framework for constraint-based mining. *Artificial Intelligence*, 244:6–29, 2017. Cited page 79.

Y. Hamadi. *Disolver : A distributed constraint solver.* Microsoft Research., 2003. Cited page 36.

Youssef Hamadi, Christian Bessiere, and Joël Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of the 13th European Conference on Artificial Intelligence*, ECAI'98, pages 219–223, Brighton, UK, 1998. Cited pages 30 and 33.

Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international joint conference on Artificial intelligence*, IJCAI'79, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. Cited page 24.

Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980. doi: 10.1016/0004-3702(80)90051-X. URL http://dx.doi.org/10.1016/0004-3702(80)90051-X. Cited pages 23, 24, and 31.

Miniar Hemaissia, Amal El Fallah Seghrouchni, Christophe Labreuche, and Juliette Mattioli. A multilateral multi-issue negotiation protocol. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 155. ACM, 2007. Cited page 62.

Holger H. Hoos and Edward P. K. Tsang. Local search methods. In *Handbook of Constraint Programming*, pages 135–167. 2006. doi: 10.1016/S1574-6526(06)80009-X. URL http://dx.doi.org/10.1016/S1574-6526(06)80009-X. Cited page 23.

Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. Argumentation as distributed constraint satisfaction: applications and results. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS'01, pages 324–331, 2001. Cited pages 24, 27, and 29.

Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008. Cited pages 38, 51, and 53.

Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 99CH37028)*, volume 6, pages 739–743. IEEE, 1999. Cited page 14.

Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968. Cited page 23.

Guoming Lai, Katia Sycara, and Cuihong Li. A pareto optimal model for automated multi-attribute negotiations. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 246. ACM, 2007. Cited pages 62 and 75.

Thomas Léauté and Boi Faltings. Coordinating logistics operations with privacy guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, IJCAI'11, pages 2482–2487, July 16–22 2011. Cited page 24.

Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons, 2013. Cited page 17.

Leslie Lipper, Philip Thornton, Bruce M Campbell, Tobias Baedeker, Ademola Braimoh, Martin Bwalya, Patrick Caron, Andrea Cattaneo, Dennis Garrity, Kevin Henry, et al. Climate-smart agriculture for food security. *Nature climate change*, 4(12):1068, 2014. Cited page 80.

Benny Lutati, Inna Gontmakher, Michael Lando, Arnon Netzer, Amnon Meisels, and Alon Grubshtein. Agentzero: A framework for simulating and evaluating multi-agent algorithms. In *Agent-Oriented Software Engineering*, pages 309–327. Springer, 2014. Cited page 37.

Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, 1977. Cited page 23.

Alan K Mackworth. Constraint-based design of embedded intelligent systems. volume 2, pages 83–86. Springer, 1997. Cited page 51.

Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'04, 2004. Cited pages 24 and 27.

Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998. Cited page 97.

Kalech Meir, A. Kaminka Gal, Meisels Amnon, and Elmaliach Yehuda. Diagnosis of multi-robot coordination failures using distributed csp algorithms. In *American Association for Artificial Intelligence, 970-975*, 2006. Cited page 52.

Amnon Meisels and Oz Lavee. Using additional information in discsp search. In *Distributed constraint reasoning workshop (DCR)*, 2004. Cited page 27.

Amnon Meisels and I. Razgon. Distributed forward-checking with conflict-based backjumping and dynamic ordering. In *Proceeding of CoSolv workshop, CP02*, Ithaca, NY, 2002. Cited page 30.

Amnon Meisels and Roie Zivan. Asynchronous forward-checking for distributed CSPs. In W. Zhang, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003. Cited pages 30 and 31.

Amnon Meisels and Roie Zivan. Asynchronous forward-checking for discsps. *Constraints*, 12(1):131–150, 2007. Cited pages 31, 36, and 92.

Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58(1-3):161–205, 1992. doi: 10.1016/0004-3702(92)90007-K. URL http://dx.doi.org/10.1016/0004-3702(92)90007-K. Cited page 23.

Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad hoc networks*, 10(7):1497–1516, 2012. Cited page 77.

Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005. Cited page 36.

Benjamin Negrevergne and Tias Guns. Constraint-based sequence mining using constraint programming. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 288–305. Springer, 2015. Cited page 79.

Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic distributed backjumping. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 71–85. Springer, 2004. Cited page 92.

Bayo Omomowo, Inés Arana, and Hatem Ahriz. Dynabt: Dynamic asynchronous backtracking for dynamic discsps. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 285–296. Springer, 2008. Cited pages 63 and 79.

Barry O'Sullivan. Opportunities and challenges for constraint programming. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012. Cited page 77.

Dinesh K Pai. Programming parallel distributed control for complex systems. In *Intelligent Control, 1989. Proceedings., IEEE International Symposium on*, pages 426–432. 1989. Cited page 51.

Dinesh K Pai. Least constraint: A framework for the control of complex mechanical systems. In *American Control Conference, 1991*, pages 1615–1621. 1991. Cited page 51.

Dorin Panescu and Carlos Pascal. A constraint satisfaction approach for planning of multi-robot systems. In *System Theory, Control and Computing (ICSTCC), 2014 18th International Conference*, pages 157–162. 2014. Cited page 52.

H Van Dyke Parunak, Allen C Ward, and John A Sauter. A systematic market approach to distributed constraint problems. In *Proceedings International Conference on Multi Agent Systems (Cat. No. 98EX160)*, pages 455–456. IEEE, 1998. Cited pages 62 and 74.

Adrian Petcu. Frodo: a framework for open/distributed optimization, 2005. Cited pages 36 and 37.

Adrian Petcu and Boi Faltings. A value ordering heuristic for local search in distributed resource allocation. In *International Workshop on Constraint Solving*

and Constraint Logic Programming, pages 86–97. Springer, 2004. Cited pages 24 and 27.

Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993. doi: 10.1111/j.1467-8640.1993.tb00310.x. URL http://dx.doi.org/10.1111/j.1467-8640.1993.tb00310.x. Cited pages 23 and 31.

Patrick Prosser, Chris Conway, and Claude Muller. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83, oct 1992. Cited page 27.

Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL http://www.choco-solver.org. Cited pages 78 and 101.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. Cited pages 51, 53, and 79.

Iyad Rahwan, Ryszard Kowalczyk, and Ha Hai Pham. Intelligent agents for automated one-to-many e-commerce negotiation. In *Australian Computer Science Communications*, volume 24, pages 197–204. Australian Computer Society, Inc., 2002. Cited pages 62 and 74.

Fenghui Ren, Minjie Zhang, and Kwang Mong Sim. Adaptive conceding strategies for automated trading agents in dynamic, open markets. *Decision Support Systems*, 46(3):704–716, 2009. Cited page 62.

Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL http://www.sciencedirect.com/science/bookseries/15746526/2. Cited pages 17 and 23.

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. Cited page 78.

Marius-Calin Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. In *DCR workshop, AAMAS-06*, 2006. Cited page 26.

Marius-Calin Silaghi and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:25–53, 2005. Cited page 30.

Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, 2000. Cited page 97.

Marius-călin Silaghi, Djamila Sam-haroud, and Boi Faltings. Abt with asynchronous reordering. In *Intelligent Agent Technology: Research and Development*, pages 54–63. World Scientific, 2001. Cited page 34.

Richard S Stansbury and Arvin Agah. A robot decision making framework using constraint programming. volume 38, pages 67–83. Springer, 2012. Cited page 52.

Evan A Sultanik, Robert N Lass, and William C Regli. Dcopolis: A framework for simulating and deploying distributed constraint optimization algorithms. 2007. Cited page 37.

Federico Thomas and Lluís Ros. Revisiting trilateration for robot localization. volume 21, pages 93–101. IEEE, 2005. Cited page 51.

Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. Dischoco 2: A platform for distributed constraint reasoning. *Proceedings of DCR*, 11:112–121, 2011. Cited pages 36 and 37.

Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. Nogood-based asynchronous forward checking algorithms. *Constraints*, 18(3): 404–433, 2013. Cited pages 93, 94, and 101.

Mark Wallace and Joachim Schimpf. Finding the right hybrid algorithm - A combinatorial meta-problem. *Ann. Math. Artif. Intell.*, 34(4):259–269, 2002. doi: 10.1023/A:1014450507312. URL http://dx.doi.org/10.1023/A:1014450507312. Cited page 23.

Richard J Wallace and Eugene C Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss, 2002. Cited pages 24 and 27.

Qihui Wu, Guoru Ding, Yuhua Xu, Shuo Feng, Zhiyong Du, Jinlong Wang, and Keping Long. Cognitive internet of things: a new paradigm beyond connection. *IEEE Internet of Things Journal*, 1(2):129–143, 2014. Cited pages 77 and 79.

William Yeoh, Ariel Felner, and Sven Koenig. Bnb-adopt: An asynchronous branch-and-bound dcop algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010. Cited page 36.

M. Yokoo. Distributed constraint satisfaction. In *:Foundation of Cooperation in multiagent Systems*. Springer, 2001. Cited page 14.

M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on MultiAgent Systems*. MIT Press, 1995a. Cited page 14.

Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceeding of CP*, pages 88–102, 1995. Cited page 30.

Makoto Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer-Verlag, London, UK, 2000. Cited pages 26 and 31.

Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*. MIT Press, 1995b. Cited page 30.

Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000. Cited pages 25 and 33.

Makoto Yokoo, Toru Ishida, Edmund H Durfee, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 614–621. IEEE, 1992. Cited pages 30, 32, 33, 36, and 92.

Makoto Yokoo, Edmund H Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on knowledge and data engineering*, 10(5):673–685, 1998. Cited pages 25, 30, 32, and 33.

Amine Benamrane Imade Benelallam Zakarya Erraji, Amal Hakkou and El Houssine Bouyakhf. A distributed constraint reasoning approach towards intelligent marketplace environment. In *International workshop on Constraint Modelling and Reformulation*. CP 2016, 2016. Cited page 78.

Safdar Zaman, Wolfgang Slany, and Gerald Steinbauer. Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, pages 1–5. 2011. Cited page 51.

Ying Zhang and Alan K Mackworth. Specification and verification of constraint-based dynamic systems. In *Principles and Practice of Constraint Programming*, pages 229–242. 1994. Cited page 51.

Yu Zhang and Alan K Mackworth. A constraint-based robotic soccer team. volume 7, pages 7–28. Springer, 2002. Cited page 51.

Roie Zivan and Amnon Meisels. Synchronous vs asynchronous search on discsps. In *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, 2003. Cited pages 31 and 92.

Résumé:

La programmation par contraintes (PPC) est un paradigme de l'intelligence artificielle qui vise la résolution de problèmes hautement combinatoires. Les problèmes de satisfaction de contraintes distribués (DisCSP) est un formalisme général de la PC développé pour la résolution des problèmes multi agents distribués. Le DisCSP a plusieurs applications dans les problèmes de coordination multi agents.

Dans cette thèse, nous présentons l'état de l'art de la résolution par les DisCSP en proposant plusieurs algorithmes et applications. Premièrement, nous proposons une nouvelle plateforme nommée JChoc DisSolver. Cette plateforme est basée sur le solveur de con- traintes Choco et la plateforme multi agents JADE. Ensuite, nous proposons une version de JChoc DisSolver dédiée aux applications robotiques pour la résolution des problèmes multi robots suivie d'une nouvelle approche qui modélise le problème du marché électronique intelligent (vente et achat de biens). Nous avons utilisé aussi notre plateforme pour présenter notre vision pour la modélisation et la résolution des problèmes d'objets connectés et comment nous pouvons les rendre autonomes et intelligents à travers l'utilisation de la programmation par contraintes. Finalement, Nous détaillons notre nouvel algorithme nommé The Improved Forward Checking Tree Algorithm (IFC). Cet algorithme destiné à la résolution des DisCSP et IFC est basé sur l'algorithme the Asynchronous Forward Checking Tree (AFC-tree). Les résultats expérimentaux montrent l'efficacité du nouvel algorithme en améliorant la performance des différentes métriques de la résolution.

Mots-clés : *Intelligence Artificielle, Programmation par contraintes, Systèmes Multi Agents, Problèmes de satisfaction de contraintes (CSP), CSP Distribuées (DisCSP), Système Multi robots, Marché électronique intelligent, Modélisation et architecture d'agent, Objets connectés cognitifs.*

Abstract:

Constraint Programming (CP) is a general paradigm used to model and to solve complex combinatorial problems. Distributed Constraint Satisfaction Problems (DisCSP) belongs to Constraint CP and is a general framework for solving distributed problems. DisCSP have a wide range of applications in multi-agent coordination.

In this thesis, we extend the state of the art in solving the DisCSPs by proposing several algorithms and applications. Firstly, we propose a new DisCSP platform named JChoc DisSolver, a platform based on Choco Solver and JADE multi-Agent platform. Then, we propose a robotic dedicated version of JChoc DisSolver platform to solve the multi-robot problems. After that, we propose a new approach to model the intelligence and autonomous marketplace environment (sell and buy goods) this approach includes a new problem formulation and Distributed Constraint Reasoning protocol to deal with e-marketplace issues. After that, we used our platform named JChoc DisSolver to model, implement and illustrate a real-world use case application based on IoT techniques. Therefore, Constraint Programming techniques have been proved to be a very elegant paradigm to handle CIoT applications. Finally, The Improved Forward Checking Tree Algorithm (IFC) for Distributed Constraint Satisfaction Problems is described. The IFC is a new algorithm based on the Asynchronous Forward Checking Tree (AFC-tree) algorithm for the DisCSP problems. The experimental results show the efficiency of this algorithm by improving different metrics of the solving performances.

Keywords: *Artificial Intelligence, Constraint Programming, Multi-Agent Systems, Realistic use, Constraint Satisfaction Problem (CSP), Distributed CSP (DisCSP), Multi-robot Systems, Intelligent Marketplace, Agent Models and Architectures, Cognitive Internet of Things.*

Année Universitaire : 2019/2020