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 : 3340

THESE

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

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

Présentée et soutenue le : 24/09/2020 par :

Ghizlane El Khattabi

Towards Complex and Ethical Distributed Constraint Satisfaction Problems

JURY

Mohammed Abdelfattah CHARIF-CHEFCHAOUNI	*PES, Institut National des Postes et Télécommunications, Rabat*	*Président*
El Houssine BOUYAKHF	*PES, Université Mohammed V, Faculté des Sciences, Rabat*	*Directeur de thèse*
Imade BENELALLAM	*PH, Institut National de Statistiques et d'Economie Appliquée, Rabat*	*Codirecteur de thèse*
Fouzia OMARY	*PES, Université Mohammed V, Faculté des Sciences, Rabat*	*Rapportrice/Examinatrice*
Mustapha BELAISSAOUI	*PES, Ecole Nationale de Commerce et de Gestion, Settat*	*Rapporteur/Examinateur*
Redouane EZZAHIR	*PH, Ecole Nationale des Sciences Appliquées, Agadir*	*Rapporteur/Examinateur*
Mohammed MAJID HIMMI	*PES, Université Mohammed V, Faculté des Sciences, Rabat*	*Examinateur*
Abdelhak MAHMOUDI	*PH, Ecole Normale Supérieure, Rabat*	*Examinateur*

Année Universitaire : 2019/2020

"Software constraints are only if you use them for what they're intended to be used for"

David Byrne

To my mother, ma mère, أمي, and my father. You are always showering me with your support, patient, and unconditional love. You are for me the examples of courage, forbearance, sacrifice, and frisking. I hope this thesis proves my love and respect.

To my brothers! No, my fathers! No, my friends Ali, Driss, and Hamid. You are still playing a triple role. You have always supported, helped, understood me, and you have even prayed for me. Please find here the testimony of my gratitude.

To my sisters−in−law, my nephews, and my whole family.

To all my friends, especially, Fatima Zahra, Nabila, Tiffy, Asmae, and my sister by heart. You made my life.

Ghizlane El Khattabi

Acknowledgements

The research presented in this thesis has been acheived in Laboratoire d'Informatique Mathématiques appliquées Intelligence Artificielle et Reconnaissance de Formes (LIMIARF), Faculty of Science, University Mohammed V, Rabat, Morocco. It was carried out under the direction of Mr. El Houssine BOUYAKHF, in Co-direction with Mr. Imade BENELALLAM.

First of all, I express my gratitude to Mr. El Houssine **BOUYAKHF**, PES in the Faculty of Sciences of Rabat, for his approval within the laboratory, his advice, and his support.

I would like to thanks Professor Mohammed Abdelfattah **CHARIF-CHEFCHAOUNI**, PES at the INPT of Rabat, for the honor he did by approving to be President of my thesis jury.

I would like to express my recognition to Mr. Imade **BENELALLAM**, PH at INSEA of Rabat, for the effort provided, the advice, and the patience in monitoring the project.

I am very honored to thank all professors for being on my thesis jury. I would like to thank:

Mr. Mustapha **BELAISSAOUI**, PES at ENCG of Settat, Reviewer, and Examinator of my thesis, for the interest that he carries to this work, and for the interesting remarks, he brings.

Mrs Fouzia **OMARY**, PES in the Faculty of Sciences of Rabat, Reviewer and Examinator of this thesis, for the time spent reading this thesis and for the judicious suggestions and remarks, it gave me.

Mr. Redouane **EZZAHIR**, PH at ENSA of Agadir, Reviewers and examinators of this thesis report, for agreeing to evaluate my work and reading my manuscript thoroughly in order to bring its valuable advice.

Mr. Mohammed **MAJID HIMMI**, PES in the Faculty of Sciences of Rabat, Examinator of this thesis report, for all the interesting remarks he made.

Mr. Abdelhak **MAHMOUDI**, PH at ENS of Rabat, Examinator of this thesis, for agreeing to examine this work.

I thank all my LIMIARF Laboratory colleagues and friends for the joyful and pleasant working environment and moments. They will still in my memory. My thanks are addressed also to all my friends, for the unforgettable moments.

I thank my parents and my brothers very much. Even if I rewrite the whole report only for thanking them, it will be insufficient. They were always there to support me, to pray for me, and especially to be patient with me. My thanks go to all my family for their moral supports.

Abstract

Distributed Constraint Satisfaction Problem (DisCSP) is a mathematical formalism solving many combinatorial distributed problems. It has proven its effectiveness in representing and solving many real problems. This thesis presents an extension of the state of the art of the DisCSP domain. Therefore, the made contributions are regrouped into four main parts. In the first part, we have studied the exiting DisCSP algorithms, prepared the dynamic DisCSP platform JChoc, and proposed new DisCSP algorithms based on the existing algorithms' merge.

The second part broaches DisCSPs with Complex local problems. We proposed the Minimal Perturbation based Asynchronous Backtracking (MP-ABT), an algorithm treating the DisCSP local problems as Minimal Perturbation Problems (MPP), the Selective Sorting and Smart Nogood (SS&SN), a general method applicable to every DisCSP algorithm so as it can solve DisCSPs with Complex local problems, and SS&SN based ABT (ABT-SS&SN) and SS&SN based AFC-ng (AFC-ng-SS&SN), the application of the SS&SN method on the DisCSP algorithms ABT and AFC-ng.

The third part approaches the creation of a new formalism named Ethical DisCSP (E-DisCSP). We integrated the ethics concept into DisCSPs. We dealt the gravity of the presence of an unethical agent into a DisCSP resolution, we proposed some methods to detect such an agent, and we suggested actions to apply on so as the DisCSP resolution is as smooth as possible. Whilst the fourth part is an initiation to the application of what we have realized on a realistic and recent problem which is the Smart Grid.

Keywords: Artificial Intelligence, Distributed Constraint Satisfaction Problems (DisCSP), DisCSPs with complex local problems, ethics, Ethical DisCSPs (E-DisCSP), Smart Grid.

Résumé

Les problèmes de satisfaction des contraintes distribuées (DisCSP) représentent de nombreux problèmes distribués combinatoires. Il a prouvé sa puissance dans l'interprétation et la résolution de nombreux problèmes réels. Cette thèse présente une extension de l'état de l'art du domaine du DisCSP. Les contributions apportées sont regroupées en quatre parties principales. Dans la première partie, nous avons étudié les algorithmes DisCSP existants, préparé la plateforme DisCSP dynamique JChoc, et proposé de nouveaux algorithmes DisCSP basés sur la fusion des algorithmes existants.

La deuxième partie entame les DisCSPs avec des problèmes locaux complexes. Dans cette partie, nous avons proposé le Minimal Perturbation based Asynchronous Back-Tracking (MP-ABT), un algorithme considérant les problèmes locaux du DisCSP comme des problèmes de minimisation des perturbation (MPP), le Selective Sorting and Smart Nogood (SS&SN), une méthode générale applicable sur chaque algorithme DisCSP afin qu'il puisse gérer les DisCSP avec les problèmes locaux complexes, et SS&SN based ABT (ABT-SS&SN), SS&SN based AFC-ng (AFC-ng-SS&SN), l'application du SS&SN sur les algorithmes DisCSP ABT et AFC-ng.

La troisième partie s'attaque à la création d'un nouveau formalisme nommé Ethical DisCSP (E-DisCSP). Nous avons intégré le concept d'éthique dans les DisCSPs. Nous avons traité la gravité de la présence d'un agent non éthique dans une résolution DisCSP, proposé quelques méthodes pour détecter un tel agent, et nous avons suggéré des actions à appliquer afin que la résolution DisCSP soit aussi normale que possible. Tandis que la quatrième partie est une initiation à l'application de ce que nous avons réalisé sur un problème réaliste et récent qui est le Smart Grid.

Mots clés: L'intelligence artificielle, les problèmes de satisfaction des contraintes distribués (DisCSP), les DisCSPs avec les problèmes locaux complexes, l'éthique, les DisCSPs éthiques (E-DisCSP), Smart Grid.

Résumé étendu

Constraint Satisfaction Problem (CSP) est un formalisme représentant plusieurs problèmes combinatoires réels. Il modélise les problèmes en un ensemble de **Variables**, définies dans un ensemble de **Domaines**, et liées par un ensemble de **Contraintes**. La nature distribuée de certains problèmes a causé l'extension du formalisme CSP en Distributed CSP. En plus des paramètres CSP, les problèmes DisCSP sont modélisés par un ensemble d'**Agents**. Les paramètres sont donc distribués entre ces agents qui se connectent via des contraintes aussi.

Plusieurs algorithmes sont apparus afin de résoudre de tels problèmes. Ils se basent en général sur l'échange des messages soit d'une manière synchrone ou asynchrone. Les messages échangés peuvent être divisé en deux catégories. Les messages qui portent les instanciations des agents, et ceux qui expriment leur blocage. Les algorithmes se diffère par la manière d'envoi des messages (synchrone ou asynchrone) et la manière d'exprimer le blocage. Parmi les algorithmes DisCSP existants, nous pouvons mentionner l'Asynchronous Backtracking ABT et le nogood based Asynchronous Forward Checking (AFC-ng). Les performances de ces algorithmes dépendent du problème à résoudre. Par exemple, AFC-ng est performant lorsque les problèmes sont denses, alors que dans le cas contraire c'est l'ABT le meilleur.

Dans un premier lieu, et avant d'entamer l'extension de l'état de l'art en ce qui concerne les méthodes, les algorithmes et les formalismes, nous avons préparé une plateforme DisCSP nommée JChoc, dans **le deuxième chapitre** de **la première partie**, après avoir abordé l'état de l'art DisCSP dans **le premier chapitre**. La plateforme permet d'exploiter les algorithmes existants, de créer de nouveaux algorithmes, de les expérimenter, et de permettre aux non-experts la résolution des problèmes DisCSPs sans avoir besoin à connaitre les algorithmes. Les résultats expérimentaux ont montré l'efficacité de la plateforme même dans des environnements dynamiques.

Dans **le troisième chapitre** de la même partie, et toujours dans le but d'exploiter et de tirer profit de l'existant, nous avons essayé de lancer plus qu'un algorithme en même temps, et proposé deux méthodes d'apprentissage des nogoods (une manière d'exprimer le blocage) entre ces algorithmes. Elles se basent sur le fait que les performances d'un algorithme dépendent du problème à résoudre. La première méthode garde les deux

algorithmes en exécution, et fait l'apprentissage mutuel des différents algorithmes, alors que la deuxième permet de piquer en quelques sortes le meilleur algorithme, au départ et d'arrêter l'autre. Les expérimentations ont prouvé l'efficacité de l'apprentissage, surtout celle de la deuxième méthode.

La conception des algorithmes DisCSP est faite de telle sorte à supporter les problèmes dont les agents sont simples ; chacun gère une seule variable. Dans le cas complexe, la communauté DisCSP utilise des méthodes qui transforment les problèmes locaux des agents. Parmi ces méthodes, on trouve la compilation qui crée pour chaque agent complexe une variable abstraite, dont le domaine est l'ensemble de ses solutions locales. On trouve aussi la décomposition, qui considère chaque variable locale comme un agent virtuel.

Dans **la deuxième partie**, nous avons abordé ces DisCSPs ayant les problèmes locaux complexes. Dans son premier chapitre (**Chapitre 4**), nous avons détaillé les méthodes précitées ainsi que les autres méthodes et algorithmes existants. Tous les algorithmes se basent en général sur soit la compilation, ou bien la décomposition.

Dans **le chapitre 5**, nous avons proposé l'algorithme Minimal Perturbation based ABT (MP-ABT), une extension de l'ABT. Dans cet algorithme, nous considérons chaque agent complexe, comme un problème de minimisation des perturbations (MPP pour Minimal Perturbation Problem). À chaque fois l'agent reçoit un message demandant le changement de son instanciation, il cherche une nouvelle instanciation très proche de l'antécédente, changeant le minimum des variables locales. La minimisation des perturbations locales a pour objectif de minimiser la perturbation des agents et donc minimiser l'échange des messages. Tous ces objectifs ont été atteints dans les résultats expérimentaux.

Dans **le chapitre 6**, nous avons proposé le Selective Sorting & Smart Nogood SS&SN. Une méthode générale à intégrer dans les algorithmes DisCSP afin qu'ils puissent résoudre les DisCSP complexes. A priori, la méthode se base sur la compilation et se compose de deux sous méthodes. La première Selective Sorting SS effectue un tri sélectif du domaine compilé de la variable abstraite créé, à chaque fois une nouvelle instanciation est requise. Le tri se fait sur la base de choisir d'abord l'instanciation qui est la plus proche de la précédente, qui va perturber les agents les moins prioritaires et qui engendra la vérification de moins de contraintes. Pour la deuxième méthode Smart Nogood SN, elle améliore la forme du nogood/le message de blocage. Le nogood envoyé lors de l'utilisation de la compilation renvoie toute l'instanciation d'un agent. Donc, un simple

changement d'une valeur d'une seule variable, est considérée comme une autre instanciation non bloquée, même si la variable changée n'a aucun rapport avec le blocage. Donc, l'objectif du SN est de concevoir un nogood renvoyant seulement les valeurs responsables du blocage.

Dans **le chapitre 7**, nous avons présenté les algorithmes ABT-SS&SN et AFC-ng-SS&SN, résultats de l'intégration de la méthode SS&SN dans les algorithmes ABT et AFC-ng. Nous avons comparé ces algorithmes avec les existants. Les résultats expérimentaux ont montré la force de la méthode SS&SN surtout lorsque les problèmes sont solvables. Dans le cas contraire, l'effort mis dans le tri des domaines se gaspillent par la déclaration de l'insolvabilité du problème. Mais le SN reste toujours bénéfique.

En parallèle avec l'étude des DisCSPs complexes, nous avons travaillé, dans une **troisième partie**, sur la possibilité d'intégrer la notion de l'éthique dans les DisCSPs. Dans **le chapitre 8**, le premier de cette partie, nous avons d'abord défini l'agent DisCSP non éthique, en présentant quelques réactions possibles de cet agent, qui peut mentir, ou bien négliger un message de blocage. Puis, nous avons présenté la gravité de la présence d'un tel agent dans une résolution DisCSP. Parmi les résultats, nous avons trouvé que la plupart des résolutions ratent la première solution, donnent des solutions incorrectes, bouclent infiniment, ou bien s'arrêtent après un time out. Afin de résoudre ce problème nous avons créé un nouveau formalisme Ethical DisCSP (E-DisCSP), qui, en plus des paramètres DisCSP, ajoutent des règles éthiques à respecter, et des actions à appliquer dans le cas du non-respect de ces règles. Dans le même chapitre nous avons conçu un système de contrôle assurant le contrôle de ces nouveaux paramètres. Ils se composent de trois unités principales. L'unité des hypothèses (Assumptions unit), qui récupère les messages échangés et les transforme en fbf, l'unité éthique (Ethical unit), qui enregistrent les règles éthiques ainsi que les actions, et l'unité TMS, qui vérifie la consistance des fbfs, récupérés depuis les messages, contre les règles éthiques enregistrées dans l'unité éthique. Lors de la détection d'une violation d'une règle, il récupère l'action appropriée.

Avant d'entamer l'intégration des règles éthiques dans l'unité éthique, nous avons veillé à contrôler, d'abord, les activités anormales communes, comme le mensonge et la négligence des messages, dans **le chapitre 9**. Nous avons proposé une méthode de conversion des messages en fbf et nous avons utilisé les résultats dans l'unité TMS, afin de détecter les activités anormales précitées. Les résultats expérimentaux ont montré que le taux de détection est très élevé d'où la force de la méthode de conversion utilisée et celle de la détection.

Une fois la détection est faite, nous avons proposé quelques actions à appliquer sur les agents responsables dans **le chapitre 10**. Nous avons proposé comme actions : le changement de la priorité de l'agent non éthique, ou bien son élimination du processus de la résolution. Les actions prises ont pu diminuer d'une manière très importante le nombre de solutions erronées, des boucles infinies, et des times out.

Dans **la quatrième partie**, nous avons pris un problème réel qui est le Smart Grid, afin d'appliquer progressivement tout ce nous avons réalisé. Dans **le chapitre 11**, nous avons commencé par la modélisation CSP, des niveaux locaux du problème. La modélisation vise, en général, à prioriser l'utilisation des énergies renouvelables, ne pas avoir une consommation locale qui dépasse les prévisions, et stocker l'énergie renouvelable, si excès.

Avant de passer à la modélisation DisCSP, nous avons réalisé que les problèmes locaux du Smart Grid sont des problèmes d'optimisations, par définition. Donc, dans **le chapitre 12**, nous avons modélisé le problème en Constraint Optimization Problem (COP), qui rajoute au CSP des fonctions objectives à optimiser. Les fonctions objectives définissant ce problème sont en général, la priorisation du fonctionnement des appareils, la maximisation du nombre d'appareils fonctionnel et la minimisation de l'écart entre la vraie consommation des locaux et les prévisions.

Contents

List of Figures

List of Tables

General Introduction

Constraint Programming is a well-known area in Artificial Intelligence. The use of this field's approaches and formalisms, especially Constraint Satisfaction Problem (CSP), allows representing stylishly several real problems as Variables taking values in Domains and linked by Constraints, and therefore solving them smartly, by assigning those variables from their domains without violating any constraint. The CSP is operative in different domains, namely drones, logistics, medicine, and industry 4.0.

Many real problems are naturally distributed among some entities. Hence the apparition of the Distributed CSP (DisCSP) formalism. It consists of distributing the CSP parameters among a set of autonomous agents. DisCSP represents therefore the problem as a set of agents, handling each one a set of constrained variables, and connecting each other via other constraints. Many real problems can be considered as DisCSP, as distributed meeting scheduling problems [Tsuruta and Shintani, 2000; Meisels and Lavee, 2004], military unmanned aerial vehicles teams [Jung, Tambe, and Kulkarni, 2001], distributed vehicle routing problems [Léauté and Faltings, 2011], and distributed sensor network problems [Jung, Tambe, and Kulkarni, 2001; Béjar et al., 2005].

To simplify the CSP use, several CSP solvers have been created, using existing algorithms to solve the problems defined by their parameters. We quote SAT4J [Le Berre and Parrain, 2010], Abscon [Merchez, Lecoutre, and Boussemart, 2001], Mistral [Hebrard, 2008], and Choco [Jussien, Rochart, and Lorca, 2008].

To solve DisCSP problems, an important number of approaches have been proposed. The most prominent ones are the Asynchronous Backtracking(ABT) [Yokoo et al., 1992; Yokoo et al., 1998; Bessière et al., 2005], the Asynchronous Weak Commitment (AWC) [Yokoo, 1995], the Asynchronous Forward Checking (AFC) [Meisels and Zivan, 2007], and the Nogood-based Asynchronous Forward Checking (AFC-ng) [Wahbi et al., 2013].

All these algorithms are based on synchronously or asynchronously exchanging messages between agents, either to send their instantiations or to express failure. For simplification assumptions, all these algorithms assume that agents are simple; there is exactly one variable per agent.

Similarly to CSPs, there are some open-source DisCSP platforms alimented by DisCSP algorithms, allowing to solve DisCSP problems, to experiment existing DisCSP algorithms, and to create new ones, namely **F**ramework for **O**pen **D**istributed **O**ptimization (**FRODO**) [Petcu, 2006; Léauté, Ottens, and Szymanek, 2009], **AgentZero** [Lutati et al., 2014], and **DisChoco** [Ezzahir et al., 2007b; Wahbi et al., 2011]. All these platforms are based on simulating each agent by a thread.

The variety of DisCSP algorithms and applications, the agents' simplicity assumption assumed by algorithms, as well as the autonomy of the agents lead to several questions.

- How about merging the DisCSP algorithms?
- Is it possible to have a realistic DisCSP platform, used not only by developers but also by simple users?
- How to solve complex problems, when agents handle more than one variable?
- And what is the fate of the DisCSP problem's resolution, if these autonomous agents make unethical decisions?

In this thesis, we are going to tackle those questions in four parts. In the first part, we are going to contribute to the DisCSPs problems' resolution, by proposing a new DisCSP platform ensuring the resolution of the problems, the integration of new DisCSP algorithms, and the experimentation of the different algorithms. In the same part, we are going to try to capitalize on the existing DisCSPs algorithms, while trying to run more than one algorithm asynchronously and learn from each other.

The second part will broach the third question, namely DisCSPs with complex local problems. We are going to tackle the different methods and algorithms proposed to handle those problems and broaden the state of the art, by proposing new algorithms and methods.

The ethics question is going to be discussed in the third part. We are going to assess the presence of an incorrect agent into DisCSP resolution, propose the Ethical DisCSP

formalism to take the unethical agents into consideration, propose some methods to detect such agents, and take decisions against them so as the DisCSP resolution is still as normal as possible.

Once the state of art enlargement is done, either by proposing methods and algorithms, or even formalisms, we must bring intention to the applicability of what we achieved. Hence the objective of the fourth part. We are going to gradually begin applying what we realized on the smart grid problem.

List of Publications

International Journals

1. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Solving Smart Grid Local Problems Based on Constraint Optimization. International Journal of Artificial Intelligence, 2020, accepted.

2. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Maintaining ethical resolution in distributed constraint reasoning. Journal of Ambient Intelligence and Humanized Computing, 2020, 1-17, In press.

3. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Selective sorting and smart nogood-based complex distributed constraint satisfaction problems. International Journal of Artificial Intelligence, 2019, Volume 17, 1-33.

4. Imade Benelallam, Zakarya Erraji, **Ghizlane El Khattabi**, El Houssine Bouyakhf. Dynamic JChoc: A Distributed Constraints Reasoning Platform for Dynamically Changing Environments. Lecture Notes in Computer Science. International Conference on Agents and Artificial Intelligence, 2015, 20-36. Springer, Cham.

International and Mediterranean Conferences

5. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Detection of Unethical Intelligent Agents in Ethical Distributed Constraint Satisfaction Problems. Proceedings of the 2nd Mediterranean Conference on Pattern Recognition and Artificial Intelligence (52-57), 2018.

6. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Contribution to Modeling Smart Grid Problem with the Constraint Satisfaction Problem Formalism. Proceedings of the 2nd Mediterranean Conference on Pattern Recognition and Artificial Intelligence (58-62), 2018.

7. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf. Nogood Learning in DisCSP Algorithm. ICAR'17, Vol-2144, 2018.

8. **Ghizlane El Khattabi**, El Mehdi El graoui , Imade Benelallam, El Houssine Bouyakhf. MP-ABT: A Minimal Perturbation Approach for Complex Local Problems. Proceedings of the 9th International Conference on Agents and Artificial Intelligence (268-275), 2017.

9. Imade Benelallam, Zakarya Erraji, **Ghizlane El Khattabi**, Jaouad Ait Haddou, El Houssine Bouyakhf. JChoc DisSolver - Bridging the Gap Between Simulation and Realistic Use. Proceedings of the 7th International Conference on Agents and Artificial Intelligence (66-74), 2015.

International Workshops

10. **Ghizlane El Khattabi**, Imade Benelallam, El Houssine Bouyakhf, Rajae Haouari. Position Paper: Towards ethical agents in Distributed Constraint Reasoning. CPAIW 2016.

Part I

Towards Distributed Constraint Satisfaction problems

1 Background

Contents

1.1 Introduction

Programming has appeared before modern computers which were born just in the 1940s. There are several types of programming: concurrent programming[Andrews, 1991], declarative programming [Myers, Giuse, and Zanden, 1992], functional programming [Reade, 1989; Hirayama and Yokoo, 1997], imperative programming [Xi, 2000], logic programming [Apt, 1990], object-oriented programming [Rentsch, 1982], procedural programming [Brilliant and Wiseman, 1996], structured programming [Vessey and Weber, 1984], and **Constraint Programming (CP)** [Podelski, 1994].

Constraint Programming is the most recent and most ideal paradigm according to E. Freuder [1]. It appeared in 1970. It allows to represent and solve big sizes' problems, using constraints which allow to simplify and reduce the problem size.

As CP parts, several paradigms have been appeared. We can mention Constraint Satisfaction Problems (CSP) [Kumar, 1992; Tsang, 1995], Constraint Optimization Problems (COP) [Amadini, Gabbrielli, and Mauro, 2014], Valued CSPs (VCSP) [Schiex, Fargier, and Verfaillie, 1995], Distributed CSPs (DisCSP) [Yokoo, 2001] and Distributed COPs (DCOP) [Liu and Sycara, 1995].

In this chapter, we are going to present the definitions related to the CSP and DisCSP paradigms, as well as examples and resolution algorithms.

1.2 Constraint Satisfaction Problems (CSP)

A Constraint Satisfaction Problem is a mathematical problem where a solution, states or objects, satisfying a number of constraints or criteria, is/are sought.

Many real problems can be modeled as CSPs [Nadel, 1990b]. We can cite Computer-Aided Design, scheduling and timetabling problems, optimization problems, and hardware configuration.

To well understand CSP modeling and resolution methods, some concepts need to be defined in more details.

1.2.1 Definitions

Definition 1 (CSP) *A CSP is a paradigm representing a mathematical problem as a triplet {X, D, C} such that:*

[1] *"Constraint Programming is one of the closest approaches to computer science. computer solves it"* E. Freuder

- $X = X_1, X_2, ..., X_n$: *is the set of variables*
- $D = D_1, D_2, ..., D_n$: *is the set of domains. Each variable* X_i *takes its values in domain* D_i.
- $C = C_1, C_2, ..., C_m$: *is the set of constraints*

The CSP objective is to assign each variable a value so as all constraints are satisfied.

Definition 2 (Constraint) *A constraint* $c \in C$ *is a mathematical relation between a subset of variables* $X' \subset X$. *This relation can be an* ***intention*** *or an* ***extension*** *one.*

An ***intention constraint*** *is defined using known mathematical properties* $(+, -, =, >, <$ *, ...). The constraint "x is equal to y" is written as* $x = y$;

An ***extension constraint*** *is defined by quoting the tuples of values belonging to the relation. The constraint "x is equal to y" is, for example, written as* $\{(1,1),(2,2),(3,3),...\}$.

Definition 3 (Assignment) $A = \{(x_1, v_1), (x_2, v_2), ..., (x_k, v_k)\}$ *is an assignment of a subset of variables* $X_A = \{x_1, ..., x_k\} \subset X$, *when those variables are instantiated with values belonging to their domains* ($v_1 \in D_1$, $v_2 \in D_2$, ..., $x_k \in D_k$).

An assignment A can be ***total*** *(ie,* $X_A = X$*) or* ***partial*** *(ie* $X_A \neq X$*),* ***consistent*** *(ie, all constraints are satisfied) or* ***inconsistent*** *(ie. at least one constraint is violated).*

Definition 4 (CSP Solution) *A CSP solution is an assignment of all variables with values from their domains, so as the existing constraints are all satisfied. It is a total consistent assignment.*

1.2.2 CSP example

There are many known CSP examples. We can mention the problems: SEND + MORE = MONEY [Lemlouma and Boudina, 2000; Nareyek, 2001], Zebra [Salavati, Hajjarzadeh, and Mazloom, 2009], Graph coloring [Gualandi and Malucelli, 2012], N queens [Nadel, 1990a], Sudoku [Mantere and Koljonen, 2006] and Meeting Scheduling problems [Sen and Durfee, 1995].

Let take a real problem that is the Meeting Scheduling Problem (MSP) [Sen and Durfee, 1995; Garrido and Sycara, 1996], as an example. It consists of scheduling m meetings, that were attended by n attendees having each one its personal calendar, split into time slots. Figure 1.1 shows an MSP example containing five meetings and three persons attending each one a subset of these meetings. Ali should assist the meetings 1 and 2, Driss has to attend the four meetings 1, 2, 3 and 4, while Hamid has the three meetings 1, 3 and 4.

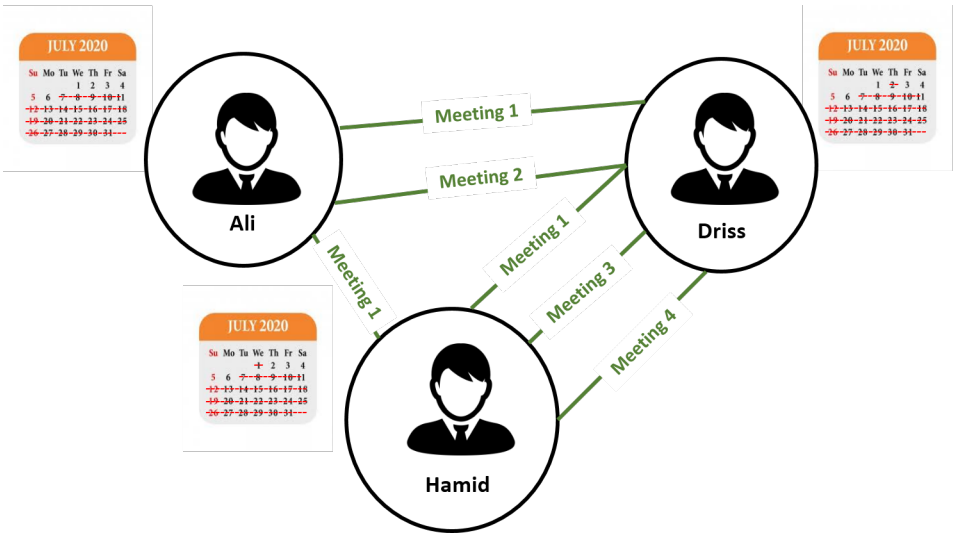


FIGURE 1.1: Meeting Scheduling Problem example

The availabilities of each attendee, the duration of each meeting, its location, as well as the displacement time between every two different meetings (Arrival-Time (AT) Constraint) should be taken into consideration.

Formally, the problem can be modeled as:

- $X = \{M_1, M_2, M_3, ..., M_m\}$ the set of m variables. Each M_i, represents a meeting. m is equal to 5 in the example shown in Figure 1.1;

- $D = \{D(M_1), D(M_2), D(M_3), ..., D(M_m)\}$ is the set of domains. $D(M_i)$ is the variable M_i 's domain. It is the shared (intersection) availabilities of the M_i's attendees.

$$D(M_i) = \bigcap_{At_j \in Attendees(M_i)} availabilities(At_j) \tag{1.1}$$

- C is the set of Arrival-Time constraints. For each person participating in at least two meetings, there is an $AT(i,j)$ constraint for each pair (M_i, M_j) of its meetings. in the example 1.1, Driss has the 4 meetings 1,2, 3 and 4, so there are 6 AT constraints $AT(1,2)$, $AT(1,3)$, $AT(1,4)$, $AT(2,3)$, $AT(2,4)$, $AT(3,4)$. $AT(i,j)$ is defined by:

$$AT(i,j) =$$
$$\|Time(M_i) - Time(M_j)\| - Duration(M_i) > Displacement(Location(M_i), Location(M_j)) \tag{1.2}$$

Once a problem is well modeled, its resolution is done via the CSP algorithms, as the BackTracking (BT) [Kumar, 1992], the Forward Checking (FC) and many other algorithms.

1.2.3 CSP Algorithms

The first algorithm, used to solve CSP problems, is Generate and Test (GT) [Kumar, 1992]. It consists of generating assignments and testing whether the generated combination satisfy the constraints or not. The first combination satisfying all the constraints is returned as a CSP solution.

The resolution with this method is blind and becomes difficult or impossible when the problems are bigger. For this, several ordered algorithms appeared, such as BackTracking (BT) [Kumar, 1992], Forward Checking (FC) [Bacchus and Grove, 1995], BackJumping (BJ) [Dechter and Frost, 2002], Arc Consistency AC-1, AC-2, AC-3, AC-4, AC-2001 [Bessiere, 1991], [Mackworth, 1977], [Mohr and Henderson, 1986], [Bessiere, 1994], [Freuder, 1995] and Maintaining Arc Consistency (MAC) algorithm [Larrosa and Schiex, 2004].

We are going to describe the BT and FC algorithms since we are based on these two algorithms in the thesis.

BackTracking (BT)

Algorithm 1 BT algorithm

```
1:  procedure BT(A,i)
2:      if X_A = X then return A
3:      else
4:          if D_1 is empty then return null
5:          else
6:              for each v_j from D_i do
7:                  if isConsistent(A ∪ (x_i, v_j)) then
8:                      A ← A ∪ (x_i, v_j);
9:                      i ← i + 1;
10:                     BT(A, i);
11:                 end if
12:             end for
13:             i ← i − 1;
14:             remove v_i from D_i;
15:             remove x_i value from A;
16:             BT(A, i);
17:         end if
18:     end if
19: end procedure
```

BackTracking [Kumar, 1992; Dechter and Frost, 1999] instantiates synchronously the variables. Each variable chooses a value from its domain and tests if all the constraints are satisfied. If so, the instantiation process passes to the next variable. Otherwise, another value is assigned to the variable. If all the variable' values are tested without finding a

consistent value, a BackTrack is made to the previous variable, to choose a new value. The algorithm ends when i) all variables are instantiated, meaning that a solution is found, or, ii) a BackTrack is performed at the first variable, meaning that the problem is insolvent.

Algorithm 1 shows the BackTracking procedure. It checks whether if the instantiation A is total, by verifying if the set X_A is exactly equal to the set X (line 2). If so, A is returned as a solution. Otherwise, the first unsigned variable is chosen and its domain is browsed to find a consistent value which satisfies all constraints (lines 6 and 7). If is that possible, it adds the chosen value to A (line 8) and a recursive call of BT is made to move on to the next variable (line 10). If the procedure does not find any consistent value, the last signed variable is removed from A (line 14), its value is deleted from its domain (line 15), and a recursive call of BT is made to find another consistent value to this variable (line 16). If the first variable's domain become empty (line 4), the problem is insolvable.

Forward Checking (FC)

Algorithm 2 FC algorithm

```
1:  procedure FC(A,i)
2:      BT(A,i);
3:  end procedure

4:  function IsConsistent(A)
5:      for each x_j from X\X_A do
6:          for each v_k from D_j do
7:              if ¬ isConsistent(A ∪ (x_j, v_k)) then
8:                  remove v_k from D_j;
9:              end if
10:         end for
11:         if D_j is empty then return false
12:         end if
13:     end for return true
14: end function
```

Forward Checking (FC) [Bacchus and Grove, 1995] algorithm improves BT algorithm by adding a propagation technique. For each instantiated variable X_i, it removes all inconsistent values from the uninstantiated variables' domains. This method allows predicting the conflict before it happens. After deleting the values from domains, the algorithm checks if there a variable domain which is empty. If this is the case, the value of the current variable X_i is changed.

Algorithm 2 shows the procedure and the function applied by the FC algorithm. The FC procedure is the same as the BT (line 2). But the consistency test made by BT is different to that for FC. *IsConsistent* function shows how FC checks if an assignment is consistent or not. For each unsigned variable (line 5), it browses its domain (line 6) to remove the inconsistent values from (lines 7 and 8). If one of the domains becomes empty the assignment A is considered inconsistent (line 11). If all unsigned variables are browsed without emptying of any domain, the assignment is considered consistent (line 13).

1.2.4 Constraint Optimization Problems (COPs)

In this report, we are going to use the COP concept in some parts, without the need to detail the resolution methods. This is why a COP definition is necessary.

A COP [Chong and Zak, 1996; Meisels, 2008] is an extension of the CSP formalism. As the CSP, the COP produces valid solutions satisfying the defined constraints. Furthermore, it allows to choose some preferred solutions. The preference is defined as a/set of function(s) to be optimized. It is/they are called Objective Function(s).

Formally, a COP is defined by the tuple (X, D, C, f), such as X, D and C are defined similarly as for CSP:

- $X = \{X_1, X_2, ..., X_n\}$: is set of n variables;
- $D = \{D_1, D_2, ..., D_n\}$: is set of domains (ie. D_i is the domain of the variable X_i);
- $C = \{C_1, C_2, ..., C_m\}$: is set of constraints which link the variables

  and

- $f : D_1 \times D_2 \times ... \times D_n \rightarrow \mathbb{R}$ is the function objective to optimize.

All CSP algorithms can solve COP problems, by looking for all solutions and keep the one which optimizes the objective functions. In addition, there are several COP algorithms as Branch and Bound [Clausen, 1999], Russian doll search [Verfaillie, Lemaître, and Schiex, 1996], and Bucket Elimination [Rollón and Larrosa, 2006].

1.3 Distributed Constraint Satisfaction Problems (DisCSPs)

The distributed nature of some problems limits the power of CSPs to model such problems adequately. Due to the complexity reasons (i.e. these problems are more complex

to be modeled in centralized way) and confidentiality (i.e. centralizing a naturally distributed problem decreases privacy), the **Distributed Constraint Satisfaction Problem (DisCSP)** concept has been created.

In DisCSPs [Yokoo et al., 1992; Yokoo et al., 1998], the parameters (X, D, C) are distributed among a set of autonomous agents in a **Multi-Agent System (MAS)** [Liu, Jing, and Tang, 2002]. These agents must exchange messages in order to find a **solution** satisfying the **intra-agent** and the **inter-agent** constraints.

1.3.1 Definitions

We are going to define terms we have already mentioned such as MAS, DisCSP, intra-agent constraint, inter-agent constraint, and DisCSP solution. Other terms will be defined as needed.

Definition 5 (MAS) *A Multi-Agent System is a set of autonomous agents, interconnected via relations. These agents share a problem and try to solve it collectively.*

Definition 6 (DisCSP) *A DisCSP is a CSP whose components (variables, domains, and constraints) are distributed among several agents (a MAS). It is defined by 5-tuple (A, X, D, C, ϕ) where:*

- $A = \{A_1, A_2, ..., A_n\}$ *is a set of n agents;*
- $X = \{X_1, X_2, ..., X_m\}$ *is a set of m variables;*
- $D = \{D_1, D_2, ..., D_m\}$ *is a set of m domains, such that D_i is the set of values that X_i can take;*
- $C = \{C_1, C_2, ..., C_p\}$ *is a set of p constraints;*
- ϕ *is the function which associates each variable to an agent.*

Definition 7 (Intra-agent constraint) *An intra-agent constraint is a constraint linking the variables of the same agent. C_{ij} is an intra-agent constraint that links X_i and X_j if and only if $\phi(X_i) = \phi(X_j)$.*

Definition 8 (Inter-agent constraint) *An inter-agent constraint is a constraint linking the variables of different agents. C_{ij} is an inter-agent constraint that links X_i and X_j if and only if $\phi(X_i) \neq \phi(X_j)$.*

Definition 9 (DisCSP Solution) *A DisCSP solution is an assignment of all variables with values from their domains, so as the inter-agent and intra-agent constraints are all satisfied.*

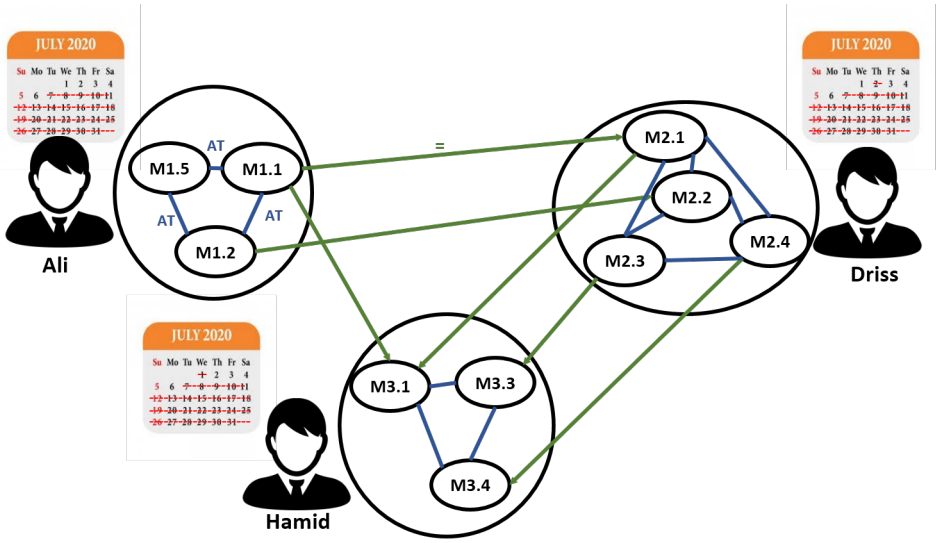


FIGURE 1.2: Distributed Meeting Scheduling Problem example

1.3.2 DisCSP Example

We are going to treat the same real problem taken in CSP example (the MSP). It was modeled and presented in section 1.2.2 as a CSP. In the CSP modelization, all informations are centralized, while the problem is distributed by nature. Generally, persons prefer to not reveal their informations for privacy reasons. This is why a DisCSP modeling of Distributed MSP (DisMSP) is proposed in [Meisels and Lavee, 2004; Tsuruta and Shintani, 2000].

Since a DisCSP problem is defined by a set of agents, in addition to the other three CSP parameters, the agents are plainly the set of persons attending the meetings. The set of variables of each agent is the different meeting attending by that agent (a meeting M_i can figure as many times as the number of participating agents). The domain of a variable/meeting is the availabilities of the agent handling the meeting. and the constraints are the Arrival-Time constraints in addition to the equality constraint which links the various apparitions of the same meeting in different agents, expressing that they represent the same meeting.

Figure 1.2 shows the DisCSP modelization of the example shown in Figure 1.1.

This example is defined by $\{A, X, D, C, \phi\}$ where:

- $A = \{A_1, A_2, A_3\}$ such as A_1 represents *Ali*, A_2 represents *Driss*, and A_3 represents *Hamid*;
- $X = \{M_{1.1}, M_{1.2}, M_{1.5}, M_{2.1}, M_{2.2}, M_{2.3}, M_{2.4}, M_{3.1}, M_{3.3}, M_{3.4}\}$;

- $D = \{D(M_{1.1}), D(M_{1.2}), D(M_{1.5}), D(M_{2.1}), D(M_{2.2}), D(M_{2.3}), D(M_{2.4}), D(M_{3.1}), D(M_{3.3}), D(M_{3.4})\}$, such as:
  * $D(M_{1.1}) = D(M_{1.2}), = D(M_{1.5}) = calendar(Ali)$,
  * $D(M_{2.1}) = D(M_{2.2}) = D(M_{2.3}) = D(M_{2.4}) = calendar(Driss)$,
  * $D(M_{3.1}) = (M_{3.3}) = D(M_{3.4}) = calendar(Hamid)$;
- $C = \{C^1_1, C^1_2, C^1_3, C^1_4, C^2_{1,2}, C^2_{1,3}, C^2_{1,4}, C^2_{1,5}, C^2_{2,3}, C^2_{2,4}, C^2_{2,5}, C^2_{3,4}\}$, such as:
  * C^1_i is the equality constraint: $M_{j.i} = M_{k.i}$
  * $C^2_{i,j}$ is the Arrival-Time (AT) constraint between $M_{k.i}$ and $M_{k.j}$ for each agent A_k. The Arrival-Time Constraint is defined by: $\|Time(M_{k.i}) - Time(M_{k.j})\| - Duration(M_{k.i}) > Displacement(Location(M_{k.i}), Location(M_{k.j}))$;
- $\phi(M_{i.j}) = A_i$.

As in CSP, there are many existing DisCSP algorithms that solve such problems.

1.3.3 DisCSP Algorithms

There are several DisCSP algorithms. Let start with the Synchronous BackTracking (SBT) [Yokoo et al., 1992] Which is the easiest one. It is based on the BackTracking algorithm [Kumar, 1992]. In the SBT, the agents instantiate its variables synchronously. The only agent that can instantiate its variables, is the one having the Current Partial Assignment (CPA) structure. After each instantiation, the agent adds the value of its variable to the CPA structure and sends it to the next agent (lower priority agent).

The synchronization is a limitation of the SBT algorithm. This is why it was necessary to create the well known and used Asynchronous BackTracking (ABT) [Yokoo et al., 1992; Yokoo et al., 1998; Bessière et al., 2005]. Contrary to the SBT, the agents instantiate their variables asynchronously.

As the asynchronism, the synchronization has also its benefits. This is the reason of the apparition of Asynchronous Forward Checking (AFC) [Meisels and Zivan, 2007]. This algorithm allows agents to instantiate their variables synchronously, and to spread the assignments asynchronously, in order to detect the failure as soon as possible. The only message that circulates among agents and handles its instantiations, is the CPA message. If an agent detects an empty domain, it generates a new Not_OK message and sends it to the agents that don't have yet a value in the CPA. The agent that receives this message and has the CPA structure, changes its values.

The authors, in [Wahbi et al., 2013], improved the AFC algorithm, by creating another new algorithm named Nogood Based AFC (AFC-ng). They are based on the fact that, in the AFC, the agents send the Not_OK messages to all agents that do not yet instantiate their variables, while it is possible to know the exact causes of the failure (the values in the CPA that causes the failure). From this fact, they proposed to send a single message nogood (Definition 10), containing the causes, to the lowest priority agent, among the responsible agents.

Definition 10 (Nogood) *A nogood is a justification of inconsistency of a variable x_k. It is written under the form $x_i = v_i \wedge x_j = v_j \wedge ... \rightarrow x_k \neq v_k$, meaning that as long as the value of x_i is equal to v_i and the value of x_j is equal to v_j, ..., x_k cannot take the value v_k.*

In this thesis, we have used the well-known algorithm ABT and the recent one AFC-ng.

Asynchronous BackTracking (ABT)

The ABT [Yokoo et al., 1992; Yokoo et al., 1998; Bessière et al., 2005] algorithm is the most well-known algorithm for solving DisCSP Problems. It is the basis of all other algorithms which have appeared after. This algorithm solves the DisCSP problem asynchronously. The order of the agents participating in the resolution is made statically beforehand, and all existing agents execute the ABT algorithm at the same time.

Algorithm 3 presents the procedures and functions used by the ABT algorithm.

When the algorithm starts, each agent chooses its first local solution, and sends it to the lower priority connected agents via an ok? message (CheckAgentView procedure line 2).

When an agent receives such a message (line 6), it stores the received value in its AgentView (ProcessInfo procedure, line 42). If the receiver AgentView contains, already, an old sender solution, the receiver updates its AgentView with the new one. After updating the AgentViews, each agent checks whether its current value is still consistent with its AgentView (CheckAgentView procedure, line 13). If so, the agent will do not anything. Otherwise, it tries to look for another consistent value in its domain(CheckAgentView procedure, line 14), while recording a nogood in its NogoodStore (ChooseValue function, line 37), justifying the inconsistency of each non-chosen value.

In the search of a new consistent value, the agent may do not find it (CheckAgentView procedure, line 19). In which case it gathers the nogoods registered in its nogoodStore

Algorithm 3 ABT algorithm

```
1: procedure ABT myvalue ← empty;end ← false;
2: CheckAgentView();
3:     while ¬end do
4:         msg ← getMsg();
5:         Switch (msg.type)
6:          Ok? : ProcessInfo(msg);
7:          ngd  : ResolveConflict(msg);
8:          stp  : end ← true;
9:          adl  : SetLink(msg);
10:    end while
11: end procedure

12: procedure CHECKAGENTVIEW(msg)
13:    if ¬consistent(myValue; myAgentView) then
14:        myValue ← ChooseValue();
15:        if myValue then
16:            for each child ∈ Γ+(self) do
17:                sendMsg : Ok? (child, myValue);
18:            end for
19:        else Backtrack();
20:        end if
21:    end if
22: end procedure

23: procedure RESOLVECONFLICT(msg)
24:    if Coherent(msg.Nogood, Γ−(self) ∪ {self}) then
25:        CheckAddLink(msg);
26:        add(msg.Nogood,myNogoodStore);
27:        myValue ← empty;
28:        CheckAgentView();
29:    else if msg.sender ∈ Γ+(self) ∧ Coherent(msg.Nogood, self) then
30:        SendMsg : Ok? (msg.Sender, myValue);
31:    end if
32: end procedure

33: function CHOOSEVALUE()
34:    for each v ∈ D(self) not eliminated by myNogoodStore do
35:        if consistent(v,myAgentView) then return (v);
36:        else
37:            add(x_j = val_j → self ≠ v, myNogoodStore);      ▷ v is inconsistent with x_j's
   value
38:        end if
39:    end for return (empty)
40: end function
```

```
41: procedure PROCESSINFO(msg)
42:     Update(myAgentView, msg.Assig);
43:     CheckAgentView();
44: end procedure

45: procedure SETLINK(msg)
46:     add(msg.sender, Γ+(self)); sendMsg:ok?(msg.sender, myValue);
47: end procedure

48: procedure CHECKADDLINK(msg)
49:     for each var ∈ lhs(msg.Nogood) do
50:         if var ∉ Γ−(self) then
51:             sendMsg:adl(var,self);
52:             add(var,Γ−(self); Update (myAgentView, var ← varValue);
53:         end if
54:     end for
55: end procedure
56: procedure BACKTRACK
57:     newNogood ← solve(myNogoodStore);
58:     if newNogood = empty then
59:         end ← true;
60:         sendMsg:stp(system);
61:     else
62:         sendMsg:ngd(newNogood);
63:         Update (myAgentView, rhs(newNogood) ← unknown);
64:         CheckAgentView();
65:     end if
66: end procedure

67: procedure UPDATE(myAgentView, newAssig)
68:     add(newAssig, myAgentView);
69:     for each ng ∈ myNogoodStore do
70:         if ¬Coherent(lhs(ng), myAgentView) then
71:             remove(ng, myNogoodStore);
72:         end if
73:     end for
74: end procedure

75: function COHERENT(nogood, agents)
76:     for each var ∈ nogood ∪ agents do
77:         if nogood[var] ≠ myAgentView[var] then
          return false;
78:         end if
79:     end for
          return true;
80: end function
```

(Backtrack procedure, line 57) to build one containing all the failure causes, and sends the newly generated nogood to the lowest priority agent, among those that exist in the resolved nogood (Backtrack procedure, line 62).

When an agent receives this message (ABT procedure, line 7), it checks if there is a value of an agent which is not linked with it. If so, it sends an **adl** message to the latter in order to add a link. Then, the receiver updates its nogoodStore (ResolveConflict procedure, line 26), and tries to find a new solution that will be consistent with its AgentView and that is not removed by any of the nogoods registered in its the nogoodStore (ResolveConflict procedure, line 28). If a solution is found, it is sent to lower priority agents. Otherwise, it sends a nogood message, in the same way as before.

After receiving a message Ok?, the update is made not only on the AgentView but also on the nogoodStore (update procedure, lines from 69 to 73), by removing the nogoods that have become obsolete (ie, they contain values which do not exist in the AgentView). This process is repeated until the algorithm stops.

The algorithm ends when a silence is detected by a master agent, which stops the resolution declaring that all agents are okay. Or when an empty nogood is found by an agent, meaning that the problem is unsolvable.

Let's apply the ABT on the example shown in Figure 1.2, supposing that each agent handles just one meeting and a reduced domain ($M_{1.2} \in \{2,3,4\}$ for A_1, $M_{2.2} \in \{1,2,3,5\}$ for A_2 and $M_{2.3} \in \{1,3,4,5\}$ for A_3) for simplification assumptions, and there is no link between A_1 and A_2. Figure 1.3 illustrates the ABT execution on the example.

In the first step (1), each agent chooses the first value from its domain and sends it the lower priority agents via Ok? message. A_1 chooses 2 and A_2 chooses 1 and send ok? messages, carrying the chosen values, to A_3. A_3 chooses 1 but does not send any message as long as it is the lowest priority agent.

In step (2), A_3 stores the received values from A_1 and A_2 in its AgentView, and remarks that its current value is not consistent with the updated AgentView. It browses its domain, value by value. It tests the first value '1', but it finds that it is inconsistent because of A_1 value. For this it stores a nogood $M_{1.2} = 2 \rightarrow M_{3.2} \neq 1$ in its nogoodStore. It does the same think for the other values, by storing a nogood for each no chosen value. After browsing all values, it finishes with a blockage. To this end, it gathers the stored nogoods and generates a summed nogood which contains the values of A_1 and A_2 $M_{1.2} = 2 \rightarrow M_{2.2} \neq 1$ (The right hand side contains the value of the lowest priority agent among A_1 and A_2) and sends it to A_2.

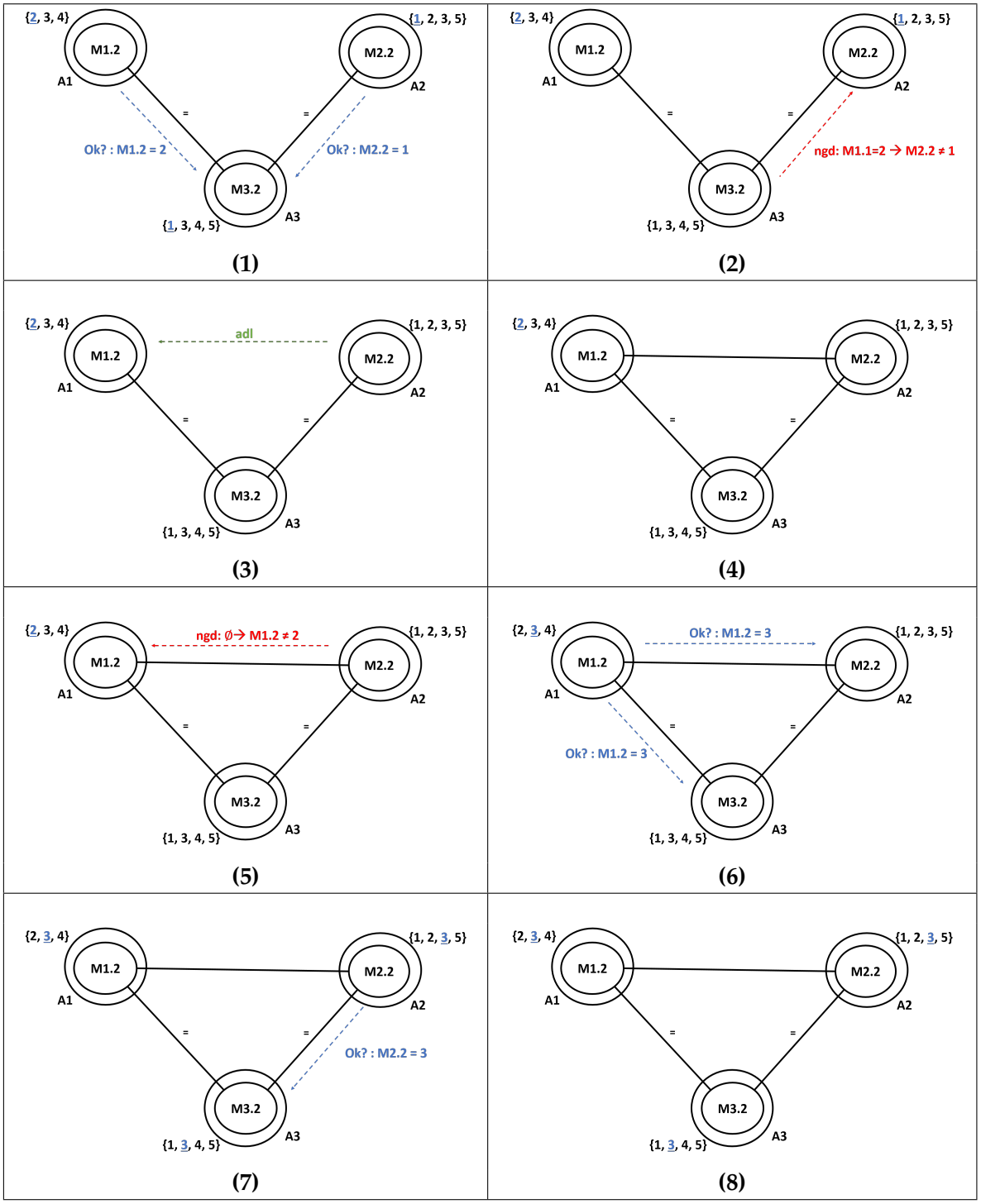


FIGURE 1.3: ABT execution example

In (3), A_2 receives the nogood, and finds that it contains the value of A_1, which is not linked with it. So, it sends an adl message to the latter.

In (4), A_1 accepts the message and adds a link with A_2.

In (5), A_2 tries to respond to the nogood message, by changing its value. But it failed because of the A_1 value. For this, it sends an ngd message to A_1, claiming to change its value.

In (6), A_1 finds another consistent value '3', which is not removed by any stored nogood, and sends it via Ok? messages to A_3 and also A_2, because it became a new child of A_1.

In (7), A_2 finds a consistent value '3' and sends it to A_3. In the same time, A_3 finds also '3' as consistent value. After receiving the new message of A_2, A_3 does not change its value, as long as it is still consistent.

After a silence (8), the solution is reported.

Nogood based Asynchronous Forward Checking (AFC-ng)

The AFC-ng algorithm [Wahbi et al., 2013] (Algorithm 4) is the most recent complete DisCSP algorithm. Like ABT, the agents' priority is made statically. But, unlike the ABT, only one agent can instantiate its variable at the same time.

At the algorithm beginning, the highest priority agent creates a CPA (Current Partial Assignments) structure (AFC-NG procedure, line 3), initializes its counter and puts its first value with the initiated counter in the CPA (Assign procedure, line 20). Then it sends the CPA to the next agent as well as the other lower priority agents which are connected to it. The agents which receive the CPA message (AFC-ng procedure, line 7) checks if the received CPA is not obsolete by comparing the counter of each agent in the received CPA with those which exist in the AgentView receiver (ProcessCPA procedure, line 44). If the CPA is new compared to the AgentView, the receiver updates its AgentView (ProcessCPA procedure, line 45), with the new received CPA and revises its domains (ProcessCPA procedure, line 47), by browsing it, value by value (Revise procedure, line 37), and adding for each inconsistent value the best nogood in the nogoodStore (Revise procedure, lines 38 and 39). The best nogood is chosen using the Highest Possible Lowest Variable (HPLV), which, between two nogoods, selects the one that will be sent to the highest priority agent.

At the end of the domain review, each agent checks whether it remains a consistent value in its domain (ProcessCPA procedure, line 49). If so, it checks whether the sending

Algorithm 4 AFC-ng algorithm

```
1:  procedure AFC-NG
2:      end ← false; AgentView.Consistent ← true ;
3:      if A_i = IA then Assign();                          ▷ IA:Initial Agent
4:      end if
5:      while ¬end do
6:          msg ← getMsg(); Switch (msg.type) do
7:           cpa : ProcessCPA(msg);
8:           ngd : ProcessNogood(msg);
9:           stp : end ← true;
10:     end while
11: end procedure

12: procedure INITAGENTVIEW
13:     for each x_j < x_i do  AgentView[j] ← {(x_j, empty, 0)};
14:     end for
15: end procedure

16: procedure ASSIGN
17:     if D(x_i) ≠ ∅ then
18:         v_i ←ChooseValue();
19:         t_i ← t_i + 1;
20:         CPA← {AgentView ∪ (x_i, v_i, t_i)};
21:     else Backtrack();
22:     end if
23: end procedure

24: procedure SENDCPA(CPA)
25:     if size(CPA) = n then  Report SOLUTION;  end ← true;
26:     else
27:         for each (x_k ∈ Γ+(x_i)) do
28:             sendMsg:cpa(CPA) to A_k;
29:         end for
30:     end if
31: end procedure

32: procedure CHECKASSIGN(sender)
33:     if predecessor(A_i = sender) then Assign();
34:     end if
35: end procedure

36: procedure REVISE
37:     for each (v ∈ D^0(x_i)) do
38:         if v is ruled out by AgentView) then
39:             Store the best nogood for v;   ▷ according to the HPLV [Hirayama and Yokoo, 2000]
40:         end if
41:     end for
42: end procedure
```

```
43: procedure PROCESSCPA(CPA)
44:     if msg.CPA is stronger than AgentView then
45:         UpdateAgentView(msg.CPA);
46:         AgentView.Consistent ← true;
47:         Revise();
48:         if D(x_i) = ∅ then Backtrack();
49:         else CheckAssign(msg.Sender);
50:         end if
51:     end if
52: end procedure

53: procedure BACKTRACK
54:     ngd ← solve(myNogoodStore);
55:     if ngd = empty then
56:         Report FAILURE;
57:         end ← true;
58:     else                                  ▷ Let x_j denote the variable in rhs(ng)
59:         for k =j+1 to i-1 do AgentView[k].value ← empty;
60:         end for
61:         for each (nogood ∈ NogoodStore) do
62:             if (¬Compatible(nogood,AgentView) ∨ x_j ∈ nogood) then
63:                 remove (nogood, NogoodStore);
64:             end if
65:         end for
66:         AgentView.Consistent ← false;
67:         v_i ← empty;
68:         sendMsg:ngd(ng) to A_j;
69:     end if
70: end procedure

71: procedure PROCESSNOGOOD(msg)
72:     if Compatible( msg.nogood, AgentView ) then
73:         add(msg.nogood,NogoodStore);          ▷ according to the HPLV
74:         if (rhs(msg.nogood).value = v_i) then
75:             v_i ← empty;
76:             Assign();
77:         end if
78:     end if
79: end procedure

80: procedure UPDATEAGENTVIEW(CPA)
81:     AgentView ← CPA;                          ▷ update values and tags
82:     for each (ng ∈ NogoodStore) do
83:         if ¬ Compatible(ng, AgentView) then
84:             remove(ng, myNogoodStore);
85:         end if
86:     end for
87: end procedure
```

agent is its predecessor (CheckAssign procedure, line 33), if so it puts its value in the CPA and sends it (Assign procedure) to the lower priority agents, in the same way. If the domain of one agent becomes empty (i.e. All values are removed by nogoods) (ProcessCPA procedure, line 48), it generates a nogood, based on the nogoods registered in its nogoodStore (Backtrack procedure, line 54) and sends it to the lowest priority agent, among those existing in the generated nogood (Backtrack procedure, line 68).

The agent receiving the nogood message (AFC-ng procedure, line 8) updates its nogoodStore, by the received nogood, while keeping the best nogood for each inconsistent value (by applying the HPLV method) (ProcessNogood procedure, line 73). The receiver checks if its domain has become empty. If so, it proceeds, in the same way, to build and send a new nogood.

In this algorithm, the agent, that owns the CPA and which is the successor of the agent sending the last CPA message, is the only one that can add its value to the CPA structure and send it to the other agents.

The algorithm ends when the CPA contains all the values of the existing agents (SendCPA procedure, line 25), meaning that a solution is found, or when an empty nogood is generated (Backtrack procedure, line 55), meaning that the problem is insolvent.

Let's take the same example taken in the ABT execution, with the same suppositions. Figure 1.4 illustrates the example's AFC-ng resolution.

In the first step (1), A_1 creates the CPA structure, takes the first value '2', adds the value to the created CPA, and sends the CPA message to the lower priority agents A_2 and A_3. Even if A_2 is not linked with A_1, it receives the CPA message of A_1, as long as it is the A_1' predecessor.

After receiving the CPA from A_1, A_3 notes, in (2), that it is not possible to choose any consistent value, because of the A_1 value. For this, it sends an ngd message to A_1.

In (3), A_1 responds the the ngd message by changing its value to '3', and sending it via CPA message to A_2 and A_3.

After receiving the first CPA from A_1, A_2 chooses a consistent value '2' in step (4), adds it to the CPA and sends it to A_3. A_3 does not take the received CPA, into consideration, because it became obsolete (the counter of the A_1 value is 1). It is inferior of the A_1's counter stored in its AgentView (it is equal to 2).

In step (5), A_2 treats the new CPA received from A_1. It chooses the value '3', adds it in the CPA, and sends it to A_3.

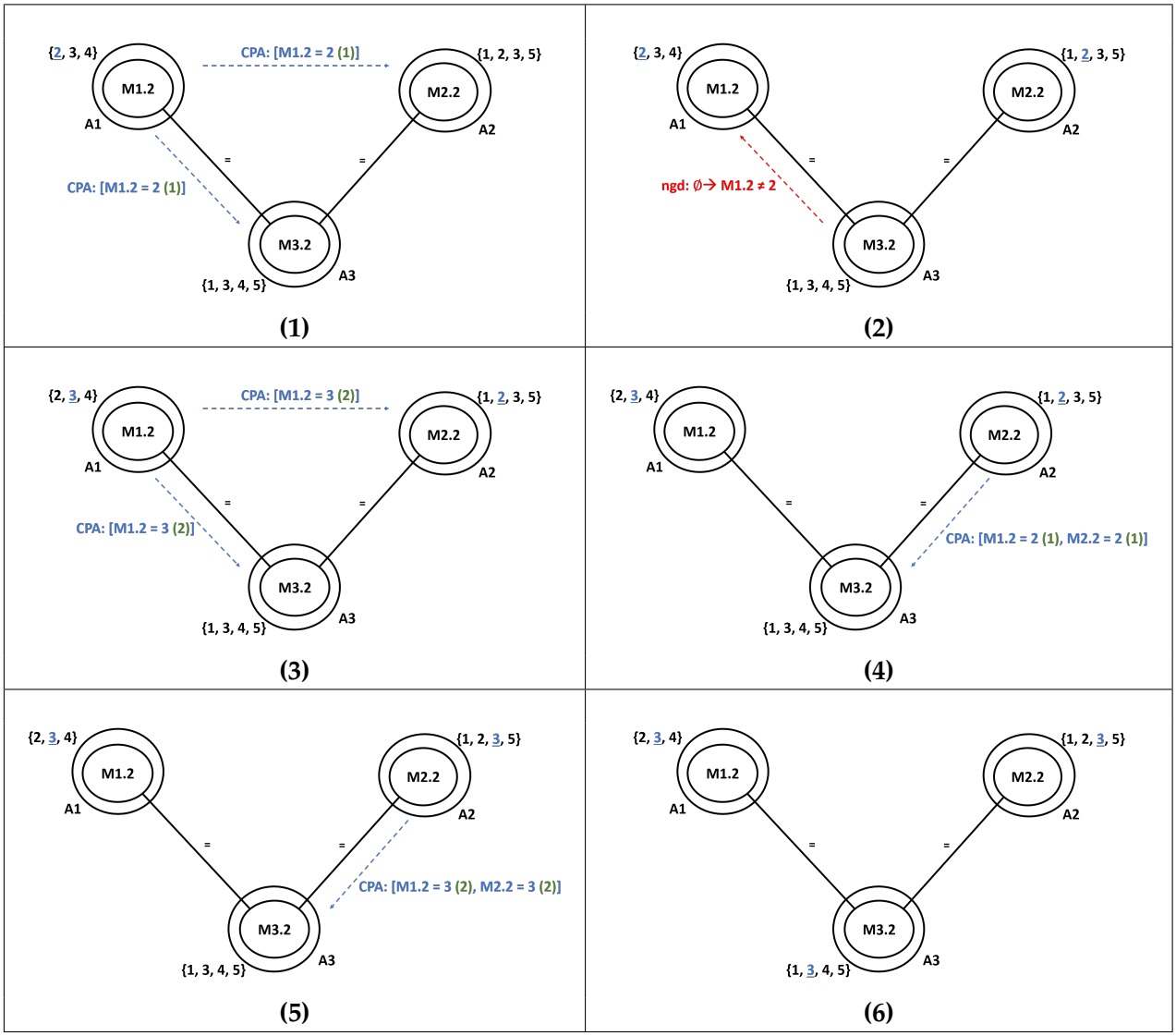


FIGURE 1.4: AFC-ng execution example

In step (6), A_3 chooses, in its role, the value '3' and adds it to the CPA. The size of the last updated CPA is equal to the number of agents. So the solution is found and reported.

1.3.4 Performance Measurement of DisCSP Algorithms

For each newly created DisCSP algorithm, its performance must be measured, by comparing it with the already existing algorithms. To do this, we compute two metrics, the number of exchanged messages MSGs and the number of Concurrent Constraint Checks CCCs (Definition 11) when the algorithm is trying to solve either random problems or real ones, as DisMSPs.

Definition 11 (Concurrent Constraint Checks CCCs) *The Concurrent Constraint Checks is a metric used to evaluate the DisCSP algorithms. It computes the number of the constraints*

checked concurrently. Each agent handles a constraint counter which is incremented after each constraint test. Each sent message carries this value. The receiver tests if the received value is greater than the its counter value. If so, it updates its own counter by the received one. When the resolution is over. The largest counter value is selected as the Concurrent Constraint Checks value.

Random problems are characterized by two parameters: the constraint tightness (Definition 12) and the constraint network density (Definition 13).

Definition 12 (Tightness) *The tightness of a constraint is the ratio of the number of its forbidden tuples and the Cartesian product of the sizes of the scope variables' domains. For a constraint reporting n forbidden tuples, of arity a, and linking variables whose domains' size is d, the tightness is equal to* $\frac{n}{d^a}$.

Definition 13 (Density) *The density of a constraint network is the number of network/DisCSP's constraints, compared to the maximum number of possible constraints. For a problem of n variables with c constraints of arity a, the density is equal to* $\frac{c}{C_n^a}$.

Based on the constraint network density, we discern between two types of random problems; sparse problems and dense ones.

1.4 Conclusion

In this chapter, we introduced the CSP and DisCSP problems, some real examples and the most known algorithms of resolution that we have used in the thesis.

In order to investigate DisCSPs, either by improving existing algorithms, or even by proposing other DisCSP algorithms, we need a DisCSP platform, gathering the existing algorithms, allowing a flexible integration of modifications, expediting the implementation of new algorithms, and containing tools to experiment the algorithms.

2

JChoc DisSolver: a Platform for DisCSP Algorithms

Contents

2.1 Introduction

One of the most advantageous points of Constraint Programming is generally the mathematical modeling. Once the problem is well modeled and defined the resolution is done via the existing solvers. Which in turn use existing algorithms, depending on the type of problem whether is CSP, COP, DisCSP or DCOP.

There are many CSP and COP solvers, implementing algorithms and allowing to solve CSP/COP problems by just declaring their parameters namely their variables, domains, and their constraints.

Unlike CSP solvers, there are few open sources DisCSP platforms that allow to: solve DisCSP problems while keeping the distributed nature of those problems, include improvements to the existing algorithms, implement new algorithms, and compare algorithms newly created with the existing ones.

In this chapter, we focus on the development of a Multi-Agent platform for Distributed Constraint Reasoning, namely **JChoc DisSolver**. This platform allows non-expert users to address and solve easily real Distributed Constraint Satisfaction Problems, as well as researchers and developers to bring improvements to the existing algorithms and even to propose new ones.

The chapter is organized as follows. Section 2.2 presents the state of art. Section 2.3 introduces the JChoc platform in static as well as dynamic environments. Section 2.4 shows the experimental results in both environments. Finally, the chapter is closed by a conclusion in section 2.5.

2.2 State of the art

The principle of Constraint Programming is modeling the problem under variables, domains, constraints, and other parameters according to the type of problem. So, the development of platforms and solvers allowing the resolution of such problems and taking into consideration their parameters, using the existing algorithms, is indispensable. There are several CSP and COP solvers such as SAT4J [Le Berre and Parrain, 2010], Abscon [Merchez, Lecoutre, and Boussemart, 2001], iZplus [Stuckey et al., 2014], Mistral [Hebrard, 2008], Ilog Solver [Lemaitre and Verfaillie, 1997], Chip [Simonis, 1995], Gecode [Schulte, Lagerkvist, and Tack, 2006], and Choco [Jussien, Rochart, and Lorca, 2008]. In the distributed case, there are few open source DisCSP / DCOP platforms, like **F**ramework for **O**pen **D**istributed **O**ptimization (**FRODO**) [Petcu, 2006; Léauté, Ottens,

and Szymanek, 2009], **AgentZero** [Lutati et al., 2014], **DisChoco** [Ezzahir et al., 2007b; Wahbi et al., 2011], and **MELY** [Orlov and Meisels, 2006].

FRODO is a MAS framework, implemented using Java language and uses several algorithms and benchmarking generators. It is based on simulating the agents in a single Java Virtual Machine (JVM), as for AgentZero.

AgentZero is also a MAS platform, supporting diverse domains, and applicable to Distributed Constraint Satisfaction and Optimization Problems. Despite the variety of application areas, but it is designed only for simulation use and it is used only by researchers.

DisChoco is a distinguished DisCSP/DCOP platform. It uses several algorithms, offers a benchmarking generator, and disposes of a graphical interface to launch and display experimentations. But as the others platforms, it is satisfied with the agents' simulation.

Melly is also a DisCSP platform implemented using Java language. Unlike other platforms, this one uses real agents' distribution using sockets. Although, it is designed only to researchers and still complex to understand so as to integrate new algorithms or to update the existing ones.

Most of these platforms are open-source and use different DisCSP and DCOP algorithms. But, for simplification assumptions, they treat DisCSP/DCOP with simple local problems, where each agent handles exactly one variable. In the case agents deal with multiple variables, each variable is considered as an agent. That can increase the resolution time and the number of exchanged messages and affect the solution.

In addition, the message exchange is simulated, since all agents' problems are collected in one site. That is not suitable because of the communication time, the cost of translation of each sub-problem in a common format, and for security and confidentiality reasons.

The centralization of all data in one single machine thwarts also developers and researchers in the integration of modifications to the existing algorithms and even in the proposition of new ones, even if the platform is open-source.

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 is the primary motivation for presenting the different ideas discussed in the present chapter.

2.3 JChoc

The effectiveness of newly proposed distributed constraints algorithms (DisCSP, DCOP, or others) is assessed by comparing it with the existing ones. Another way can prove the forceful of those algorithms is to use them in realistic multi-platform agents, so as the gap between theory and practice is reduced.

JChoc is a distributed constraints multi-agent platform proposed for:

- Solving combinatorial problems within a specific distributed environment;
- Implementing new algorithms;
- Making updates to the algorithms proposed by researches;
- Testing and comparing algorithms.

This platform is presented in the form of an Application Program Interface (API) as well as applications for different types of users. Hence, JChoc is addressed to three main users:

- **Developers:** to conceive client, web or mobile applications within distributed constraints programming, based on JChoc API;
- **Researchers:** to make updates, to add, or to experiment distributed constraints programming algorithms;
- **Non-expert user:** to interact and resolve problems based on distributed constraints programming;

To allow realistic use, agents should have a tool of communication, that allows sending and receiving multiple messages. This is possible by means of JADE model (JADE, 2013). It is designed for the development of MAS, according to the Intelligent Physical Agents (FIPA) standard. FIPA is an IEEE Computer Society standards organization instigating standards of agent-based technology with other technologies. In addition to JADE, the platform uses Choco solver in each agent, to consider even agents with multiple variables. Choco will be used to find solutions to the CSP local problem of the agent.

2.3.1 JChoc Architecture

Based on FIPA standards, JChoc an open-source Java platform, including two other open-source platforms **JADE** and **Choco**. Figure 2.1 shows the JChoc architecture.

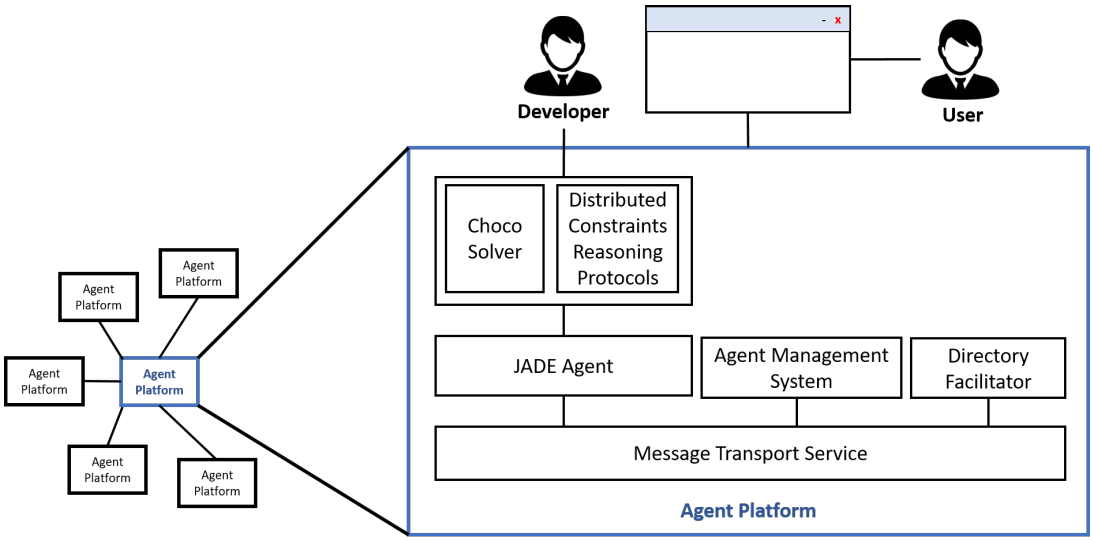


FIGURE 2.1: JChoc architecture

JChoc supports complex problems, when agents can handle multiple variables connected via intra-agent constraints. Hence, the **Choco Solver (CS)** is used to solve the local problems of these agents.

Once local problems are resolved and local solutions are identified, agents must collaborate, send messages, and react after receiving new ones, in order to find a global solution. This is done via the **Distributed Constraints Reasoning Protocols (DCRP)**, as ABT, AFC, and AFC-ng.

To ensure good communication between agents without bugs, each agent is identified with a unique ID. **JADE** awards an Agent Identifier (AID) to each agent.

The Agent Management System (AMS) plays the role of agents' monitor. It has global visibility on the platform events, the agents' authentication, their access to the system, and the exchanged messages. The information available to the AMS can be communicated to another component called **Sniffer** GUI. This Sniffer is used to display and track the various messages exchanged between a set of selected agents. Each message exchanged can be selected, viewed in detail, or even saved.

The management of the communication between agents is done via the **Message transport Service** or **Agent Communication Channel (ACC)**. In addition, the platform provides a service of "yellow pages" via the Directory Facilitator (DF).

The platform is used in a static environment and can be used even in dynamic problems when agents, variables, domains or constraints can be changed, deleted or added during the resolution process.

2.3.2 JChoc in a Static Environment

To use the platform, three mean components have to be declared, **the agents** participating in the resolutions, their **sub-problems**, and the **master agent** which will monitor the resolution by deciding when launching the resolution and when stopping it. To illustrate how these components are declared, we are going to use the DisMSP problem, taken in chapter 1 figure 1.2, as example. The problem is composed of three agents Ali: A_1, Driss: A_2, and Hamid: A_3, handling each one its personal local problem, containing variables, domains, intra-agent constraints and connecting with the others via inter-agent constraints.

Defining Sub-Problems

To participate in the resolution, an XML file is prepared for each agent, containing its local problem and the necessary information to communicate with its neighbors. Figure 2.2 represents the XML file of Driss(A_2)' sub-problem.

Firstly, the file exposes the domain. In case every variable is defined in a different domain, the number of domains is introduced and every domain is put with a different name. To define a domain, two ways are possible. In case, the values of the domains are continuous, we put the first and the last value separated by "..". In case they are discrete, all values are mentioned and separated with a ";".

For variables, the number of variables has to be mentioned and every variable has to be defined by a unique name, a unique id, and the name of the domain, in which the variable is defined (D_1 in Figure 2.2). The name is a concatenation of a character, the id of the owner agent, and the identifier of the variable in the current sub-problem. The variable $M_{2.3}$ is the 3^{th} variable of A_2 (2 is the A_2 id and 3 is the variable id in A_2' sub-problem).

For constraints, two constraints' types are possible, intention constraints with predefined functions as: equal $eq(X, Y)$, greater than $gt(X, Y)$, greater than or equal $ge(X, Y)$, lower than $lt(X, Y)$, lower than or equal $le(X, Y)$, and extension constraints by defining the tuples of values in a relation and mentioning the type of the relation either it is forbidden or allowed.

In the example, there are 6 intention constraints of Arrival-Time constraints, which are defined as a predicate in the XML file. These constraints link $M_{2.1}$ with $M_{2.2}$, $M_{2.3}$ and $M_{2.4}$; $M_{2.2}$ with $M_{2.3}$ and $M_{2.4}$; and $M_{2.3}$ with $M_{2.4}$.

```
<?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="5">1;3;4;5;6</domain>
   </domains>

   <variables nbVariables="4">
      <variable name="M2.1" id="1" domain="D1" description="M_1" />
      <variable name="M2.2" id="2" domain="D1" description="M_2" />
      <variable name="M2.3" id="3" domain="D1" description="M_3" />
      <variable name="M2.4" id="4" domain="D1" description="M_4" />
   </variables>

   <constraints nbConstraints="6">

      <constraint model="TKC" name="C0" reference="ArrivalTime"
         scope="M2.1 M2.2" arity="2">
         <parameters>M2.1 M2.2 1</parameters>
      </constraint>
      <constraint model="TKC" name="C1" reference="ArrivalTime"
         scope="M2.1 M2.3" arity="2">
         <parameters>M2.1 M2.3 1</parameters>
      </constraint>
      <constraint model="TKC" name="C2" reference="ArrivalTime"
         scope="M2.1 M2.4" arity="2">
         <parameters>M2.1 M2.4 1</parameters>
      </constraint>
      <constraint model="TKC" name="C3" reference="ArrivalTime"
         scope="M2.2 M2.3" arity="2">
         <parameters>M2.2 M2.3 1</parameters>
      </constraint>
      <constraint model="TKC" name="C4" reference="ArrivalTime"
         scope="M2.2 M2.4" arity="2">
         <parameters>M2.2 M2.4 1</parameters>
      </constraint>
      <constraint model="TKC" name="C5" reference="ArrivalTime"
         scope="M2.3 M2.4" arity="2">
         <parameters>M2.3 M2.4 1</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="C3" reference="eq"
                  scope="M2.1 M1.1" arity="2">
                  <parameters>M2.1 M1.1</parameters>
               </constraint>
               <constraint model="TKC" name="C4" reference="eq"
                  scope="M2.2 M1.2" arity="2">
                  <parameters>M2.2 M1.2</parameters>
               </constraint>
            </constraints>
         </agent>
      </agents_parent>

      <agents_children>
         <agent name="A3" id="3" variable="M2.1 M2.3 M2.4" />
      </agents_children>


   </agents_neighbours>
</instance>
```

FIGURE 2.2: Definition of DMS sub-problem in XDisCSP format

To allow an agent to communicate with others, a list of its neighbors must be entered. This list is divided into two types. *i)* Parent agents' list which contains higher priority agents, as well as, the inter-agent constraints linking the current agent with each of these agents. And *ii)* Child agents' list which includes the lower priority agents, as well as the names of variables constrained with those agents.

Defining the Master

JChoc architecture allows functioning in a really distributed environment. Each agent can be launched in a separate physical machine, knowing that an agent is represented by the predefined sub-problem XML file. All the agents must connect in the same network where the master is running, which can be executed in another physical machine too.

Figure 2.3 shows how the master agent launches its communication interface listening on the network. We start by instantiating the dissolver object (line 7). This class modelizes the distributed problem when JChoc is used to solve a problem in a real distributed environment. All information on the 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). Afterward, we trigger the container and we set the number of agents to follow (lines 9 and 10). Finally, we launch the master (lines 10).

```
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(3);
11       js.run();
12    }
13
14 }
```

FIGURE 2.3: How the master launches its communication interface.

Defining the agents

Figures 2.4, 2.5, and 2.6 show how to launch agents Ali : A_1, Driss : A_2, and Hamid A_3. As the JChoc Master definition, the agent launching starts with the DisSolver object (line 6), pursued by the agent type (line 8). Then we declare the agent, by adding its name and its sub-problem XML file (line 9). Afterward, we associate the agent to the network by declaring the container containing the master with its IP address (line 10). Finally, we launch the agent.

```
1  package DisSolver;
2
3  public class Ali
4  {
5     public static void main(String[] args) {
6        DisSolver ds = new DisSolver();
7
8        ds.setType("AgentABT");
9        ds.addAgent("A1","problem1.xml","A1",true, true);
10       ds.setContainer("192.168.59.2");
11       ds.run();
12    }
13 }
```

FIGURE 2.4: Launching the agent "Ali : A_1"

```
1  package DisSolver;
2
3  public class Driss
4  {
5     public static void main(String[] args) {
6        DisSolver ds = new DisSolver();
7
8        ds.setType("AgentABT");
9        ds.addAgent("A2","problem1.xml","A2",true, true);
10       ds.setContainer("192.168.59.2");
11       ds.run();
12    }
13 }
```

FIGURE 2.5: Launching The agent "Driss : A_2"

```
1  package DisSolver;
2
3  public class Hamid
4  {
5     public static void main(String[] args) {
6        DisSolver ds = new DisSolver();
7
8        ds.setType("AgentABT");
9        ds.addAgent("A3","problem1.xml","A3",true, true);
10       ds.setContainer("192.168.59.2");
11       ds.run();
12    }
13 }
```

FIGURE 2.6: launching The agent "Hamid : A_3"

Once all agents are connected, the master sends them a start message to launch a research protocol in order to find the solution. In the platform's first version, they use the ABT algorithm.

Figures 2.7 and 2.8 show the sniffer agent at the beginning and the end of the ABT resolution. A message is represented with an arrow, and the message' type is identified by a different color. The blue represents ABT's Ok? messages.

2.3.3 JChoc in a Dynamic Environment

The use of JChoc in a dynamic environment; when some agents, variables, domains, or/and constraints can be changed anytime during the resolution process; is similar to that of a static environment. The difference is only in the sub-problem XML file of agents.

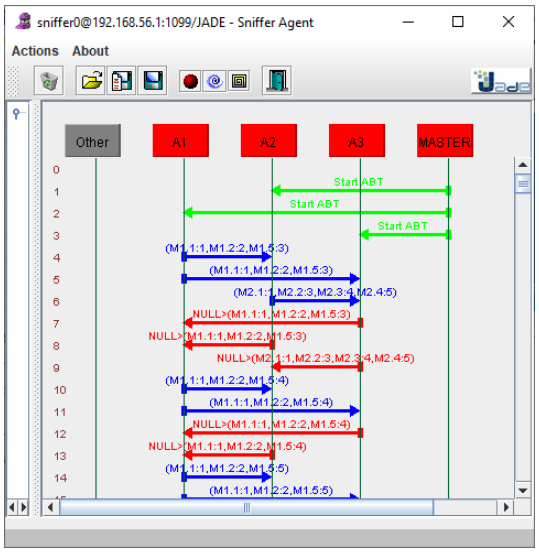


FIGURE 2.7: The start on sniffer agent

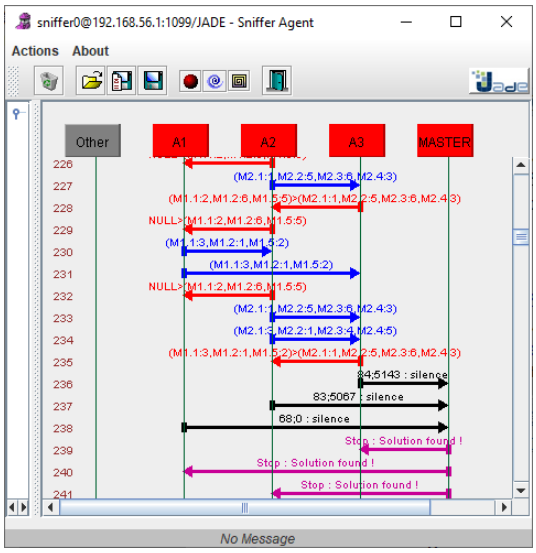


FIGURE 2.8: The finish on sniffer agent

Let take the same example used in the static environment, and suppose that the constraint linking Driss' $M_{2.1}$ with Ali' $M_{1.1}$ will be added after 4 seconds and the constraint connecting $M_{2.2}$ with Ali' $M_{1.2}$ will be removed after 6 seconds (Figure 2.9).

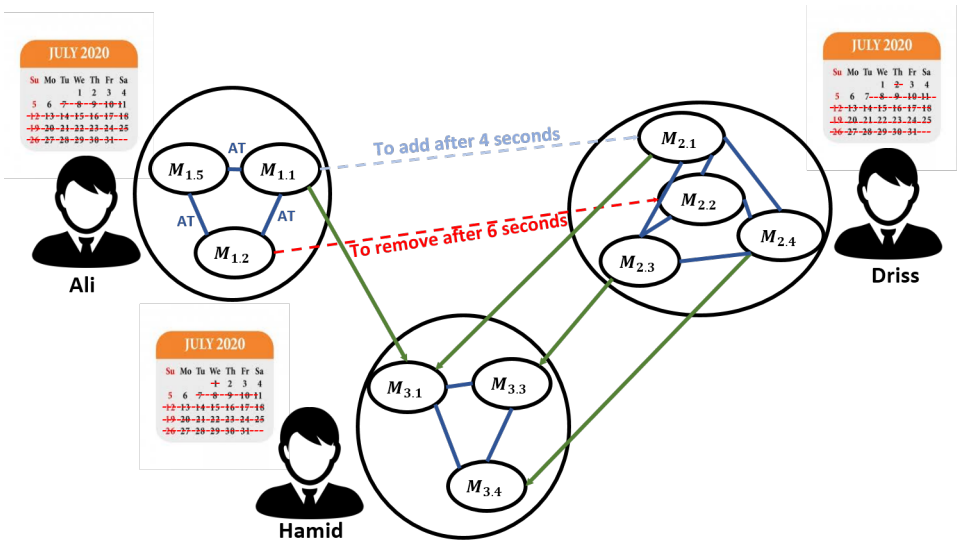


FIGURE 2.9: Dynamic Distributed MSP example

Figure 2.10 shows how the sub-problem XML file of the agent Driss is updated, so as it takes the last cited changes into consideration.

Once the sub-problem XML files are defined, all other steps of launching the master and the agents are still the same as in a static environment.

```
<?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="5">1;3;4;5;6</domain>
   </domains>

   <variables nbVariables="4">
      <variable name="M2.1" id="1" domain="D1" description="M_1" />
      <variable name="M2.2" id="2" domain="D1" description="M_2" />
      <variable name="M2.3" id="3" domain="D1" description="M_3" />
      <variable name="M2.4" id="4" domain="D1" description="M_4" />
   </variables>

   <constraints nbConstraints="6">

      <constraint model="TKC" name="C0" reference="ArrivalTime"
         scope="M2.1 M2.2" arity="2" change = "no">
         <parameters>M2.1 M2.2 1</parameters>
      </constraint>
      <constraint model="TKC" name="C1" reference="ArrivalTime"
         scope="M2.1 M2.3" arity="2" change = "no">
         <parameters>M2.1 M2.3 1</parameters>
      </constraint>
      <constraint model="TKC" name="C2" reference="ArrivalTime"
         scope="M2.1 M2.4" arity="2" change = "no">
         <parameters>M2.1 M2.4 1</parameters>
      </constraint>
      <constraint model="TKC" name="C3" reference="ArrivalTime"
         scope="M2.2 M2.3" arity="2" change = "no">
         <parameters>M2.2 M2.3 1</parameters>
      </constraint>
      <constraint model="TKC" name="C4" reference="ArrivalTime"
         scope="M2.2 M2.4" arity="2" change = "no">
         <parameters>M2.2 M2.4 1</parameters>
      </constraint>
      <constraint model="TKC" name="C5" reference="ArrivalTime"
         scope="M2.3 M2.4" arity="2" change = "no">
         <parameters>M2.3 M2.4 1</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="C3" reference="eq"
                  scope="M2.1 M1.1" arity="2" change = "add_4">
                  <parameters>M2.1 M1.1</parameters>
               </constraint>
               <constraint model="TKC" name="C4" reference="eq"
                  scope="M2.2 M1.2" arity="2" change = "remove_6">
                  <parameters>M2.2 M1.2</parameters>
               </constraint>
            </constraints>
         </agent>
      </agents_parent>

      <agents_children>
         <agent name="A3" id="3" variable="M2.1 M2.3 M2.4" />
      </agents_children>


   </agents_neighbours>
</instance>
```

FIGURE 2.10: Definition of Dynamic sub-problem in XDisCSP format

2.4 Experimental Results

2.4.1 Performance of JChoc in Static Environment

Since JChoc uses the already existing algorithms in a physically distributed environment, we are going to assess JChoc against the number of messages and the resolution time, in order to evaluate if the real distribution will beget more message exchanging, or more resolution time comparing to the normal when the algorithm is launched in a single machine.

The exchange of more messages is explained by the fact that messages are received lately. The consumption of more time means the use of the communication network is no longer good.

To assess our platform, we consider a large number of DisMSP instances. They are characterized by $< p; m; n; d; h; t; a >$:

p is the number of participants;

m is the number of meetings;

n is the number of inter-agent constraints;

d determines the number of days;

h is the number of hours per day;

t is the duration of the meetings;

a is the percentage of availability for each participant.

The instances we take belong to the class $< p; m; n; 5; 10; 1; 90\% >$ such as the three parameters p, m, and n vary, and the available meetings are distributed among the agents, so as there are 2 meetings per agent.

Table 2.1 shows the number of messages; and the run time execution, according to the three parameters p, m, and n.

The results show that JChoc performs rapidly in small instances when the number of participants does not exceed 14 ($\#p < [4, 14]$). The number of messages increases when the number of participants is between 15 and 18 ($\#p \in [15, 18]$) and reduces when p exceeds 18 ($\#p > 18$). This scalability behavior is due to the complexity of DisMSP problems. When the instance is dense the problem can be solved rapidly.

Participants #p	Meetings #m	Intra-agent constraints #n	Number of 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

TABLE 2.1: Performance of JChoc platform using ABT protocol on the Meeting Scheduling Problem (MSP).

This behavior is the same as ABT in a simulation environment. That proves the scalability and effectiveness of JChoc even if it uses a realistic environment.

2.4.2 Performance of JChoc in Dynamic Environment

If a problem parameter is changed during the resolution, the process relaunches from scratch, to take the new updates into consideration. This is when the platform does not support dynamic problems. It uses only DisCSP algorithms. But in a dynamic environment, the changes should be real-time taken into consideration during the resolution process. This is possible if a Dynamic DisCSP algorithm is used.

So, to assess the performance of JChoc in dynamic problems, we made our experiments using ABT that can't solve such a problem dynamically, and Dynamic ABT (Dyn-ABT) [Omomowo, Arana, and Ahriz, 2008] that can adapt the problem and continue the solving process.

The experiments are done against random problems. They are characterized by $< a; i; n; d; p_1; p_2; \delta >$:

a: the number of agents = 20;

i: the number of instances = 10;

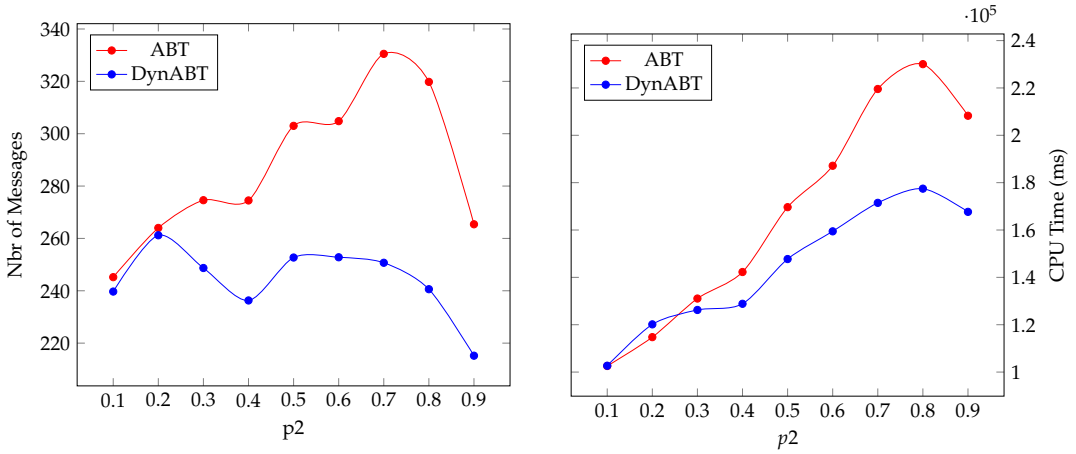


FIGURE 2.11: ABT vs DynABT

n: the number of variables = 20;

p_1: the density of constraints = 20%;

p_2: the tightness of constraints. It varies from 10% to 90% with 10% as step;

δ: the change rate = 20%. It represents the ratio of changed constraints against the constraints shaping the problem.

The range of tightness [10%, 40%] contains solvable problems, 50% contains both solvable and unsolvable problems, and [60%, 90%] are unsolvable problems.

Figure 2.11 shows the number of exchanged messages and the execution run time, spent by ABT vs DynABT. The obtained results show that Dyn-ABT outperforms significantly ABT in the dynamic changing environment. This comparison shows the benefit of solving dynamic distributed problems in a real distributed changed environment with a dynamic DisCSP algorithm, implemented in a suitable platform.

2.5 Conclusion

In this chapter, we have presented a modular distributed constraints platform, called JChoc DisSolver. It benefits the power of the Choco solver, the multi-agent platform JADE, the DisCSP formalism, and also the DCOP algorithms. It can be used easily for static as well as dynamic combinatorial problems.

Comparing to other platforms, JChoc offers the possibility to construct new constraints based applications, to easily update the existing implemented algorithms, to implement other algorithms, and to propose new ones. The platform is designed to support secured, Complex, and also ethical algorithms and applications.

The experimental results have proven the power of JChoc to support and resolve real problems, in static and dynamic environments, thanks to DisCSP and Dynamic DisCSP' algorithms.

Before engaging in the proposition of new algorithms, we are going to implement another existing DisCSP algorithm AFC-ng, in addition to ABT, and try to take advantage of the commonalities between these algorithms.

3

Nogood Learning in DisCSP Algorithms

Contents

3.1 Introduction

As we saw in Chapter 1, several existing DisCSP algorithms can resolve any DisCSP problem effectively. The only difference between those algorithms is the performances of each one, namely the number of exchanged messages and the non-concurrent constraints checks. It depends on the treated problem.

The performance of an algorithm is not absolute. For example, AFC-ng is proven better than ABT in dense problems. Otherwise, is ABT the best. So, before delving into the proposal of a new algorithm, let try to reap the benefits of two different algorithms.

The idea is to use machine learning [Michie, Spiegelhalter, and Taylor, 1994; Mitchell, 1997; Alpaydin, 2020] principle, which uses two different data concepts, namely training and testing. In our contribution, the two concepts are going to be replaced by the two algorithms ABT and AFC-ng.

The choice of the training algorithm or the testing one is not going to be frozen statically at the start. The pick is going to be done dynamically and cleverly during the resolution process. At a given point, the fast algorithm which finds a failure the first will be the training one. This allotment of rules is not definitive. It is changed during the resolution process.

We are going to describe the contribution details more accurately. Hence, the chapter proceeds as follows. Section 3.2 presents the justification for the two algorithms ABT and AFC-ng choice, as well as the two learning methods we proposed. Section 3.3 shows the experimental results of problems with lower as well as higher density. Finally, the chapter is concluded in section 3.4.

3.2 Nogood Learning Based ABT and AFC-ng

3.2.1 Why ABT and AFC-ng?

We have chosen ABT and AFC-ng because they have multiple common properties:

- **The AgentView content**: In the two algorithms, the agents have the same AgentView structure. It contains the higher priority agent assignments. The only difference resists in the use of counters by AFC-ng agents. It is still a piece of additional information, not influencing ABT in any term adversely;

- **The nogood structure**: the nogoods generated by the two algorithms have the same architectures. They are composed of two sides. The Left Hand Side (LHS) is built from the conjunction of higher priority agents' values causing the failure. The Right Hand Side (RHS) represents the prohibited value because of the LHS combination;

- **The NogoodStore content**: In the two algorithms, agents have the same NogoodStore structure. It contains the nogoods generated by the agent itself when a tested value is inconsistent, and the nogoods received from lower priority agents;

These common points are the guides to launch the two algorithms in the same DisCSP problem, to learn, to collaborate, and to find a solution, either with less message exchanged and fewer tested constraints or in a minimum of time. Based on nogoods, we did that using two different learning ways, The two learning methods differ in how the two algorithms are running concurrently.

3.2.2 Learning-1: First Nogood Learning Method

In the first method, the two algorithms are running synchronously. All agents start the ABT algorithm. At the same time, the initial agent, holding the highest priority, generates and sends the CPA structure to start the AFC-ng algorithm too.

In that case, each agent can receive and send five types of messages:

- CPA;
- Ok?;
- ngd_AFCng: nogood message sent using AFC-ng algorithm;
- ngd_ABT: nogood message sent using ABT algorithm;
- stp: stop message. It can be sent either by ABT or AFC-ng master.

For each received message, the applied procedure is the same as the original algorithm. The algorithm 5 presents only procedures and functions we changed.

Algorithm 5 Learning-1

```
1: procedure ABT-AFCNG SHARING 1
2:     end ← false; AgentView_AFCng.Consistent ← true ;
3:     if A_i = IA then
4:         Assign();
5:     end if
6:     CheckAgentView();
7:     while ¬end do
8:         msg ← getMsg();
9:         Switch (msg.type) do
10:         cpa          : ProcessCPA(msg);
11:         ngd_AFCng    : ProcessNogood(msg);
12:         Ok?          : ProcessInfo(msg);
13:         ngd_ABT      : ResolveConflict(msg);
14:         stp          : end ← true;
15:    end while
16: end procedure

17: procedure BACKTRACK_ABT
18:     newNogood ← solve(myNogoodStore);
19:     if newNogood = empty then
20:         the same ABT treatment;
21:     else
22:         the same ABT treatment;
23:         add(newNogood, SentNogoodsByABT)
24:     end if
25: end procedure

26: procedure BACKTRACK_AFCNG
27:     ngd ← solve(myNogoodStore);
28:     if ngd = empty then
29:         The same AFC-ng Treatment;
30:     else
31:
32:         The same AFC-ng Treatment;
33:         add(ngd, SentNogoodsByAFCng);
34:     end if
35: end procedure
```

```
36: function CHOOSEVALUE_ABT()
37:     for each ngd ∈ SentNogoodsByAFCng do
38:         if (istheAgentViewAlreadyInconsistent(ngd) then
39:             Clear NogoodStore_ABT;
40:             for each (v ∈ D(self)) do
41:                 nogood ← ngd.lhs∧ngd.rhs→ x_i = v;
42:                 add (nogood, nogoodStore_ABT);
43:             end for
44:             return null;
45:         end if
46:     end for
47:     for each v ∈ D not eliminated by NogoodStore_ABT do
48:         if v is eliminated by a coherent nogood from NogoodStore_AFCng then
49:             add(ngd, NogoodStore_ABT);
50:         else
51:             if consistent(v,myAgentView) then return (v);
52:             else
53:                 add(x_j = val_j → self ≠ v, NogoodStore_ABT);
54:                                          ▷ v is inconsistent with x_j's value
55:             end if
56:         end if
57:     end for return (empty)
58: end function

59: procedure REVISE
60:     for each ngd ∈ SentNogoodsByABT do
61:         if (istheAgentViewAlreadyInconsistent(ngd) then
62:             for each (v ∈ D(self)) do
63:                 ngd ← ngd.lhs∧ngd.rhs→ x_i = v;
64:                 if v is eliminated by nogoodStore_AFCng then
65:                     keep the best nogood between the eliminating nogood and ngd;
66:                                                  ▷ According to the HPLV
67:                 else
68:                     add (ngd, nogoodStore_AFCng;
69:                 end if
70:             end forreturn null;
71:         end if
72:     end for
73:     for each (v ∈ D^0(x_i)) do
74:         if (v is ruled out by AgentView or eliminated by nogoodStore_ABT ) then
75:             Store the best nogood for v;
76:                                  ▷ according to the HPLV [Hirayama and Yokoo, 2000]
77:         end if
78:     end for
79: end procedure
```

In addition to AgentView_ABT, AgentView_AFCng, NogoodStore_ABT and NogoodStore_AFCng structures, each agent creates two new structures SentNogoodsByABT to store the nogoods sent using ABT algorithm and SentNogoodsByAFCng which will contain the nogoods sent using AFCng. When an agent creates and sends a new nogood using ABT(backtrack_ABT procedure), it stores the last in the new structure SentNogoodsByABT. The same treatment is done for nogoods generated using AFC-ng.

Before trying to choose a value, using ABT (ChooseValue_ABT() function), or revising the domain, using the AFC-ng algorithm (Revise() procedure), the agent checks if it has already sent a nogood using the other algorithm, which is coherent with its AgentView. If so, ChooseValue_ABT procedure clears the NogoodStore_ABT, generates a new nogood whose left-hand side is the nogood's components conjunction (LHS and RHS), and the right-hand side is empty. Then, it browses the agent whole domain, creates a copy of the last nogood for each value, and edits the RHS of the nogood copy by the tested value, to store in the current algorithm NogoodStore (NogoodStore_ABT). Finally, it returns null, signifying that there is no consistent value, without testing any constraint since the current agentView has already been tested by the other algorithm (AFC-ng).

Otherwise, the procedure browses the domain, value by value, and checks if there is a nogood in the nogoodStore of the other algorithm (NogoodStore_AFCng) which is compatible with the AgentView_ABT and eliminating the tested value.

If so, it adds the found nogood in the NogoodStore_ABT. Otherwise, it checks whether the tested value is eliminated by the current algorithm's NogoodStore (NogoodStore_ABT) or not.

If this is not the case, it tests if the value is consistent with AgentView_ABT. If so, it returns the value, otherwise, it stores a nogood, as ever.

For the Revise() method, it tests if there is a nogood in the SentNogoodByABT structure, which is coherent with the AgentView_AFC-ng. If so, it browses the domain to construct a new nogood, whose left-hand side is the LHS and RHS conjunction of the found nogood and the RHS is the tested value. If the tested value is already removed by a nogood (in NogoodStore_AFCng), the method keeps the better nogood (the constructed nogood or the found one), according to the HPLV method. Otherwise, it adds the generated nogood to the NogoodStore_AFCng.

If there is no nogood sent by the ABT algorithm, the Revise() procedure browses the domain. It not only checks whether the value is inconsistent with AgentView_AFCng but also if it is eliminated by nogoodStore_ABT. If so, it keeps the best nogood using the

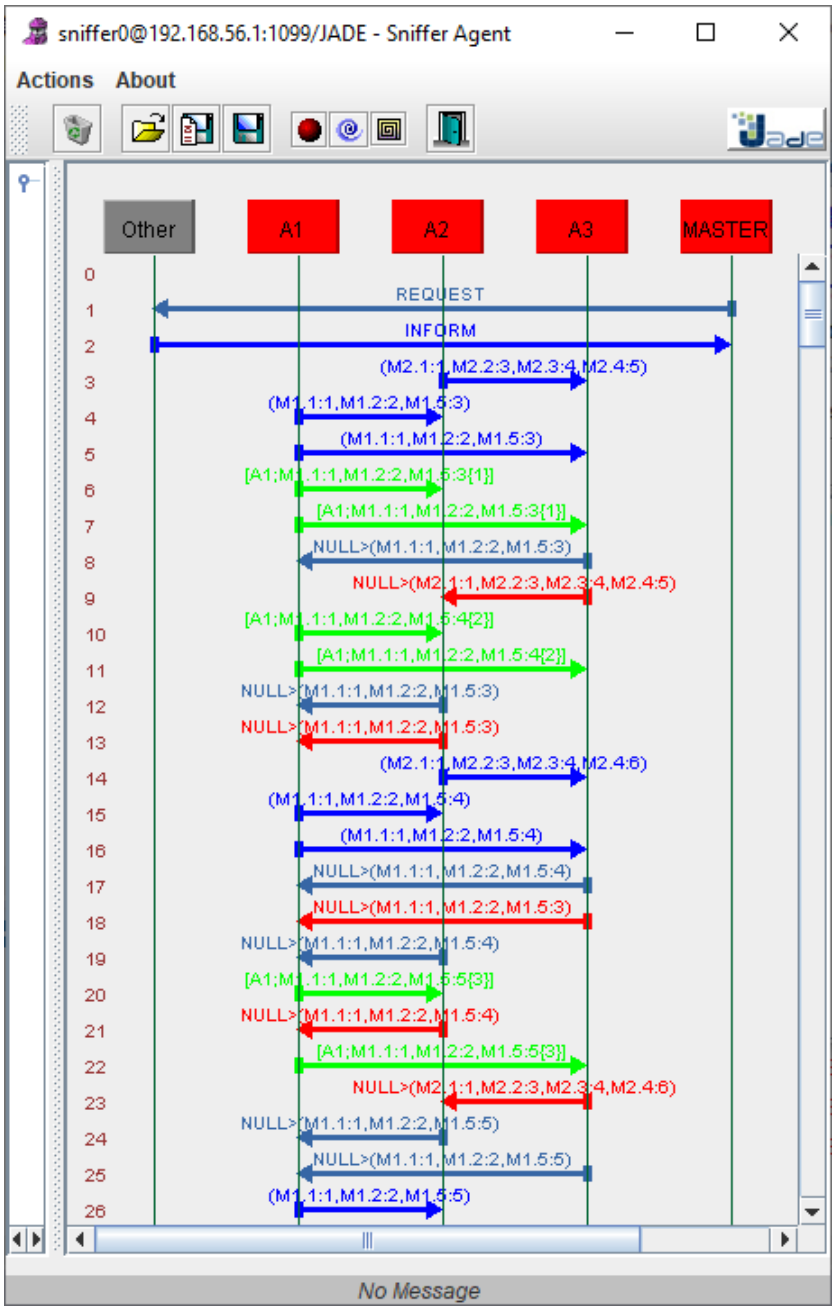


FIGURE 3.1: The problem 1.2 resolution using Learning-1 method

HPLV method.

Figure 3.1 presents some exchanged messages using the Learning-1 method to resolve the problem (Figure 1.2 of Chapter 1). The light blue arrows represent the OK? messages, the green ones show the CPA messages, the reds indicate the nogood messages sent by the ABT algorithms, and the gray arrows represent the nogoods sent by AFC-ng.

The two nogood messages 12 and 13 are the same. The first one sent using the AFC-ng and the second one using ABT. The second message is generated not from checking constraints, but from the first nogood that is coherent with ABT's AgentView.

3.2.3 Learning-2: Second Nogood Learning Method

The second learning method follows the same methodology as the first method. The only difference is noticeable when one of the two algorithms generates a non-empty nogood. Before sending or adding the nogood in SentNogoods structures (SentNogoodsByABT or SentNogoodsByAFCng), the agent checks if the generated nogood was not already sent by the other algorithm.

If an algorithm determines that the newly generated nogood has already been sent by the other algorithm, it stops the resolution and lets the other to continue its resolution process, assuming that the first algorithm finding the nogood is the fastest.

Algorithm 6 Learning-2

```
1:  procedure BACKTRACK_AFCNG
2:      ngd ← solve(myNogoodStore);
3:      if ngd = empty then
4:          The same AFC-ng Treatment;
5:      else
6:          if ¬ SentNogoodsByABT contains ngd then
7:              The same AFC-ng Treatment;
8:              add(ngd, SentNogoodsByAFCng);
9:          end if
10:     end if
11: end procedure

12: procedure BACKTRACK_ABT
13:     newNogood ← solve(myNogoodStore);
14:     if newNogood = empty then
15:         the same ABT treatment;
16:     else
17:         if ¬ SentNogoodsByAFCng contains ngd then
18:             the same ABT treatment;
19:             add(newNogood, SentNogoodsByABT)
20:         end if
21:     end if
22: end procedure
```

Figure 3.2 presents some exchanged messages using the Learning-2 method to resolve the problem 1.2 of Chapter 1.

The figure shows that from message 16 onwards, only ABT sends messages. Because after receiving the two CPA messages 14 and 15, the agent A_3 tried to generate a nogood, using AFC-ng and noticed that it has already sent the generated nogood using ABT (message 16). However, it stops AFC-ng and lets ABT continue the resolution.

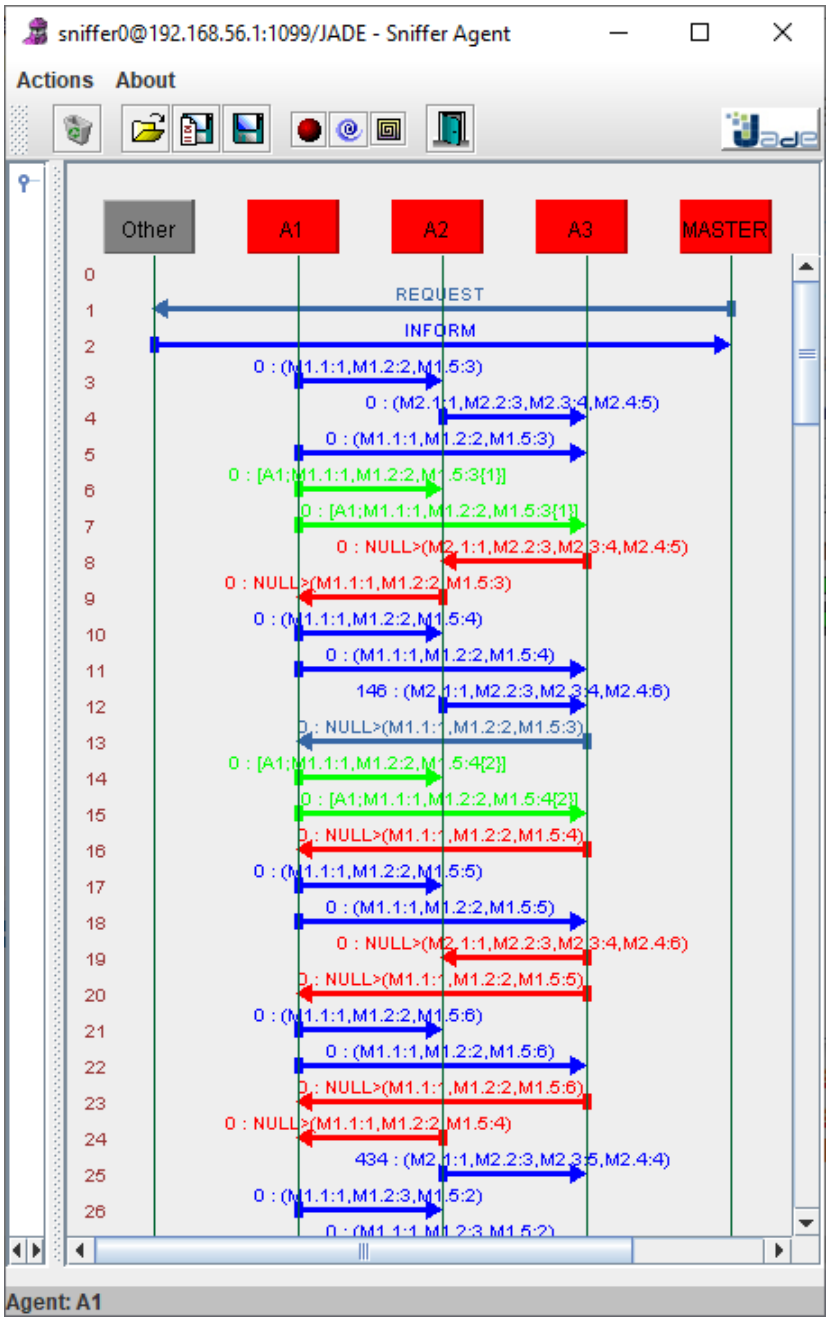


FIGURE 3.2: The problem 1.2 resolution using Learning-2 method

3.3 Experimental results

To assess the nogood's learning methods, we compare the existing algorithms ABT and AFC-ng with the two learning methods (Learning-1 and Learning-2) and also with ABT/AFC-ng_Learning-1. The ABT/AFC-ng_Learning-1 algorithm is obtained by using the Learning-1 method in the two algorithms ABT and AFC-ng, computing just the number of MSGs and CCCs in ABT for ABT_Learning-1 and AFC-ng for the AFC-ng_Learning-1 algorithm, and keep the results of the fastest algorithm.

The experimentations are made against the number of exchanged messages (# MSGs) and the Concurrent Constraint Checks (# CCCs), using the jChoc platform.

We assess the five algorithms in random problems characterized by the parameters (n, d, p_1, p_2) where:

- n is the number of agents. $n = 20$;
- d is the domain size. $d = 10$;
- p_1 is the problem density;
- p_2 is the tightness of constraints.

We evaluate the algorithms into two types of problems. With low-density value (density p_1 equal to 0.2) and with high-density value (density p_1 equal to 0.7). For the two kinds of problems, we variate the tightness p_2 from 0.1 to 0.9 by 0.05 as a step.

3.3.1 Lower Density Problems

Figure 3.3 shows the number of exchanged messages and Concurrent Constraint Checks using the five algorithms, for sparse graphs (20, 10, 0.2,p_2).

The experimentation results show that Learning-1 and Learning-2 methods' performances lie between ABT and AFC-ng, knowing that we compute the number of the exchanged message by the two algorithms ABT and AFC-ng, even if, it is just one algorithm which finds the solution (or detects the unsolvability).

These results show that the algorithms learn from each other. Because if there is not any nogood learning, the number of exchanged messages would be the sum of the two algorithms' exchanged messages. it would be very large in such a way, the number of messages exchanged by ABT and AFC-ng would be negligible in front of the sum.

For CCCs, we are at least like the better of them.

When we compute the number of messages exchanged by only the algorithm finding the solution the first (it can be found by the ABT or the AFC-ng, according to the fastest algorithm), we get very important results. The ABT/AFCng_Learning-1 exchanges fewer messages and tests fewer constraints concurrently. That confirms the last results.

The choice of computing the MSGs and CCCs of only the algorithm discovering the solution is not obsolete. The machine learning principle is serving to store the exchanged nogood messages following each launch of the training algorithm. Then, use the stored messages, to learn and fastly resolve the DisCSP problem. In our case, we should not

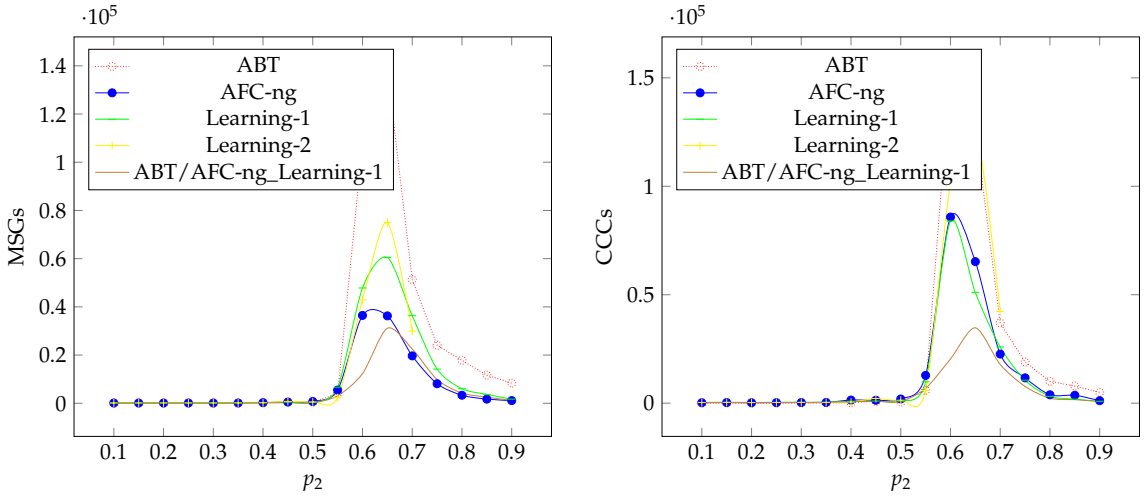


FIGURE 3.3: Learning benchmarking, $n = 20$, $v = 10$, $P_1 = 0.2$, p_2

compute the number of MSGS or CCCs of the training step because it is already done once.

3.3.2 Higher Density Problems

For dense random problems (20, 10, 0.7, p_2), figure 3.4 shows the evaluation results. As for the previous figure, it represents the number of exchanged messages and CCCs of the five algorithms. For p_2, it is ranging from 0.1 to 0.9 by 0.05 as a step, except the Learning-2 method which we evaluate just from 0.1 to 0.25.

Learning-2 is better than the ABT, AFC-ng and the Learning-1 methods, because, when an algorithm finds that the other has already sent the same nogood message, it stops the resolutions.

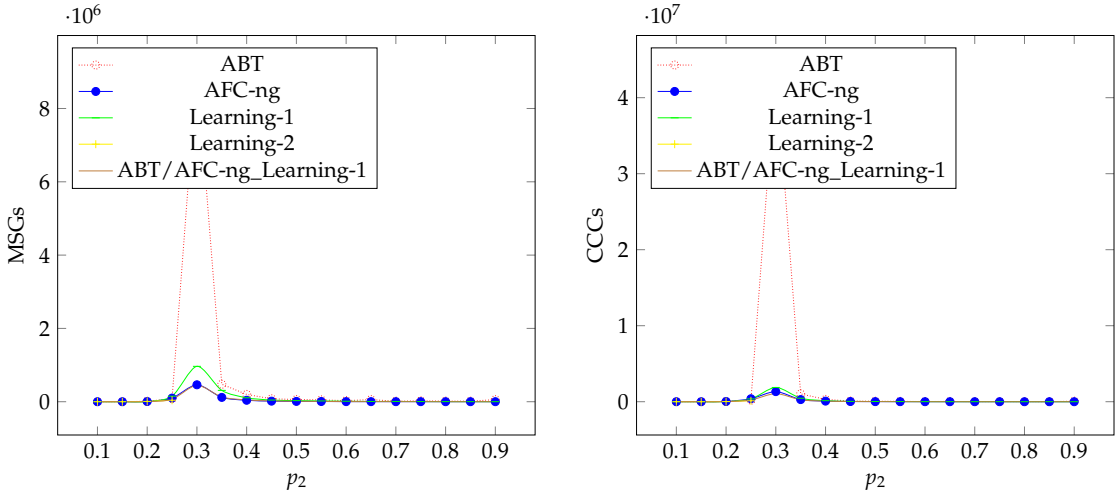


FIGURE 3.4: Learning benchmarking, $n = 20$, $v = 10$, $P_1 = 0.7$, p_2

This is feasible when problems are simple. But for complex problems, it is less likely, to find the same nogood generated with the two algorithms. In that case, it is Learning-1 the more feasible.

The results make clear that the two learning methods we used (Learning-1 or Learning-2) saves the exchanged messages and the tested constraints, even if we compute the resources used by the two algorithms. The importance of learning becomes more legible when we evaluate the ABT/AFC-ng_Learning-2.

3.4 Conclusion

Before trying to propose any new DisCSP algorithms or ameliorate the existing ones, we suggested, in this chapter, to invest in done efforts using the learning principle. To this end, we proposed two methods Learning-1 and Learning-2, that make it possible to launch at least two DisCSP algorithms at the same DisCSP problem and share the nogoods content between the participant algorithms.

The first method permits to share the nogoods until one of the algorithms finds a solution or detects the problem insolvency. The second method serves to detects which algorithm seems to be fast to resolve the problem to let it alone continue the problem resolution. Assessments prove the effectiveness of the two proposed methods for saving exchanged messages and concurrent constraints checks.

Part II

Complex Distributed Constraint Satisfaction Problems (Complex DisCSP)

4

Complex Distributed Satisfaction Problems' Resolution Methods and Algorithms

Contents

4.1 Introduction

The different existing complete DisCSP algorithms we have already seen have a common major limitation. For simplification assumptions, they assume that each agent is simple. It handles just one variable. Researchers assume that every Complex DisCSP problem can be updated to a simple problem, so that those algorithms can solve them as normal.

There are two major techniques to reformulate the complex local CSP problems, so as there is exactly one variable per agent. **The decomposition** and **the compilation** transform the original problem to make it as a simple one, so as the existing algorithms may look for the global solution as though the initial problem is simple.

The decomposition and the compilation methods are not sufficient, and can not be applied in some cases, especially when the local variables are growing in number. Figure 4.1 shows an example of a DisCSP problem with complex local problems with an important number of local variables. To this end, two other methods are created to improve the compilation method, namely the **Interchangeability** and the **Neighborhood Partial Interchangeability (NPI)**. Several algorithms based on the decomposition or the compilation method have appeared. But most of these algorithms are incomplete.

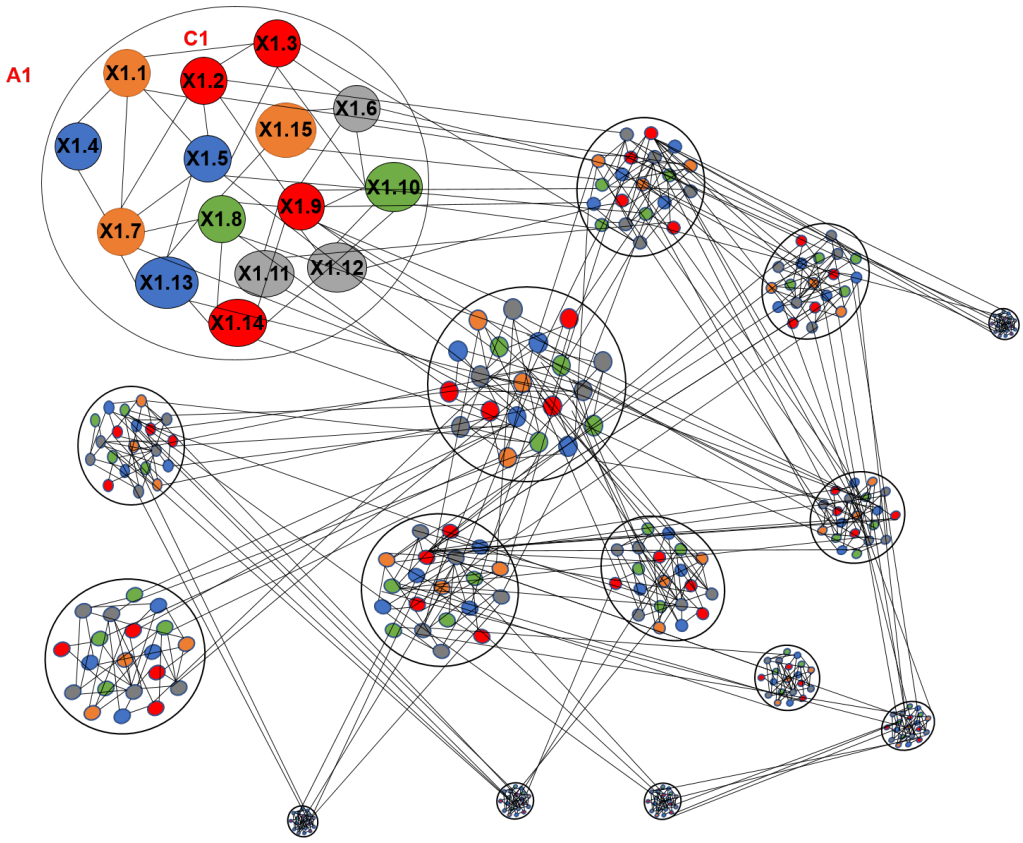


FIGURE 4.1: DisCSP example with complex local problems

In this chapter, we are going to present the different existing methods and algorithms to support DisCSP with complex local problems.

4.2 Methods

The proposed methods aim to transform the local problems, not the algorithm, so that the whole problem can be solved using one of the existing DisCSP algorithms.

To well understand the different methods, we are going to take the same example shown in figure 1.2, Chapter 1. It is an example of a DisCSP with complex local problems, composed by three agents A_1, A_2, and A_3. A_1 handles three variables $M_{1.1}$, $M_{1.2}$, and $M_{1.5}$, A_2 has four variables $M_{2.1}$, $M_{2.2}$, $M_{2.3}$, and $M_{2.4}$, and the local problem of A_3 is composed of three variables $M_{3.1}$, $M_{3.3}$, and $M_{3.4}$.

4.2.1 Decomposition

The decomposition [Fioretto, Yeoh, and Pontelli, 2016] method is used to create a virtual agent, for each variable, to manage its domain.

The application of this method on the example generates the creation of a new simple problem composed of ten virtual agents $M_{1.1}$, $M_{1.2}$, $M_{1.5}$, $M_{2.1}$, $M_{2.2}$, $M_{2.3}$, $M_{2.4}$, $M_{3.1}$, $M_{3.3}$, and $M_{3.4}$ (Figure 4.2). Each virtual agent handles its own variable. Several algorithms use the decomposition, but they are all incomplete.

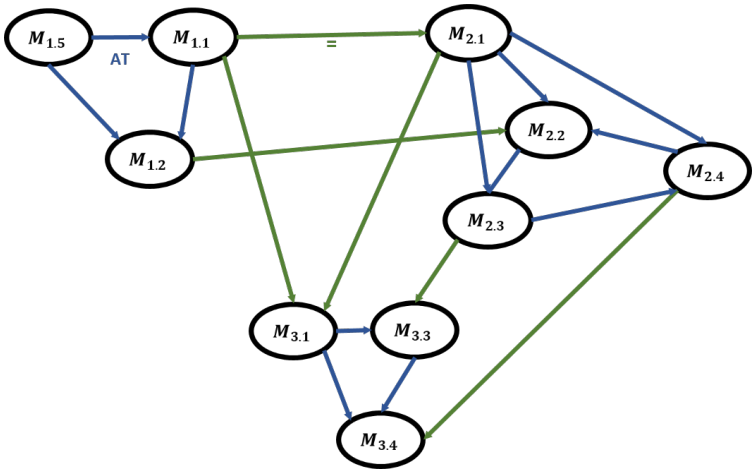


FIGURE 4.2: Decomposition

4.2.2 Compilation

The compilation [Weigel and Faltings, 1999] consists of creating a single new abstract variable for each complex agent. The domain of the newly created variable is the set of solutions of the agent local CSP problem.

A_1, and all other complex agents A_2 and A_3, create a new abstract variable whose domain values are their local solutions. Table 4.1 shows a part of the local solutions of A_1. The newly created abstract variable has 120 values in its domain $\{v_1, v_2, v_3, v_4, v_5, \ldots, v_{120}\}$, which is an important number of solutions.

The size of the abstract variable's domain depends on the number of variables, the constraints' type, and also on the domain size of each local. The more variables and the domains are large, the larger the abstract variable domain is. The fewer constraints there are, the larger the domain too.

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
v_1	1	2	3
v_2	1	2	4
v_3	1	2	5
v_4	1	2	6
v_5	1	3	2
v_6	1	3	4
v_7	1	3	5
v_8	1	3	6
v_9	1	4	2
v_{10}	1	4	3
v_{11}	1	4	5
v_{12}	1	4	6
v_{13}	1	5	2
v_{14}	1	5	3
v_{15}	1	5	4
v_{16}	1	5	6
v_{17}	1	6	2
v_{18}	1	6	3
v_{19}	1	6	4
v_{20}	1	6	5
v_{21}	2	1	3
v_{22}	2	1	4
v_{23}	2	1	5
...	...	...	...
v_{120}	6	5	4
The compiled domain size			120

TABLE 4.1: The compiled domain of A_1

4.2.3 Interchangeability

To reduce the size of the created abstract variable's domain, the authors proposed, in [Burke and Brown, 2006], the interchangeability as an improvement of the compilation. Its principle is based on the fact the external variables are the only variables having a direct effect on the resolution process in a local problem.

Definition 14 (External variables) *The external variables are variables that are linked to the other agents' variables. In the example, the variables $M_{1.1}$ and $M_{1.2}$ are the two external variables of A_1, because they are linked with the A_2's variables $M_{2.1}$ and $M_{2.2}$ and also to the A_3's $M_{3.1}$.*

Interchangeable solutions are the local solutions having the same values of external variables. These local solutions are removed from the compiled domain.

Table 4.1 shows that the newly created abstract variable's domain contains the 120 local solutions of A_1, whereas there are some interchangeable values. For example, in the four first local solutions, the values $M_{1.1} = 1$, $M1.2 = 2$ are the same. By applying the interchangeability, we keep just one of the four solutions. No matter the value of $M_{1.5}$ is, since it does not influence the resolution.

After applying the interchangeability principle, the domain of the abstract variable, created for the A_1's local problem, is containing 30 values, instead of 120 (Figure 4.3).

The interchangeability reduces the domain size, if and only if, at least one of the local variables is not external. For the agents A_2 and A_3 of the same example, all their variables are external. Suddenly, the interchangeability application will not make any changes to their compiled domains.

4.2.4 Neighborhood Partial Interchangeability

The Neighborhood Partial Interchangeability (NPI) is a special case of the Interchangeability. It is applied with respect to a specified constraint.

We say that a set of local solutions are neighborhood partial interchangeable with respect to a given inter-agent constraint when the scope of the constraint (the variable/s linked by the constraint), has the same value in all those local solutions.

Let name the first inter-agent constraint of the example, that links the two variables $M_{1.1}$ and $M_{2.1}$, "C_1" and the second one, which connects $M_{1.2}$ and $M_{2.2}$, "C_2".

The Neighborhood Partial Interchangeability with respect to C_1 concerns all local solutions having the same value of $M_{1.1}$ that is the scope of C_1. While the NPI respecting to C_2 corresponds to the local solutions having the same value of $M_{1.2}$.

(a) Compiled domain of Agent A_1

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
v_1	1	2	3
v_2	1	2	4
v_3	1	2	5
v_4	1	2	6
v_5	1	3	2
v_6	1	3	4
v_7	1	3	5
v_8	1	3	6
v_9	1	4	2
v_{10}	1	4	3
v_{11}	1	4	5
v_{12}	1	4	6
v_{13}	1	5	2
v_{14}	1	5	3
v_{15}	1	5	4
v_{16}	1	5	6
v_{17}	1	6	2
v_{18}	1	6	3
v_{19}	1	6	4
v_{20}	1	6	5
v_{21}	2	1	3
v_{22}	2	1	4
v_{23}	2	1	5
v_{24}	2	1	6
v_{25}	2	1	3
v_{26}	2	1	4
v_{27}	2	1	5
v_{28}	2	1	6
...	...	...	...
v_{120}	6	5	4
The compiled domain size			**120**

(b) Compiled domain after applying the interchangeability

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
v_1	1	2	3
v_5	1	3	2
v_9	1	4	2
v_{13}	1	5	2
v_{17}	1	6	2
v_{21}	2	1	3
v_{25}	2	1	3
v_{29}	2	4	1
v_{33}	2	5	1
v_{37}	2	6	1
v_{41}	3	1	2
v_{45}	3	2	1
v_{49}	3	4	1
v_{53}	3	5	1
v_{57}	3	6	1
v_{61}	4	1	2
v_{65}	4	2	1
v_{69}	4	3	1
v_{73}	4	5	1
v_{77}	4	6	1
v_{81}	5	1	2
v_{85}	5	2	1
v_{89}	5	3	1
v_{93}	5	4	1
v_{97}	5	6	1
v_{101}	6	1	2
v_{105}	6	2	1
v_{109}	6	3	1
v_{113}	6	4	1
v_{117}	6	5	1
The compiled domain size			**30**

FIGURE 4.3: The compiled domain of A_1 before and after the interchangeability application

(a) the NPI with respect to C_1

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
v_1	1	2	3
v_{21}	2	1	3
v_{41}	3	1	2
v_{61}	4	1	2
v_{81}	5	1	2
v_{101}	6	1	2
The compiled domain size			**6**

(b) the NPI with respect to C_2

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
v_1	1	2	3
v_5	1	3	2
v_9	1	4	2
v_{13}	1	5	2
v_{17}	1	6	2
v_{21}	2	1	3
The compiled domain size			**6**

FIGURE 4.4: The application of the NPI on A_1 domain

Figure 4.4 shows the compiled domains of A_1 after applying the NPI with respect to C_1 and to C_2. The displayed values can be considered as an abstract of the whole domain. For example, in the first table, v_1 represents all local solutions from v_1 to v_20, since all these local solutions have the value 1 for the variable $M_{1.1}$.

4.3 Algorithms

Many algorithms have appeared to solve DisCSPs with complex local problems. They are all based either on the decomposition and/or its ameliorations or the compilation.

There are several algorithms that use the decomposition method, such as Multi-Aynchronous Weak Commitment (Multi-AWC) [Xiong et al., 2010], the Multiple local variables Distributed Breakout (Multi-DB [Hirayama and Yokoo, 2002; Hirayama and Yokoo, 2005]), and the Multi Dynamic Priorisation with Penalties (Multi-DynApp) [Magaji, Arana, and Ahriz, 2014]. The common point between these methods is the incompleteness. They are all incomplete.

Whereas the compilation, the Interchangeability, and the Neighborhood Interchangeability are used by Ezzahir et al. in [Ezzahir et al., 2007a] to create a new algorithm named ABT-cf (Algorithm 7) based on the ABT algorithm.

As in the ABT algorithm, ABT-cf uses 4 types of messages (waitForArrivalMessages() procedure), **Info** which replaces the ABT's Ok? message, **Nogood**, **AddLink**, and **Stop**.

In addition to the ABT procedures, the ABT-cf uses a procedure to test the feasibility of the local problem. If the agent's local problem is solvable, it waits for the reception of

Algorithm 7 ABT-cf algorithm

```
1: procedure ABT-CF
2:     if local problem is not feasible then
3:         end ← true;
4:         sendMsg:stp(system);
5:     else
6:         seeReceivedMessages();
7:         Ic(i) ← {};
8:         CheckAgentView();
9:         while ¬end do
10:            waitForArrivalMessages();
11:        end while
12:    end if
13: end procedure

14: procedure WAITFORARRIVALMESSAGES
15:    msg ← getMsg();
16:    Switch (msg.type)
17:     Info  : ProcessInfo(msg);
18:     Nogood  : ResolveConflict(msg);
19:     Stop  : end ← true;
20:     AddLink  : SetLink(msg);
21: end procedure

22: procedure CHECKAGENTVIEW(msg)
23:    if ¬consistent(LS, myAgentView) or Ic(i) = {} then
24:        LS ← ChooseValue();
25:        if LS then
26:            Ic(i) ← LS ↓ Nv;
27:            I(i) ← (Ic(i) − Ip(i));
28:            for each child_j ∈ Γ+(self) do
29:                sendMsg : Info (child_j, I(i) ↓ Nv_j);
30:            end for
31:        else Backtrack();
32:        end if
33:    end if
34: end procedure

35: function CHOOSEVALUE()
36:    for each LS ∈ Compiled_D(self) not eliminated by myNogoodStore do
37:        if consistent(LS,myAgentView) then return (v);
38:        end if
39:    end for return (empty)
40: end function
```

```
41: procedure RESOLVECONFLICT(msg)
42:     add msg.nogood to myNogoodStore
43:     for each LS from Compiled_D do
44:         if LS holds msg.nogood.RHS then then
45:             remove LS from Compiled_D;
46:         end if
47:     end for
48:     CheckAddLink(msg.nogood.LHS);
49:     Ip(i) ← Ic(i);
50:     CheckAgentView();
51: end procedure

52: procedure PROCESSINFO(msg)
53:     Update(myAgentView, msg.Assig);
54:     CheckAgentView();
55: end procedure

56: procedure SETLINK(msg)
57:     add(msg.sender, Γ+(self));
58:     sendMsg:Info(msg.sender, myValue);
59: end procedure

60: procedure CHECKADDLINK(msg)
61:     for each agt ∈ lhs(msg.Nogood) do
62:         if agt ∉ Γ−(self) then
63:             sendMsg:adl(agt,self);
64:             add(agt,Γ−(self);
65:             Update (myAgentView, agt ← agtValue);
66:         end if
67:     end for
68: end procedure

69: procedure BACKTRACK
70:     newNogood ← solve(myNogoodStore);
71:     if newNogood = empty then
72:         end ← true;
73:         sendMsg:stp(system);
74:     else
75:         sendMsg:ngd(newNogood) to A_j agent enclosed in newNogood.RHS;
76:         Remove (newNogood.RHS from MyAgentView;
77:         CheckAgentView();
78:     end if
79: end procedure
```

```
80: procedure UPDATE(myAgentView, newAssig)
81:     add(newAssig, myAgentView);
82:     for each ng ∈ myNogoodStore do
83:         if ¬Coherent(lhs(ng), myAgentView) then
84:             remove(ng, myNogoodStore);
85:         end if
86:     end for
87: end procedure

88: function COHERENT(nogood, agents)
89:     for each agt ∈ nogood ∪ agents do
90:         if nogood[agt] ≠ myAgentView[var] then
91:             return false;
92:         end if
93:     end for
         return true;
94: end function
```

messages (Lines 2 . . . 6). Otherwise, it stops the resolution from the start, reporting that the problem is insolvent.

Unlike the Ok? message, which is used to send the current instantiation $Ic(i)$ from agent i, the Info message is used to send a part of its instantiation. The content of the message depends on the recipient agent. For each lower priority agent, the agent i sends the partial assignment containing the common external variables, from its whole current instantiation.

The other changes compared to the ABT algorithm are seen in the choice of a new value using the ChooseValue() function and when a Nogood message is received (ResolveConflict() procedure).

The choice of a new value is replaced by the choice of a local solution from the agent's compiled domain (Line 36). This is done after applying the interchangeability principle which tries to reduce the size of its compiled domain.

For ResolveConflict () Procedure, the received Nogood does not report the whole local solution LS of the receiver agent. It reports only a portion. In that case, the agent applies the NPI principle to remove all local solutions which hold the right part of the received Nogood, from its compiled domain (Lines 43 . . . 47).

In this report, all the new proposals in the context of DisCSP problems with Complex problems will be compared with ABT-cf and the other existing methods, namely the compilation and the decomposition.

Comparing a method with an algorithm no longer makes sense. For this, we integrate

the methods into DisCSP algorithms. In order not to be limited to a single algorithm, the integration is made both into the ABT and also the AFC-ng.

This gives rise to four other algorithms that are alimented in JChoc Platform:

- Compilation based ABT **ABT-comp**. It differs from ABT in the ChooseValue() procedure. The agents browse their compiled domain to choose a local solution (as ABT-cf);

- Decomposition based ABT **ABT-decomp**. Instead of running the ABT algorithm by each agent, it is launching by each local variable, which is considered as a virtual agent;

- Compilation based AFC-ng **AFC-ng-comp**. As for ABT-comp, the AFC-ng-comp agent chooses a local solution instead of choosing a value to a variable, since the agent's local problem is complex;

- Decomposition based AFC-ng **AFC-ng-decomp**. The AFC-ng is running by each variable.

4.4 Conclusion

The different existing DisCSP algorithms assume that the participating agents are simple for simplification assumptions. There is exactly one variable per agent. In the case of complex agents, the local problems are transformed using the compilation or the decomposition method, so the existing algorithms can resolve the problem as if the local problems are simple.

Even the proposed algorithms are all based on either decomposition or compilation. Whereas this is not feasible, because ignoring local resolution strategy can lead to a global costly resolution.

5

MP-ABT: A Minimal Perturbation Approach for Complex Local Problems

Contents

5.1 Introduction

The algorithms of DisCSPs with complex local problems use the decomposition or the compilation methods to reformulate the local problems so as there is exactly one variable per agent. These transformations change either the number of agents or the number of domain values of each agent. Although, ignoring the local resolution strategy can lead to a global costly resolution.

Managing the trade-off between complex local problems and distributed search effort can give away to great improvements. For that, we are focusing, to guardedly choose the local solutions, after receiving a new message, without updating the original global problem.

We propose a new algorithm, entitled Minimal Perturbation based Asynchronous Backtracking (MP-ABT), to resolve DisCSP problems with complex local problems. MP-ABT considers each complex agent as a Minimal Perturbation Problem (MPP) and each received messages as a new intra-constraint perturbation event. When a message is received, the local problem is updated and a new MPP local solution is reported using the Hybrid Search for Minimal Perturbation Problems (HS-MPP) algorithm. All real Complex DisCSPs can benefit from this trade-off strategy, as the meeting scheduling problem, road traffic, and multi-robot exploration.

In the following, we define the Minimal Perturbation Problem in section 5.2. Then we describe the MP-ABT algorithm, show its properties, and apply it on our DisMSP example in section 5.3. Finally, we exhibit experimental results illustrating the efficiency of our newly developed algorithm in section 5.4.

5.2 Minimal Perturbation Problem (MPP)

5.2.1 Definition

A Minimal Perturbation Problem (MPP) [Barták, Muller, and Rudová, 2003] is a altered CSP problem after being solved, where the main task is to find a new solution in such a way the latter is as close as possible to the first CSP solution. The MPP can be formulated as:

- a CSP problem P,
- an initial solution S_1 of P;

- a distance function f defining the distance between any two assignments.

A solution of an MPP problem is a solution S_2, such as the distance $f(S_1, S_2)$ between S_1 and S_2 is minimal.

5.2.2 Hybrid Search for Minimal Perturbation Problems (HS-MPP)

The Hybrid Search for Minimal Perturbation Problems (HS-MPP) [Zivan, Grubshtein, and Meisels, 2011; El Graoui, Benelallam, and Bouyakhf, 2016] is an approach to solve Minimal Perturbation Problems considering the Hamming Distance [Norouzi, Fleet, and Salakhutdinov, 2012] as a distance function.

Definition 15 (Hamming Distance) *The Hamming Distance is a mathematical concept that computes the number of positions where two entities, with the same length, differ. In our case, on two different assignments of the same local problem (i.e., the same agent), this measure computes the number of variables having different values.*

Let take an example A_i's local problem with three local variables $X_{i.1}$, $X_{i.2}$, and $X_{i.3}$, having an initial solution $S_1 = \{X_{i.1} = 1, X_{i.2} = 1, X_{i.3} = 1\}$. After a constraint modification, the solution becomes $S_2 = \{X_{i.1} = 1, X_{i.2} = \mathbf{2}, X_{i.3} = \mathbf{2}\}$. So, the Hamming distance of S_1 and S_2 is equal to 2, since the values of the 2 variables $X_{i.2}$ and $X_{i.3}$ are changed.

The HS-MPP algorithm takes the CSP variables, domains, and constraints as well as the last CSP solution as parameters and returns the closest solution or null. Null signifies that there is no solution to the declared MPP problem.

5.3 Minimal Perturbation based Asynchronous Backtracking MP-ABT

The main contribution of MP-ABT is highlighted when an agent receives a new message Ok? or Nogood. In the ABT algorithm, if such a message requires a new assignment, the receiver chooses a consistent local solution randomly. This assignment is chosen without considering the former local solution. In MP-ABT, the MPP formalism is used to benefit from the former solution and minimize the local perturbations so as the number of disturbed neighbor agents is reduced.

The MP-ABT merges the ABT and HS-MPP algorithms. It extends the ABT, to tackle DisCSPs with complex local problems, with fewer perturbations. The idea is to consider

each local complex problem as an MPP problem and each newly received message as a new intra-agent constraint perturbation. The aim is to find a new local solution that is as close as possible to the former local solution.

5.3.1 Description of the Algorithm

The local search of ABT can not be replaced directly by the HS-MPP approach. In ABT's local search, the agent stores nogoods when looking for a new consistent local solution. In case the consistent local solution is not found, the agent generates a new nogood from the nogoods stored for each inconsistent assignment. For the sake of HS-MPP, the local consistent value is looked for in a single execution. Therefore, if the returned value is null (i.e., there is no consistent local solution), the agent has no justifications to construct a new nogood. That is why we have to make several changes to the original pseudo-code of ABT.

In the following, we are going to describe how the MP-ABT algorithm is running. Algorithm 8 provides the procedures and functions executed by each MP-ABT agent, which are not existing in the ABT pseudo-code, or which are changed.

As for the ABT, each MP-ABT agent assigns values to its variables, sends them to the corresponding agents, and then switches to the listening position to respond to incoming messages (ABT pseudo-code 3).

After the reception of the Ok? message, the receiver filters the domain of each local variable. It removes all values, which are inconsistent with sender values, from the variable domain. For each filtered value, it adds a nogood to its 'Justifications' structure (Update procedure, line 37). The stored nogood contains the deleted value and the variable that cause this value removal. It contains only the partial assignment that causes the inconsistency, not the whole local solution of the sender. Hence it enjoys the benefits of the interchangeability principle to speed up the resolution process.

During the filtering process, the agent tests the whole domain even if, it may contain values that are already deleted. These redundant justifications aim to save all suppression causes of each value.

Foremost, the agent updates the Justifications and NogoodStore structures, by removing the nogoods that become obsolete (Update procedure, lines 76 and 81). Then, it restores values to the variables domains (Update procedure, line 83). A value is restored if and only if all the corresponding justifications are removed (Update procedure,

Algorithm 8 MP-ABT algorithm

```
1: procedure MP-ABT myvalue ← empty;
2: end ← false;
3: CheckAgentView();
4:     while ¬end do
5:         msg ← getMsg();
6:         Switch (msg.type)
7:         Ok? : ProcessInfo(msg);
8:         ngd  : ResolveConflict(msg);
9:         stp  : end ← true;
10:        Adl  : SetLink(msg);
11:    end while
12: end procedure

13: function CHOOSEVALUE(VariableDomains, NOGOODSTORE)
14:    if (one of VariableDomains is empty) then
         return null
15:    end if
16:    Create a MPP problem with the same variables of the initial local problem;
17:    Attribute VariableDomains to the variables;
18:    Add the non redundant rhs(nogood) of the NogoodStore as a not equal constraint
       of the MPP problem;
         return MPP.getSolution
19: end function

20: procedure UPDATE(myAgentView, newAssig)
21:    add(newAssig, myAgentView);
22:    for { each ng ∈ myNogoodStore} do
23:        if (¬Coherent(lhs(ng), myAgentView)) then
24:            remove(ng, myNogoodStore);
25:        end if
26:    end for
27:    for {each ng ∈ Justifications} do
28:        if (¬Coherent(lhs(ng), myAgentView)) then
29:            remove(ng, Justifications);
30:            if (rhs(ng) is not removed by a nogood in Justifications ) then
31:                restore rhs(ng).value to the currentDomains;
32:            end if
33:        end if
34:    end for
35:    for each local variable do
36:        Remove inconsistent values from currentDomain;
37:        Add a nogood in justifications for each removed value;
38:    end for
39: end procedure
```

```
40: procedure BACKTRACK
41:     newNogood ← solve(myNogoodStore ∪ Justifications);
42:     if newNogood = empty then
43:         end ← true;
44:         sendMsg: stp (system);
45:     else
46:         sendMsg:ngd(newNogood);
47:         Update (myAgentView, rhs(newNogood) ← unknown);
48:         CheckAgentView();
49:     end if
50: end procedure
```

line 82). Finally, it chooses a new local solution (ChooseValue procedure), following the succeeding steps:

1. It checks if there is an empty filtered domain (line 14);
2. If so, it returns **null**, to send a nogood message, without looking for a new local solution;
3. Otherwise, it declares a new MPP problem. The latter consists of its local problem variables which are defined on their new corresponding filtered domains, its local constraints (intra-agent constraints), and the stored valid nogoods, that are compatible with the AgentView, as new constraints (lines 16, 17, and 18);
4. Finally, it looks for a new closest solution to its current solution, using the HS-MPP algorithm.;
5. If a new consistent solution is found, it is sent to the corresponding agents (CheckAgentView procedure);
6. Otherwise, a new nogood is generated, using the stored justifications and the received nogoods (Backtrack procedure, line 41).

After the reception of a nogood message, the receiver checks if it is coherent with its AgentView, as in ABT. If so, it stores it in its NogoodStore, and chooses a new closest solution, with the same manner described previously.

5.3.2 MP-ABT Properties

The MP-ABT has the same properties as the ABT: the soundness, completeness, and the termination.

Soundness

Since the local search of ABT is replaced by the MPP approach in MP-ABT. The only risk that may cause the unsoundness of MP-ABT is that it has no way to record the deletion reasons for each local solution. But from the moment that it records justifications during the filtering operation, therefore it remains also sound, as the ABT algorithm.

completeness

The MP-ABT uses the filtered domain, the existing intra-agent constraints and the received valid nogoods as inputs to the MPP algorithm. Since the HS-MPP algorithm is sound, the returned solution will satisfy all constraints (the original constraints, and the received nogoods). So the local solution of each agent, satisfy its inter-agent and intra-agent constraints. So the algorithm is complete.

termination

The filtering process speeds up the search, and can never be trapped in an infinite loop since the domains are finite. In addition, since the HS-MPP algorithm finds the solution in a limited time, so the whole problem can be solved also in a finite time. On the contrary, it speeds up the search, because it gives the local decision in just one loop, without doing the compilation nor the decomposition process.

5.3.3 Application Example

To well understand the MP-ABT, we are going to apply it in our DisMSP example (Figure 1.2), which is made up of three complex agents A_1, A_2 and A_3. A_1 and A_3 handle three variables, while A_2 manipulates four variables.

When the MP-ABT starts, each agent chooses its first local solution and communicates it to the lower priority agents via Ok? messages. A_1 chooses and sends $(M_{1.1} = 1, M_{1.2} = 2, M_{1.5} = 3)$ to A_2 and A_2. Whereas A_2 chooses $(M_{2.1} = 1, M_{2.2} = 3, M_{2.3} = 4, M_{2.3} = 5)$ and sends it to A_3.

After receiving the OK? message from A_1, A_2 stores the received values in its AgentView, removes inconsistent values from its variables' domains, and stores a justification for each removed value.

It deletes all values from $M_{2.1}$ domain except 1, because $M_{1.1} = 1$ and $M_{2.1}$ should be equal to $M_{1.1}$. It adds the nogoods $M_{1.1} = 1 \rightarrow M_{2.1} \neq v$, $v \in \{3, 4, 5, 6\}$ as deletion justifications in its 'Justifications' structure, and checks if its current local solution remains consistent. The same treatment is done for the second variable $M_{2.2}$. The stored justifications for the second variable contains the value $M_{1.2} = 2$ as cause.

After storing the nogoods, A_2 finds that the $M_{1.2}$'s domain becomes empty. So it gathers the stored nogoods to generate $\emptyset \rightarrow (M_{1.1} \neq 1, M_{1.2} \neq 2)$. The generated nogood contains only $M_{1.1} = 1$ and $M_{1.2} = 2$ which are the real causes of the failure.

For A_3, it receives two Ok? messages (from A_1 and A_2). After receiving the A_1 message, it proceeds as A_2, by storing a justifying nogoods for each inconsistent value. The only variable constrained with A_1 is $M_{3.1}$, so all justifications contain the same cause $M_{1.1} = 1$. After the deletion process, A_3 notes that its domain becomes empty. So, it sends the nogood $\emptyset \rightarrow M_{1.1} \neq 1$ to A_1.

Even the A_2's Ok? message causes a failure to A_3, because of the three variables $M_{2.1}$, $M_{2.3}$,and $M_{2.4}$. In this case, it sends another nogood message to A_2.

After receiving the first nogood $\emptyset \rightarrow (M_{1.1} \neq 1, M_{1.2} \neq 2)$ from A_2, A_1 stores it in its NogoodStore and tries to find a new local solution responding to the received nogood and which is as close as possible to its current local solution $(M_{1.1} = 1, M_{1.2} = 2, M_{1.5} = 3)$, using the HS-MPP method.

The local solution returned by the HS-MPP is $(M_{1.1} = 4, M_{1.2} = 2, M_{1.5} = 3)$, Which changes only the value of the first variable. This assignment is sent to A_2 and A_3 via Ok? messages. The advantage of choosing an MPP solution can be seen when the variable $M_{1.5}$ of the agent A_1 is constrained with another external variable of another agent. In that case, A_1 will not disrupt the other agents. Therefore, MP-ABT minimizes the number of global perturbations, by minimizing it locally.

The reception of the second nogood message $\emptyset \rightarrow M_{1.1} \neq 1$ does not provoke any new message except its storage in A_2's NogoodStore since the last A_2's local solution responds already the received nogood.

For the nogood message received by A_2, it requires to be saved and to look for a new consistent value with a minimal number of changed values, while taking into account the nogoods stored in A_2 NogoodStore. The nogoods are going to be added to the MPP' declaration as constraints.

The agents are going to continue this way, by minimizing the number of variables' values changes when looking for a new local solution and generating the appropriate

nogoods, until finding a global solution to the problem.

One of the advantages of the MP-ABT we saw is the pointed nogood's content. It allows detecting the real cause of conflict and implicating the responsible variable, not only the responsible agent, as for the ABT algorithm. We note also that the filtering process helps to detect the conflict without the need of searching a new solution.

5.4 Experimental Results

In this section, we compare the MP-ABT algorithm with ABT-comp, ABT-decomp, and ABT-cf. As we have mentioned in Chapter 4, ABT-comp (respectively ABT-decomp) is the result of the merge of the compilation (respectively the decomposition) with the ABT algorithm.

The assessments are made against the number of exchanged messages (# MSGs), which can be considered as a measure of the number of the agents' perturbations and the communication load. The experimentations are also made against the Concurrent Constraint Checks (# CCCs). To have significant results, the checked constraints during the compilation and the decomposition are taken into consideration too.

The four algorithms are evaluated on Random Complex DisCSP problems. All problems are characterized by (i_a, i_b, c, n, d, v, p_2) while:

- i_a is the intra-agent density;
- i_b is the inter-agent density;
- c is the connection density;
- n is the number of agents;
- d is the domain size;
- v is the number of variables per agent;
- p_2 is the tightness of constraints.

We select problems in four most representative zones classes of constraints: $<$ 0.3, 0.7, 0.3, 5, 5, 2, p_2 $>$, $<$ $0.7, 0.3, 0.7, 5, 5, 2, p_2$ $>$, $<$ $0.3, 0.7, 0.3, 5, 5, 4, p_2$ $>$, and $<$ $0.7, 0.3, 0.7, 5, 5, 4, p_2$ $>$. The tightness p_2 varies from 0.1 to 0.9 by 0.1 as a step. For each fixed set (i_a, i_b, c, n, d, v, p_2), we generate 10 instances, and we take the average.

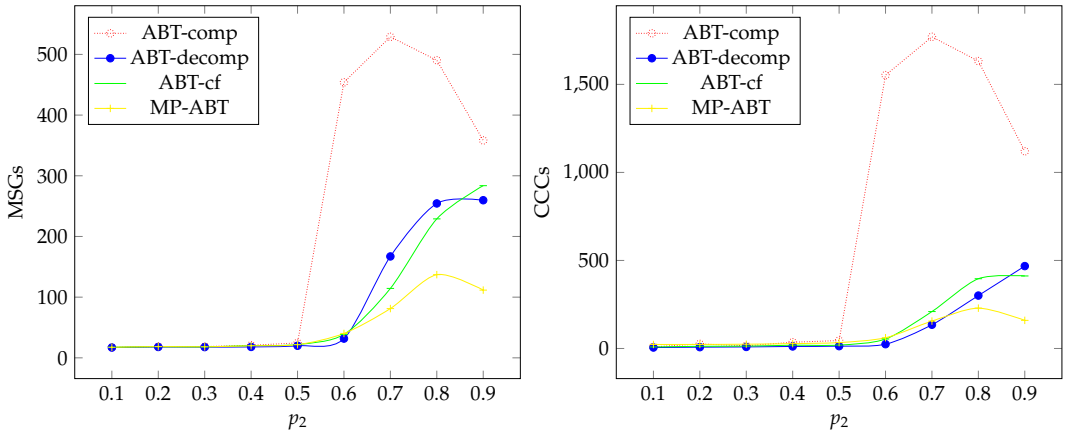


FIGURE 5.1: MP-ABT benchmarking with $< 0.3, 0.7, 0.3, 5, 5, 2, p_2 >$

Figures 5.1 and 5.2 show the behavior of the four algorithms ABT-comp, ABT-decomp, ABT-cf, and MP-ABT, while solving problems with **2** variables per agent and belonging to the first class of constraints $<$ 0.3, 0.7, 0.3, 5, 5, 2, $p_2 >$ (Figure 5.1), and also to the second class of constraints $<$ 0.7, 0.3, 0.7, 5, 5, 2, $p_2 >$ (Figure 5.2).

The results exhibit that the MP-ABT exchanges fewer messages (less disturbance) than the other algorithms. In this case, our principal aim is achieved which is minimizing the number of perturbations. MP-ABT agents exchange fewer messages than the three other algorithms, especially, when problems become denser.

In the first-class problems, agents handle few intra-agent constraints (the intra-agent density = 0.3). So even if few or zero constraints are tested during compilation (ABT-comp and ABT-cf), MP-ABT checks fewer constraints concurrently than the other algorithms. The same for ABT-decomp, that wastes several messages without any importance in the decomposition, such as nogood and AddLink messages.

For 4 variables per agent, the evaluation results are shown in figures 5.3 and 5.4.

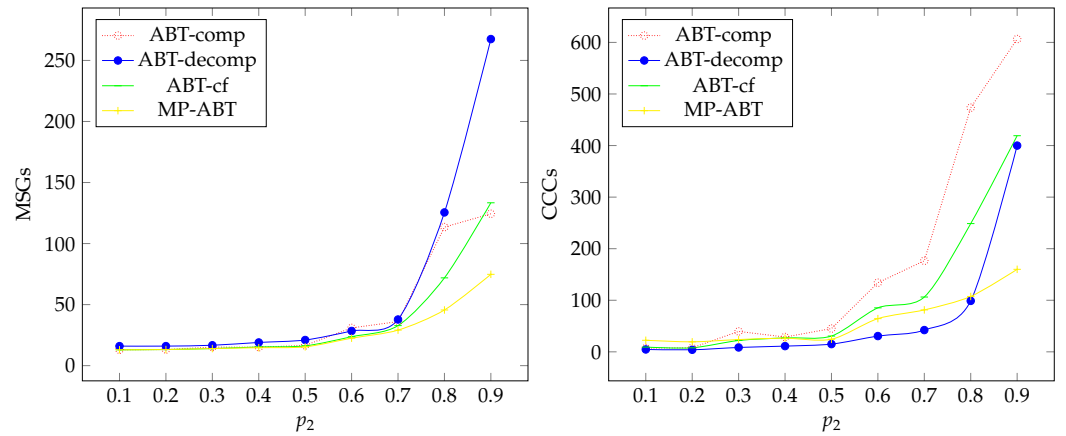


FIGURE 5.2: MP-ABT benchmarking with $< 0.7, 0.3, 0.7, 5, 5, 2, p_2 >$

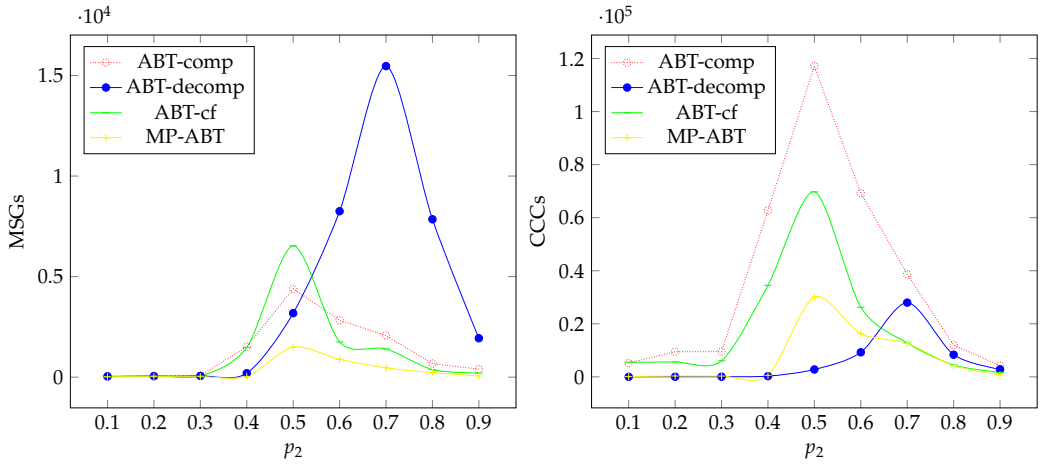


FIGURE 5.3: MP-ABT benchmarking with $< 0.3, 0.7, 0.3, 5, 5, 4, p_2 >$

In this kind of problem, the performance of the MP-ABT algorithm becomes increasingly important. On the contrary to the first results (i.e., 2 variables per agent), the outperformance began to be remarkable even in problems with low tightness too.

Local problems become more constrained. So, it is too hard and complex to find the whole compiled domain, in ABT-comp and ABT-cf. Thus, the domain contains several values. Therefore more constraints are tested. For the ABT-decomp, it makes a big effort. It can loop several times to find one consistent local solution.

We observe that, in general, the ABT-comp is the less-performance algorithm, especially in the first class of problems. It exchanges more messages (more disturbance) and checks more constraints concurrently. This is due to the size of the newly constructed domain by the compilation.

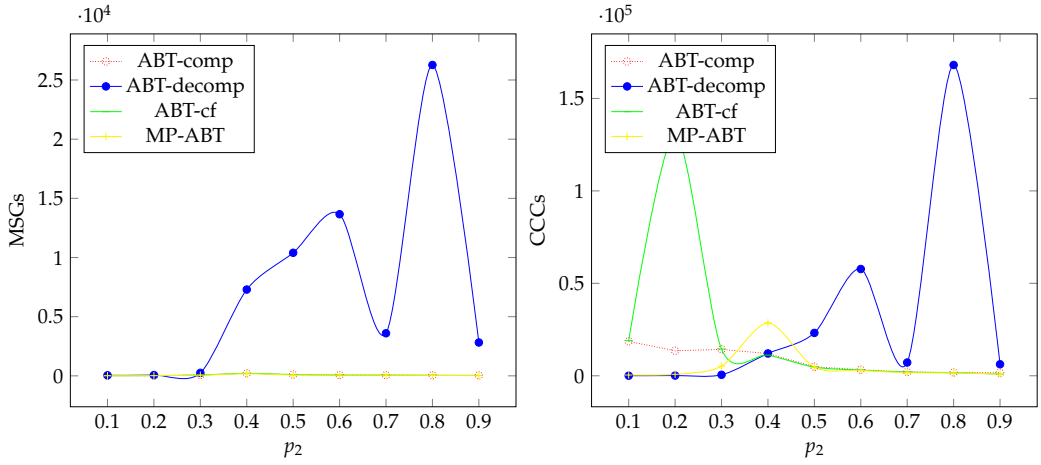


FIGURE 5.4: MP-ABT benchmarking with $< 0.7, 0.3, 0.7, 5, 5, 4, p_2 >$

ABT-cf is expected to be better than ABT-comp because it reduces the size of the compiled domain by deleting the interchangeable solutions. But it does not exceed the MP-ABT, because the content of used nogoods does not help to speed up the search and interchangeability is not really applied when local problems become dense.

Even if the domain size is still the same in the ABT-decomp algorithm, it is also less efficient than MP-ABT, because of the creation of the virtual agents, which increases the size of the global problem. And so, there are more exchanged messages, and then more concurrent constraint checks.

We aimed to minimize the number of perturbations (i.e., the number of exchanged messages). Such results exhibit that our principal aim is achieved, in the case of two variables per agent as well as four variables. In addition to our main goal, we have also reduced the number of non-concurrent constraint checks, in most tightness values, especially when the local constraint network becomes dense.

The different evaluation results demonstrate that the MP-ABT is a very effective way to decrease the number of messages and therefore the number of perturbations. We observe that the more variables we have, the less disturbance MP-ABT does. And even in unsolvable problems. The MP-ABT algorithm remains better, due to the nogood content, which contains just the solution parts that cause the failure (contrary to ABT-comp that reports all the solution with the different variables). So, instead of deleting just one solution after the reception of a nogood message, we can delete a group of solutions that contains the responsible variable value. Besides, the filtering process used in MP-ABT minimizes the number of checked constraints.

5.5 Conclusion

In this chapter, we presented the complete Minimal Perturbation based Asynchronous Backtracking algorithm **MP-ABT**, which is an upgrade of the ABT algorithm to solve DisCSPs with complex local problems while minimizing perturbations locally.

The existing methods are based on the compilation of the complex agent domains or the decomposition to make local problems as simple, so as the DisCSP algorithms can be applied without any modification. Whereas MP-ABT manages the trade-off between the complex local problems and the distributed search effort, to minimize the local changes for minimizing the agents' perturbations.

The MP-ABT algorithm examines each complex local problem as an MPP problem and each received message as a perturbation of the intra-agent constraints. After each message reception, the MP-ABT agent returns the closest local solution to its current one, using the HS-MPP algorithm.

Experimentation results show that the MP-ABT algorithm outperforms the different ABT versions, in terms of the number of exchanged messages and therefore the number of perturbations while minimizing the computational effort, especially when problems become dense and contain more variables per agent.

In the next chapter, we are going to propose a most general approach that can reinforce several DisCSP algorithms to support DisCSPs with Complex local problems.

6

Selective Sorting and Smart Nogood

Contents

6.1 Introduction

We proposed in the previous chapter the MP-ABT, an upgrade of ABT. It considers each complex local problem as a Minimal Perturbation Problem (MPP). The agent constructs a new MPP problem, solves it, and communicates the picked up result, every time a local solution is needed. This process allows recovering a new local solution with minimal changes against the last agent local solution, so as to disturb fewer agents and then rise the ABT's performance. The perturbation minimization did not take into consideration the number of agents linked, the number of variables of each linked agents, and even the identity of those agents.

In the first part of this work, we are going to continue with the same minimal reasoning, but by improving the compilation method, which is one of the most used methods, and taking other parameters into consideration in the minimization process. In the second part, we are going to enhance the nogood construction. Those proposed methods can be applied to several DisCSP algorithms, so as they can solve DisCSPs with complex local problems.

The improvements brought to the compilation are based on *i)* a Selective Sorting (SS) of the local solutions making up the compiled domain, using the COP formalism, and *ii)* a Smart Nogood (SN) construction.

The chapter will be organized as follow: We explain the Selective Sorting (SS) method, the algorithm, as well as its properties in section 6.2. We do the same thing for the second method Smart Nogood (SN) in 6.3, and we conclude the chapter in section 6.4.

6.2 Selective Sorting (SS)

In our first contribution, we consider the local problems as Constraint Optimization Problems (COPs), instead of CSPs, so as we can insert some preferences serving to hold steady the complexity of the DisCSP algorithms.

6.2.1 Description

The complex local problems are, by definition, CSPs plus objective functions to optimize. In the first contribution SN, we consider each local problem as a COP problem with four objective functions to minimize:

- Hamming Distance (HD);
- Lower Priority Agents (LPA);
- Constraints of Changed Variables (CCV;
- Most Priority Agents (MPA).

The use of an existing multi-objective COP algorithm can solve the preference problem, and give a local solution. Although, the COP algorithm is going to look for the set of local solutions each time a local solution is required, whereas the set of the CSP problem solutions are already prepared once for all. To use the last cited advantage and optimize the choice at the same time, we are going to keep the compiled domain and just reorder it, according to our preferences, and using our Selective Sorting method.

6.2.2 Algorithm

After the reception of a message Ok?, CPA or nogood, according to the used DisCSP algorithm, making the current local solution of the receiver inconsistent, our first contribution takes place (Algorithm 9). This method is based on four steps:

1. Sort the domain in ascending order of the Hamming Distance (HD) of each browsed local solution, against the current local solution of the agent (SortDomain() procedure, lines 4 and 5 and HD() function).

   The HD(*Csol*, *Lsol*) function returns the Hamming Distance of two solutions *Csol* and *Lsol*2. Knowing that the two solutions are two instantiations of the same local problem, they have the same number of variables. The function initializes a counter C with 0 (line 23). goes through the variables of the first solution (line 24) or the second, since they have the same size. For each tested variable, the function checks if its value is not equal to the other local solution variable value. If so, the counter is incremented by 1.

   Let's take the agent A_1 of our DisMSP example (figure **??**), which has three variables $M_{1.1}$, $M_{1.2}$, $M_{1.5}$. We assume that the Current local Solution of A_1 is the first one *Csol* = ($M_{1.1} = 1$, $M_{1.2} = 2$, $M_{1.5} = 3$), and we have to return the HD of this solution

Algorithm 9 Selective Sorting (SS)

```
1: procedure SORTDOMAIN((Solution Csol))
2:     for j=1 to domain.size do
3:         for k=j+1 to domain.size do
4:             if HD(Csol,domain(k)) < HD(Csol,domain(k)) then
5:                 Swap (domain(j),domain(k));
6:             else if HD(Csol,domain(k)) = HD(Csol,domain(k))
7:                 if LPA(Csol,domain(k)) < LPA(Csol,domain(k)) then
8:                     Swap (domain(j),domain(k));
9:                 else if LPA(Csol,domain(k)) = LPA(Csol,domain(k))
10:                    if CCV(Csol,domain(k)) < CCV(Csol,domain(k)) then
11:                        Swap (domain(j),domain(k));
12:                    else if CCV(Csol,domain(k)) = CCV(Csol,domain(k))
13:                        if MPA(Csol,domain(k)) < MPA(Csol,domain(k)) then
14:                            Swap (domain(j),domain(k));
15:                        end if
16:                    end if
17:                end if
18:            end if
19:        end for
20:    end for
21: end procedure

22: function HD((Solution sol1, Solution sol2))
23:     C ← 0;
24:     for i=1 to sol1.nbrOfVariables do          ▷ sol1 and sol2 have the same number of
    variables
25:         if sol1.getvariable(i) ≠ sol1.getvariable(i) then  C ← C + 1;
26:         end if
27:     end for  return C;
28: end function

29: function MPA((Solution sol1, Solution sol2))
30:     MPA ← "";
31:     for each Agent from LPA(sol1,sol2) do
32:         if Agent < MPA then  MPA ← Agent;
33:         end if
34:     end for  return MPA;
35: end function
```

with respect to the Local Solution $Lsol$ = ($M_{1.1} = 1$, $M_{1.2} = 3$, $M_{1.5} = 2$). The HD function compares the values of $M_{1.1}$ in $Csol$ and $Lsol$. We find that they are equal. In this case, the counter is still equal to 0. Next, it compares the values of $M_{1.2}$ in the two solutions and notes that they are different. So, the counter becomes equal to 1 this time. The same thing is done with the third variable $M_{1.5}$ which causes the increase of the counter to 2. The function returns finally the value 2 (Table 6.1).

```
36: function LPA((Solution sol1, Solution sol2))
37:     LPA = ;
38:     for i=1 to sol1.nbfOfVariables do
39:         if sol1.getvariable(i) ≠ sol2.getvariable(i) then
40:             for each Agent from sol1.getvariable(i).getConnectedLowerPriorityAgent()
    do
41:                 if Agent ∉ LPA then add Agent to LPA;
42:                 end if
43:             end for
44:         end if
45:     end for return LPA;
46: end function

47: function CCV((Solution sol1, Solution sol2))
48:     CCV = ;
49:     for i=1 to sol1.nbfOfVariables do
50:         if sol1.getvariable(i) ≠ sol2.getvariable(i) then
51:             for each Constraint from sol1.getvariable(i).getCstrOfLowerPriorityAgent()
    do
52:                 if Constraint ∉ CCV then add C to CCV;
53:                 end if
54:             end for
55:         end if
56:     end for return CCV.size;
57: end function
```

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
Csol	1	2	3
Lsol	1	3	2
Accumulated HD	0	1	**2**

TABLE 6.1: HD computation example

2. If the HD of two local solutions are equal (SortDomain procedure, line 6), the local solutions are sorted against the LPA, the number of Lower Priority Agents which are going to be affected by the changes (SortDomain procedure, lines 7 and 8 and LPA function). For example, if the HD of a local solution is equal to 2, we compute the number of lower priority agents, which are going to be affected by these two changes. If the two variables are connected only with one agent, the LPA returns the value 1.

   The LPA (*Csol*, *Lsol*) function shows the instructions to follow to get the value of the LPA of two solutions *Csol* and *Lsol*. The function creates an empty list LPA (line

30), roams the variables of one of the two solutions, variable by a variable (line 31), and checks if the variable values in the two solutions are not equal (line 32). If so, it gets the set of lower priority agents connected with the tested variable (line 33) and adds those not already existing in the LPA list (line 34).

Let apply this function to the same example as the previous step. The first variable $M_{1.1}$ does not change its value, so the LPA is still empty. While the second variable $M_{1.2}$ is changed and connected to A_2. In this case, the LPA list is alimented by A_2. Whereas the third variable $M_{1.5}$ is changed but it is not connected with any lower priority agent, thus the function returns a list with just one element A_2 (Table 6.2).

3. If the LPAs of two local solutions are equal (SortDomain procedure, line 9), we sort against CCV, the number of **C**onstraints linking the **C**hanged **V**ariables with those of the Lower Priority Agents (SortDomain procedure, lines 10 and 11 and CCV function). For example, if two variables are changed, and the first variable is connected via two constraints, and the second with just one constraint, the CCV returns the value $3 = 2 + 1$.

   The details of this step are shown in the CCV function. It follows the same treatment as the previous function, in its three first steps (lines 41, 42 and 43). But, instead of recuperating the set of Lower Priority Agents, it gets the variables of these agents and adds those not already existing to the CCV list. In the end, the function returns the size of the CCV list.

   n the earlier example, the second variable $M_{1.2}$ is connected with just $M_{2.2}$ of the agent A_2, while $M_{1.5}$ is not connected to any variable or agent. In this case, the function returns the value 1 (Table 6.3). The number of variables connected to the first variable $M_{1.1}$ is not computed since its value is not changed.

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
Csol	1	2	3
Lsol	1	3	2
Connected agents	$\{A_2, A_3\}$	$\{A_2\}$	$\emptyset$
Accumulated LPA	$\emptyset$	$\{A_2\}$	$\{A_2\}$

TABLE 6.2: LPA computation example

	$M_{1.1}$	$M_{1.2}$	$M_{1.5}$
Csol	1	2	3
Lsol	1	3	2
Connected variables	$\{M_{2.1}, M_{3.1}\}$	$\{M_{2.2}\}$	$\emptyset$
Accumulated CCV	$\emptyset$	$\{M_{2.2}\}$	$\{M_{2.2}\}$
CCV size		**1**	

TABLE 6.3: CCV computation example

4. If the CCVs of two solutions are equal (SortDomain procedure, line 12), we sort according to the identities of connected agents, not only their number (SortDomain procedure, lines 13 and 14 and MPA function). If the first solution will affect the two agents A_i and A_j ($A_i < A_j$) and another solution will disturb A_k and A_l ($A_k < A_l$). the MPA compares the priority of the highest priority agent in the two cases A_i and A_k. If A_k is the lowest priority agent, the corresponding local solution is placed before, assuming that changing a solution of a lower priority agent will not disturb more, in comparison to changing the solution of a higher priority agent.

   The MPA(*Csol*, *Lsol*) function generates a string (line 52). Then, it roams the list of Lower Priority Agents (recovered by the function LPA applied on the two solutions *Csol* and *Lsol*), agent by agent (line 53). If the agent tested is a higher priority than MPA, the MPA is changed by the name of the current agent. Since the priority is made lexicographically, the test of priority is made according to the lexicons of strings (line 54).

The HD of the local solution, corresponding to the inconsistent current local solution, takes always the greatest value. It is equal to the number of local variables + 1, starting from the premise that an inconsistent solution should not be chosen.

The transition to the last three steps is done if and only if the local solution has a Hamming Distance smaller than the number of variables. If the HD is equal to the number of variables, the whole local solution is going to be modified and all the other measures are going to be the same.

The transition from one step to another is possible if and only if the preceding step gives equal values.

Sorting the compiled domain according to the Hamming Distance, when looking for a new local solution, is going to give good results since the idea is taken from the Minimal Perturbation based ABT (MP-ABT) algorithm. It considers each complex local problem as

a Minimal Perturbation Problem and uses the HS-MPP algorithm to find a local solution with minimal disruption, whenever it is required. The HS-MPP is based on this famous Hamming Distance and the results presented were very interesting.

6.2.3 Properties

This improvement will, in no way, change the used DisCSP algorithm properties. if it is complete it will remain complete, because the SS method does not remove any value from the domain nor change its content. It just reorders local solutions of the compiled domain, which still static when looking for a new local solution. So, the agent does not miss any value. This is true for the termination and the soundness of the used DisCSP algorithm.

6.2.4 Example

To well understand the Selective Sorting method, let take our DisMSP example (Figure 1.2). As we have seen in Chapter 4, the compiled domain of A_1 contains 120 local solutions. To reduce the number of local solutions, let assume that the variables' domains have only the three first values {1,2,3}. In this case, the compilation produces a new abstract variable with 6 local solutions (Figure 6.1, Table (0)).

After receiving a message requiring the change of A_1 current local solution, which is supposed to be equal to $Csol$ = ($M_{1.1} = 1$, $M_{1.2} = 2$, $M_{1.5} = 3$), A_1 sorts its compiled domain following the SS's four steps. The resulted sorted domain is shown in figure 6.1.

6.3 Smart Nogood (SN)

6.3.1 Description

The second contribution concerns the nogood construction. In ABT, AFC-ng, or any other algorithm, relying on nogoods in their resolution, the built nogood is expressing that an instantiation of this agent is blocking the tested value. In the case of a Complex Local Problem, summed up in a variable whose domain is the set of local solutions found during the compilation process, the principle of the generation of a nogood remains the same. The generated nogood involves a whole local solution, without destining the true

steps	updated domain
(0)	$M_{1.1}$ | $M_{1.2}$ | $M_{1.5}$ $\underline{1}$ | $\underline{2}$ | $\underline{3}$ 1 | 3 | 2 2 | 1 | 3 2 | 3 | 1 3 | 1 | 2 3 | 2 | 1
(1)	$M_{1.1}$ | $M_{1.2}$ | $M_{1.5}$ | **HD** 1 | 3 | 2 | **2** 2 | 1 | 3 | **2** 3 | 2 | 1 | **2** 2 | 3 | 1 | **3** 3 | 1 | 2 | **3** $\underline{1}$ | $\underline{2}$ | $\underline{3}$ | **4**
(2)	$M_{1.1}$ | $M_{1.2}$ | $M_{1.5}$ | **LPA** 2 | 1 | 3 | **0** 1 | 3 | 2 | **1** 3 | 2 | 1 | **1** 2 | 3 | 1 | **-** 3 | 1 | 2 | **-** $\underline{1}$ | $\underline{2}$ | $\underline{3}$ | **-**
(3)	$M_{1.1}$ | $M_{1.2}$ | $M_{1.5}$ | **CCV** 2 | 1 | 3 | **-** 1 | 3 | 2 | **1** 3 | 2 | 1 | **1** 2 | 3 | 1 | **-** 3 | 1 | 2 | **-** $\underline{1}$ | $\underline{2}$ | $\underline{3}$ | **-**
(4)	$M_{1.1}$ | $M_{1.2}$ | $M_{1.5}$ | **MPA** 2 | 1 | 3 | **-** 1 | 3 | 2 | **A3** 3 | 2 | 1 | **A3** 2 | 3 | 1 | **-** 3 | 1 | 2 | **-** $\underline{1}$ | $\underline{2}$ | $\underline{3}$ | **-**

FIGURE 6.1: The SS four steps, followed by the A_1, to sort its domain with respect to the solution ($M_{1.1} = 1, M_{1.2} = 2, M_{1.5} = 3$)

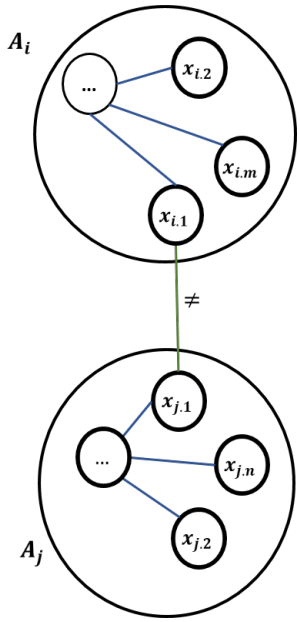


FIGURE 6.2: DisCSP example with complex local problems

cause of inconsistency and what is the exact inconsistent variable. In our second contribution, we propose to reason by variable instead of reasoning by agent/local solution, when generating a new nogood (Algorithm 10).

Take the example of a general agent A_i, handling m variables ($x_{i.1}$, $x_{i.2}$, ..., $x_{i.m}$), and connecting with another agent A_j, whose local problem contains n variables ($x_{j.1}$, $x_{j.2}$, ..., $x_{j.n}$). We assume that the two agents are linked with a binary constraint C:$x_{i.1} \neq x_{j.1}$ (Figure 6.2). When A_i chooses the value (V_1, V_2, ..., V_m), the variable $x_{j.1}$ of A_j should be different from V_1. In the ordinary reasoning, the A_j creates the nogood $A_i = (V_1, V_2, ..., V_m) \rightarrow A_j \neq (V_1, ..., V_n)$ for each local solution whose $x_{j.1}$ is equal to V_1, while a simple nogood as $x_{i.1} = V_1 \rightarrow x_{j.1} \neq V_1$ can resolve this issue. It can remove all local solutions which have the value V_1 in the variable $x_{j.1}$. This will decrease the storage and the number of exchanged messages and then the number of checked constraint.

6.3.2 Algorithm

Algorithm 10 shows the GetJustification function. It points out the detailed instructions to follow to get the Smart Nogood, justifying the inconsistency of a solution regarding a received one. The function takes in parameters the higher priority agent's local solution senderSol, the current local solution Csol of the agent desiring to check its consistency against the received one, and the constraint cstr to test.

Algorithm 10 Smart Nogood (SN)

```
1: function GETJUSTIFICATION(Solution senderSol, Solution Csol, Constraint cstr)
2:     ngd ← "";
3:     if !cstr.isSatisfied(senderSol, Csol) then   ▷ cstr contains the name of the variables
   and the constraint type
4:         for each element from SenderSol.getVariables do        ▷ element is the variable
   with its value
5:             if cstr.getVariables.contains(element.getName) then
6:                 ngd ← ngd + element;    break;
7:             end if
8:         end for
9:         ngd ← ngd + "→";
10:        for each element from Csol.getVariables do
11:            if cstr.getVariables.contains(element.getName) then
12:                ngd ← ngd +element;    break;
13:            end if
14:        end for
15:    end if    return ngd;
16: end function
```

The function generates an empty nogood ngd (line 2) and checks if the Csol and the senderSol do not violate the constraint cstr (line 3). If so, the function returns the generated empty ngd. Otherwise, it roams the variables of SenderSol, to know which variable is connected by cstr (lines 4 and 5). The variable connected with cstr is going to be added to the left-hand side of ngd (line 6), to guarantee that the ngd left-hand side contains just the inconsistent variable.

In the same way, CSol's variables are roamed to identify which variable is responsible for the inconsistency (lines 10 and 11). The identified variable is going to be added on the right-hand side of ngd (line 12).

The returned nogood contains just one variable on the left-hand side and one other on the right-hand side. It is able to remove all local solutions handling the value that exists on its right-hand side.

6.3.3 Properties

The Smart Nogood method removes more values than the simple nogood from the domain. This does not affect the completeness, or the correctness of the DisCSP algorithms in any way since it has the same principle as the decomposition method which is proved that it keeps the DisCSP algorithm properties. It is used to consider each local variable as a virtual agent. The agent generates a nogood that calls into question the virtual agent (ie. the variable), not the real agent. This exactly what we do with the SN.

For the termination, the method helps to speed up the algorithms since it removes a set of local solutions with just one small nogood. it also reduces the number of exchanged messages as well as the non-concurrent constraints checks.

6.4 Conclusion

In this chapter, we proposed a new method that can be used by most DisCSP algorithms, to resolve DisCSPs with Complex Local Problems. It is based on the compilation's improvements, which replaces each complex agent by an abstract variable, whose domain is the set of local solutions of the agent. The first improvement is the Selective Sorting of the compiled domain, compared to four criteria HD, LPA, CCV, and MPA. The second improvement is based on reasoning by variable instead of agent/local solution in the construction of a Smart Nogood. We are going to apply these methods to some DisCSP algorithms, namely ABT and AFC-ng, and evaluate their effectiveness against the existing methods.

7

Selective Sorting and Smart Nogood based DisCSP algorithms

Contents

7.1 Introduction

The methods Selective Sorting of the complex agent compiled domain we have proposed to improve the compilation as well as the Smart Nogood can be integrated to the DisCSP algorithms so as they solve DisCSPs with complex local problems, especially those based on the nogoods. The ABT and the AFC-ng are the two most employed algorithms which use nogoods.

In this chapter, we are going to apply the two methods SS and SN into ABT and AFC-ng and compare the resulting algorithms ABT-SS&SN and AFC-ng-SS&SN with the existing algorithms to assess the proposed methods.

We are going to describe the propositions' details more accurately. Hence, the chapter proceeds as follows. Section 7.2.1 presents the first algorithm ABT-SS&SN Section 7.2.2 tackles the second one AFC-ng-SS&SN Section 7.3 shows the experimental results. And Finally, section 7.4 concludes the chapter.

7.2 Selective Sorting and Smart Nogood based DisCSP algorithms

The SS and SN methods are designed to be integrated into DisCSP algorithms, so as they can resolve DisCSPs with complex local problems. In this section, we are going to incorporate the two methods into ABT and AFC-ng.

7.2.1 Selective Sorting and Smart Nogood based ABT

The integration of SS and SN into ABT causes a modification in the ABT algorithm. Algorithm 11 shows the modified procedures and functions with respect to the ABT algorithm.

The two changed functions are *ChooseValue* and *Coherent*. In the *ChooseValue* function (line 1), which is applied when an agent looks to instantiate its variables/local problem, the agent sorts its compiled domain using the *SortDomain* procedure of the SS method (line 2). Afterward, it continues its process by browsing the compiled domain, solution by solution (line 3), while looking for a local solution that is not eliminated by its nogoodStore. For the last test, it is enough that the agent finds only one variable's value of its local problem is removed by a nogood, all the local solutions containing this variable's value are going to be eliminated by this nogood.

If a local solution is not removed by any nogood, the agent tests whether it is consistent with all the higher priority agents' local solutions, existing in its AgentView (line 5). To do this, it uses the function $GetJustification$ of the SN method (line 7). If the function returns an empty nogood, the tested constraint is checked and verified. Otherwise (line 8), the examined constraint is not consistent and the recovered nogood is going to be saved in the agent's *NogoodStore* (line 9) as justification for inconsistency.

If the agent crosses all instantiations existing in the AgentView without saving any nogood, that means that the tested local solution is consistent. Thereby, the function returns the tested value (line 15).

Algorithm 11 ABT-SS&SN algorithm (Changed parts in relation to ABT)

```
1:  function CHOOSEVALUE()
2:      SortDomain(myValue)                          ▷ integrate the first compilation improvement
3:      for each v ∈ D(self) not eliminated by myNogoodStore do                              ▷
    if only one value of a variable is deleted by a nogood, all v variables will be eliminated, (ie all the local
    solution will be deleted momentarily)
4:          coherent ← true;
5:          for each agent of myAgentView do
6:              for each Constraint of agent.getConstraints do
7:                  ngd ← GetJustification(v_j,self,Constraint);
8:                  if ngd ≠ "" then
9:                      add(ngd, myNogoodStore);
10:                     coherent ← false;                    ▷ one justification for an agent
11:                     break;
12:                 end if
13:             end for
14:         end for
15:         if coherent then return v
16:         end if
17:     end for    return (empty)
18: end function

19: function COHERENT(nogood, agents)
20:     for each var ∈ nogood ∪ agents do
21:         if nogood[var] ∉ myAgentView[var].getVariables then return false;
22:         end if
23:     end for
          return true;
24: end function
```

The second changed function *Coherent* (line 19) is used to test the validity of a nogood against the local solutions stored in the AgentView.

For each variable existing in the nogood (line 20), the agent does not check if this variable value is equal to the owner agent's local solution figuring in the AgentView. It checks only if the higher priority agent local solution contains the tested variable (line 21). If not, that means the variable value (in the AgentView) is different from that of the nogood. That implies the nogood is incoherent.

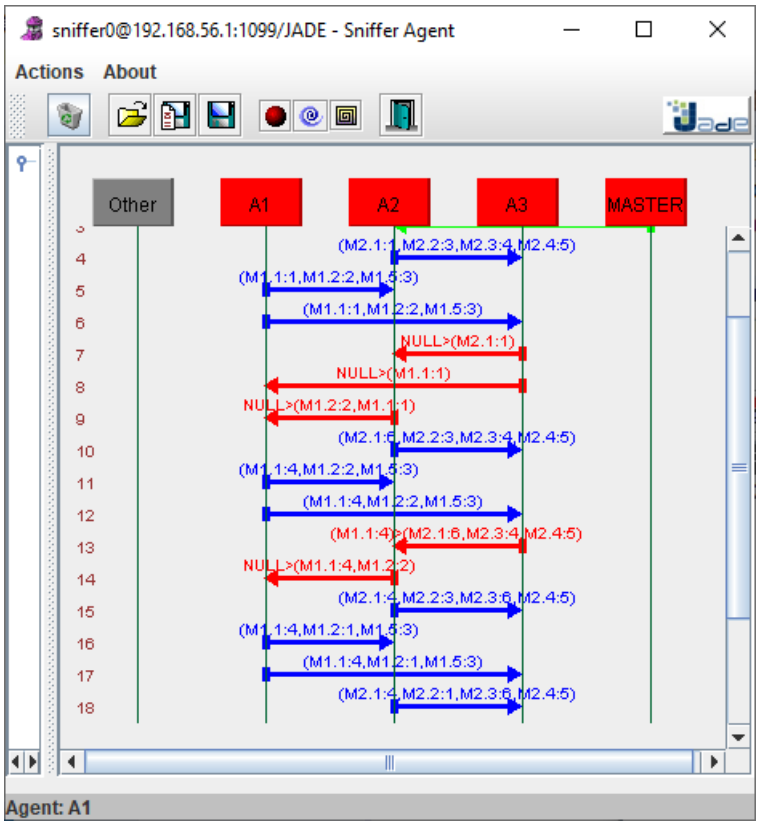


FIGURE 7.1: The Figure 1.2 problem resolution with ABT-SS&SN

Figure 7.1 shows the DisMSP example (Figure 1.2) resolution process, using the ABT-SS&SN algorithm.

The agents choose their first local solutions and send its to the lower agents via Ok? Messages. A_1 chooses the local solution ($M_{1.1} = 1, M_{1.2} = 2, M_{1.5} = 3$) and sends it to A_2 and A_3 (lines 5 and 6 in the figure 7.1's sniffer). While A_2 chooses the value ($M_{2.1} = 1, M_{2.2} = 3, M_{2.3} = 4, M_{2.4} = 5$) and sends it to A_3 (line 4).

After receiving the Ok? messages, A_3 finds that it can not choose any local solution because of the value A_1's $M_1.1 = 1$ and A_2's $M_2.1 = 1$. Therefore, it constructs nogoods according to the SN method and sends them to the responsible (lines 7 and 8). Similarly, A_2 is blocked because of two variables $M_1.1 = 1$ and $M_1.2 = 2$ of the same agent A_1 (line 9).

After receiving the nogood $\emptyset \leftarrow M_{1.1} \neq 1$ from A_3, A_2 deletes all local solutions having the value $M_{1.1} = 1$ from its compiled domain, sorts the domain using the SS method, and choose the first consistent local solution $M_{2.1} = 6, M_{2.2} = 3, M_{2.3} = 4, M_{2.4} = 5$. The SS has permitted to change only the first variable's value, which has the HD = 1. A_1 does the same treatment as A_2 and chooses the local solution $M_{1.1} = 2, M_{1.2} = 2, M_{1.5} = 3$ which differs from its first one by only one value $M_{1.1} = 2$.

The new local solution of A_2 is blocking A_3. This is why A_3 sends a new nogood to A_2 reporting all its local solution. The latter is also blocked. Therefore, it sends a new nogood to the highest priority agent (line 14). The sent nogood is a resolution of the stored nogoods as well as the newly received one.

The agents continue this process, using the SS before selecting a new instance and the SN in generating nogoods justifying the inconsistency of their local solutions until silence is detected or an empty nogood is generated. After only 15 messages, ABT-SS&SN finds a solution to our DisMSP example.

7.2.2 Selective Sorting and Smart Nogood based AFC-ng

As we did for the ABT algorithm, we are going to integrate the two enhancements SS and SN to the latest complete DisCSP algorithm AFC-ng. This merge gives birth to the AFC-ng-SS&SN algorithm. The changes compared to the original algorithm are shown in algorithm 12.

The changed methods in AFC-ng-SS&SN are *Revise* and *CheckAssign* procedures and the methods testing the compatibility of a nogood against the AgentView, namely the *Backtrack*, *ProcessNogood*, and $UpdateAgentView$. The algorithm 12 presents the *Revise* procedure, the *CheckAssign* procedure, as well as the $Compatible$ function which is called by the functions checking the compatibility of a nogood.

In *Revise* procedure, the agent checks the consistency of its local solutions against the values existing in its AgentView. It goes through its compiled domain (Revise procedure, line 2). For each tested local solution, it browses its AgentView (Revise procedure, line 3) to test the coherence of the local solution taken by each higher priority agent with the tested local solution, against the constraints linking the tested variables. For each constraint (Revise procedure, line 4), the agent calls the $GetJustification$ function of the SN algorithm (Revise procedure, line 5), which returns a nogood. If the nogood is empty, the constraint tested is checked and verified. Otherwise (Revise procedure, line 6), it is unsatisfied, and the nogood returned will be stored in its NogoodStore as a justification of the inconsistency. If there is, already, a nogood that removes the tested value, the agent stores the best one between the stored nogood and the generated one, using the HPLV method (Revise procedure, line 7).

In *CheckAssign* procedure, the agents sort their compiled domains (line 15), using the SS method, before choosing its local solution.

Algorithm 12 AFC-ng-SS&SN algorithm (Changed parts in relation to AFC-ng)

```
1:  procedure REVISE
2:      for each (v ∈ D^0(x_i)) do
3:          for each agent of myAgentView do
4:              for each Constraint of agent.getConstraints do
5:                  ngd ← GetJustification(v_j,self,Constraint);
6:                  if ngd ≠ "" then
7:                      Store the best nogood for v between ngd and the stored nogood, if it already exists; ▷
    according to the HPLV
8:                  end if
9:              end for
10:         end for
11:     end for
12: end procedure

13: procedure CHECKASSIGN(sender)
14:     if predecessor(A_i = sender) then
15:         SortDomain(Csolution);
16:         Assign();
17:     end if
18: end procedure

19: function COMPATIBLE((nogood, AgentView))
20:     for each element of nogood do
21:         if element ∉ AgentView.get(element.owner) then return false;
22:         end if
23:     end for
24: return true;
25: end function
```

The *compatible* function returns true if a nogood is compatible with an AgentView. As in ABT-SS&SN, the AFC-ng-SS&SN agent tests if each element of nogood exists in the AgentView (*Compatible* function, line 21). if it is not, the whole nogood is incompatible.

Figure 7.2 shows the DisMSP example (Figure 1.2) resolution process, using the AFC-ng-SS&SN algorithm.

The agents follow a bit the same steps as ABT-SS&SN. The difference is seen in the messages types. The highest priority agent A_1 chooses the first local solution, generates the CPA structure, and sends it to A_2 and A_3. The successor agent sends its local solution when it receives the CPA from its predecessor.

The number of messages differs because of obsolete nogoods. Otherwise, the agents use the same methods to choose a new local solution when it is required and to store nogoods in their NogoodStores.

AFC-ng-SS&SN keeps the same priorities of AFC-ng, including the way the algorithm stops.

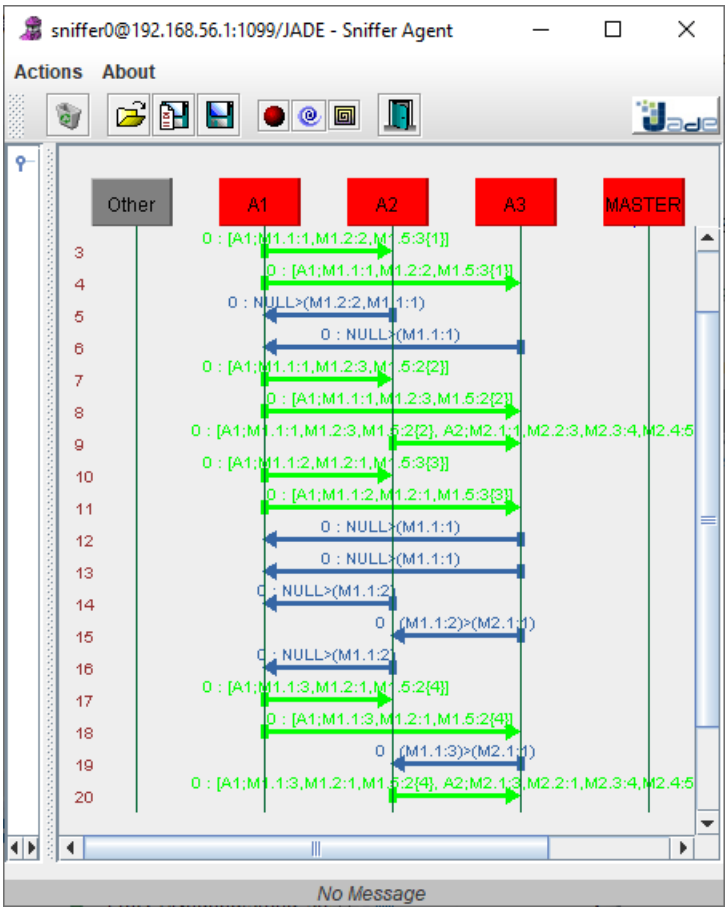


FIGURE 7.2: The Figure 1.2 problem resolution with AFC-ng-SS&SN

7.3 Experimental Results

During the conception of the algorithms, we expected that our new algorithms will outperform the existing approaches. We have to prove that either theoretically or experimentally. This is not feasible theoretically, since we do not use mathematical equations. So, the only way we have to use is to experiment with our proposed algorithms against the existing ones. The experiments should cover all areas, in order to draw strengths as well as the weaknesses. Testing the algorithms in problems with different parameters will help to identify in which regions our proposed algorithms are strong and likewise the opposite.

In order to evaluate the validity of the ABT-SS&SN algorithm and to bring out its strong points as well as its weak ones, we have to compare it to existing algorithms. To do this, we tested the behavior of the ABT-SS&SN against ABT-Comp and ABT-cf. The same for AFC-ng-SS&SN, we are going to compare it with AFC-ng-comp and AFC-ng-cf.

We evaluate the algorithms on problems characterized by six parameters (n, d, p_1, p_2, p_3, p_4), where n is the number of agents, d the number of variables per agent, p_1 is the

constraints' density, p_2 is the inter-agents density, p_3 is the connection density, and p_4 is the constraint tightness.

In the experiments, we take the parameters encompassing all problems' regions. We choose the most representative values which are: n=5 and d=5 (equivalent to 5*5=25 variables), p_1 varies from 10 to 70 by steps of 10, p_2 takes values between 20 and 100 by steps of 10 too, p_3 = 6, and p_4 takes two values 20 and 70 to represent the two types of problems, sparse and dense ones.

Experiments are performed on the number of exchanged messages (MSGs), that of the Concurrent Constraint Checks (CCCs), and the time in milliseconds.

To evaluate the impact of the problem complexity, the density, the inter-agent density and the constraints tightness on the proposed algorithms ABT-SS&SN and AFC-ng-SS&SN (sections 7.3.1 and 7.3.2), we plot the results of each algorithm on 3D for each complexity kind (p_4 = 20 for sparse graphs, and p_4 = 70 for dense one).

Section 7.3.1 represents the experimental results of ABT family algorithms, namely ABT-comp, ABT-cf and ABT-SS&SN. While section 7.3.2 shows the behaviors of AFC-ng family algorithms (i.e. AFC-ng-comp, AFC-ng-cf and AFC-ng-SS&SN).

7.3.1 ABT Family Algorithms

Figure 7.3, respectively 7.4, shows the number of Exchanged messages, the number of Concurrent Constraints Checks and the consumed time in milliseconds by the ABT-SS&SN algorithm, against the constraints' density p_1 and the inter-agents density p_2, while resolving sparse problems, respectively dense problems.

The figures show that ABT-SS&SN behaves correctly, in all zones, like any other complete existing algorithm. The Concurrent Constraint Checks match, generally, the exchanged messages in the two types of problems. Furthermore, the consumed time does not exceed 8.10^4 Ms in sparse problems and 8.10^5 Ms in dense problems. Compared to the complexity of the problems, namely the number of variables and the number of constraints, these values are so effective.

Each time a local solution is required, the selective sorting of the domain appears to be a waste of time, and it can move beyond the solution in many cases. It is going to exchange more messages and test more constraints. To evaluate the effect of the sorting, we added only the SN to ABT (and even to AFC-ng).

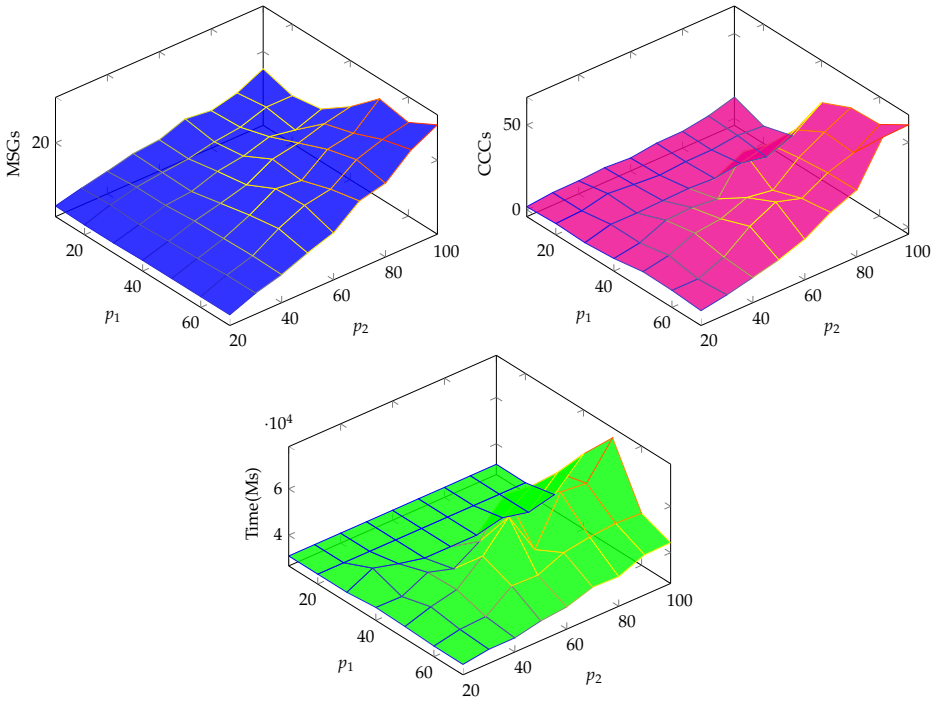


FIGURE 7.3: ABT-SS&SN behavior for $p_4 = 20$

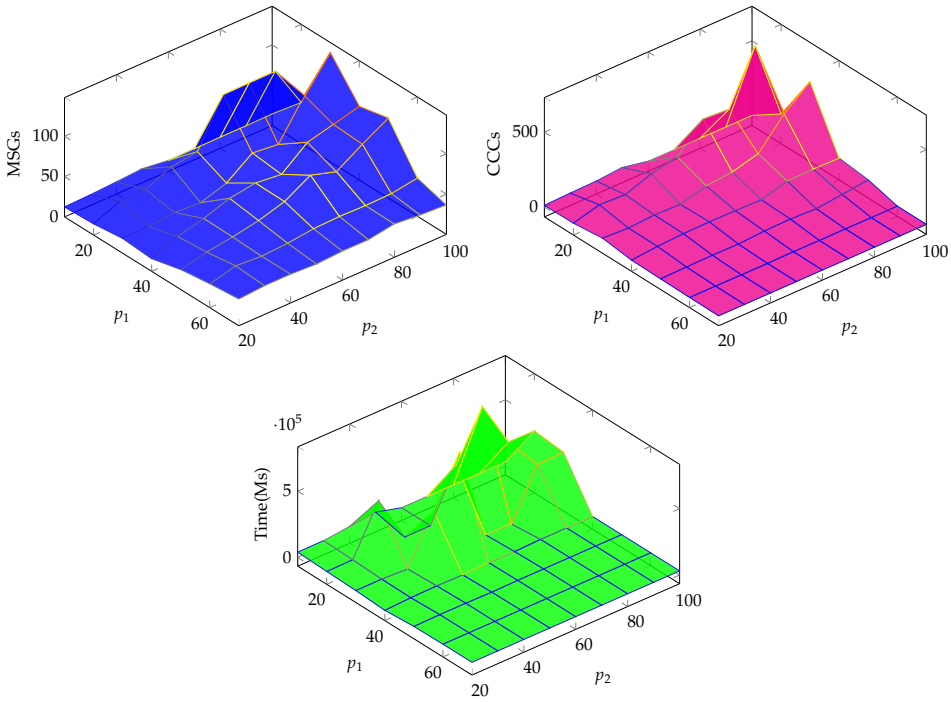


FIGURE 7.4: ABT-SS&SN behavior for $p_4 = 70$

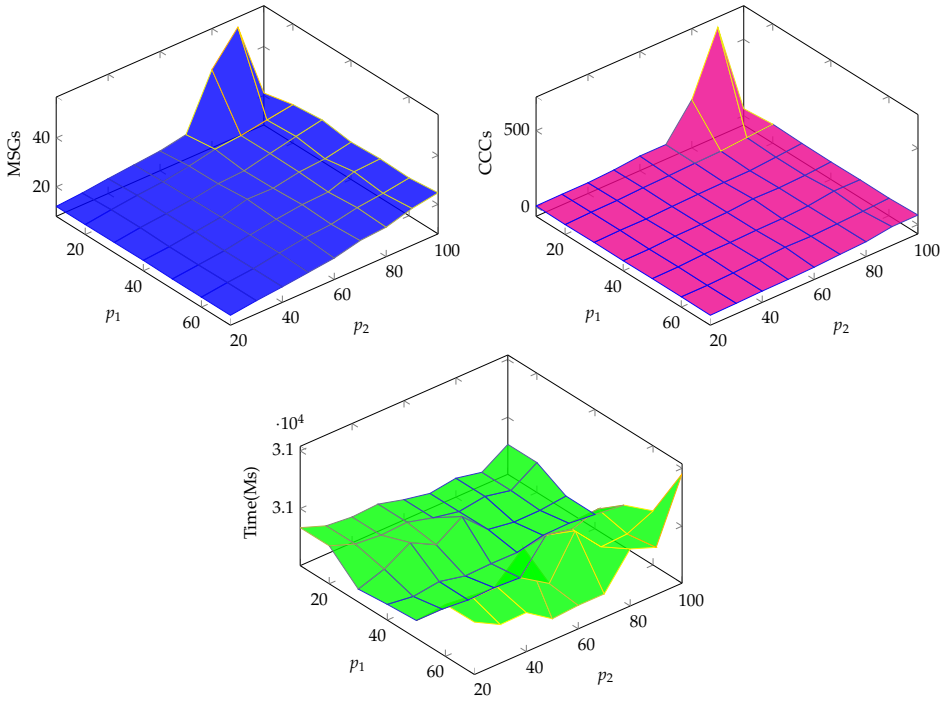


FIGURE 7.5: ABT-SN behavior for $p_4 = 20$

Figures 7.5 and 7.6 represent the ABT-SN behavior compared to the number of exchanged messages, Concurrent Constraint Checks, and time, in sparse problems (Figure 7.5), as well as dense ones (Figure 7.6).

In sparse graphs (Figure 7.5), the resolution time in the ABT-SN algorithm does not exceed $3,1.10^4 Ms$, compared to $8.10^4 Ms$ in the ABT-SS&SN. That expresses that sorting the domain in this kind of problem consumes more time. While in dense problems (Figure 7.6), ABT-SN can consume up to $1,5.10^6 Ms$ compared to $8.10^5 Ms$ with ABT-SS&SN, which proves that, in dense graphs, the Selective Sorting of the domain has a very important role in term of time.

Even if the resolution time of the ABT-SS&SN appears bigger than ABT-SN in sparse graphs, the number of exchanged messages in this type of problems (the maximum value is equal to 30 messages) and also in the densest problems (the maximum value is equal to 150 messages) is much less than those exchanged by ABT-SN (60 messages as the maximum value in sparse problems and $1,5.10^4$ in dense problems). We notice that the difference, between ABT-SS&SN and ABT-SN, in the number of messages in dense problems, is much greater. This proves that the sorting of domains becomes more and more important with the increase of the problem complexity, in terms of exchanged messages.

Regarding the Concurrent Constraint Checks, ABT-SN is better than ABT-SS&SN, in the two types of problems. So, we conclude that the two algorithms ABT-SS&SN and

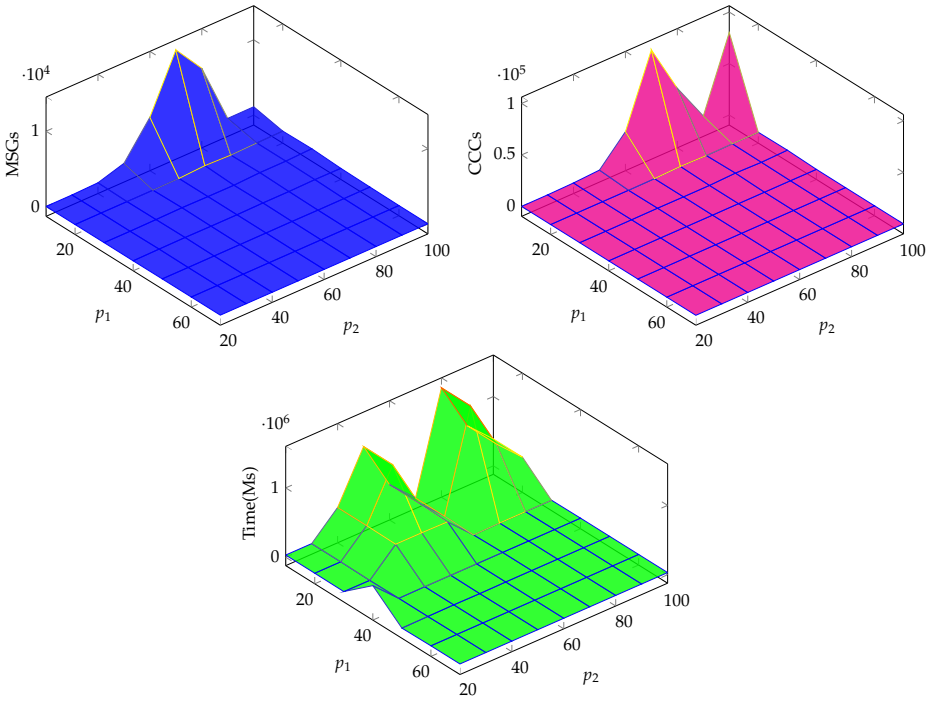


FIGURE 7.6: ABT-SN behavior for $p_4 = 70$

ABT-SN can be used according to the problem type. If we look for a quality solution, with minimum perturbation of agents, without taking into account the number of Concurrent Constraint Checks, the ABT-SS&SN is better than ABT-SN whatever is the problem complexity. But if a solution of a sparse problem is intended with a minimum of Concurrent Constraint Checks in a minimum time, we can adapt the ABT-SN instead of ABT-SS&SN. Then, when we have a dense problem, and we look for a solution with minimum perturbations, minimum exchanged messages and in minimum time, the ABT-SS&SN is better than ABT-SN.

Sparse Problems

We vary p_1 from 10 to 70 and p_2 from 20 to 100 for ABT-comp and ABT-cf, as we did in ABT-SS&SN and ABT-SN, to compare our proposed algorithms with the existing ones (ABT-comp and ABT-cf). Displaying the results in a 3D plot will not be effective because some graphs are going to hide others. To overcome this, we have selected the most representative results. We have chosen 2 values of p_1 (10 and 50) and for each one of them, we varied p_2 from 20 to 100. We did the same treatment to compare AFC-ng-comp and AFC-ng-cf with AFC-ng-SS&SN and AFC-ng-SN. The results of the last comparison are shown in subsection 7.3.2.

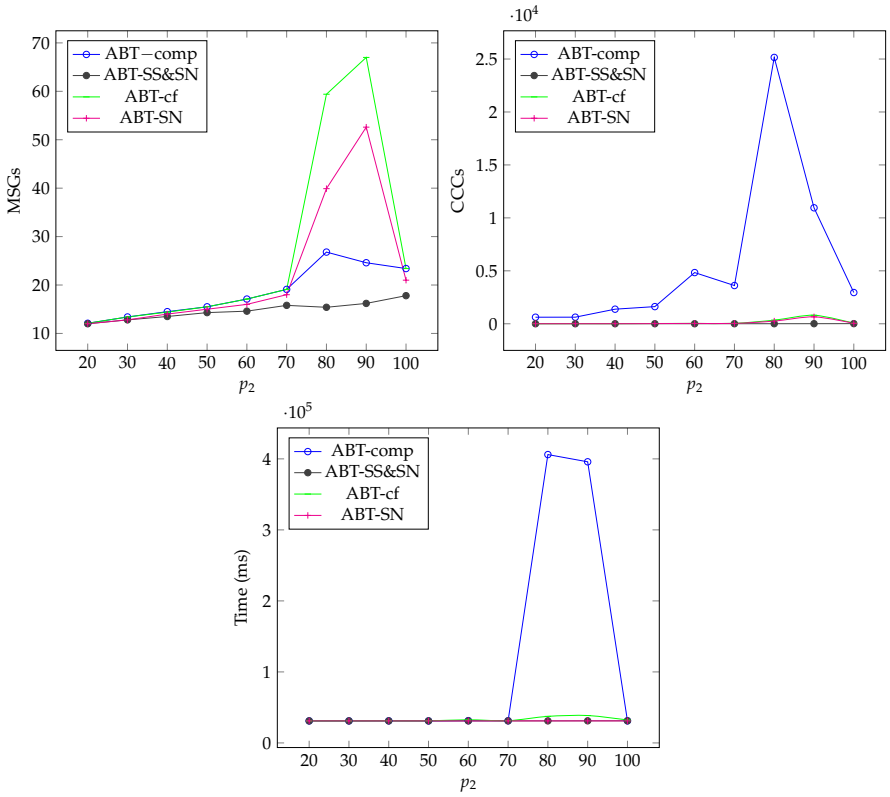


FIGURE 7.7: ABT family benchmarking, for $p_1 = 10$, $p_4 = 20$ and $p_2 \in [20, 100]$

Figure 7.7 represents the comparison results against the number of exchanged messages, that of Concurrent Constraint Checks, and the consumed time in Milliseconds, for $p_1 = 10$. The results show that ABT-SS&SN is the best one. It spends fewer messages, fewer CCCs and less time. The graph of ABT-SN shows that it is better than ABT-comp, which means that the manner the nogood is constructed has a great effect. We observe also that ABT-SN is not better than ABT-SS&SN, which proves that the Selective Sorting of the domain has also a great impact on the resolution. In addition, the ABT-cf which improves the ABT-comp surpasses ABT-comp in this kind of problem, in terms of exchanged messages.

For $p_1 = 50$, figure 7.8 shows that the 4 algorithms behave nearly in the same way, in terms of exchanged messages. But precisely, it is ABT-SS&SN algorithm the best. The ABT-SN algorithm is generally like the ABT-cf algorithm. In terms of constraints, our two algorithms ABT-SS&SN and ABT-SN outperform the others. For the time, we notice that ABT-SS&SN consumes more time, because the majority of problems in $p_1 = 50$ become insolvent, and sorting domains, in this case, becomes obsolete. In that event, ABT-SN is sufficient because the way the nogood is building is going to accelerate the detection of the insolvency of problems. This is what is shown in the results.

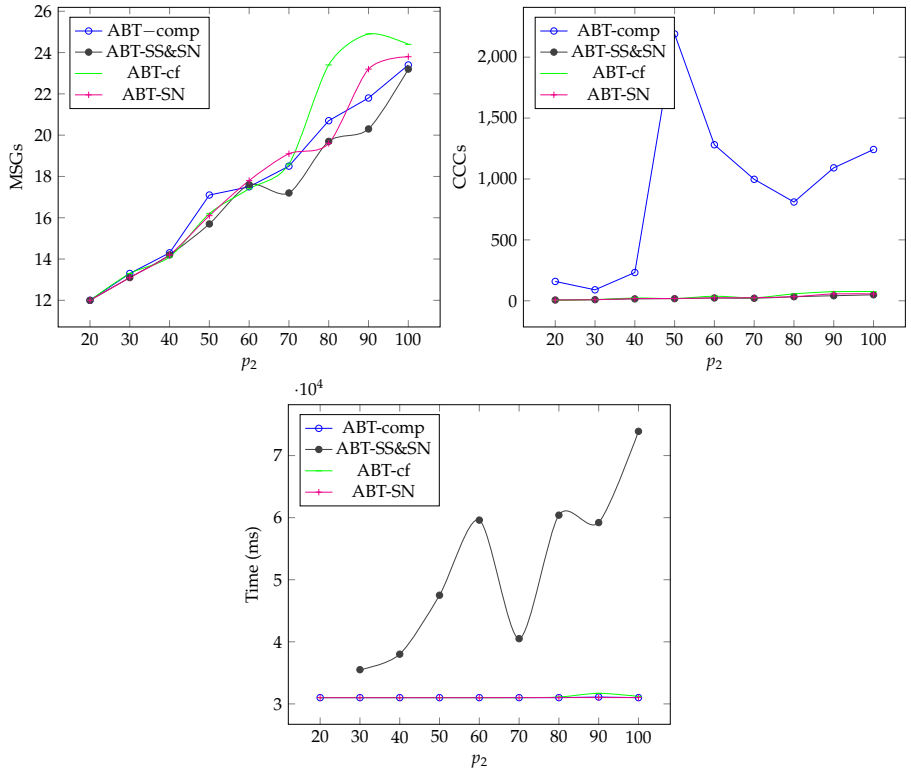


FIGURE 7.8: ABT family benchmarking, for $p_1 = 50$, $p_4 = 20$ and $p_2 \in [20, 100]$

Dense Problems

We experienced dense problems ($p_4 = 70$), in the same way as sparse ones.

Figure 7.9 shows the results of the experiments for $p_1 = 10$. We note that ABT-SN behaves in the same way as ABT-cf, which is normal in this kind of problem. In ABT-cf, the agent removes all interchangeable solutions after compilation, and during the resolution, it removes the NPI solutions, taking into consideration each constraint. Whereas in the ABT-SN algorithm, the agent reasons by variables instead of agents, in the construction of a nogood. This nogood removes all solutions that contain the variable(s) existing in the nogood. This process is equivalent to the two deletions made by the ABT-cf. This equivalence is not noticeable in the sparse problems because the ABT-SN becomes equivalent to ABT-cf when it starts generating nogoods. The more it generates the nogoods, the closer it gets to the ABT-cf, while minimizing storage. Whereas in sparse graphs, the generation of nogoods is very minimal.

For this value of p_1, we note that the ABT-SS&SN is the best in terms of exchanged messages, concurrent constraint checks and time, since it is not only based on the generation of nogood, but also on the Selective Sorting of the domain, which minimizes agent disruption, and so number of exchanged messages and concurrent constraint checks.

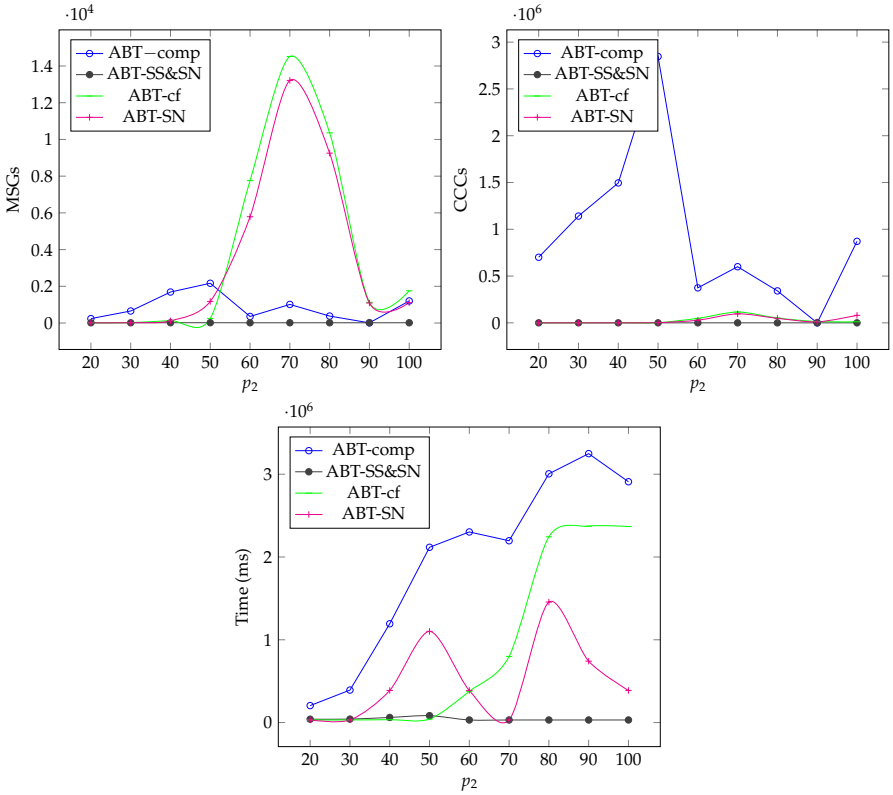


FIGURE 7.9: ABT family benchmarking, for $p_1 = 10$, $p_4 = 70$ and $p_2 \in [20, 100]$

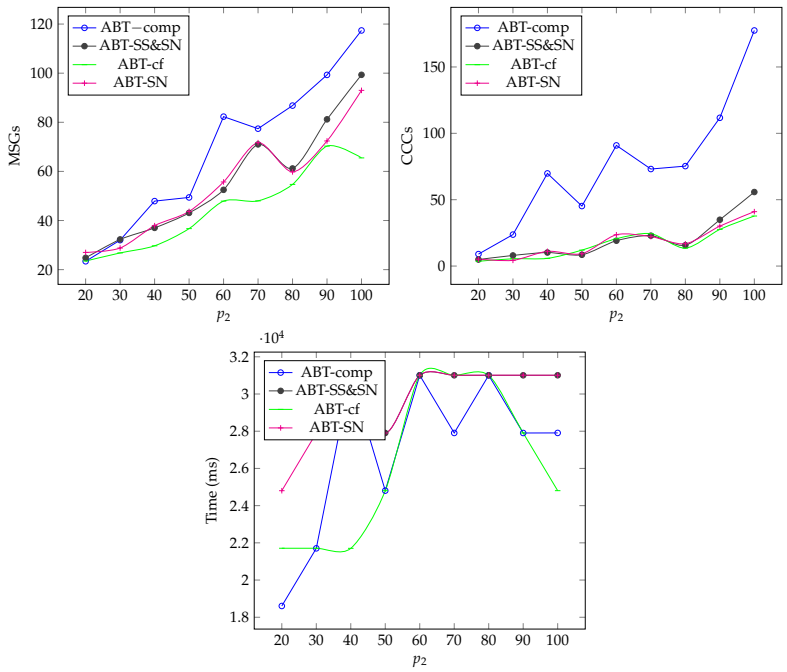


FIGURE 7.10: ABT family benchmarking, for $p_1 = 50$, $p_4 = 70$ and $p_2 \in [20, 100]$

For p_1 = 50 (figure 7.10), we perceive that ABT-SS&SN, ABT-SN, and ABT-cf have almost the same behavior, in terms of exchanged the messages and concurrent constraint checks. That proves the domains' selective sorting does not influence the resolution of this kind of problem. Because all agent variables are constrained with the other agents' variables. In this case, the four steps, made by the SS contribution, give equal values and the selection will be made in a random way. Contrariwise, the way we generate the nogoods influences the resolution. In the worst results, our algorithms, especially ABT-SN, behave like the best among the other algorithms.

7.3.2 AFC-ng Family Algorithms

As we did for the ABT family algorithms, we plot the graphs of AFC-ng-SS&SN (Figure 16 for sparse problems and Figure 17 for dense problems) and AFC-SN (Figure 18 for sparse problems and Figure 19 for dense problems) in 3D; and this according to p_1 and p_2.

We also compare AFC-ng-SS&SN and AFC-ng-SN with AFC-ng-comp and AFC-ng-cf in the same way as the ABT family algorithms.

Figures 7.11 and 7.12 show that AFC-ng-SS&SN behaves like ABT-SS&SN and as any other complete algorithm. The exchanged messages correspond to the constraint checks.

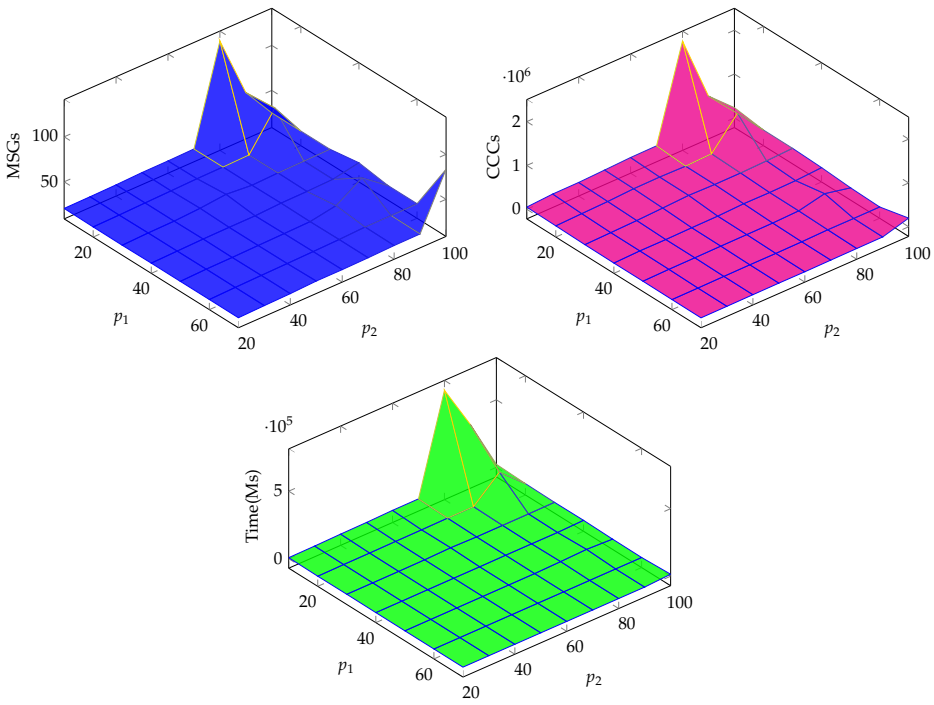


FIGURE 7.11: AFC-ng-SS&SN behavior for $p_4 = 20$

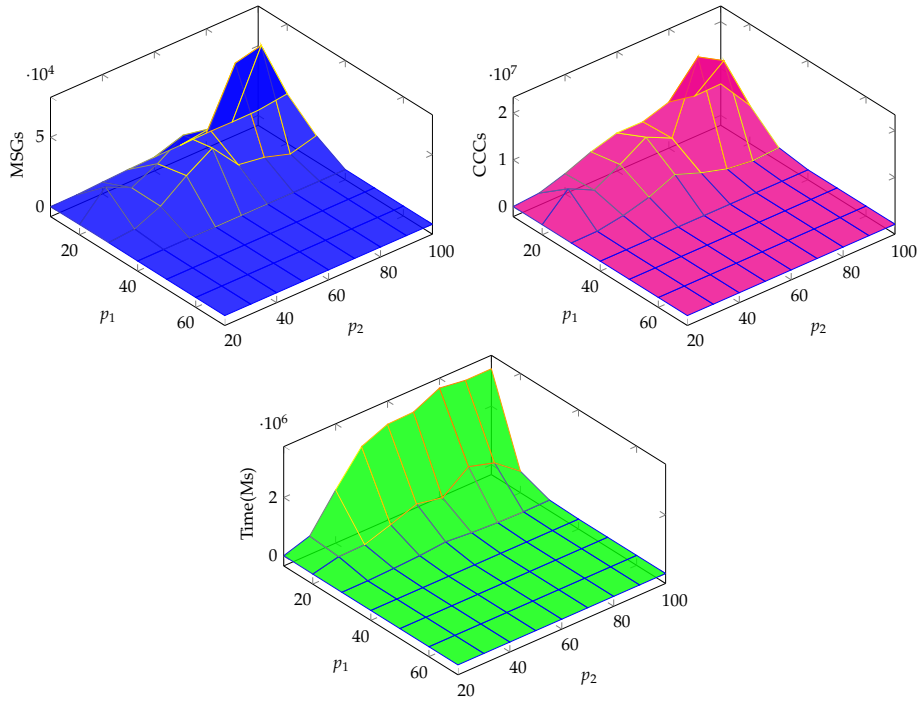


FIGURE 7.12: AFC-ng-SS&SN behavior for $p_4 = 70$

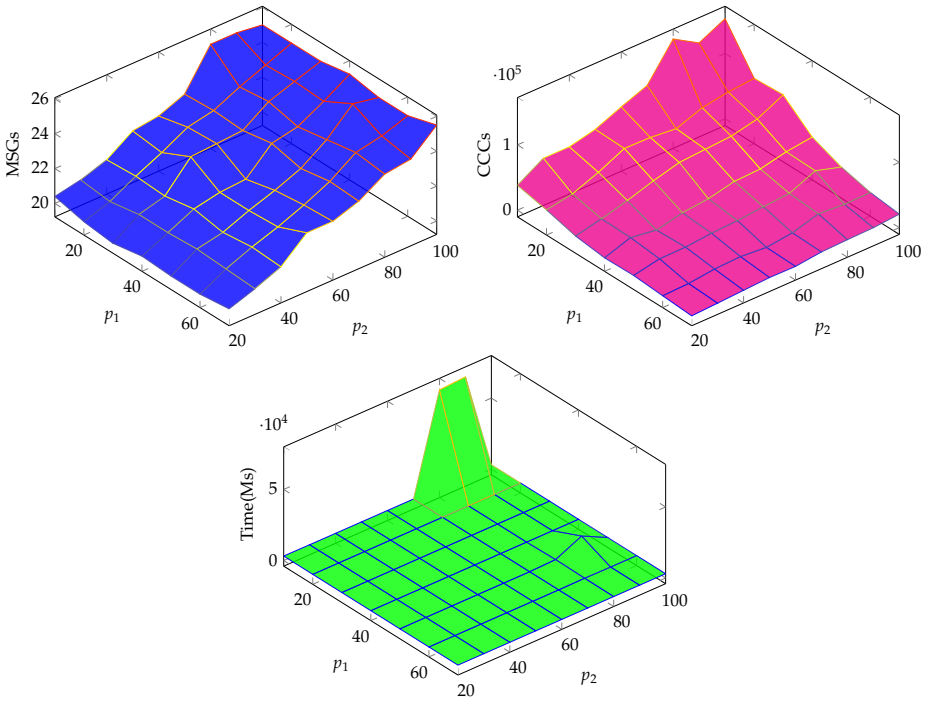


FIGURE 7.13: AFC-ng-SN behavior for $p_4 = 20$

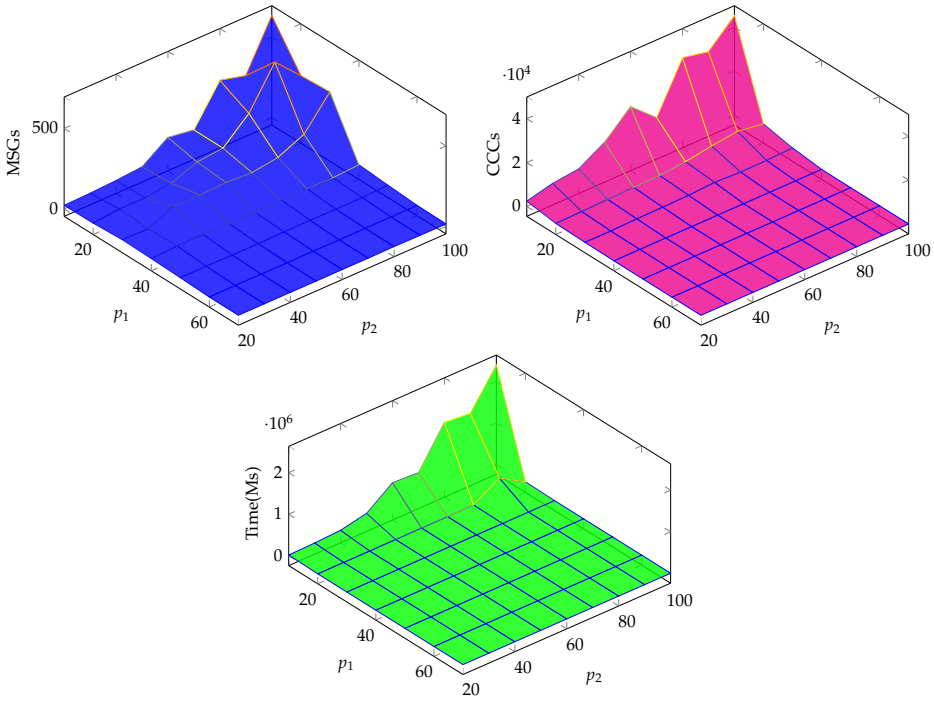


FIGURE 7.14: AFC-ng-SN behavior for $p_4 = 70$

We notice that they have the same look. The time in its maximum value does not exceed $10^6 Ms$ in sparse graphs and $4.10^6 Ms$ in dense ones.

For the AFC-ng-SN, the figures 7.13 and 7.14 show that, even for this algorithm, the exchanged messages correspond to the tested constraints, in both types of problems.

We note that the messages exchanged by AFC-ng-SS&SN are numerous compared to those exchanged by AFC-ng-SN and even the concurrent constraint checks. And this in both sparse and dense problems. This is just a global view of the paces, the details will be subsequently discussed.

Sparse Problems

In sparse problems, we remark that for $p_1 = 10$ (Figure 7.15) and $p_1 = 50$ (Figure 7.16), AFC-ng-SS&SN and AFC-ng-SN have the same behaviors, in terms of exchanged messages, and tested constraints. This means that sorting the compiled domain does not affect the resolution; but, when we see that they are the same, even in terms of time, we can say that sorting is almost not done in this type of problem. Why?

In this type of problem, there are few constraints, and the probability that the first local solution of each agent is consistent is very large.

In ABT, instantiations are done asynchronously. In the beginning, each agent takes a local solution and sends its to lower priority agents, without waiting for messages from

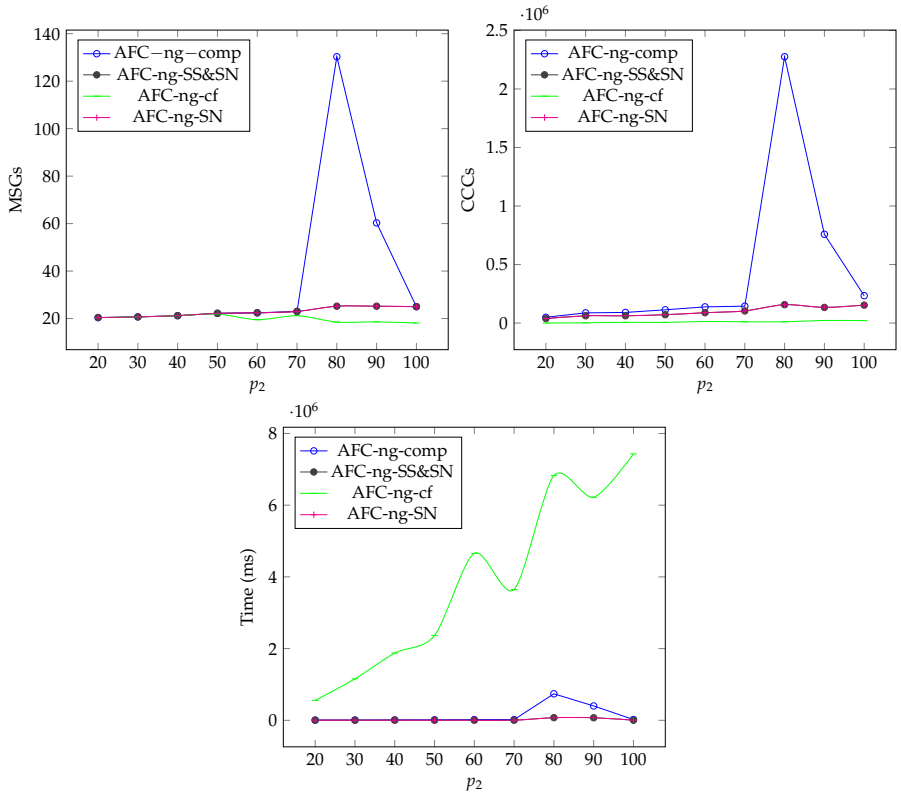


FIGURE 7.15: AFC-ng family benchmarking, for $p_1 = 10$, $p_4 = 20$ and p_2 varies from 20 to 100

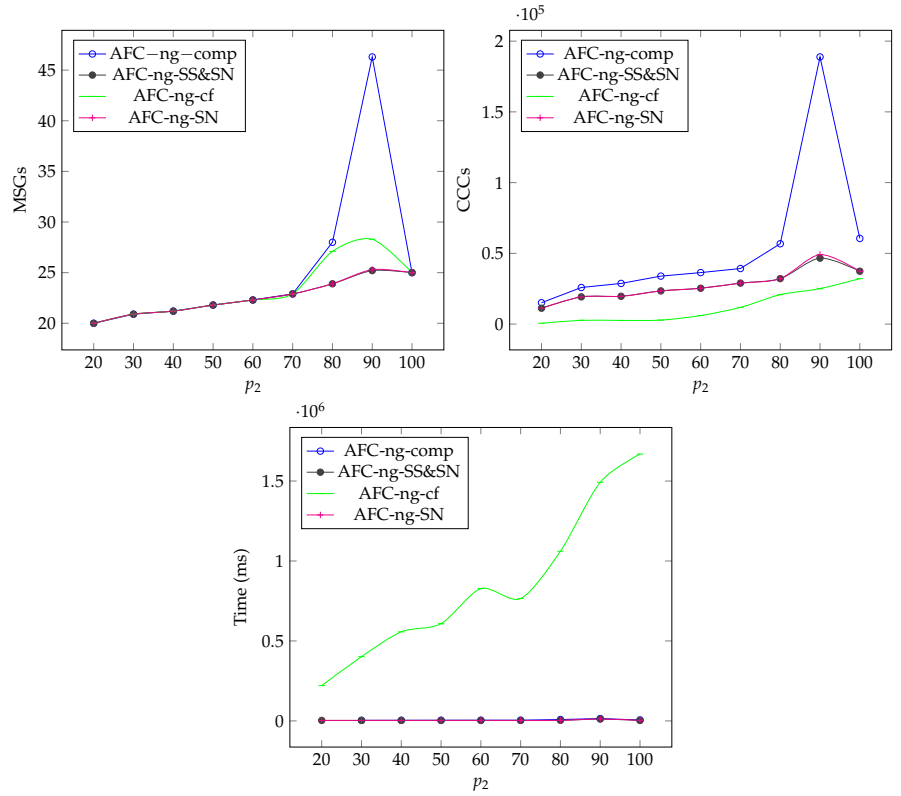


FIGURE 7.16: AFC-ng family benchmarking, for $p_1 = 50$, $p_4 = 20$ and $p_2 \in [20, 100]$

others. When these receive the value, they must, generally, look for a new consistent value. In this case, each agent receiving an Ok? message must sort its domain against its old solution, before searching a new one. So the sorting of the domain is taking place at least once, and the influence of the sorting appears.

While in AFC-ng, the agent instantiates its problem only when it receives the instantiation of its predecessor. So it will not sort the domain since it is its first instantiation. And since the problems are sparse, the transition to the sorting is very rare. That's why sorting is not done in these problems.

On the other hand, the SN contribution has an important role in the resolution. This is because AFC-ng-SN, as well as AFC-ng-SS&SN, are much better than AFC-ng-comp for all values of p_1. But, we notice that the AFC-ng-cf is a bit better in terms of concurrent constraint checks because the size of each agent domain is reduced before the resolution launching, by applying the interchangeability method. In this case, the agents test fewer values and fewer constraints. While with our nogood contribution, we reduce the size of the domain, just, after the generation of nogoods. Whilst nogoods are very rare in sparse problems.

Although this does not prevent that the elimination of interchangeable solutions in AFC-ng-cf, for each agent, consumes a lot of time. This is confirmed in figures 7.15 and 7.16.

Dense Problems

Figures 7.17 and 7.18 show the exchanged messages, the concurrent constraint checks, and the time consumed by AFC-ng-comp, AFC-ng-cf, AFC-ng-SS&SN and AFC-ng-SN algorithms in solving dense problems. The obtained results confirm the explanations we concluded in the previous section.

For the two values of p_1 10 and 50, we note that AFC-ng-SS&SN and AFC-ng-SN are always better in terms of exchanged messages, concurrent constraint checks, and resolution time. In this kind of problem, agents are linked by more constraints, and the generation of nogoods becomes surer. Besides, at least one agent local solution is inconsistent, and then the advantage of the domain Selective Sorting takes part.

The ABT-SS&SN is efficient when the problems are dense and solvent, because when problems are insolvent, the Selective Sorting will, in no way, minimize the disturbances, since the problems are going to be declared insolvent in a given moment. While, Smart Nogood allows detecting the failure/insolvency as soon as possible.

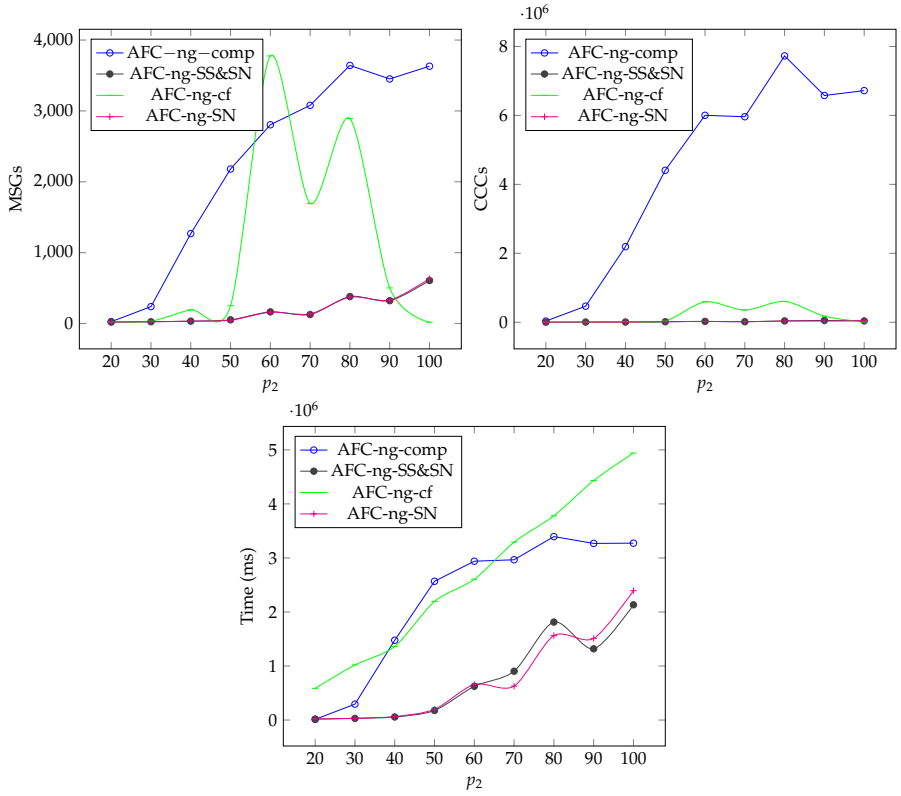


FIGURE 7.17: AFC-ng family benchmarking, for $p_1 = 10$, $p_4 = 70$ and $p_2 \in [20, 100]$

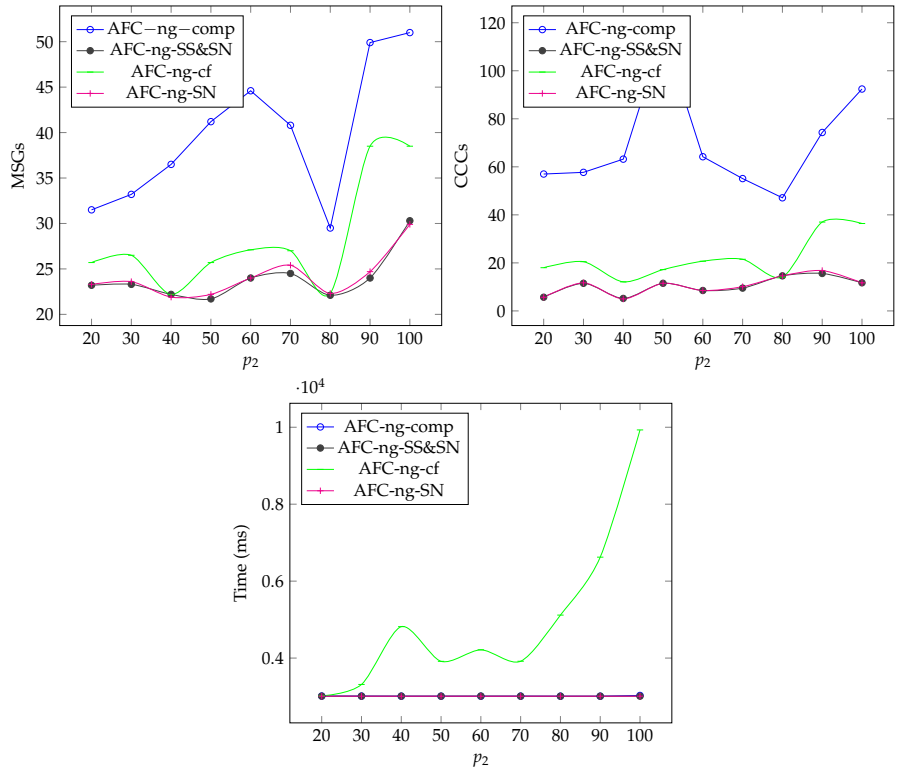


FIGURE 7.18: AFC-ng family benchmarking, for $p_1 = 50$, $p_4 = 70$ and $p_2 \in [20, 100]$

7.4 Conclusion

In this chapter, we integrated the Selective Sorting and Smart Nogood methods to ABT and AFC-ng algorithms. This produced the ABT-SS&SN and AFC-ng-SS&SN algorithms. Before evaluating the performance of each proposed algorithm, we integrated only the Smart Nogood method to both algorithms ABT and AFC-ng, in order to assess each method apart. The experimental results show the efficiency of our proposed algorithms, ABT-SS&SN, ABT-SN, AFC-ng-SS&SN, and AFC-ng-SN, especially when the problems are solvable and contain quite constraints.

The algorithms showed their strengths in terms of exchanged messages and concurrent constraint checks. In some cases, our algorithms behave like the best among other algorithms. When the problems are insolvent, the proposed algorithms consume more time. To overcome this, we plan to add in a future work a test sorting the domain. So as sorting will be done only if there is at least one consistent value in the domain.

Part III

Ethical Distributed Constraint Satisfaction Problems (Ethical DisCSP)

8

Presence of Unethical Agents in Distributed Constraint Satisfaction Problems

Contents

8.1 Introduction

The increasing use of multi-agent technologies has led to the emergence of ethics in multi-agent systems [Robbins and Wallace, 2007]. The agents in such systems have a very high level of autonomy. Hence the need to make ethical decisions and to regulate resolution in such systems.

Distributed Constraint Satisfaction Problem is a multi-agent technology. All DisCSP algorithms assume that agents involved in the resolution are all ethicals. While the autonomy of the agents allows them to make unethical decisions such as lying or non-considering a received message.

In this chapter, we are going to figure out the existing works in the ethics field, define the unethical DisCSP agent, discuss the gravity of its presence in a DisCSP resolution, and propose a new Ethical DisCSP formalism. Hence, the chapter is organized as follows. Section 8.2 presents the state of the art. Section 8.3 defines the unethical agent and exposes the gravity of its presence in a DisCSP resolution. Section 8.4 broaches the creation of the E-DisCSP formalism. Finally, section 8.7 concludes the chapter.

8.2 State of the Art

Ethics is an all-time discipline. It was guiding human behaviors and actually it is guiding also non-human ones. It has appeared in: (i) the embedded drones and decision support in military [McMahan, 2013], (ii) the systems managing private data in medicine [Cook, Dickens, and Fathalla, 2003], social networks [Light and McGrath, 2010], Big Data [Zwitter, 2014; Richards and King, 2014], and (iii) the autonomous systems, especially those sociable [McCarthy, 2009; Arkin, 2016]. It becomes important in several domains including Artificial Intelligence (AI). [Bostrom and Yudkowsky, 2014] is one of the most recent works on Ethical Artificial Intelligence. It treats the AI moral status philosophically and considers that the current AI systems do not have any moral status. For this, it perceives to produce an algorithm generating a super-ethical behavior.

"Machine Ethics" [Anderson, Anderson, and Armen, 2004; Moor, 2006; Anderson and Anderson, 2011] and Military Robot (or killer robots) [Sparrow, 2007] are two other applications of the ethics in AI. The main purpose of Machine Ethics is to create an Artificial Moral Agent (AMA) model of a moral machine following a set of ethical principles to make decisions. For Military Robot, the authors wondered whether "We Want Robot Warriors to Decide Who Lives or Dies?". While assuming that the robot term, which has appeared to produce low-cost goods, has ended with non-ethic robots that kill the human race because they have varying degrees of autonomy which will be complete autonomy in the future and fearing that robotic weapons may trigger a world war.

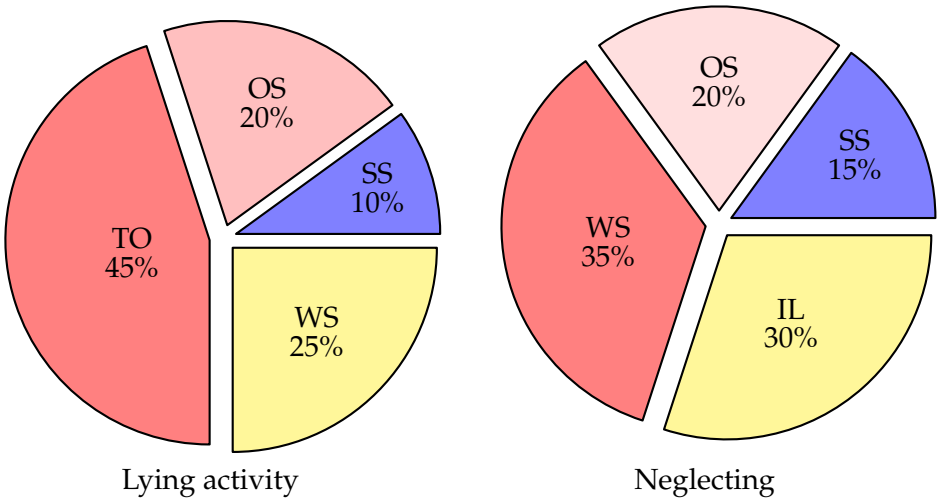
The Multi-Agent System is among the applications that have required attention in ethics. The most recent works dealing with ethical multi-agent systems proposed just the architectures and steps to follow in order to have future algorithms treating such systems. The ETHICAA team has defined, in [Belloni et al., 2015], a theoretical framework enabling the MAS to dynamically lead ethical collisions, taking into account the ethics of each independent individual agent as well as that of MAS. Cointe and al. have done the same thing as ETHICAA team in [Cointe, Bonnet, and Boissier, 2016], by proposing a framework to produce some ethics and to apply some actions. The latter is just a qualitative approach.

8.3 Unethical DisCSP Agent

8.3.1 Definition

The DisCSP algorithms are based on messages' exchanging, but there are no parameters or functions allowing to control such messages. Knowing that this formalism can represent human or robotic problems, and we have no guarantee that the participating agents are all ethical, it is the necessity to control the agents' behaviors. An unethical agent in a DisCSP problem resolution can make three abnormal activities:

- **lying**, by choosing a value and sending a false one;
- **neglecting**, by not considering a received nogood/backtrack message;
- **blocking the resolution**, by generating several false nogoods and sending them.



OS: Other Solution;
SS: Same Solution;
WS: Wrong Solution;
TO: Time Out;
IL: Infinite Loop.

FIGURE 8.1: Presence of one unethical agent in random problems

8.3.2 Gravity

To assess the seriousness of the presence of an unethical agent in a DisCSP resolution, we choose some solvable random problems and DisMSPs.

We take 20 random problems with the connectivity p_1 = 50, the tightness p_2= 45 and one unethical agent among 20. We take also 20 distributed MSPs having 20 agents (one unethical) each one, the number of availabilities per agent g = 9, and the number of meetings m_a = 2.

The results (Figure 8.1) show that in the presence of just one liar agent in random problems, the resolution can pass beyond the solution and find catastrophic results. 45% of resolution processes are stopped after a timeout, 25% of resolutions are given wrong solutions and just 30% (20% + 10%) have found true solutions.

In the presence of neglecting agents, the results are different. The number of wrong solutions becomes bigger (35%) and other abnormal results appear. 30% of problems loop infinitely.

The results of integrating one unethical agent into DisMSP problems are shown in figure 8.2. The results show that in the case the unethical agent is lying, 45% of found solutions are incorrect, 10% of the resolution processes are stopped after a time out, 25%

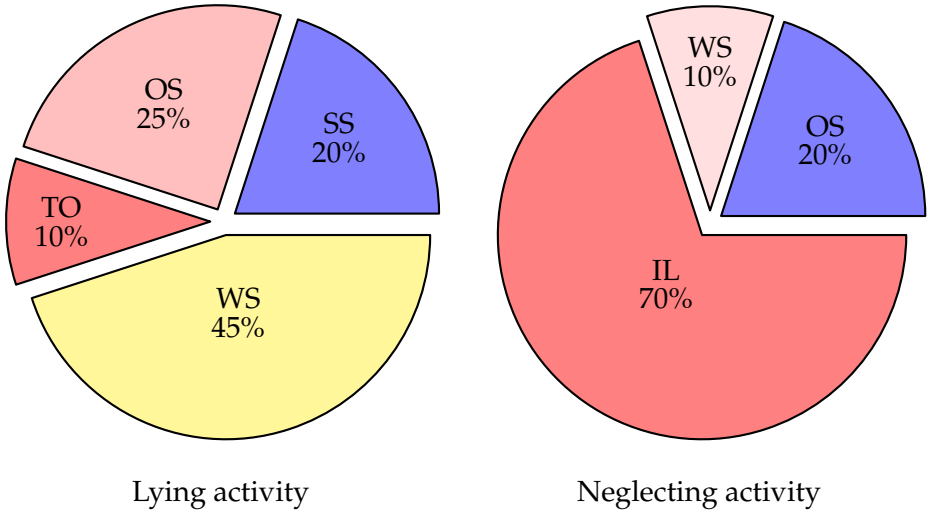


FIGURE 8.2: Presence of one unethical agent in distributed MSP problems

find true solutions but they are not the first ones, and only 20% find the first solution.

The presence of a neglecting agent is more grave in such problems. 70% of resolutions loop infinitely, 10% find wrong solutions, and only 20% find true solutions, but they are not the first ones. The first solution is missed.

8.4 Ethical Distributed Constraint Satisfaction Problem (E-DisCSP)

Based on the Machine Ethics, we have proposed a new extension of the DisCSP, called Ethical DisCSP (E-DisCSP), in order to:

- Detect abnormal activity in the presence of one or more unethical agent (s);
- Identify them;
- Take measures against their behaviors.

This new formalization is based on the same parameters as the DisCSP formalism, namely the agents, variables, domains, constraints and the function which associates each variable with an agent, while adding other components, namely the ethical rules and the actions to be applied after each violation of a rule.

More formally, the E-DisCSP is defined by the parameters (A, X, D, C, ϕ, R, A_c, f) such as:

- $A = \{A_1, A_2, \ldots, A_n\}$ is the set of n agents;

- $X = \{X_1, X_2, \ldots, X_m\}$ is the set of m variables;
- $D = \{D(X_1), D(X_2), \ldots, D(X_m)\}$ is the set of domains;
- $C = \{C_1, C_2, \ldots, C_k\}$ is the set of constraints;
- $\phi : X \rightarrow A$ is the function associating each variable to an agent;
- $R = \{R_1, R_2, \ldots, R_p\}$ is the set of rules;
- $A_c = \{A_{c1}, A_{c2}, \ldots, A_{cq}\}$the set of actions;
- $f\ R \rightarrow A_c$ is the function which associates each rule to an action. $f(R_i) = A_{cj}$ indicates the action A_{cj} is going to be applied when the rule R_i is violated.

8.5 Control System

To take the three newly added parameters, namely R, A_c, and f, into consideration and control the messages exchanged by the agents, we use a Control System (Figure 8.3). It is based on three principal units, namely the Assumption Unit, the Ethical Unit, and the TMS Unit.

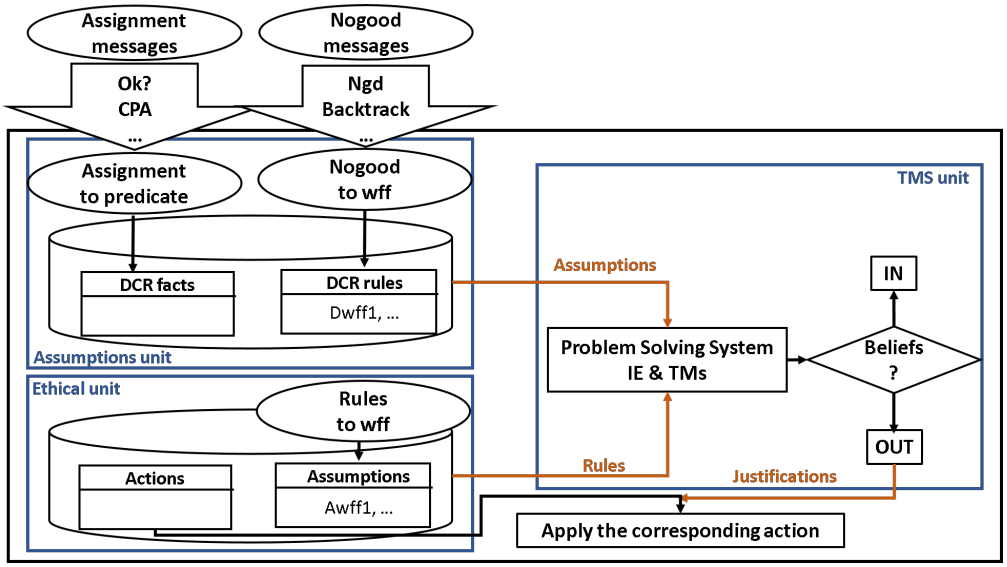


FIGURE 8.3: Ethics control framework

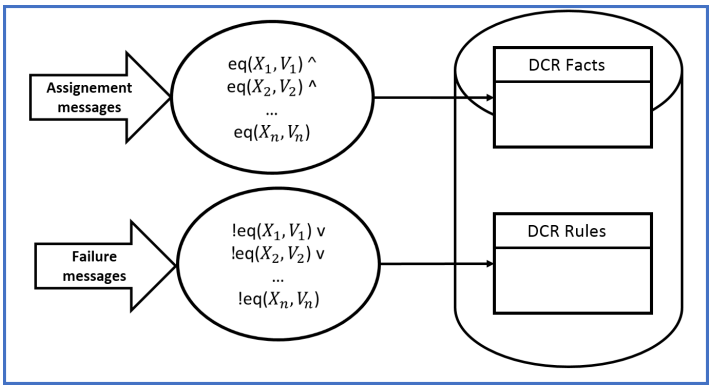


FIGURE 8.4: The Assumption Unit

8.5.1 The Assumption Unit

The Assumption unit (Figure 8.4) is used to convert the exchanged messages, namely Ok?, CPA, nogood, backtrack, etc, according to the used DisCSP protocol, into Well Formed Formula (wff) and predicates.

A Well Formed Formula wff is a finite sequence of symbols presenting a proposition, a structure or a logical form. It is used by logical languages. It uses capital letters and symbols to express in general the negation of an expression (õr !), the conjunction of a set of expressions ($\wedge$), their disjunction ($\vee$), and many others.

It has three major rules, namely:

- Each capital letter is a wff;
- Each wff prefixed by a is wff too;
- More than one wff can be gathered using an expression such as $\wedge$ and $\vee$. It results in a wff too.

The assumption unit groups messages into two categories. i) Assignment messages which represent messages handling the assignments of agents, as Ok? and CPA, and ii) Failure or Nogood messages that gather messages expressing that agents are finding a failure, as nogood exchanged by ABT or AFC-ng.

After conversion, assignment messages are stored as facts, while failure messages are stored as rules.

8.5.2 The Ethical Unit

In addition to the rules stored in the assumption unit, an expert can add some ethical rules.

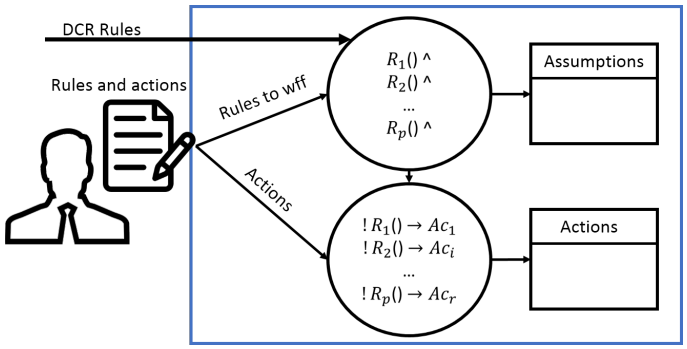


FIGURE 8.5: The Ethical Unit

The ethical unit permits (Figure 8.5) to translate those rules as wff, so as the machine can understand. The new recovered rules as well as the DCR rules are stored as assumptions.

This unit collects even the actions to be applied for each violated rule, from the expert. Wholesale, it allows to add the three parameters of the E-DisCSP, namely the rules, the actions, and the rules-actions link.

8.5.3 The TMS Unit

To help users of hypothetical reasoning systems, the conception of specific algorithms is required, in order to be able to efficiently manage hypotheses. The implementation of hypothetical reasoning is traditionally done using Truth Maintenance Systems (TMS). The latter is working in parallel with an Inference Engine.

Inference Engine

The Inference Engine [Negnevitsky, 2005] is the backbone of an expert system. It can bring updates to the knowledge base with these two components, namely the fact base and even the rule base. Each rule consists of its action and the conditions of the action

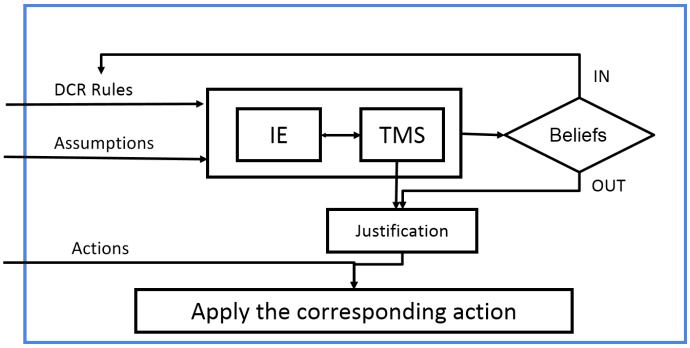


FIGURE 8.6: The TMS Unit

application (the facts). Rules are triggered by comparing their conditions with elements of the fact base.

The inference engine is characterized by its reasoning strategy, which is based on front or back chaining. It has three phases:

- The detection of active rules;
- The selection of the rule to apply;
- The trigger of the selected rule.

Truth Maintenance System

A Truth Maintenance System (TMS) [Doyle, 1979; Diri, 2009] is a program working in parallel with an Inference Engine or with any other agent to which it offers certain services by recording the reasonings, they have made. Among the offered services by this system: the detection of abnormal activities (detection of contradiction if they arise and to report them), providing explanations on request and perform revisions [Forbus and De Kleer, 1993].

A TMS determines also a set of beliefs from a set of reasons or justifications. This system allows updating this set of beliefs taking into account any new reason; which is added or retracted during a reasoning process. A belief that is put out of the set of beliefs is called OUT. It is called IN when it is introduced. According to [Diri, 2009], IN and OUT can be interpreted respectively as "it is true" and "it is false" that a fact exists on the basis of beliefs.

The initial beliefs that can be retracted are called "assumptions". So a TMS must determine the consequences of any change in the state of these assumptions as it should be able to trace the reasons that led to these changes. The same process is undertaken if a belief is to be reconsidered [Doyle, 1979]. The set of changes in the TMS base simulates the state in which the facts base of the inference engine should be if it is actually modified.

In general, a TMS is used to create a dependency graph (ie, a set of linked nodes), with a label corresponding to each node. labeling differs, depending on the type of the used TMS. Many types of truth maintenance systems exist as JTMS (Justification based TMS) and ATMS (Assumption based TMS).

In Justification-based TMSs (JTMS) [Martins, 1990], the language used is limited to Horn formulas [McCarthy, 1987], with just one context (i.e just one assumption).

In Assumption-based TMSs (ATMS), [Martins, 1990] the Language is limited to Horn formulas, but several alternatives (contexts) can be explored at the same time. The last point is the strength of ATMS compared to JTMS and meets our control system requirements, which should support multiple assumptions at the same time (multiple messages).

The TMS Unit receives the stored DCR rules (converted messages) from the Assumptions unit, tests coherence between them and against the assumptions (rules) stored in the Ethical unit, and recovers the actions corresponding to a rule if detected violated. Otherwise, it continue recovering the updates DCR rules from the Assumption unit. The consistency test is done using the TMS system.

8.6 Control Types

The control system can monitor one or more agents at the same time. Depending on the number of supervised agents, three types of control can be distinguished: centralized, distributed, and hybrid.

8.6.1 Centralized Control

In centralized control, all existing agents are supervised by one single control system (Figure 8.7). The latter collects the messages exchanged by all agents in order to be able to test the coherence of messages and their consistency regarding the stored rules.

The advantage of this control type is simplicity. But the fact of gathering the information of all agents in a single entity seems to be complex and insecure.

8.6.2 Distributed Control

Unlike centralized control, each agent is supervised by its own control system (Figure ref D control). In order to make a decision vis-à-vis a rule violation, all controllers must communicate.

This type of control responds to the security problem encountered in centralized control, but the communication between the different control systems can overload the network.

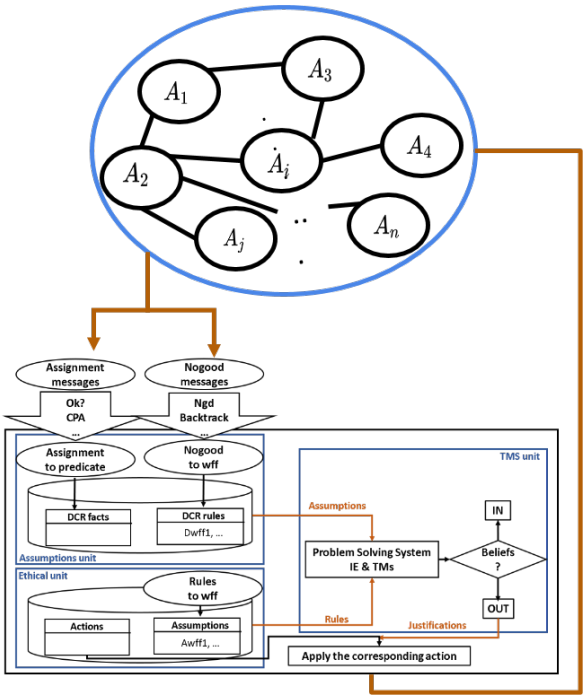


FIGURE 8.7: Centralized Control

8.6.3 Hybrid Control

To deal with the security and complexity issues challenged in centralized control, and those of communication overload confronting distributed control, the hybrid control proposal can be a solution.

In this type of control, agents are distributed in sub-sets. Each sub-set is supervised

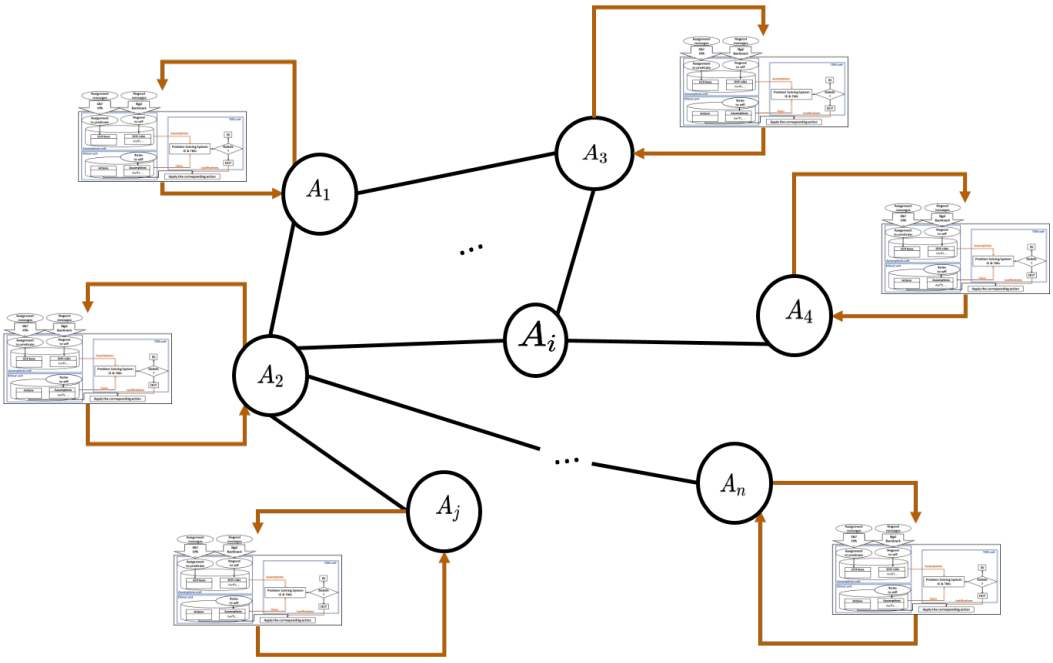


FIGURE 8.8: Distributed Control

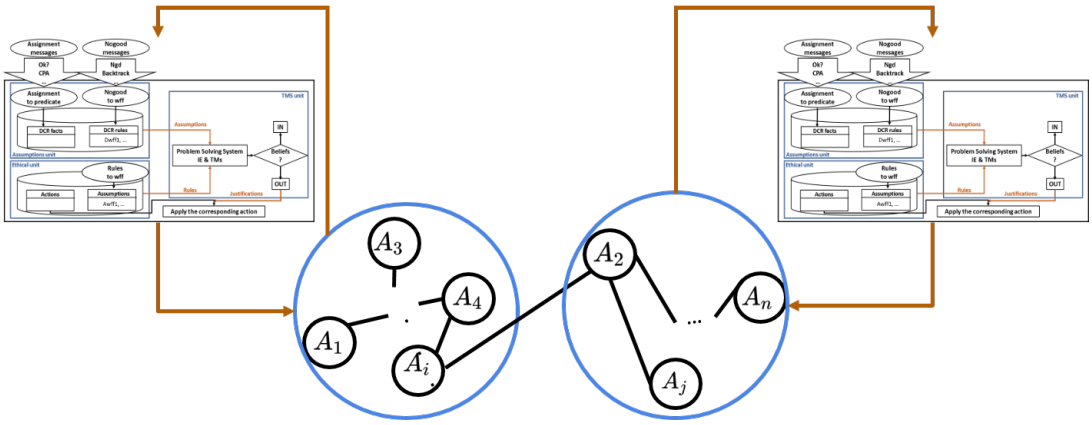


FIGURE 8.9: Hybrid Control

by a separate control system (Figure 8.9). As in distributed control, these control systems communicate in order to make a decision. This will not overload the networks as long as the number of control systems is reduced.

8.7 Conclusion

In this chapter, we introduced ethics in DisCSP. We discovered the gravity of the presence of unethical agents in a DisCSP resolution. It completely deviates from the normal resolution process. To this end, we proposed a new formalism called Ethical DisCSP which makes possible the integration of ethical rules as well as actions which are going to be applied if a rule is violated. This formalism is based on a control system allowing to monitor exchanged messages, compared to the added rules and to apply the appropriate actions.

Before supervising messages, the control system has to convert them to well formed formulas. In next chapter, we are going to see how messages are converted, and how the TMS unit uses these messages to detect unethical agents.

9

Detection of Unethical Agents in Ethical Distributed Constraint Satisfaction Problems

Contents

9.1 Introduction

The Ethical DisCSP formalism we proposed takes into consideration ethical rules and actions to apply if any of the rules are violated. We designed a control system considering, in addition to these ethical rules, the coherence of the exchanged messages.

To pursue the exchanged messages and check their consistency, a conversion to wff should be made so as a Truth Maintenance System in collaboration with an Inference Engine can detect the failure justification if exist.

In this chapter, we are going to propose a wff conversion of exchanged messages and study how much the latter proposal helps to detect abnormal activities and so unethical agents.

For this purpose, the chapter is organized as follow: Section 9.2 presents the wff conversion, giving an example. Then, section 9.3 shows the experimental results that prove the method efficiency. Finally, section 9.4 concludes the chapter.

9.2 Assumption and TMS Unit

The architecture of the Ethical DisCSP's control system contains three principal units, assumption unit, ethical unit, and TMS unit. In this chapter, we are going to discover how exchanged messages are converted to wff in the Assumption unit and how the results are used by TMS to detect unethical agents.

The messages' types used in the different existing algorithms can be summed up two:

- **Assignment message**: the message used to send the local solution of an agent;
- **Nogood message**: the message used to express a blockage.

9.2.1 Assignment Conversion

The conversion of the first message comes down to the conversion of the local solution of the agent. The $SolutionToWff$ function of Algorithm 13 shows the different steps to follow in order to get the wff formula of a local solution.

To well understand the function, let's take the agent A_1 of our DisMSP example (Figure 1.2) and convert its first local solution ($M_{1.1} = 1$, $M_{1.2} = 2$, $M_{1.5} = 3$), which is going to be sent to lower priority agents, using a message Ok?, CPA, or other, according to the used algorithm.

Algorithm 13 Assignment Conversion

```
1: function SOLUTIONTOWFF(Solution ChosenValue)
2:     wff ← empty ;
3:     for each variable of ChosenValue do
4:         wff ← wff + eq(variable.name, variable.value) + ∧;
5:     end for
6:     wff ← wff.substring(0,wff.length-1);
       return wff;
7: end function
```

The *SolutionToWff* function (Algorithm 13) creates an empty wff (Line 2) and browses the variables of the local solution (Line 3). For each variable, it recuperates the name of the variable and its value and conjuncts the predicate eq(*the variable name*, *the variable value*) to wff (Line 4).

In the above example, the function tips firstly on $M_{1.1}$ and adds "$eq(M_{1.1}, 1)\wedge$" to the created wff. Then, it goes on to $M_{1.2}$ to update the wff with "$eq(M_{1.1}, 1) \wedge eq(M_{1.2}, 2)\wedge$", and finally on $M_{1.3}$. In the last iteration, the wff becomes equal to "$(M_{1.1}, 1) \wedge eq(M_{1.2}, 2) \wedge eq(M_{1.5}, 3)\wedge$". Before returning the resolved wff, the function removes the last conjunction character $\wedge$ (Line 6).

9.2.2 Nogood Conversion

For the second type of messages, it is the nogood the most general message. The backtrack is a nogood's special case.

A nogood $(X_1 = v_1) \wedge (X_2 = v_2) \wedge ... \wedge (X_i = v_i) \rightarrow (X_j \neq v_j)$ is converted to a wff as follow:

Algorithm 14 Nogood Conversion

```
1: function NOGOODTOWFF(nogood ngd)
2:     wff ← empty;
3:     for each variable of ngd.getLHS() do
4:         wff ← wff + !eq(variable.name, variable.value) + ∨;
5:     end for
6:     for each variable of ngd.getRHS() do
7:         wff ← wff + !eq(variable.name, variable.value) + ∨;
8:     end for
9:     wff ← wff.substring(0,wff.length-1);
       return wff;
10: end function
```

$$eq(X_1, v_1) \wedge eq(X_2, v_2) \wedge ... \wedge eq(X_i, v_i) \rightarrow !(eq(X_j, v_j))$$
$$\Leftrightarrow$$
$$!(eq(X_1, v_1) \wedge eq(X_2, v_2) \wedge ... \wedge eq(X_i, v_i)) \vee !(eq(X_j, v_j))$$
$$\Leftrightarrow$$
$$!eq(X_1, v_1) \vee !eq(X_2, v_2) \vee ... \vee !eq(X_i, v_i) \vee !(eq(X_j, v_j))(*)$$

The taken formula is the last one $(*)$, that represent a horn formula.

The $NogoodTowff$ function shows the steps to convert such a message. It generates an empty wff and browses the left-hand side (Line 3) as the right-hand side (Line 6) of the nogood. Instead of conjuncting the pointed variables, the function disjuncts the $!eq(\textit{the variable name}, \textit{the variable value})$ (Lines 4 and 7) in wff.

9.2.3 Unethical Agent Detection

When the converted messages are recovered, the TMS unit of the control system uses them in order to test their coherence and returns an IN or OUT decision with the justifications of the inconsistency.

In order to well understand how the abnormal activity is detected using the converted messages, let take our DisMSP example (Figure 1.2), and assume that the agent A_2 is an unethical agent, and its abnormal activity is the non consideration of the nogood message $ngd = (M_{1.1} = 1, M_{1.2} = 3, M_{1.5} = 3) \rightarrow (M_{2.1} \neq 2, M_{2.2} \neq 1, M_{2.3} \neq 3, M_{2.4} \neq 4)$.

After receiving the message $ngd = (M_{1.1} = 1, M_{1.2} = 3, M_{1.5} = 3) \rightarrow (M_{2.1} \neq 2, M_{2.2} \neq 1, M_{2.3} \neq 3, M_{2.4} \neq 4)$, A_2 (assumed unethical) can keep its current solution $(M_{2.1} \neq 2, M_{2.2} \neq 1, M_{2.3} \neq 3, M_{2.4} \neq 4)$ and resend it to the lower priority agents (A_3), as if the nogood does not delete its value.

This will cause an infinite loop without finding the DisCSP solution. The loop is detected by following the messages, and observing that the exchanged messages are repeated many times.

With the control system, and the proposed message conversion, the message ngd sent by A_3 $ngd = (M_{1.1} = 1, M_{1.2} = 3, M_{1.5} = 3) \rightarrow (M_{2.1} \neq 2, M_{2.2} \neq 1, M_{2.3} \neq 3, M_{2.4} \neq 4)$ is going to be converted to

$$\begin{aligned} !eq(M.1.1, 1) \vee !eq(M.1.2, 2) \vee !eq(M.1.5, 3) \vee !eq(M.2.1, 5) \vee \\ !eq(M.2.2, 1) \vee !eq(M.2.3, 3) \vee !eq(M.2.4, 4) \end{aligned} \tag{9.1}$$

and the message Ok?/CPA sent by the agent A_2, after reception of the message ngd, is converted to

$$eq(M.2.1, 5) \wedge eq(M.2.2, 1) \wedge eq(M.2.3, 3) \wedge eq(M.2.4, 4) \tag{9.2}$$

$$\begin{aligned}(!eq(M.2.1, 5) \vee !eq(M.2.2, 1) \vee !eq(M.2.3, 3) \vee !eq(M.2.4, 4)) \wedge \\ eq(M.2.1, 5) \wedge eq(M.2.2, 1) \wedge eq(M.2.3, 3) \wedge eq(M.2.4, 4)\end{aligned} \tag{9.3}$$

The two generated predicates are going to be the inputs of our control system. The TMS unit can detect the contradiction, easily, and even the responsible of this abnormal activity, which is A_2.

9.3 Experimental Results

In order to evaluate the performance of the unethical agents' detection method, we carried out experiments on random problems as well as on Distributed MSP problems.

Random problems are characterized by the parameters <n, v, d, p_1, p_2, u>, where n is the number of agents, v is the number of variables, d is the size of the domains, p_1 is the network connectivity, p_2 is the constraint tightness, and u is the number of unethical agents to be introduced.

The random problems we take have 20 agents ($n = 20$), 20 variables ($v = 20$, each agent handles a variable), 10 values in the domain of each variable ($d = 10$), three values of p_1 (20,50 and 80) to take up problems with different complexities (ie. less complex, medium complexity, more complex), p_2 varies between 10 and 90 with 5 as a step and 5 values of the number of unethical agents (1, 2, 3, 4, and 5 unethical agents).

The Distributed MSP problems are defined by <m, n, m_a, g, u> such as m is the number of Meetings, m is the number of agents, m_a is the number of meetings per agent and g is the size of the grid (ie. the number of availabilities of each agent).

The DisMSP problems we evaluate have two values of the number of meetings m (9 and 10 meetings), 20 agents ($n = 20$), four values of the number of meetings per agent m_a (2 and 3 meetings per agent), the size of the grid g varies from 3 to 21 by 2 as a step, and the same values u of the number of unethical agents as random problems.

The unethical agents' detection method is integrated into ABT algorithm.

9.3.1 Random Problems

Tables 9.1 and 9.2 show the detection success rate of unethical agents, when introducing one, two, three, four or five unethical agent(s) **neglecting** the nogood messages (Table 9.1) and **lying** (Table 9.2) in random problems.

The results show that the detection achieves, in most problems, more than 50%, especially when the unethical agents are lying. This is normal given that in all DisCSP algorithms, the safest messages to circulate are the messages carrying the local solutions of agents (Ok? in the ABT algorithm). By contrast, the failure message (nogood in the ABT algorithm) is sent by an agent just after detecting a failure. While the unethical agents which neglect the nogood message have no way to apply their abnormal activity if their lowest successors do not send any nogood message.

For sparse problems, it is noted that in the first values of p_2, the detection of neglecting agents is 0% (Table 9.1). In this kind of problem, the agents are connected with few constraints and we know that the less the agents have constraints to satisfy, the more they are worthy. And the more they must satisfy constraints the more they are blocked and they send nogood messages. So the unethical agent will be more likely to apply its neglecting activity. This is confirmed by the results shown in Table 9.2. Even in sparse problems, the detection rate is always upper than 0%, because in this kind of problem the unethical agents do not neglect the nogood messages, but they lie by sending false values. While this kind of messages are always exchanged and the probability that an unethical agent sends such messages becomes greater. We also note that even with more unethical agents, we can detect the majority of them with a very high rate too.

All results (ie. u = 1, 2, 3, 4, and 5 for the two abnormal activities) highlight the efficiency of our unethical agents' detection method. The percentage is increased when p_2 is raised for the majority of values.

The detection success rate decreases when the number of unethical agents increases. This rate represents the number of agents detected compared to the number of introduced unethical agents. These percentages prove that the detection of abnormal activity is certain, except that we do not detect all the unethical agents. This is due to the fact that once just one unethical agent is detected the resolution is stopped since we know, beforehand, that the presence of such an agent causes infinite loops. And this is a great strength because the rates represented in the tables show the concurrent detection rate of unethical agents.

TABLE 9.1: Detection rate in random problems, when unethical agents are neglecting ngd messages

	$u = 1$			$u = 2$			$u = 3$			$u = 4$			$u = 5$		
p_1 / p_2	20	50	80	20	50	80	20	50	80	20	50	80	20	50	80
10	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
15	0%	0%	10%	0%	0%	5%	0%	0%	3.3%	0%	0%	7.5%	0%	0%	10%
20	0%	10%	50%	0%	10%	25%	0%	10%	33.3%	0%	5%	20%	0%	4%	28%
25	0%	40%	60%	0%	35%	50%	0%	20%	33.3%	2.5%	12.5%	32.5%	2%	30%	34%
30	0%	80%	90%	15%	40%	65%	3.3%	33.3%	50%	2.5%	25%	47.5%	2%	22%	42%
35	3%	80%	**100%**	0%	55%	65%	6.7%	36.7%	53.3%	5%	40%	62.5%	4%	44%	58%
40	0%	**100%**	**100%**	0%	70%	75%	6.7%	53.3%	70%	7.5%	35%	75%	6%	44%	58%
45	10%	**100%**	**100%**	10%	55%	75%	10%	63.3%	80%	12.5%	55%	85%	14%	56%	72%
50	20%	**100%**	**100%**	20%	70%	90%	16.7%	63.3%	80%	25%	75%	80%	20%	68%	68%
55	50%	**100%**	**100%**	55%	80%	95%	36.7%	86.7%	76.7%	37.5%	75%	80%	32%	62%	74%
60	90%	**100%**	**100%**	40%	90%	95%	46.7%	80%	73.3%	32.5%	80%	75%	36%	78%	78%
65	90%	**100%**	**100%**	65%	85%	85%	50%	80%	**93.3%**	47.5%	67.5%	82.5%	44%	78%	80%
70	**100%**	**100%**	**100%**	70%	85%	95%	66.7%	80%	73.3%	55%	72.5%	77.5%	60%	**88%**	78%
75	**100%**	**100%**	**100%**	70%	95%	**100%**	70%	80%	83.3%	57.5%	80%	77.5%	56%	86%	82%
80	**100%**	**100%**	**100%**	85%	**100%**	**100%**	83.3%	90%	**93.3%**	65%	85%	80%	80%	**88%**	**84%**
85	**100%**	**100%**	**100%**	70%	**100%**	95%	80%	**93.3%**	**93.3%**	80%	92.5%	**90%**	76%	82%	82%
90	**100%**	**100%**	**100%**	**95%**	95%	95%	**86.7%**	86.7%	63.3%	**92.5%**	**90**%	63.3%	84%	86.7%	80%
Averages	45%	77%	83%	35%	63%	71%	26%	49%	56%	29%	50%	65%	30%	52%	59%
		68%			56%			44%			48%			47%	
								53%							

TABLE 9.2: Detection rate in random problems, when unethical agents are lying

	$u = 1$			$u = 2$			$u = 3$			$u = 4$			$u = 5$		
p_1 / p_2	20	50	80	20	50	80	20	50	80	20	50	80	20	50	80
10	60%	70%	100%	70%	87.5%	92.5%	78.33%	90%	93.33%	75%	88.75%	96.25%	86%	91%	98%
15	60%	100%	95%	70%	90%	95%	85%	88.33%	95%	72.5%	91.25%	96.25%	78%	88%	94%
20	60%	100%	100%	67.5%	87.5%	95%	78.33%	86.67%	93.33%	87%	92.5%	96.25%	72%	89%	94%
25	70%	100%	100%	67.5%	97.5%	97.5%	68.33%	95%	91.67%	81.25%	92.5%	96.25%	76%	91%	96%
30	80%	90%	100%	67.5%	90%	97.5%	68.33%	91.67%	95%	76.25%	96.25%	93.75%	68%	94%	93%
35	80%	90%	90%	62.5%	85%	100%	68.33%	83.33%	91.67%	78.75%	83.75%	91.25%	76%	77%	95%
40	90%	90%	85%	85%	90%	95%	75%	83.33%	91.67%	82.5%	87%	96.25%	83%	87%	95%
45	80%	100%	95%	75%	90%	97.5%	75%	96.67%	98.33%	73.75%	96.25%	95%	77%	90%	90%
50	80%	90%	100%	85%	92.5%	80%	83.33%	95%	93.33%	78.75%	90%	93.75%	72%	87%	96%
55	80%	90%	100%	77.5%	87.9%	100%	80%	90%	90%	73.75%	87.5%	98.75%	78%	83%	98%
60	90%	100%	100%	75%	87.9%	92.5%	81.67%	90%	95%	81.25%	92.5%	95%	80%	91%	92%
65	60%	85%	100%	90%	97.5%	92.5%	85%	86.67%	93.33%	78.75%	95%	88.75%	82%	90%	95%
70	80%	95%	100%	82.5%	85%	97.5%	70%	85%	98.33%	66.25%	90%	87%	77%	90%	90%
75	90%	90%	90%	90%	100%	95%	83.33%	93.33%	90%	80%	91.25%	90%	80%	93%	89%
80	90%	90%	90%	82.5%	85%	92.5%	75%	81.67%	88.33%	87%	77.5%	98.75%	73%	79%	93%
85	80%	80%	90%	82.5%	87.5%	90%	76.67%	96.67%	91.67%	82.5%	92.5%	97.5%	77%	92%	92%
90	70%	90%	90%	75.5%	90%	92.5%	78.33%	95%	100%	73.75%	90%	100%	78%	88%	95%
Averages	76%	91%	96%	80%	89%	94%	78%	91%	94%	82%	90%	93%	77%	88%	94%
		88%			88%			88%			89%			86%	
								88%							

Furthermore, the number of 0% in the detection success rate decreases when the number of unethical agents increases, which enhances what we said (the detection becomes more certain). Following this reasoning, we can say that if we introduce more unethical agents the detection will be safer.

9.3.2 Distributed Meeting Scheduling Problems

As we did with random problems, we evaluate the unethical agents' detection method using also real problems. To this end, we experiment in the DisMSP problems we defined. The results of the assessments are illustrated in Tables 9.3, 9.4, 9.5 and 9.6. They display the detection rate of unethical agents when the problem contains one, two, three, four, and five unethical agents.

Table 9.3 shows the results when the unethical agents are lying and the number of meetings is equal to 9. Whereas Table 9.4 displays the results of problems with 10 meetings.

The results demonstrate that the rate is always important in this kind of problem and the values are generally stable. All results are close to the average which is a very important rate.

For the neglecting activity, the results are clarified in Tables 9.5 (for 9 meetings in the problem) and 9.6 (for 10 meetings in the problem). Unlike the first results, they are not very large but the detection is always sure. These values are completely normal. The unethical agents are not detected, not because of the method, but because they did not have the opportunity to apply their abnormal activities. How?

In DisMSP, the local problems are not simple as random problems, they are complex. Each agent handles g meetings (ie. 2 or 3 meeting per agent). In this kind of problems, the agent does not participate with its domain, but with its compiled domain containing the set of its local solutions. So, the more it has variables, the more the number of local solutions becomes important. And the more the number of choices is grand, the less the agents are blocked, the less they send nogood messages and the less the unethical agents have the chance to apply their abnormal activities. This is confirmed by the difference between the values of results when m_a changes from 2 to 3.

As we have already raised, if there are few numbers of constraints, the unethical agent does not any neglecting activity. It is confirmed another time with the difference between the two tables 9.5 and 9.6. Having 10 meetings to be distributed among the

TABLE 9.3: Detection rate of liar agents in Distributed MSP problems with 9 meetings

	$u = 1$		$u = 2$		$u = 3$		$u = 4$		$u = 5$	
g	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$
3	100%	85%	95%	95%	90%	95%	88.75%	95%	97%	93%
5	80%	90%	87.5%	92.5%	93.33%	95%	92.5%	95%	92%	95%
7	85%	90%	87.5%	85%	93.33%	93.33%	85%	95%	91%	94%
9	90%	100%	95%	90%	91.67%	90%	88.75%	92.5%	91%	96%
11	80%	90%	90%	95%	86.67%	93.33%	88.75%	92.5%	87%	94%
13	90%	90%	82.5%	90%	91.67%	81.67%	85%	83.75%	93%	86%
15	100%	90%	82.5%	85%	91.67%	75%	93.75%	78.75%	89%	76%
17	100%	70%	92.5%	75%	90%	76.67%	90%	68.75%	90%	76%
19	100%	75%	95%	72.5%	93.33%	81.67%	92.5%	83.75%	87%	75%
21	95%	35%	82.5%	70%	93.33%	61.67%	91.25%	68.75%	92%	64%
Averages	92%	82%	94%	86%	90%	89%	87%	95%	91%	85%
	87%		90%		89%		91%		88%	
	89%									

TABLE 9.4: Detection rate of liar agents in Distributed MSP problems with 10 meetings

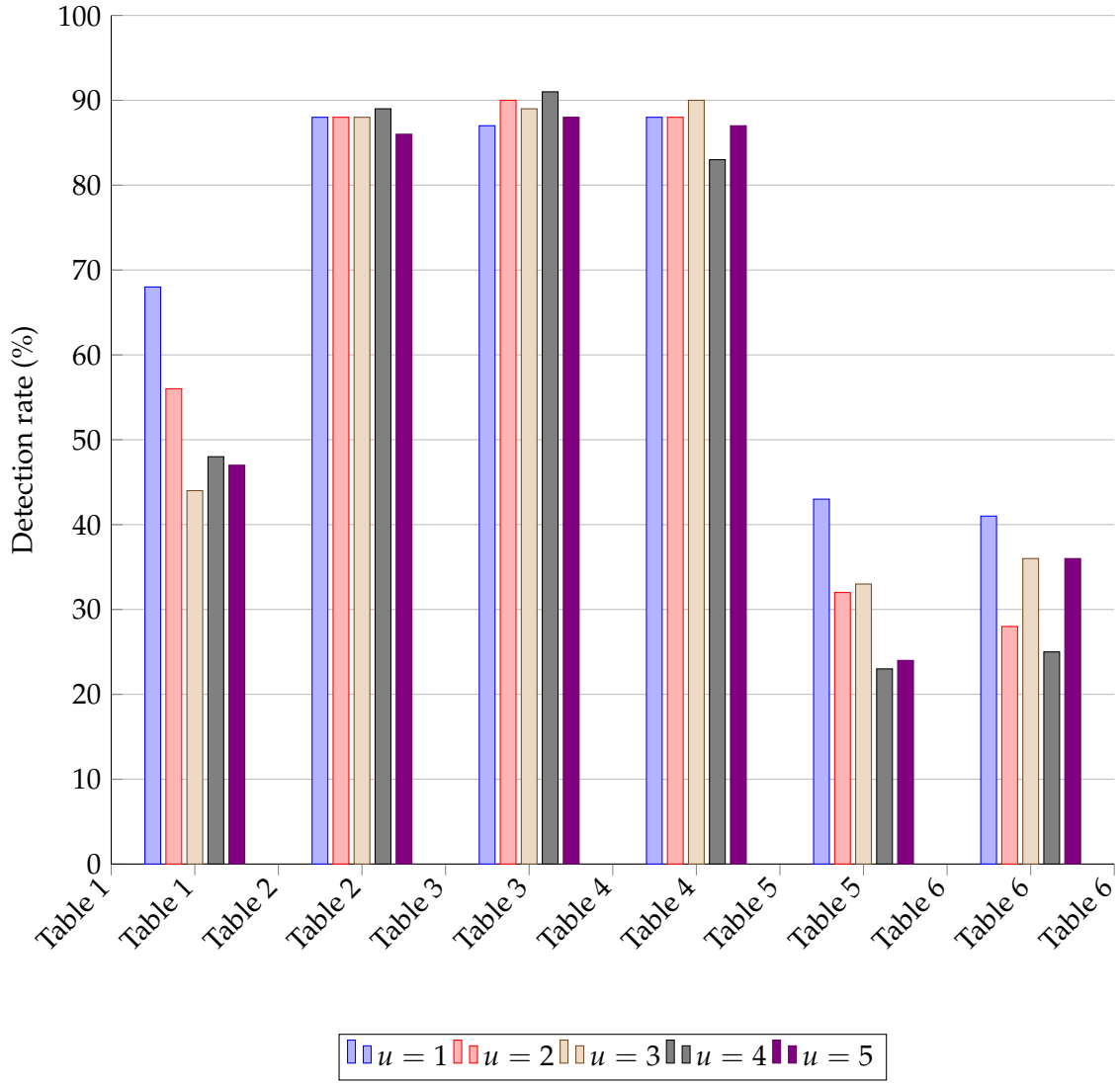
	$u = 1$		$u = 2$		$u = 3$		$u = 4$		$u = 5$	
g	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$	$m_a = 2$	$m_a = 3$
3	70%	80%	90%	92.5%	85%	93.33%	86.25%	92.5%	88%	92%
5	85%	100%	85%	87.5%	86.67%	96.67%	88.75%	95%	86%	97%
7	100%	95%	90%	92.5%	83.33%	95%	90%	96.25%	86%	93%
9	95%	100%	90%	82.5%	90%	91.67%	85%	91.25%	89%	94%
11	95%	85%	87.5%	92.5%	91.67%	86.67%	83.75%	92.5%	91%	92%
13	85%	80%	90%	95%	88.33%	85%	87.5%	83.75%	88%	94%
15	95%	95%	97.5%	67.5%	95%	76.67%	86.25%	77.5%	88%	81%
17	90%	80%	85%	82.5%	91.67%	88.33%	88.75%	85%	87%	84%
19	95%	85%	87.5%	62.5%	85%	71.67%	86.25%	75%	87%	69%
21	65%	80%	85%	80%	90%	71.67%	80%	70%	86%	76%
Averages	88%	88%	88%	88%	89%	90%	85%	81%	88%	87%
	88%		88%		90%		83%		87%	
	87%									

TABLE 9.5: Detection rate of agents neglecting ngd messages in Distributed MSP problems with 9 meetings

	$u=1$		$u=2$		$u=3$		$u=4$		$u=5$	
g	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$
3	55%	20%	30%	17.5%	35%	20%	41.25%	17.5%	35%	15%
5	50%	25%	40%	20%	26.67%	28.33%	35%	27.5%	28%	22%
7	55%	45%	40%	27.5%	38.33%	21.67%	30%	21.25%	30%	16%
9	75%	45%	47.5%	32.5%	45%	30%	26.25%	22.5%	26%	27%
11	45%	75%	37.5%	45%	28.33%	40%	28.75%	27.5%	27%	23%
13	55%	50%	52.5%	20%	41.67%	23.33%	36.25%	26.25%	32%	21%
15	60%	10%	47.5%	17.5%	36.67%	11.67%	36.25%	10%	28%	19%
17	55%	25%	50%	27.5%	41.67%	16.67%	40%	26.25%	32%	19%
19	50%	15%	47.5%	15%	46.67%	11.67%	36.25%	10%	27%	9%
21	45%	5%	32.5%	20%	38.33%	15%	28.75%	7.5%	30%	10%
Averages	55%	32%	40%	24%	40%	26%	35%	10%	30%	18%
	43%		32%		33%		23%		24%	
	31%									

TABLE 9.6: Detection rate of agents neglecting ngd messages in Distributed MSP problems with 10 meetings

	$u=1$		$u=2$		$u=3$		$u=4$		$u=5$	
g	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$	$m_a=2$	$m_a=3$
3	55%	20%	35%	12.5%	36.67%	18.33%	38.75%	11.25%	34%	17%
5	60%	15%	27.5%	27.5%	46.67%	18.33%	37.5%	18.75%	26%	16%
7	60%	15%	30%	30%	41.67%	18.33%	27.5%	17.5%	27%	19%
9	60%	10%	35%	32.5%	35%	36.67%	33.75%	20%	31%	30%
11	60%	10%	27.5%	25%	43.33%	33.33%	31.25%	22.5%	30%	25%
13	65%	25%	32.5%	37.5%	33.33%	35%	35%	36.25%	33%	27%
15	75%	30%	40%	20%	38.33%	11.67%	28.75%	15%	27%	17%
17	60%	10%	40%	20%	40%	16.67%	37.5%	18.75%	32%	17%
19	75%	10%	40%	12.5%	31.67%	23.33%	30%	20%	32%	19%
21	70%	30%	37.5%	5%	38.33%	18.33%	37.5%	18.75%	35%	22%
Averages	64%	18%	37%	20%	38%	35%	33%	18%	31%	21%
	41%		28%		36%		25%		26%	
	31%									



* **Table 1**: Neglecting ngd messages in Random problems;
* **Table 2**: Lying in Random problems;
* **Table 3**: Lying in the DisMSP problems having 9 meetings;
* **Table 4**: Lying in the DisMSP problems having 10 meetings;
* **Table 5**: Neglecting ngd messages in the DisMSP problems having 9 meetings;
* **Table 6**: Neglecting ngd messages in the DisMSP problems having 10 meetings;

FIGURE 9.1: The detection rate of unethical agents in Random and DisMSPs' problems

existing agents is not similar to having just 9. The second case will generate more equal constraints compared to the first case.

9.3.3 Summary

The averages displayed in all tables (9.1, 9.2, 9.3, 9.4, 9.5 and 9.6) show that, for each value of u, the more problems are denser the more the detection rate is important (averages' first line). They demonstrate also that the detection rates, for all number of unethical agents (line 2 of averages) are always close to the general average (line 3).

These all about meaning that the detection is sure regardless of the tightness of the problem, its density, its complexity, and even the number of introduced unethical agents.

The last affirmation is proved by figure 9.1, which attests that for each type of problem and for each abnormal activity, the detection rate is more or less stable regardless of the number of unethical agents.

9.4 Conclusion

In this chapter, we tackled the conversion of messages exchanged into wff made by the assumption unit and the use of the resulted conversions by the TMS unit to detect abnormal activities as well as the responsible agent.

Experimental results showed the effectiveness of the method used in both random and DisMSP problems, regardless of the complexity of the problem and the number of unethical agents.

In the next chapter, we are going to see what actions can be taken against these unethical agents and what will be the effects.

10

Maintaining Ethical Resolution in Distributed Constraint Reasoning

Contents

10.1 Introduction

The presence of just one unethical agent in a DisCSP resolution, as we saw previously, causes a lot of issues such as looping infinity, missing a solution, or finding a solution completely wrong.

The proposal of the E-DisCSP formalism as well as the control system, which takes the ethics parameters into account, appear to be useful in such a case. The detection methods we proposed could detect most of these agents. But after the detection, the resolution is completely arrested. While the main goal of the E-DisCSP formalism is to keep the DisCSP resolution as normal as possible.

In this chapter, we are going to study the different actions we can apply to the detected unethical agents and see what will be the influence on the resolution.

For this purpose, the chapter is going to be organized as follows. Section 10.2 presents the possible actions we can take, which are detailed in subsections 10.2.1 and 10.2.2. Section 10.3 exposes the experimental results. And section 10.4 includes the chapter after all.

10.2 Ethical Actions

The detection of unethical agents is not enough, the control system must react to minimize the damage caused by those agents. In this chapter, we propose three actions to solve this issue, namely:

1. Change the priority of the unethical agent by making it as the lowest priority agent. Since the lowest priority agent in most of the DisCSP algorithms does not send assignment messages, it just receives them;

2. Get rid of the detected unethical agent, by removing it from the DisCSP problem;

3. Stop the resolution.

If the control system detects an abnormal activity, it sends a warning message to the control entity, which can be one of the participating agents, a master agent or a human being gathering the justifications obtained from the TMS unit as well as the actions to apply.

Algorithm 15 Ethical actions

```
1: procedure CONTROLLERAGENT
2:     while ¬end do
3:         msg ← getMsg();
4:         Switch (msg.type)
5:          PriorityChange  : ChangePriority(msg);
6:          AgentElimination  : Eliminate(msg);
7:          stp  : end ← true;
8:     end while
9: end procedure
```

When the control entity receives such a message, extracts the name of the unethical agent from the received warning message and broadcasts a PriorityChange, an AgentElimination, or a stop message (according to the chosen action) to all agents, carrying the extracted agent name.

Each participating agent is always in listening mode. In addition to the protocol messages, it is waiting for three other messages PriorityChange, AgentElimination and stop messages. The algorithm 15 shows the executed instructions when receiving such messages.

The agent tests the received messages' type (Line 4). If the message type is ChangePriority, the agent applies the *ChangePriority* procedure (Line 5). If it is an EliminateAgent message, it calls the *Eliminate* procedure (Line 6). While if the received message is a stop one, the agent stops the listening mode (Line 7).

10.2.1 Change Unethical Agents' Priority

In the *ChangePriority* procedure (Algorithm 16), the agent checks if its name is not equal to the agent name received in the ChangePriority message (Line 8). If so, it looks for the agent in its antecedents (higher priority agents than self Γ^-) to remove it (Lines 9 and 10) and puts it in its set of successors (lowest priority agents than self Γ^+) (Line 11).

Otherwise, it moves all the agents existing in its successors (Γ^+) to its antecedents (Γ^-) (Lines 12, 13 and 14), so that all agents become higher priority than self.

Assume that the agent A_2 of our DisMSP example shown in Figure 1.2 is detected unethical. Figure 10.1 shows the updated problem after applying the *PriorityChange* procedure. The oriented arrows indicate the priority of agents.

Algorithm 16 Change Priority

```
1:  procedure CHANGEPRIORITY(agent)
2:      if self = agent then
3:          for each parent ∈ Γ+(self) do
4:              Γ+(self).remove(parent);
5:              Γ−(self).add(parent);
6:          end for
7:      else
8:          for each parent ∈ Γ−(self) do
9:              if parent = agent then
10:                 Γ−(self).remove(parent);
11:                 Γ+(self).add(parent);
12:                 break;
13:             end if
14:         end for
15:     end if
16:     Restart the resolution;
17:     CheckAgentView;
18: end procedure
```

10.2.2 Eliminate Unethical Agents

In the *Eliminate* procedure (Algorithm 17), the $self$ agent checks if its name is not equal to the unethical agent name received in the Eliminate message (Line 2). If so, it browses the set of its successors Γ^+ (Line 3) as well as its predecessors Γ^- (Line 9), to look for the unethical agent and remove it from the two sets (Lines 5 and 11).

The deleting process deletes even the constraints linking $self$ agent with the unethical

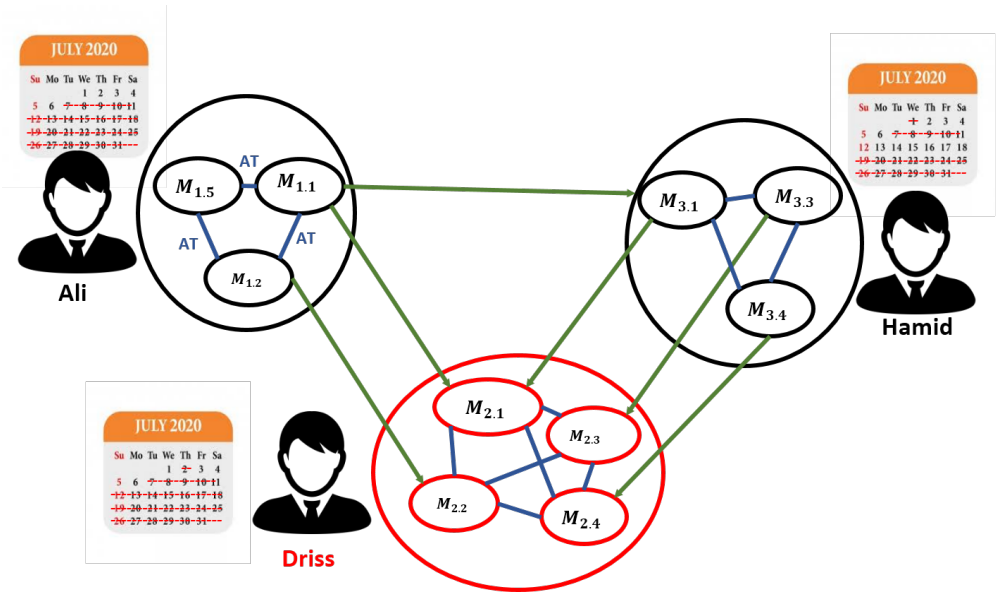


FIGURE 10.1: Changing the priority of the agent A_2

agent. If $self$ finds that the unethical agent exists in the set of its successors, it does not check the list of predecessors, since the unethical agent can be either lower or higher priority than $self$.

After eliminating unethical agents from resolution, two scenarios can be produced. The first scenario is produced when the other agents are connected with each other, in addition to the connection with the unethical agent. In that case, a new problem is regenerated with the same number of agents minus one.

Figure 10.2 illustrates the resulting problem after applying the elimination action on the agent A_2. It generates a new problem with just 2 agents. The master agent has to wait for the silence of the two agents to declare a solution.

The second scenario is provoked when the unethical agent is a bridge between two or more subsets of agents. In this case, two or more subproblems are going to be created.

Undertake that all agents (A_1 and A_3 in the taken example) were connected with just the unethical agent (A_2 in the example) and there are no other connections between them. The elimination action will produce two independent subproblems (Figure 10.3). This will not cause any problem, because the master agent has to wait for the silence of all agents and will not declare any solution until it detects the silence of all participant agents.

After changing the priority of the unethical agent or eliminating it. Agents have to

Algorithm 17 Eliminate

```
1:  procedure ELIMINATE(agent)
2:      if self ≠ agent then
3:          for each child ∈ Γ+(self) do
4:              if child = agent then
5:                  Γ+(self).remove(child);
6:                  Return;
7:              end if
8:          end for
9:          for each parent ∈ Γ−(self) do
10:             if parent = agent then
11:                 Γ−(self).remove(parent);
12:                 Return;
13:             end if
14:         end for
15:     end if
16:     Restart the resolution;
17:     CheckAgentView;
18: end procedure
```

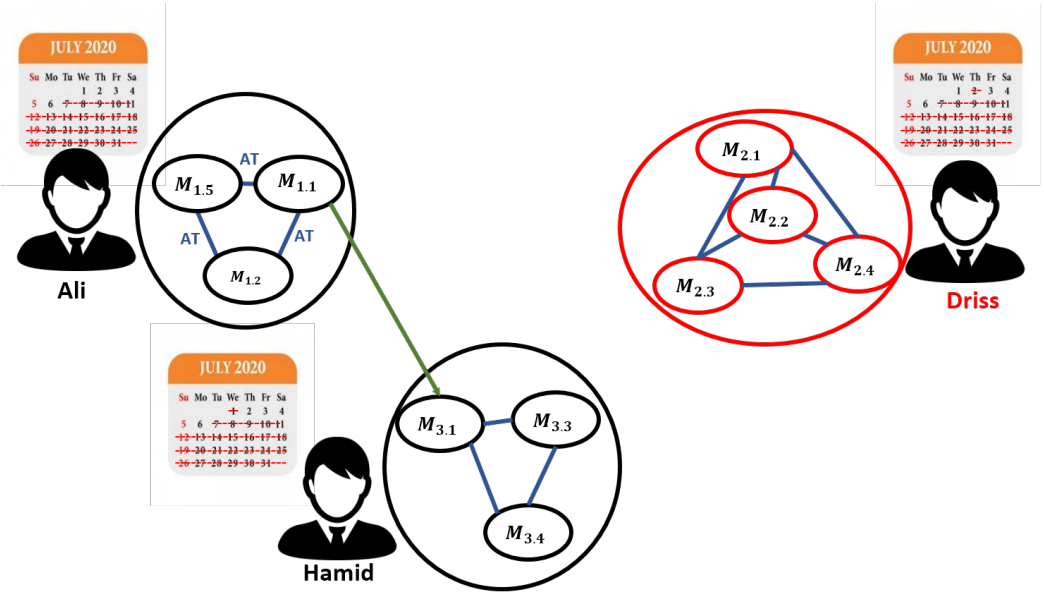


FIGURE 10.2: Case 1: Consequences of eliminating the unethical agent

restart the resolution and look for a new solution again. The resolution restart is made by clearing the AgentViews, the NogoodStores and the local solution of each agent.

10.3 Experimental Results

As any new method or algorithm, the proposed actions must be evaluated, by comparing it with the state of art. Since there is no similar method to compare with, we are going to

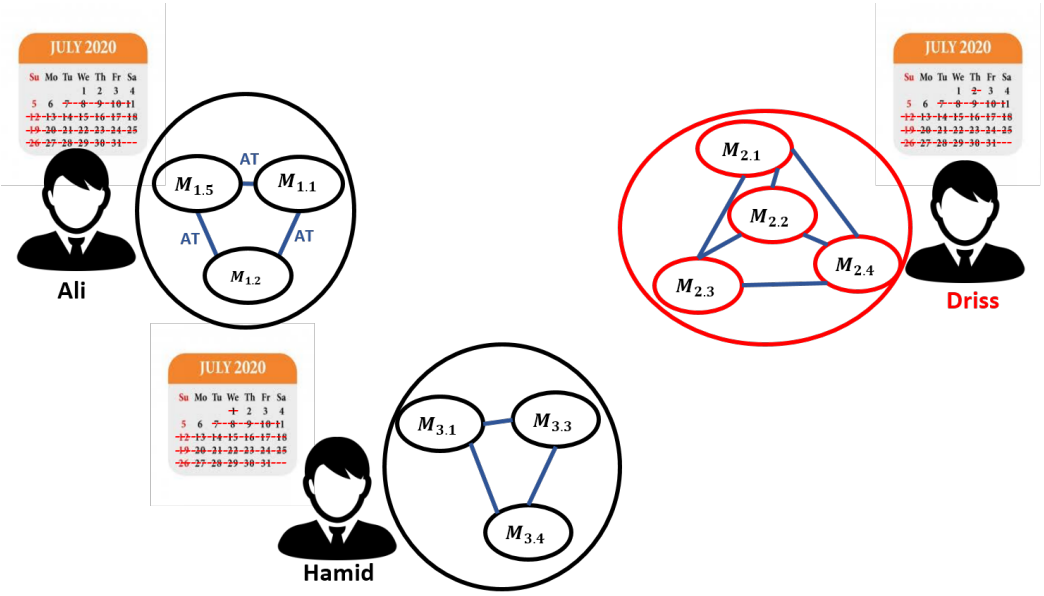


FIGURE 10.3: Case 2: Consequences of eliminating the unethical agent

compare the results after applying the two proposed actions with those without applying any action.

For this, we are going to choose one random problem kind and another DisMSP. The choice is done according to the results found in Chapter 9. We have to choose the problems which are solvable and where the detection of the unethical agents is surer. We chose 20 random problems with p_1 = 50%, p_2= 45% and u = 1, and 20 DisMSP with g = 9, m_a = 2 and u = 1.

We have applied the two proposed actions, namely the priority change and the unethical agent elimination on the two types of abnormal activities, namely the lying and the neglecting, which brings four combinations:

- Applying the priority change action on the liar agents;
- Applying the priority change action on the neglecting agents;
- Applying the elimination action on the liar agents;
- Applying the elimination action on the neglecting agents.

10.3.1 Random Problems

The results of the application of the proposed actions on the unethical agent present in random problems are displayed in Figures 10.4 and 10.5.

In the case the unethical agent is lying, the results of the actions are illustrated in figure 10.4. The first pie is showing the gravity of the presence of such an agent if we do not apply any action (the same results found in chapter 8, figures 8.1 and 8.2). The second one is presenting the gravity after applying the change priority action. While the third one is indicating the results after applying the second action, namely eliminating the unethical agent.

If we do not react against unethical agents, the resolution pass beyond the solution and finds catastrophic results. 45% of resolution processes are stopped after a timeout (TO), 25% of resolutions give wrong solutions, and just 30% (20% + 10%) find true solutions.

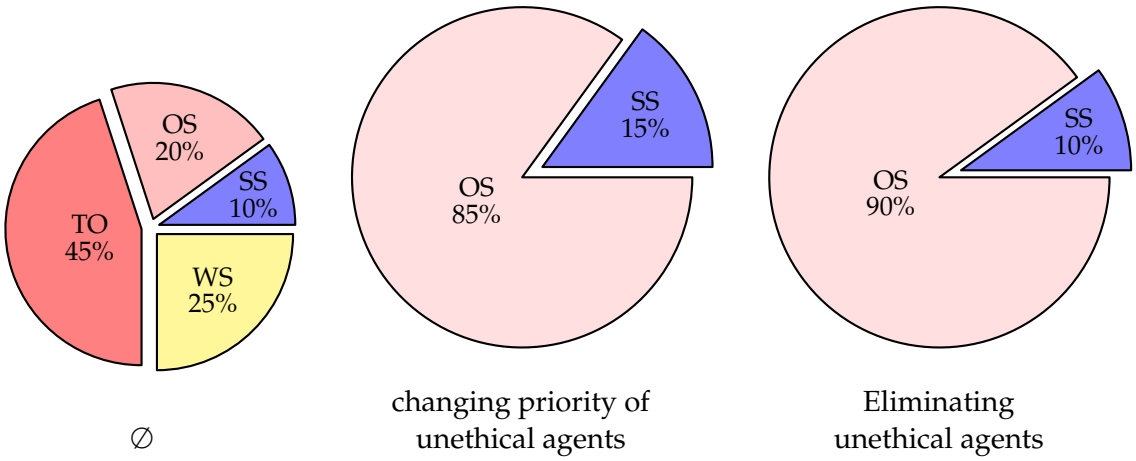


FIGURE 10.4: Lying activity in Random problems

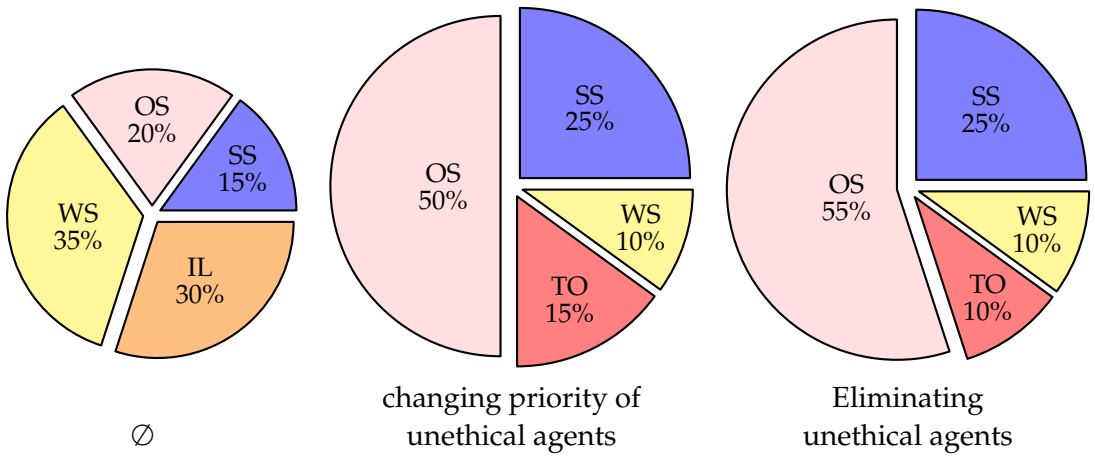


FIGURE 10.5: ngd neglecting activity in Random problems

After applying the priority change or the unethical agent elimination action, all the problems give true solutions. They sometimes miss the first solution which is normal. But at least they give correct solutions.

In the presence of unethical agents neglecting ngd messages in random problems (figure 10.5), the results become increasingly catastrophic. Without actions, the number of wrong solutions becomes bigger (35%) and 30% of problems loop infinitely without finding any solution.

After applying the two actions, the infinite loops disappear, more than 75% of problems find true solutions, just 10% of problems give wrong solutions and less than 15% are stopped after a timeout. The timeout detected in some problems is due to the number of instructions followed by every agent to detect the abnormal activity which should be

improved in future work.

10.3.2 Distributed MSP Problems

As we did for random problems, the results of applying the actions on unethical agents within the DisMSP problems are presented in figures 10.6 and 10.7.

When unethical agents are lying in DisMSP problems (figure 10.6), 45% of found solutions are incorrect, and 10% of problem resolutions are stopped after a timeout.

While, after changing the priority of the liar agents, or eliminating them, all problems find consistent solutions with 100%. They indeed miss the first solution, as in random problems, but the found solutions are correct.

The first solution's miss is always normal, since the two actions reformulate the original problem, either by reordering agent priorities or changing the number of agents. Therefore, the solutions' order is changed.

In the presence of unethical agents neglecting nogood messages in DisMSP problems' resolution, 65% (30% + 35%) of problems find the true solutions after applying the priority change action and 85% (65% + 20%) after applying unethical agent elimination action, compared to just 20% in case we do not apply any action (70% of problems looped infinitely and 10% found wrong solutions).

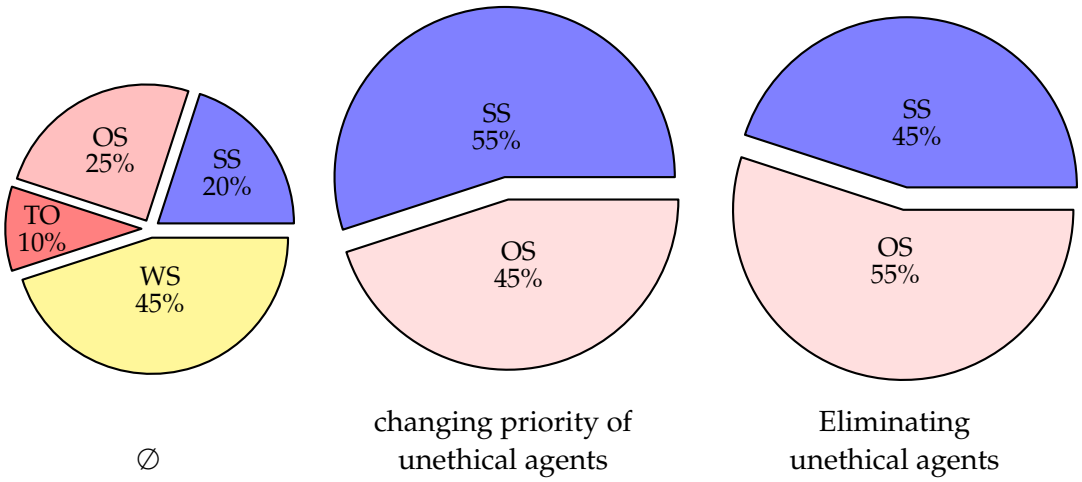


FIGURE 10.6: Lying activity in DisMSP problems

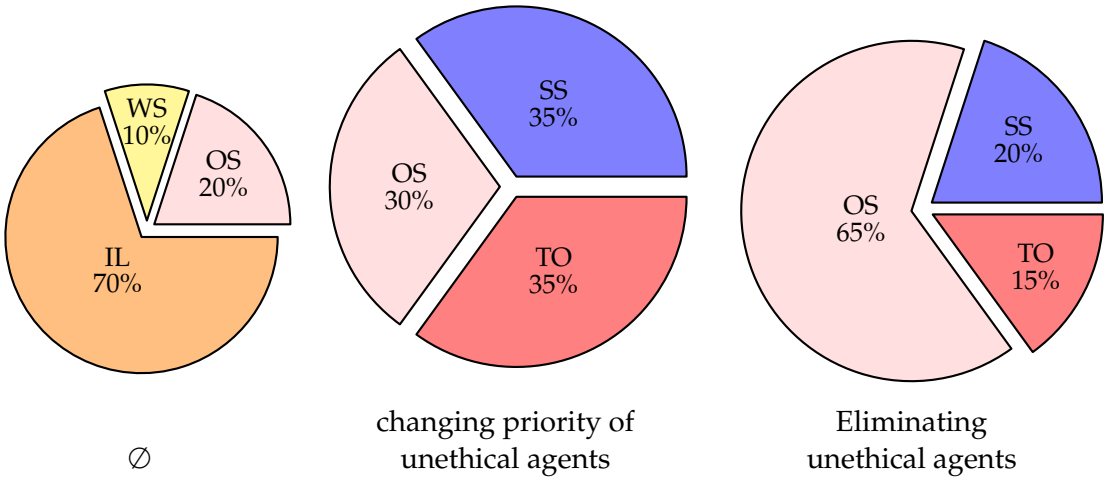


FIGURE 10.7: nogood neglecting activity in DisMSP problems

10.4 Conclusion

The great importance that ethics has taken in Artificial Intelligence and especially in Multi-Agent Systems has guided to deal with it in DisCSPs problems. This has inspired us to propose a new formalism called Ethical DisCSP, to support both ethical rules and actions.

In order to deal with ethics in this new formalism, we proposed a control system; to be integrated into a distributed, centralized or hybrid way; able to monitor the messages circulating between the agents in order to test their consistency with each other and their coherence towards the ethical rules to add, and then make decisions and actions if necessary.

We suggested methods of detecting anomalies in messages circulating and we have applied some actions which gave very satisfactory results.

In future work, we perceive to create the other types of controllers, namely centralized and hybrid, propose a manner to develop each agent, in order to be able to detect the abnormality of other agents; based on the voting principle; and finally, amplify the ethical unity of the control system, trying to apply some ethical rules on the resolution.

Part IV

Applications

11

Contribution to Modeling Smart Grid Problem with the Constraint Satisfaction Problem Formalism

Contents

11.1 Introduction

As proven in previous chapters, Constraint Programming can represent many real problems efficiently. One of the most recent real problems that can be represented and solved by CSPs, Complex DisCSPs, and why not our newborn E-DisCSP is the Smart Grids.

Smart Grids are electricity distribution networks that use intelligent IT methods to match the adequacy between consumer demand and provider offers. They are derived from the fact that the use of electricity is not stable throughout the day. There are schedules where the use of electricity remains minimal, and others the opposite. The modeling of these networks using computer methods is a rather complex task, due to the large amount of information offered by these networks and their complexity.

Modeling this problem using CP is an adequate solution. In this chapter, we are going to start with the very first CP formalism which is CSP, and try to move towards more general formalisms gradually.

For this purpose, the paper is organized as follows. Section 11.2 defines the smart grid problem. Section 11.3 presents related works. Section 11.4 exposes the CSP model of the Smart Grid local level. Section 11.5 broaches some experimental results. And section 11.6 exhibits the chapter conclusion.

11.2 Smart Grid

Power grids Eyer and Corey, 2010 are infrastructures routing and distributing energy from suppliers to consumers.

The high electricity demand (Figure 11.1) will unable current electricity networks to meet all customer needs because of the appearance of new electrical devices as electric cars and electric brushes. This will cause power cuts. According to the US Department of Energy, the current cutouts cause an annual loss of $80 billion and the improvement of these networks could save between 46 and 117 billion dollars from 2010 to 2023, hence the apparition of Smart Grids.

The smart networks aspect was appeared by the automatic meter reading in 1980, the sending of statistics of daily consumption using the meters in 1990, and the launching of Telegestore in 2000 (the first Smart Grid project linking 27 billion local).

11.2.1 Definition

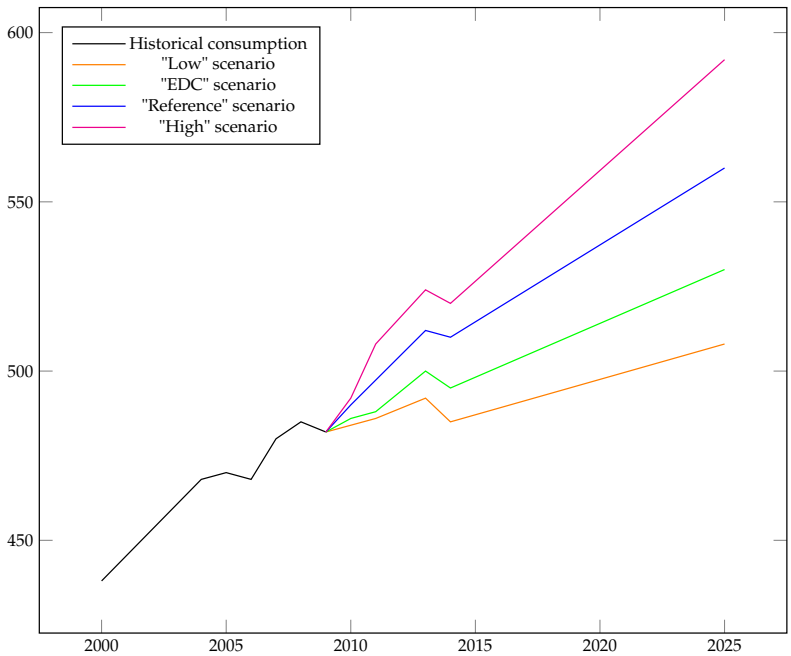
A Smart Grid Farhangi, 2009, Gungor et al., 2011 is an electrical network which uses the Information and Communication Technology (ICT) tools to optimize the production, the consumption, the storage of energy, as well as the losses.

Smart Grids contain three main components (Figure 11.2).

- Consumers who regulate their own consumption by using smart meters (Definition 16);
- Producers who try to meet the needs of consumers in real-time;
- Entities transporting and distributing electricity between producers and consumers.

Definition 16 (Smart Meter) *A smart meter is a meter used to facilitate monitoring tasks. It measures the energy consumption of domotics and it communicates in real-time with control centers. It can record the history of a subscriber's energy consumption and offer it at any time.*

The Smart Grid Problem is modeled in Ahat et al., 2013 by three major levels:



"Low" scenario: Assumptions reducing consumption
"High" scenario: Assumptions raising consumption
"EDC" scenario: Assumptions taking into account accelerating Energy Demand Control (EDC) through actions on consumer behavior.

FIGURE 11.1: Forecast of electricity consumption (source: Electricity Transmission Network in France)

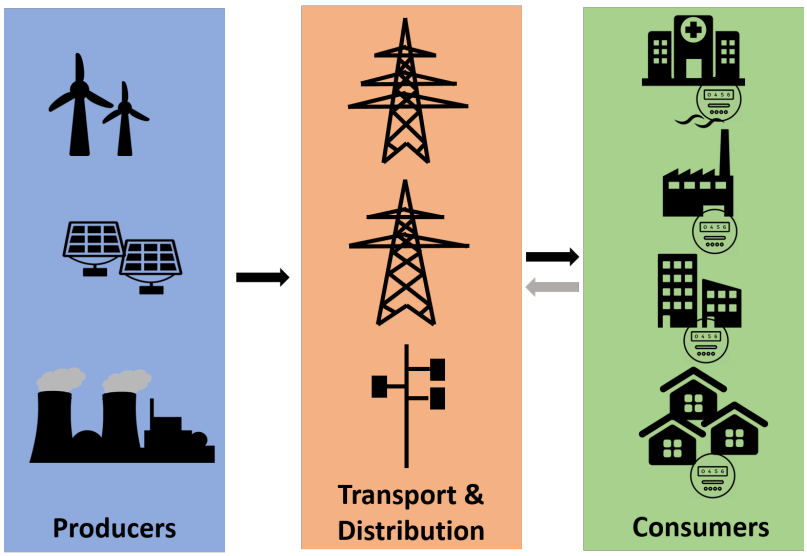


FIGURE 11.2: Main components of Smart Grids

- **Local level**: It represents the consumer. It includes all entities controlled by a smart meter. We can mention houses, factories, hospitals, buildings, etc;
- **Microgrid level**: It represents the bridge between consumers and producers. Its function is to spread the energy while equilibrating the demand and the supply;
- **Transmission and Distribution (T&D) level**: It represents the two other components of Smart Grid (producers and entities of transport and distribution). The main function of T&D is to transmit the energy offered by producers to microgrids.

The main objective of Smart Grids is to promote the use of renewable energies in order to reduce the overload made on power stations, reduce the greenhouse gas emissions, regulate the consumption, satisfy consumers, balance supply and demand, and to distribute energy without overcrowding.

The use of renewable energy sources is not an obvious task, as these sources depend on several uncontrollable factors. Wind turbines depend on the wind and photovoltaics depend on the sun. In this case, the periods of the production of the energy does not necessarily correspond to the periods of the consumption peaks. Hence the necessity to store the energy to use in needs.

11.2.2 Smart Grid Local Level

In this chapter, we are going to model and solve the local level, using CSP. Therefore, more local level specifications are required.

Studying the local level comes back to look into the devices composing the local. Some devices require a high energy value, while they do not have to operate in a specific time. The arbitrary functioning of these devices and at the same time causes a fairly large rise in consumption.

To control devices' functioning at the local level, a new variable is added for each device, called the functioning priority Guérard, 2014. It expresses the need to switch on a given device. The implementation of that priority functioning management will allow to smooth the consumption curve.

Devices whose priorities are small, can either be put in standby, postpone the start-up, or stop consuming during their operation. The action to be taken is chosen according to its priority value.

Some devices are not affected by this variable, as they must imperatively work. For example desktop computers, lights, or refrigerators. The priority value of these devices will default to 0 (the higher is the priority value, the lower is the priority). While others take their values based on user behavior and usage statistics. In this case, it is possible that some devices have a priority that grow or decrease over time.

11.2.3 Smart Grid Characteristics and Objectives

Smart Grids are characterized by:

- Flexibility: the networks allow the interconnection of distributed production and energy storage at any moment. They provide the opportunity to integrate new technologies.
- Self-Healing: they automatically detect potentially defective devices and reconfigure the providing system to meet customers without penalties.
- Predictability: they predict future events and adapt the system taking into account these events, using statistics, machine learning, and other methods.
- Security: they encrypt and protect the two-way exchanges to avoid any external manipulation of the data.
- Accessibility: they promote the integration of renewable energy sources throughout the network.
- Economy: they save energy and reduce costs even in production as in consumption.

The main Smart Grids' objectives are:

- avoidance of congestion in energy distribution;
- consumer satisfaction;
- consumption regulation;
- equilibration of supply and demand.

11.3 Related Works

The proposition of computer science algorithms can somehow solve this problem. But the realization of a single algorithm to meet all the requirements of all levels is a very complex task or even impossible. Even in the case where researchers arrive to develop this kind of algorithm, it will take exponential time to meet all the needs of each consumer, at each local level and each microgrid. While the key identity of Smart Grids is the real-time response.

For this reason, researchers interested in the development of Smart Grid algorithms exhibit an algorithm for each level (local, microgrid, and T&D).

In Marah and El Hibaoui, 2018, the authors have proposed an algorithm for the local level and another for the T&D one. They have modeled the local level as a "**Knapsack problem** Ross and Tsang, 1989, Chu and Beasley, 1998" and have proposed to solve it with a new version of the "**Branch and Bound**" algorithmKolesar, 1967, Dyer, Kayal, and Walker, 1984.

Figure 11.3 presents an example of the Knapsack Problem. It contains a set of objects. Each object is identified with a value and a weight. The total weight exceeds the bag capacity. The Knapsack principle is to maximize the number of valued objects while not overtaking the bag capacity. If we project this problem on the Smart Grid, the bag models the local (house, building, ...) and the objects represent the local devices. The object value is presented by a priority of the device and the object weight is the device consumption.

In the last cited work, the possible priority values they consider are "0" and "1". "0" to express that the device should be ON, and "1" to say that the device functioning is not so obligatory.

In Marah and El Hibaoui, 2018, the algorithm used to resolve the local level is a merge of the "Branch Bound" algorithm and their own algorithm, namely the "Priority Management Algorithm". The latter gives the priority to devices with "0" priority to use the

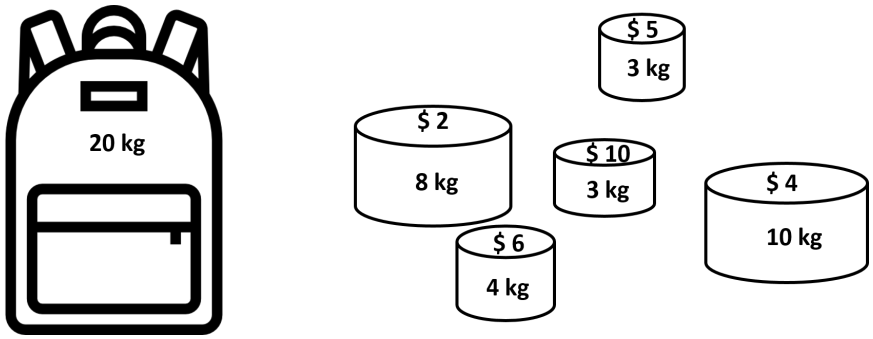


FIGURE 11.3: Knapsack problem example

energy and subtracts the sum of their energy consumptions from the total energy. The rest of the energy is given to the Branch and Bound algorithm as an input, to distribute it among the other devices, by maximizing the profit.

In Ahat et al., 2013, the authors treat Smart Grid as an Optimization Problem whose objective is to manage the consumer side, by regularizing the consumption curve, and to synchronize data for effective results, by running each station and each piece of equipment in the same time. To this end, they modeled the local problem as a Knapsack problem too.

For the microgrid level, authors proposed to use "Game networks" La Mura, 2000 which is a formalism in game theory Osborne and Rubinstein, 1994, Myerson, 2013 used to permit the players to play multiple games at the same time. For the T&D level, they proposed to use Max Flow Shiloach and Vishkin, 1982 Boykov and Kolmogorov, 2004 and Equilibrium algorithms Florian and Hearn, 1995 to avoid routing and congestion problems. These algorithms are known in electric networks.

11.4 CSP based Smart Grid's Local Level

The local problems of Smart Grids contain data predefined at the outset, such as forecasts, the devices' consumption, their priorities, and the battery capacity. So before defining the model, we are going to identify what are the problem data.

11.4.1 Data

Any Smart Grid's local problem is compiled by a set of data:

- The number of devices n;
- The number of slots, of which it is supposed to plan consumption s;

- The predicted amount of energy at each slot i $E_i / i \in \{1, 2, \cdots, s\}$;
- The predicted amount of energy, offered by renewable energies' sources, at each slot i $Re_i / i \in \{1, 2, \cdots, s\}$;
- The consumption of each device j $C_j / j \in \{1, 2, \cdots, n\}$;
- The priority of each device j $P_j / j \in \{1, 2, \cdots, n\}$;
- The maximal energy the power plant can produce E_{Pmax};
- The battery capacity B.

Modeling the problem amounts to identify variables, domains, as well as constraints, while using the data we defined.

11.4.2 Variables & Domains

The set of variables is

$$X = \{X_{c_i}, X_{E_{p_i}}, X_{E_{r_i}}, X_{E_{b_i}}, X_{ON}\}$$

,

1. The local consumption of the period i:

$$X_{c_i} \in [0, E_i]$$

2. The energy amount we will recover from power plant at the slot i:

$$X_{Ep_i} \in [0, EPmax]$$

3. The energy amount we will recover from renewable energies' sources at the slot i:

$$X_{Er_i} \in [0, Re_i]$$

;

4. The energy amount to store in the battery at the slot i:

$$X_{Eb_i} \in [0, B]$$

5. The functioning state of each device j at the slot i:

$$X_{ON_{i,j}} \in \{0, 1\}$$

1 is signifying that device is ON, while 0 is meaning the device if OFF;

11.4.3 Constraints

The set of constraints is

$$C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$$

C1 The local consumption should be less than or equal to the expected consumption at each slot i

$$X_{c_i} \leq E_i / \forall i \in \{1, 2, \cdots, s\} \tag{11.1}$$

C2 The total consumption at the slot i is equal to the sum of consumptions of running devices

$$X_{c_i} = \sum_{j=1}^{n} C_j \times X_{ON_{i,j}} \tag{11.2}$$

C3 The used energy recovered from power plant X_{Ep_i}, renewable energies' sources X_{Er_i}, and battery X_{Eb_i} should be greater than or equal to the total consumption. X_{c_i}

$$X_{Ep_i} + X_{Er_i} + X_{Eb_i} \geq X_{c_i} / \forall i \in \{1, 2, ..., s\} \tag{11.3}$$

C4 The energy recovered from the power plant should be equal to '0' when the energy produced by renewable energies' sources is sufficient

$$X_{Er_i} \geq X_{c_i} \Rightarrow X_{Ep_i} = 0 \tag{11.4}$$

C5 If the energy produced by renewable energy sources is insufficient; even after adding the energy stored in the battery; it should be stored in the battery. But before energy storage, the battery capacity should be checked. If it is exceeded, we keep the maximum energy which is the battery capacity itself

$$X_{Er_i} < X_{c_i} \vee X_{Er_i} + X_{Eb_i} < X_{c_i} \Rightarrow \begin{cases} X_{Eb_{i+1}} = X_{Eb_i} + X_{Er_i}, & \text{if } X_{Eb_i} + X_{Er_i} \leq B \\ X_{Eb_{i+1}} = B, & \text{else} \end{cases} \tag{11.5}$$

C6 Otherwise (ie. the sum of the energy produced by sources of renewable energies and that stored in the battery is sufficient), The battery will contain the remainder, taking the battery capacity into consideration

$$X_{Er_i} + X_{Eb_i} \geq X_{c_i} \Rightarrow \begin{cases} X_{Eb_{i+1}} = X_{Eb_i} + X_{Er_i} - X_{c_i}, & \text{if } X_{Eb_i} + X_{Er_i} - X_{c_i} \leq B \\ X_{Eb_{i+1}} = B, & \text{else} \end{cases} \tag{11.6}$$

C7 If the sum of devices' consumptions having the highest priority ("1") does not exceed the forecasts, then all these devices will be functional. For other priorities, the same test will be done but by comparing the devices' consumptions with the remaining energy, after subtracting the energy used to operate the previous devices from the predicted energy.

$$\begin{cases} \begin{cases} \sum_j C_j \leq E_i / \\ C_j \textit{ is the consumption of devices with priority } 1 \end{cases} \\ \textbf{or} \\ \begin{cases} \forall p \in \{2, \cdots, p_{max}\}, \forall i \in \{1, 2, \cdots, s\} \\ \sum_j C_j \leq E_i - \sum_l C_l / \\ C_j \textit{ is the consumption of devices with priority } p \\ C_l \textit{ is the consumption of devices with priority } p-1 \end{cases} \end{cases} \rightarrow \begin{cases} \forall k \in \{1, \cdots, n\} / P_k = p, \\ X_{ON_k} = 1 \end{cases} \tag{11.7}$$

Slots	1	2	3	4	5	6	7	8	9	10
E	4561	4424	3961	3958	4139	4445	4879	5216	5459	5543
Re	2000	4500	4000	3400	3500	3600	4000	4100	4200	4400

TABLE 11.1: Predictions

Devices	TV	Fr	L	VC	HD	EO	ER	WM	D	Fz	C	H
C	190	3900	200	600	100	2050	160	1200	50	175	160	1400
P	1	1	2	2	3	1	3	2	1	2	2	1

TABLE 11.2: Consumptions and priorities of used devices

11.5 Experimental Results

To prove the feasibility and scientific diligence of the proposed modeling, the model is implemented using the Choco 4 solver in a workstation with 3.50GHz in processor and 32 Go in RAM.

We took a home example with twelve devices: Television (TV), Computer (C), Washing Machine (WM), Vacuum Cleaner (VC), Electric Razor (ER), Hair Dryer (HD), Lamp (L), Dishwasher (D), Heating (H), Fridge (Fr), Freezer (Fz) and Electric Oven (EO) with different priority values and capacities (table 11.2). For predictions' values, we chosen the values shown in table 11.1.

The results of the implementation, for 1 slot, 2, 3, ... and 10 slots, namely the number of found solutions and the consumed time (in Millisecond Ms), are shown in table 11.3.

The table shows that we achieve important results, namely 13 solutions for 1 slot. This number increases exponentially by increasing the number of hours, which proves the

Number of hours	Number of solutions	Time (Ms)
1	13	160
2	221	212
3	525	359
4	789	382
5	2685	656
6	13176	2447
7	74844	31970
8	420768	145462
9	2151144	493017
10	10519200	4238801

TABLE 11.3: CSP modelization results

$slots$	X_{c_i}	E_i	Production			Devices									
			X_{Ep_i}	X_{Er_i}	X_{Eb_i}	TV	Fr	L	VC	HD	EO	ER	WM	D	Fz	C	H
1	2000	4561	0	2000	0				x								x
2	3900	4424	0	4500	600		x										
3	600	3961	0	4000	4000				x								
4	3450	3958	0	3400	50			x			x		x				
5	0	4139	0	3500	3500												
6	3600	4445	0	3600	3550	x					x		x			x	
7	4090	4879	0	4000	90	x	x										
8	600	5216	0	4100	3590				x								
9	4225	5459	0	4200	25			x	x		x		x		x		
10	5200	5543	0	4400	800		x			x			x				

TABLE 11.4: One solution proposed by our implementation

efficiency of the model. It does not besiege the user with a limited number of solutions. That allows him to choose a convenient solution. We also notice that the resolution time does not evolve in the same way as the number of solutions. This shows the performance of the model.

Table **??** shows one of the solutions proposed by our implementation, for the same example but for 10 slots. It shows the local consumption, the prognosis, the production of the power plant and renewable energies, the energy stored in the battery, and also the state of devices (running or not) at each slot.

We note that:

- the consumption is always lower than prognosis;
- the power station is used if and only if the energy offered by the battery and renewable energies is insufficient;
- When the power plant energy is used the others are stored in the battery;
- the devices' priorities are taken into consideration;
- the battery capacity is taken into consideration.

11.6 Conclusion

In this chapter, we modeled the Smart Grid local problems using CSP, which is rather a crucial problem due to the huge quantity and the complexity of the information to process. To achieve this goal, we deeply described the global and local operations of the Smart Grid problem and described the data as well as the CSP parameters identifying the problem.

Experimental results showed the feasibility of the modeling. But there are some parameters to adjust, as maximizing the number of running devices, minimizing the gap between prognosis and the consumption, and optimizing many other parameters. In the next chapter, we are going to model the same problem using COP, so as the optimization of the parameters can be possible.

12

Modelization of Smart Grid Local Problems Based on Constraint Optimization

Contents

12.1 Introduction

The main objectives of the Smart Grid make it an optimization problem by definition, especially local levels. Many parameters have to be optimized, the number of functional devices, the favor of higher priority devices, as well as the lag between consumption and forecasting.

In this chapter, we are going to improve the last proposed modeling in order to treat the Smart Grid's local level as a COP problem. For this purpose, the chapter will be organized as follow. Details of our COP modelization are described in Section 12.2. section 12.4 presents an experimental evaluation of our proposed model and we conclude the chapter in section 12.5.

12.2 Smart Grid' Local Level as an Optimization Problem

As seen in the previous chapter, almost all studies treat Smart Grid' local level as a knapsack problem, which is quite correct, since we are trying to maximize the number of functional devices having the highest priority.

Moreover, the characteristics and objectives of the smart Grid require minimization of the gap between forecasts and actual consumption. Therefore the Smart grid problem, especially the local level, is an optimization problem by definition, whose objective functions are:

- promote the operation of devices with a higher priority;
- maximize the number of functional devices;
- minimize the lag between consumption and forecasting;

12.3 COP based Smart Grid's Local Level

Modeling the problem amounts to identify the variables X, the domains D, the constraints C, and the objective functions O. The first 3 parameters are the same as the CSP modeling made in the previous chapter:

- $X = \{X_{c_i}, X_{E_{p_i}}, X_{E_{r_i}}, X_{E_{b_i}}, X_{ON}\}$;
- $D = \{D(X_{c_i}), D(X_{E_{p_i}}), D(X_{E_{r_i}}), D(X_{E_{b_i}}), D(X_{ON})\}$;

- $C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$;

For objective functions, we identify four:

$$O = \{O_1, O_2, O_3, O_4\}$$

O1 Maximize the number of running devices, at each slot i

$$Min(\sum_{i=1}^{s} m - \sum_{j=1}^{n} X_{ON_{i,j}}) \tag{12.1}$$

O2 Minimize losses by minimizing the lag between total consumption and forecast, at each slot i

$$Min(\sum_{i=1}^{s} E_i - X_{c_i}) \tag{12.2}$$

O3 The constraint C_7 is used to distribute the energy in priority order. In the case where the remaining energy suffices all the devices of the same priority, it is distributed between them. But in the case where the remaining energy is no longer sufficient, there is no constraint to control its distribution. The purpose of the objective function O_3 is therefore to distribute this energy by maximizing the number of functional devices having the highest priority.

$$Min(\sum_{j=1}^{s} \sum_{i=1}^{n} P_i \times X_{ON_{i,j}}) \tag{12.3}$$

The more the priority p is high, the higher the term $P_i \times \sum_{i=1}^{n} X_{ON_{i,j}}$ is (p is highest

Device	d_1	d_2	d_3	d_4	d_5	
Priority	1	2	1	3	2	O_3
X_{ON}	**1**	**1**	**1**	**0**	**0**	**4**
X_{ON}	1	1	0	1	0	6
X_{ON}	1	1	0	0	1	5
X_{ON}	1	0	1	1	0	5
X_{ON}	**1**	**0**	**1**	**0**	**1**	**4**
X_{ON}	1	0	0	1	1	6

TABLE 12.1: O_3 values in different scenarios

$\rightarrow$ devices with p are the lowest priority ones). So, to give the hand to the higher priority devices before the lower priority ones, we will try to minimize this term. In the example shown in table 12.1, we suppose to have 5 devices with different priorities 1, 2, or 3. The example shows six energy distribution scenarios and computes

the objective function O_3, for each scenario. It shows that the two scenarios where O_3 is minimal are the best.

O4 In some cases, the number of higher priority devices is very high compared to the number of lower priority devices. So the objective function O_3 is insufficient. To this end, we have to increase the coefficient associated with the priority ($10^{priority}$) and link it with the number of functional devices with priority *priority* compared to the number of all functioning ones.

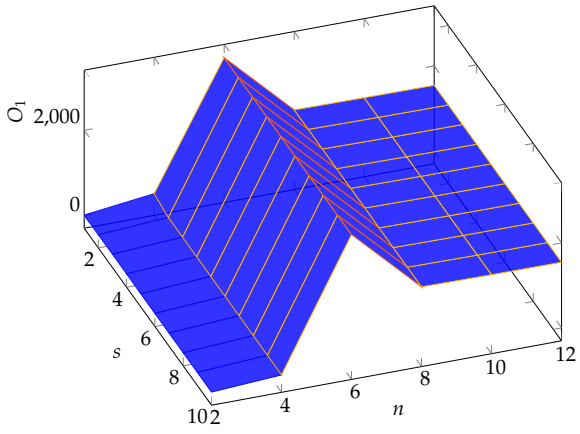
$$Min(\sum_{j=1}^{s} \frac{\sum_{i=1}^{n} P_i \times 10^i \times X_{ON_{i,j}}}{\sum_{k=1}^{n} X_{ON_{k,j}}}) \tag{12.4}$$

In order to prove the efficiency of the COP modelization, we have to demonstrate that either theoretically or experimentally. Since the problem is real, we have to experiment with the model in different realistic instantiations, in a real local, with a given number of slots, a given number of devices, and some realistic predicted energy values. We are going to use the same instances 11.2 as the precedent chapter.

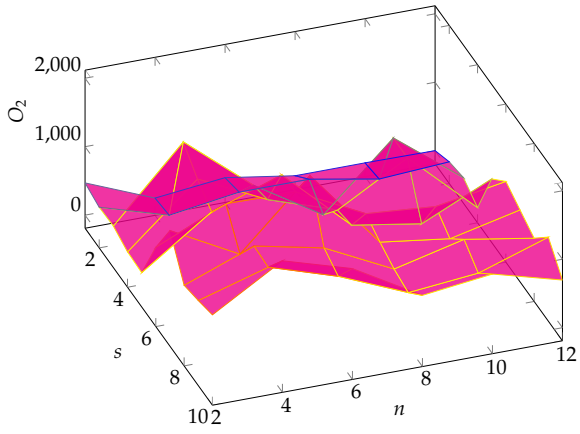
12.4 Experimental Results

To evaluate the performance of the COP modelization and study its complexity, we applied it to local problems characterized by $< n, s, E, Re, C, P, B >$.

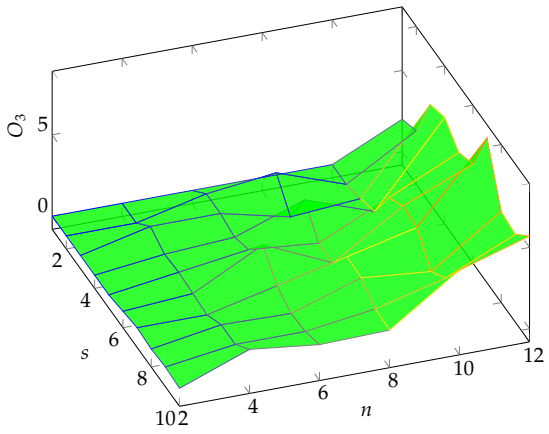
- n is the number of devices, which takes values from $\{2, 4, 6, 8, 10, 12\}$ (2 = the two first devices from table 11.2, 4 = the four first devices, ...);
- s is the number of slots. It varies from 1 to 10 by 1 as step;
- E is the set of predicted consumed energy values for all slots (Table 11.1);
- Re is the set of predicted energies offered by renewable energies' sources for all slots (Table 11.1);
- C is the set of consumptions of the devices;
- P is the set of priorities of the devices. Table 11.2 shows the names of the devices that can be integrated into the local, the capacity of each device and its priority;
- B is the capacity of the battery. It is equal to 4000.



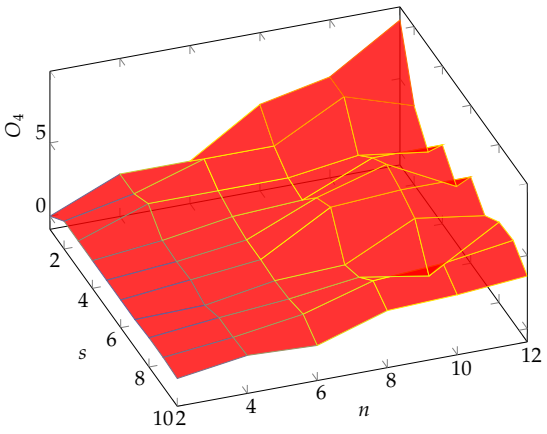
The objective function O_1



The objective function O_2



The objective function O_3



The objective function O_4

FIGURE 12.1: Objective functions evaluation

In addition to the quality of the returned solutions, the performance of the COP modeling is also be measured by the objective functions' values, so as to confirm, they are really minimal or not. But first, let study the complexity of each objective function separately.

Figure 12.1 shows the variation of the four objective functions O_1, O_2, O_3 and O_4 comparing to the number of slots s and the number of devices n composing the local. The problems' resolution can give several optimized solutions. In that case, the taken value is the average ($\frac{O_i}{The\ number\ of\ solutions}$, i = 1, 2, 3 or 4). In the case of $n > 1$, we also take the average of each objective function values ($\frac{\frac{O_i}{The\ number\ of\ solutions}}{n}$).

The plots show that the first objective function O_1 does not depend on the number of

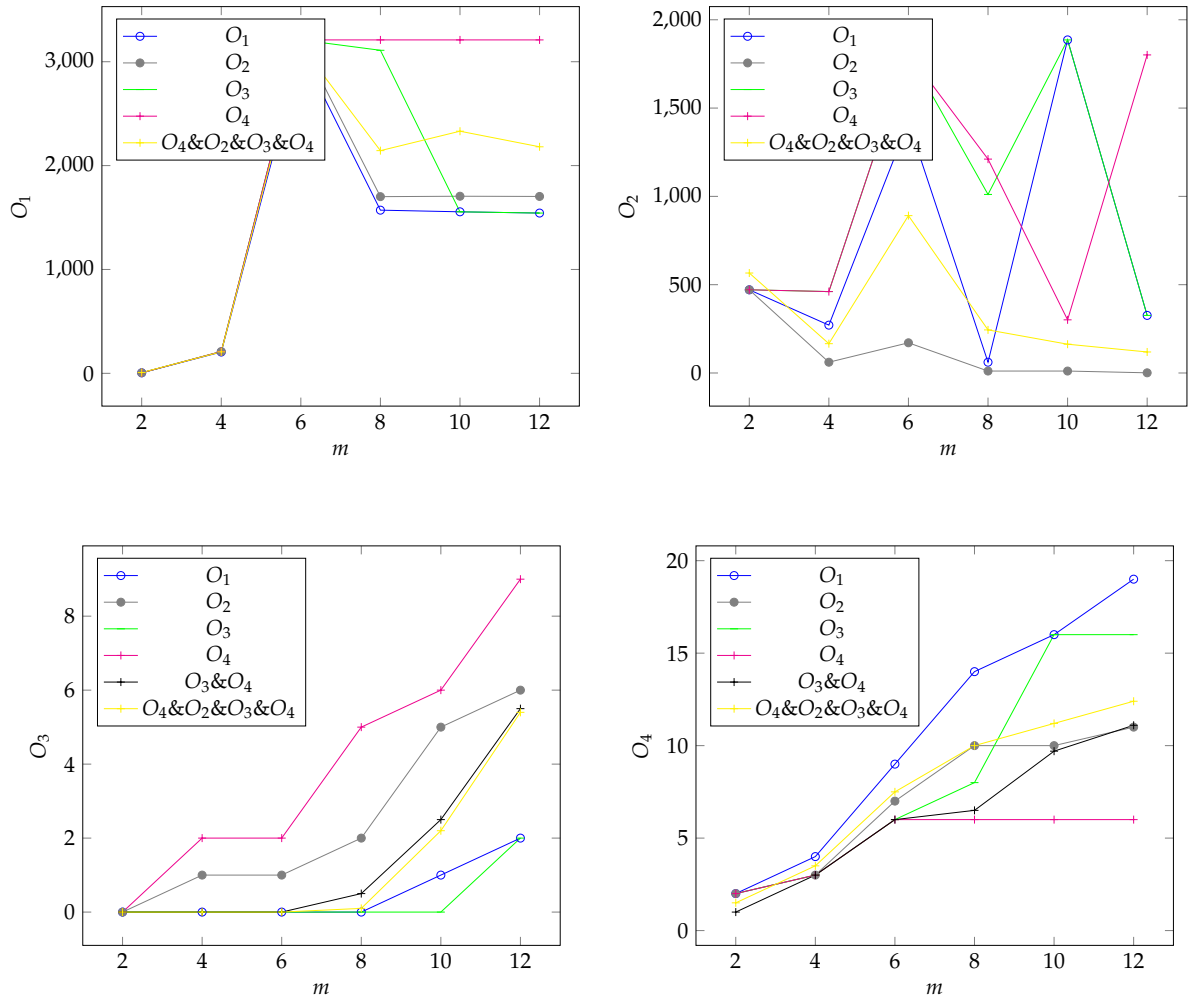


FIGURE 12.2: The influence of objective functions' optimization when n = 1

slots. It is still constant when the number of devices is fixed, which is normal because the objective of O_1 is to maximize the number of the running devices (minimizing the recall between the existing devices and the functioning ones). The variation of O_1 according to the number of devices can be considered linear except when n is equal to 6 because the six first devices include three devices having the higher priority, but cannot be served all in the same time, because the sum of their consumptions surpasses the provided value. So only two devices will have the priority to function, and the remaining values will be distributed among the lowest priority devices while the remaining energy amount is low.

The variation of O_2 is linear too, according to the number of devices and the number of slots. When the devices are more numerous, O_2 becomes minimal. Which is normal, because O_2 aims to minimize the lag between the local consumption and the forecast. When we have a bit number of devices, even we make all devices ON, we still far from

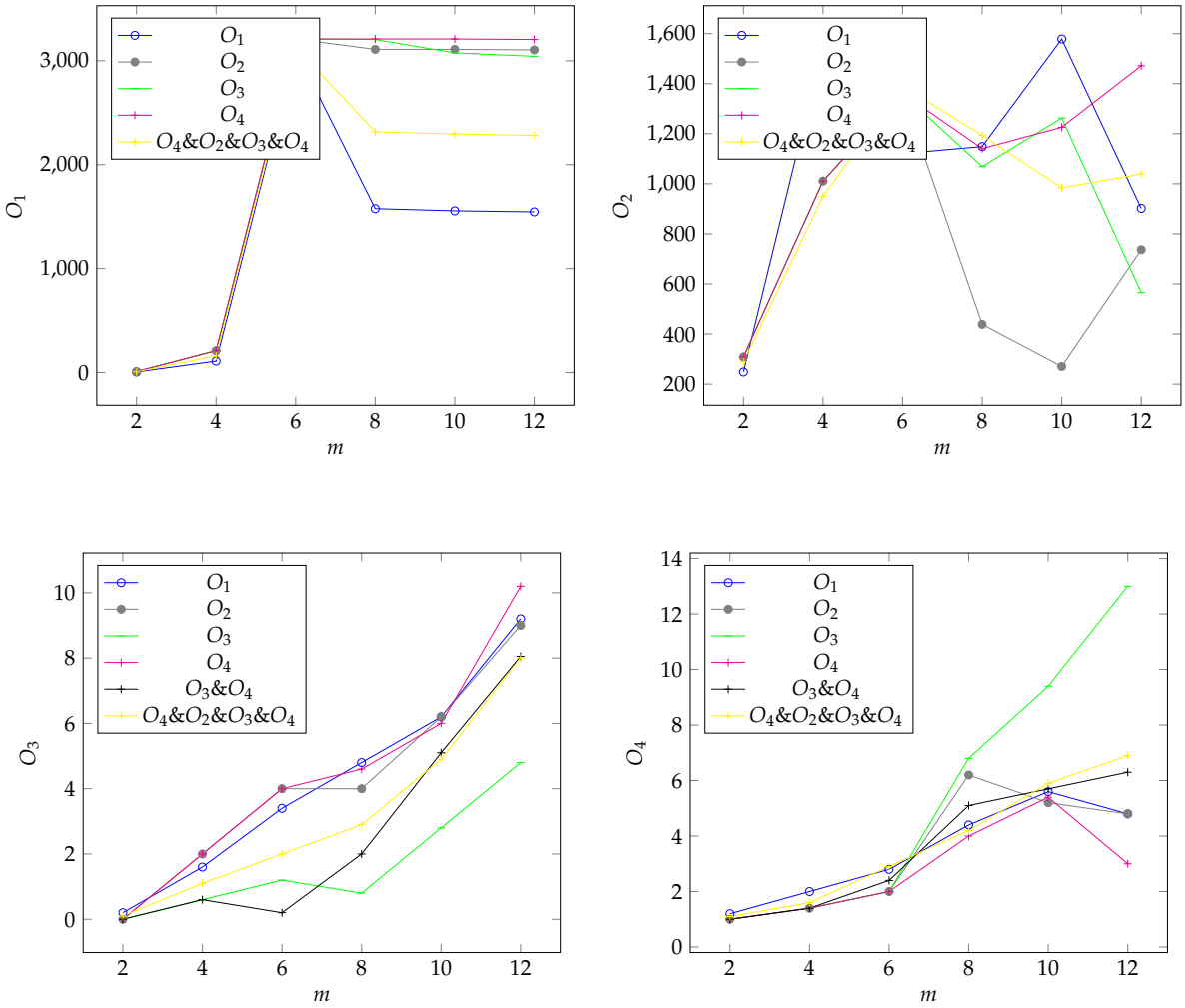


FIGURE 12.3: The influence of objective functions' optimization when n = 5

the forecast. The last point is a highlight of O_2.

For O_3, it is linear too, according to the two parameters. It becomes more important when the number of slots and the number of devices increase. This is very normal according to its expression. It is an increasing function, summing the priorities of devices which are active. So the more we have active devices, the more their priorities are considered. The values of objective functions will indeed be divided by the number of slots to recover their averages. So the fact that the O_3 expression depends on the number of slots does not explain why it is growing when the number of slots is larger. The real reason is to have more energy stored in the battery and so the possibility of operating more devices.

O_4 behave similarly to O_4 which can be considered a completeness function of O_3. Except that it depends inversely on the number of active devices. So, contrary to O_3,

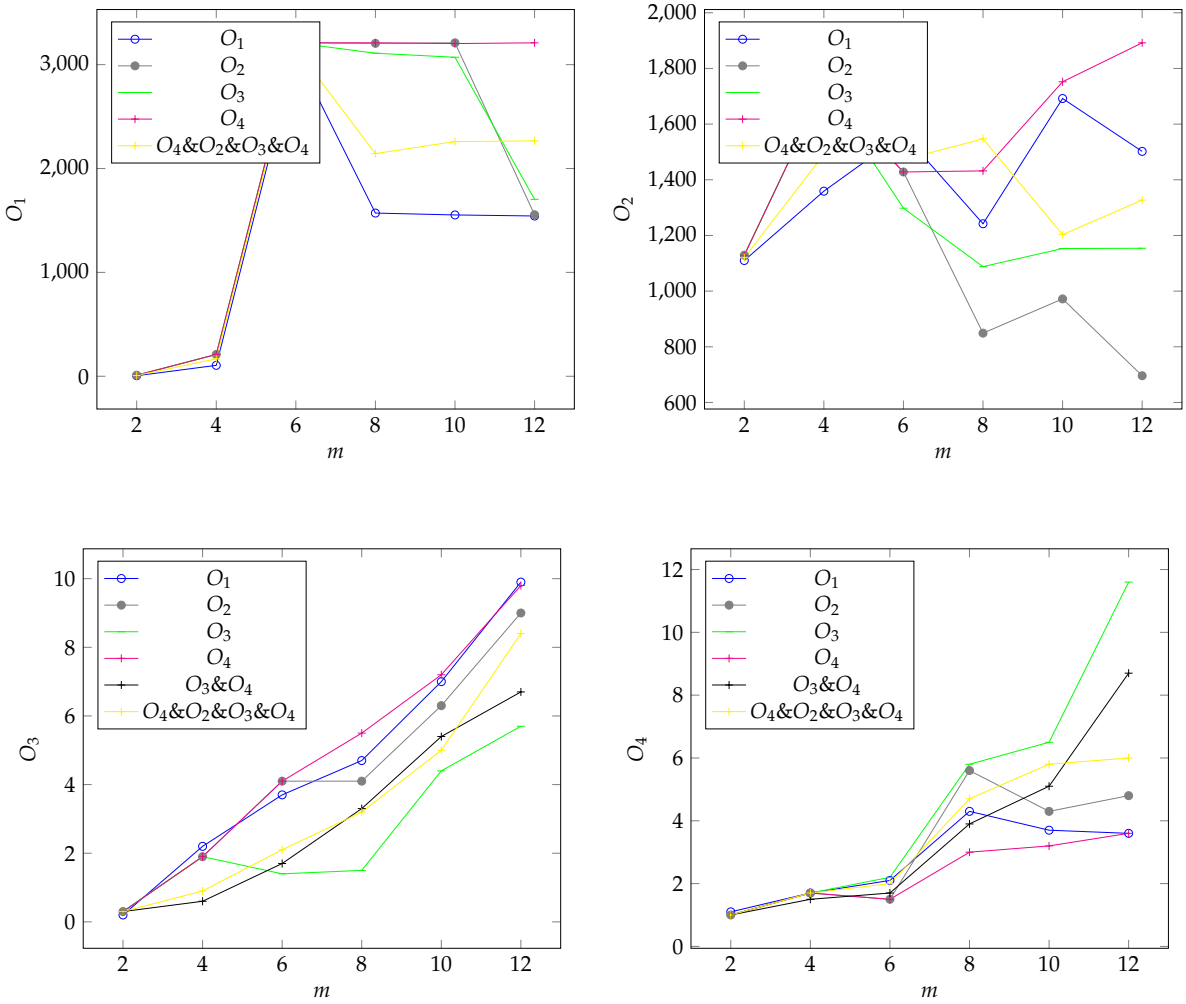


FIGURE 12.4: The influence of objective functions' optimization when n = 10

which is increasing, O_4 is decreasing when the number of devices is increasing.

After evaluating each objective function separately, figures 12.2, 12.3 and 12.4 show how much their optimizations make the difference. According to figure 12.1, the behavior of a local depends on devices it contains. So, we choose 3 values of n to scan all types of problems. A small value n = 1 (Figure 12.2), a medium value n = 5 (Figure 12.3) and another large n = 10 (Figure 12.4).

We can check that a function is optimized, only if we compare it with its different values when the optimization is not carried out. Such a COP problem returns multiple non-optimal solutions. So, to evaluate the objective function optimization against non-optimized solutions, we have to select some general cases (4 or 5 cases).

For this purpose, we select the cases when we optimize the other objective functions, others than the one to assess. Assuming that there will be no values greater than those

returned by these cases, where the resolution process makes more effort, to optimize the other objective functions.

To this end, we are going to evaluate each objective function in 5 cases. The first case concerns the optimization of the objective function itself, the second, the third and the fourth case concern the optimization of the three other functions and the fifth case refers to the optimization of the four objective functions concurrently, based on the principle that the four cases present four possible behaviors of the function without optimization (minimization).

For the first objective function O_1, it has the same behavior in the three cases (n=1, n=5 and n=10). Optimizing this function affects the solution. Its optimization is better than all values of other solutions' O_1. Especially when the number of devices is large, the difference is very important.

Unlike O_1, O_2 has not the same behavior when the number of slots is changed. For $n = 1$, the optimization of O_2 makes a very remarkable difference. It is very better than other cases. It can be considered stable according to the number of devices, contrary to other instances. For $n = 5$ and $n = 10$ it behaves in the same manner. It is also the best but the optimization impact can be seen when the number of devices is higher.

For O_3 and O_4, they behave, for the three values of n, in the same way. The optimization becomes more and more important whenever the number of devices increases.

For the three values of n, we tried to evaluate the impact of optimizing all objective functions concurrently on O_1, O_2, O_3, and O_4. The figures show that the obtained results are so satisfying. They are all near to the objective function optimization itself.

To give the obtained results a life, we consider one of the processed problems (the same as Chapter 11) and show the values taken by the variables defining the problem.

Tables 11.4 and 12.3 show the variables' values without (Table 12.2) and with (Table 12.3) the optimization of all function objectives concurrently.

The two tables show that all defined constraints are satisfied as perceived in chapter 11. The local consumption never exceeds forecast and it is equal to the sum of running devices' consumptions, the use of the energy produced by renewable energy sources is the most favored, and the unused energy is stored in the battery.

We must recall that O_1 aims to maximize the number of running devices compared to the existing ones. So, the values shown in the table represent, more precisely, the number of nonfunctional devices. It is noticed that is almost always minimal when O_1

i	X_{c_i}	E_i	Production			Devices											Objective functions			
			X_{Ep_i}	X_{Er_i}	X_{Eb_i}	TV	Fr	L	VC	HD	EO	ER	WM	D	REF	PC	H	O_1	O_2	O_3	O_4
1	2000	4561	0	2000	0				x								x	10	2561	3	105
2	3900	4424	0	4500	600		x											11	524	1	10
3	600	3961	0	4000	4000				x									11	3361	2	200
4	3450	3958	0	3400	50			x			x		x					9	508	5	136.7
5	0	4139	0	3500	3500													12	4139	0	-
6	3600	4445	0	3600	3550	x					x		x			x		8	845	6	105
7	4090	4879	0	4000	90	x	x											10	489	2	10
8	600	5216	0	4100	3590				x									11	4616	2	200
9	4225	5459	0	4200	25			x	x		x		x		x			7	1234	10	162
10	5200	5543	0	4400	800		x			x			x					9	343	6	1070
																		9.8	1962	3.7	222

TABLE 12.2: One found solution using the CSP modelization **el2018contribution**

i	X_{c_i}	E_i	Production			Devices												Objective functions			
			X_{Ep_i}	X_{Er_i}	X_{Eb_i}	TV	Fr	L	VC	HD	EO	ER	WM	D	REF	PC	H	O_1	O_2	O_3	O_4
1	4275	4561	2275	2000	0		x	x							x			**9**	**286**	5	136.7
2	4140	4424	0	4500	360	x	x							x				**9**	**284**	3	**10**
3	1840	3961	0	4000	2520	x		x						x			x	**8**	**2121**	5	**73.3**
4	3785	3958	0	3400	2135						x				x	x	x	**8**	**173**	6	**105**
5	4110	4139	0	3500	1525		x					x		x				**9**	**29**	5	**1006**
6	3950	4445	0	3600	1175				x	x	x		x					**8**	**495**	8	802.5
7	4595	4879	595	4000	0		x	x				x			x	x		**7**	**284**	9	722
8	4850	5216	0	4100	750		x	x	x	x				x				**7**	**336**	9	684
9	2335	5459	0	4200	2615				x			x			x		x	8	3124	**8**	855
10	5425	5543	1025	4400	0	x		x			x	x	x	x	x		x	**4**	**118**	13	455
																		7.7	728	7.1	484.9

TABLE 12.3: One found solution using the COP modelization

is optimized and, admittedly, the number of devices in the second case is great than the first one (table 12.2).

For O_2, which aims to minimize the recall between the consumed energy and the forecasted one, it is also almost minimal when it is optimized.

Unlike O_1 and O_2, the values O_3 and O_4 are not minimal, even with optimization. This is due to the number of functional devices. When O_1 is minimal, the number of running devices is large. So O_3, which compute the priorities of running devices, will be large too. In table 12.2 , O_3 is better, because the number of devices is low. This can be proved by the solution of the ninth slot (s = 9). In this instance, O_1 is not minimal but O_3 is.

For O_4 which is considered as an O_3 complement, We effectively notice that half of the values are low in table 12.3 compared to table 12.2, and that is outstanding by the good distribution of energy among devices, which takes into consideration the priority of devices.

12.5 Conclusion

In this chapter, we modelized the Smart Grid's local level as a COP problem, which is a decisive problem. The quantity of parameters to be considered makes the problem's purpose complex. But the found results prove the feasibility, the intelligence, and the strength of our proposal. The whole Smart Grid problem can be summed up as a set of local levels. Therefore, it is intended to solve the whole Smart Grid problem, using DisCSP and DCOP with Complex Local Problems.

Conclusion and Perspectives

In this thesis, we have addressed the Distributed Constraint Satisfaction Problem formalism (DisCSP) and its possible extensions, namely the DisCSP with complex local problems and the ethical DisCSP. We have proposed new effective methods, algorithms, and even a new formalism.

After presenting the Constraint Satisfaction Problem (CSP) formalism, its applications, as well as its algorithms, we have exposed the DisCSP in the same way. To illustrate and explain the existing as well as realized algorithms and methods in a real-world example, we have chosen the Distributed Meeting Scheduling Problem (DisMSP).

In the first part of the thesis, we have thought of taking benefit of the existing DisCSP methods and algorithms, by proposing a new DisCSP JChoc platform, enriched by the exiting DisCSP protocols and enabling the integration of new ones, as well as experimenting them. Additionally, it is designed in such a way it is aimed even at users, for applications. In addition to the platform, we have proposed two learning methods, in order to allow launching more than one algorithm at the same time, since the performance of a DisCSP algorithm depends on the type of problem to solve.

The investigation of these algorithms in detail, allowed us to identify some limitations. The algorithms are designed for simple problems when there is exactly one variable per agent, which does not represent reality. In addition, these algorithms run only in the perfect case, when all agents are correct. While the agents' autonomy allows them to make incorrect and sometimes unethical autonomous decisions. For this, we have worked on two disciplines in two other different parts.

In the second part, we have tackled DisCSPs with complex local problems, where agents can handle multiple variables. At first, we have presented different exiting methods and algorithms. The methods are based on transforming local problems, so as there is exactly one variable per agent, to enable the DisCSP algorithms to solve them.

Then we have proposed the Minimal Perturbation based Asynchronous BackTracking MP-ABT, an extension of the DisCSP algorithm ABT, so as it supports complex local problems. The strength of the algorithm is realized in the treatment of local problems as Minimal Perturbation Problems. The objective is to minimize local variables' perturbation and so message exchanging reduction. Experimental results have proven the effectiveness of the proposed algorithm.

In the same part, we have designed the Selective Sorting & Smart Nogood SS&SN method. A general method to integrate into DisCSP algorithms to deal with complex DisCSP. It is based on sorting the compiled domain of each complex agent, according to four functions. The integration of the method into the DisCSPs algorithms ABT and AFC-ng has given very important results, especially when problems are solvable.

In the third part, we have designed our new formalism Ethical DisCSP E-DisCSP, to tackle unethical agents, especially as their presence causes catastrophic results, as we have proven. We have proposed methods to detect the unethical agents, and to react against those agents, so as the DisCSP resolution is as smooth as possible. The assessments have demonstrated how much we have been able to save the DisCSP resolution process.

Once we have updated the state of the art of DisCSPs, DisCSPs with complex local problems, as well as Ethical DisCSPs, we have started, in the forth part, to grasp the Smart Grid real-world problem and try to gradually apply the formalisms, and methods we have deliberated. Up to now, we have treated it by the CSP and the COP formalisms.

Perspectives

In future research, many approaches and methods can be either ameliorated or created:

- The JChoc platform can be improved by; adding other DisCSP, Distributed COP, and Dynamic DisCSP algorithms; preparing experimentations' interface; and improving the user front end.

- The selective sorting of the complex agents' domain we did is effective especially when problems are solvable, and when there are some consistent values in the domain. So in future work, we are going to improve the sorting so as it is done if only there are at least two consistent values.

- In E-DisCSP, the detection process and actions are realized against incorrect agents doing general abnormal activities, like lying of neglecting some messages. In future works, we expect to add ethical rules and ensure such rules' respect.
- In applications, we are going to extend the Smart Grid modelization to the DisCSP, DCOP, and also the E-DisCSP.

Bibliography

Ahat, Murat et al. (2013). "Smart grid and optimization". In: *American Journal of Operations Research* 3.1A, pp. 196–206.

Alpaydin, Ethem (2020). *Introduction to machine learning*. MIT press.

Amadini, Roberto, Maurizio Gabbrielli, and Jacopo Mauro (2014). "Portfolio approaches for constraint optimization problems". In: *International Conference on Learning and Intelligent Optimization*. Springer, pp. 21–35.

Anderson, Michael and Susan Leigh Anderson (2011). *Machine ethics*. Cambridge University Press.

Anderson, Michael, Susan Leigh Anderson, and Chris Armen (2004). "Towards machine ethics". In: *Proceedings of AAAI 2004 Workshop on Agent Organizations: Theory and Practice*.

Andrews, Gregory R (1991). *Concurrent programming: principles and practice*. Benjamin/Cummings Publishing Company San Francisco.

Apt, Krzysztof R (1990). "Logic Programming." In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)* 1990, pp. 493–574.

Arkin, Ronald C (2016). "Ethics and autonomous systems: Perils and promises [point of view]". In: *Proceedings of the IEEE* 104.10, pp. 1779–1781.

Bacchus, Fahiem and Adam Grove (1995). "On the forward checking algorithm". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 292–309.

Barták, Roman, T Muller, and Hana Rudová (2003). "Minimal perturbation problem-a formal view". In: *Neural Network World* 13.5, pp. 501–512.

Béjar, Ramón et al. (2005). "Sensor networks and distributed CSP: communication, computation and complexity". In: *Artificial Intelligence* 161.1-2, pp. 117–147.

Belloni, Aline et al. (2015). "Dealing with ethical conflicts in autonomous agents and multi-agent systems". In: *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 1–7.

Bessiere, Christian (1991). "Arc-Consistency in Dynamic Constraint Satisfaction Problems." In: *AAAI, Anaheim, California*. Vol. 91, pp. 221–226.

— (1994). "Arc-consistency and arc-consistency again". In: *Artificial intelligence* 65.1, pp. 179–190.

Bessière, Christian et al. (2005). "Asynchronous backtracking without adding links: a new member in the ABT family". In: *Artificial Intelligence* 161.1-2, pp. 7–24.

Bostrom, Nick and Eliezer Yudkowsky (2014). "The ethics of artificial intelligence". In: *The Cambridge handbook of artificial intelligence* 1, pp. 316–334.

Boykov, Yuri and Vladimir Kolmogorov (2004). "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 9, pp. 1124–1137.

Brilliant, Susan S and Timothy R Wiseman (1996). "The first programming paradigm and language dilemma". In: *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pp. 338–342.

Burke, David A and Kenneth N Brown (2006). "Applying interchangeability to complex local problems in distributed constraint reasoning". In: *Workshop on Distributed Constraint Reasoning (AAMAS 06), Hakodate, Japan*, pp. 1–15.

Chong, Edwin KP and Stanislaw H Zak (1996). "An introduction to optimization, John Wiley & Sons". In: *Inc., New York*, pp. 73–90.

Chu, Paul C and John E Beasley (1998). "A genetic algorithm for the multidimensional knapsack problem". In: *Journal of heuristics* 4.1, pp. 63–86.

Clausen, Jens (1999). "Branch and bound algorithms-principles and examples". In: *Department of Computer Science, University of Copenhagen*, pp. 1–30.

Cointe, Nicolas, Grégory Bonnet, and Olivier Boissier (2016). "Ethical Judgment of Agents' Behaviors in Multi-Agent Systems." In: *AAMAS*, pp. 1106–1114.

Cook, Rebecca J, Bernard M Dickens, and Mahmoud F Fathalla (2003). *Reproductive health and human rights: integrating medicine, ethics, and law*. Clarendon Press.

Dechter, Rina and Daniel Frost (1999). *Backtracking algorithms for constraint satisfaction problems*. Tech. rep. Technical Report.

— (2002). "Backjump-based backtracking for constraint satisfaction problems". In: *Artificial Intelligence* 136.2, pp. 147–188.

Diri, Driss (2009). "Élaboration d'un système de maintien de vérité: une approche orientée objet". PhD thesis. Université Laval.

Doyle, Jon (1979). "A truth maintenance system". In: *Artificial intelligence* 12.3, pp. 231–272.

Dyer, ME, N Kayal, and J Walker (1984). "A branch and bound algorithm for solving the multiple-choice knapsack problem". In: *Journal of computational and applied mathematics* 11.2, pp. 231–249.

El Graoui, El Mehdi, Imade Benelallam, and El-Houssine Bouyakhf (2016). "A commentary on "Hybrid search for minimal perturbation in Dynamic CSPs"". In: *Constraints* 21.2, pp. 349–354.

Eyer, Jim and Garth Corey (2010). "Energy storage for the electricity grid: Benefits and market potential assessment guide". In: *Sandia National Laboratories* 20.10, p. 5.

Ezzahir, Redouane et al. (2007a). "Compilation formulation for asynchronous backtracking with complex local problems". In: *2007 International Symposium on Computational Intelligence and Intelligent Informatics*. IEEE, pp. 205–211.

Ezzahir, Redouane et al. (2007b). "DisChoco: A platform for distributed constraint programming". In: *Proceedings of the IJCAI*. Vol. 7, pp. 16–21.

Farhangi, Hassan (2009). "The path of the smart grid". In: *IEEE power and energy magazine* 8.1, pp. 18–28.

Fioretto, Ferdinando, William Yeoh, and Enrico Pontelli (2016). "Multi-variable agents decomposition for DCOPs". In: *Thirtieth AAAI Conference on Artificial Intelligence*.

Florian, Michael and Donald Hearn (1995). "Network equilibrium models and algorithms". In: *Handbooks in Operations Research and Management Science* 8, pp. 485–550.

Forbus, Kenneth D and Johan De Kleer (1993). *Building problem solvers*. Vol. 1. MIT press.

Freuder, Eugene C (1995). "Using inference to reduce arc consistency computation". In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95). 23, 28, 33*. Morgan Kaufmann Publishers Inc., pp. 592–598.

Garrido, Leonardo and Katia Sycara (1996). "Multi-agent meeting scheduling: Preliminary experimental results". In: *Proceedings of the Second International Conference on Multiagent Systems*, pp. 95–102.

Gualandi, Stefano and Federico Malucelli (2012). "Exact solution of graph coloring problems via constraint programming and column generation". In: *INFORMS Journal on Computing* 24.1, pp. 81–100.

Guérard, Guillaume (2014). "Optimisation de la diffusion de l'énergie dans les smarts-grids". PhD thesis. Versailles-St Quentin en Yvelines.

Gungor, Vehbi C et al. (2011). "Smart grid technologies: Communication technologies and standards". In: *IEEE transactions on Industrial informatics* 7.4, pp. 529–539.

Hebrard, Emmanuel (2008). "Mistral, a constraint satisfaction library". In: *Proceedings of the Third International CSP Solver Competition* 3.3.

Hirayama, Katsutoshi and Makoto Yokoo (1997). "Distributed partial constraint satisfaction problem". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 222–236.

— (2000). "The effect of nogood learning in distributed constraint satisfaction". In: *20th International Conference on Distributed Computing Systems, Taipei, Taiwan*. IEEE, pp. 169–177.

— (2002). "Local search for distributed SAT with complex local problems". In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, pp. 1199–1206.

— (2005). "The distributed breakout algorithms". In: *Artificial Intelligence* 161.1-2, pp. 89–115.

Jung, Hyuckchul, Milind Tambe, and Shriniwas Kulkarni (2001). "Argumentation as distributed constraint satisfaction: Applications and results". In: *Proceedings of the fifth international conference on Autonomous agents*, pp. 324–331.

Jussien, Narendra, Guillaume Rochart, and Xavier Lorca (2008). "Choco: an open source java constraint programming library". In:

Kolesar, Peter J (1967). "A branch and bound algorithm for the knapsack problem". In: *Management science* 13.9, pp. 723–735.

Kumar, Vipin (1992). "Algorithms for constraint-satisfaction problems: A survey". In: *AI magazine* 13.1, pp. 32–32.

La Mura, Pierfrancesco (2000). "Game networks". In: *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence, San Francisco, CA*. Morgan Kaufmann Publishers Inc., pp. 335–342.

Larrosa, Javier and Thomas Schiex (2004). "Solving weighted CSP by maintaining arc consistency". In: *Artificial Intelligence* 159.1-2, pp. 1–26.

Le Berre, Daniel and Anne Parrain (2010). "The Sat4j library, release 2.2". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.2-3, pp. 59–64.

Léauté, Thomas and Boi Faltings (2011). "Coordinating logistics operations with privacy guarantees". In: *Twenty-Second International Joint Conference on Artificial Intelligence*.

Léauté, Thomas, Brammert Ottens, and Radoslaw Szymanek (2009). "FRODO 2.0: An open-source framework for distributed constraint optimization". In: *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*. CONF, pp. 160–164.

Lemaitre, Michel and Gérard Verfaillie (1997). "Daily management of an earth observation satellite: comparison of ILOG solver with dedicated algorithms for valued constraint satisfaction problems". In:

Lemlouma, Tayeb and Abdelmadjid Boudina (2000). "Programmation logique avec contraintes (CLP) Etude et application au puzzle:'Send plus More equal to Money'". In: *Revue d'Informatique Scientifique et Technique (RIST)* 10.1-2, pp. 63–85.

Light, Ben and Kathy McGrath (2010). "Ethics and social networking sites: a disclosive analysis of Facebook". In: *Information Technology & People*.

Liu, Jiming, Han Jing, and Yuan Yan Tang (2002). "Multi-agent oriented constraint satisfaction". In: *Artificial Intelligence* 136.1, pp. 101–144.

Liu, Jyi-Shane and Katia P Sycara (1995). "Exploiting Problem Structure for Distributed Constraint Optimization." In: *ICMAS*. Vol. 95, pp. 246–254.

Lutati, Benny et al. (2014). "Agentzero: A framework for simulating and evaluating multi-agent algorithms". In: *Agent-Oriented Software Engineering*. Springer, pp. 309–327.

Mackworth, Alan K (1977). "Consistency in networks of relations". In: *Artificial Intelligence* 8.1, pp. 99–118.

Magaji, Amina Sambo, Ines Arana, and Hatem Ahriz (2014). "Local search for discsps with complex local problems". In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. Vol. 3. IEEE, pp. 56–63.

Mantere, Timo and Janne Koljonen (2006). "Solving and rating sudoku puzzles with genetic algorithms". In: *New Developments in Artificial Intelligence and the Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP*, pp. 86–92.

Marah, Rim and Abdelaaziz El Hibaoui (2018). "Algorithms for smart grid management". In: *Sustainable cities and society* 38, pp. 627–635.

Martins, Joao P (1990). "The truth, the whole truth, and nothing but the truth". In: *AI Magazine* 11.4, p. 7.

McCarthy, John (1987). "Generality in artificial intelligence". In: *Communications of the ACM* 30.12, pp. 1030–1035.

McCarthy, Natasha (2009). "Autonomous Systems: social, legal and ethical issues". In:

McMahan, Jeff (2013). *Killing by remote control: the ethics of an unmanned military*. Oxford University Press.

Meisels, Amnon (2008). "Constraints Optimization Problems-COPs". In: *Distributed Search by Constrained Agents*. Springer, pp. 19–26.

Meisels, Amnon and Oz Lavee (2004). "Using additional information in DisCSP search". In: *Proc. 5th workshop on distributed constraints reasoning, DCR*. Vol. 4.

Meisels, Amnon and Roie Zivan (2007). "Asynchronous forward-checking for DisCSPs". In: *Constraints* 12.1, pp. 131–150.

Merchez, Sylvain, Christophe Lecoutre, and Frédéric Boussemart (2001). "Abscon: A prototype to solve csps with abstraction". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 730–744.

Michie, Donald, David J Spiegelhalter, CC Taylor, et al. (1994). "Machine learning". In: *Neural and Statistical Classification* 13.1994, pp. 1–298.

Mitchell, Tom M et al. (1997). *Machine learning*.

Mohr, Roger and Thomas C Henderson (1986). "Arc and path consistency revisited". In: *Artificial intelligence* 28.2, pp. 225–233.

Moor, James H (2006). "The nature, importance, and difficulty of machine ethics". In: *IEEE intelligent systems* 21.4, pp. 18–21.

Myers, Brad A, Dario A Giuse, and Brad Vander Zanden (1992). "Declarative programming in a prototype-instance system: object-oriented programming without writing methods". In: *ACM Sigplan Notices* 27.10, pp. 184–200.

Myerson, Roger B (2013). *Game theory*. Harvard university press, pp. 1–5.

Nadel, Bernard A (1990a). "Representation selection for constraint satisfaction: A case study using n-queens". In: *IEEE Intelligent Systems* 3, pp. 16–23.

— (1990b). *Some applications of the constraint satisfaction problem*. Wayne State University, Department of Computer Science.

Nareyek, Alexander (2001). "The "Send More Money" Problem". In: *Constraint-Based Agents: An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*, pp. 139–139.

Negnevitsky, Michael (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson education, pp. 30–31.

Norouzi, Mohammad, David J Fleet, and Russ R Salakhutdinov (2012). "Hamming distance metric learning". In: *Advances in neural information processing systems*, pp. 1061–1069.

Omomowo, Bayo, Inés Arana, and Hatem Ahriz (2008). "DynABT: Dynamic asynchronous backtracking for dynamic disCSPs". In: *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer, pp. 285–296.

Orlov, Michael and Amnon Meisels (2006). "Descending requirements search for DisCSPs". In: *Proceedings of Distributed Constraint Satisfaction Workshop at ECAI-2006*.

Osborne, Martin J and Ariel Rubinstein (1994). *A course in game theory*. MIT press, pp. 1–8.

Petcu, Adrian (2006). *Frodo: A framework for open/distributed constraint optimization*. Tech. rep.

Podelski, Andreas (1994). "Constraint Programming: Basics and Trends 1994 Châtillon Spring School Châtillon-sur-Seine, France, May 16–20, 1994 Selected Papers". In: *Conference proceedings TCS School*. Springer, p. 49.

Reade, Chris (1989). *Elements of functional programming*. Vol. 248. Addison-Wesley Reading, MA.

Rentsch, Tim (1982). "Object oriented programming". In: *ACM Sigplan Notices* 17.9, pp. 51–57.

Richards, Neil M and Jonathan H King (2014). "Big data ethics". In: *Wake Forest L. Rev.* 49, p. 393.

Robbins, Russell W and William A Wallace (2007). "Decision support for ethical problem solving: A multi-agent approach". In: *Decision Support Systems* 43.4, pp. 1571–1587.

Rollón, Emma and Javier Larrosa (2006). "Bucket elimination for multiobjective optimization problems". In: *Journal of Heuristics* 12.4-5, pp. 307–328.

Ross, Keith W and Danny HK Tsang (1989). "The stochastic knapsack problem". In: *IEEE Transactions on communications* 37.7, pp. 740–747.

Salavati, Soroor, Sahar Hajjarzadeh, and Masoud Mazloom (2009). "An Optimized Method for Solving Zebra Puzzle". In: *2009 Second International Conference on Computer and Electrical Engineering*. Vol. 1. IEEE, pp. 448–451.

Schiex, Thomas, Helene Fargier, Gerard Verfaillie, et al. (1995). "Valued constraint satisfaction problems: Hard and easy problems". In: *IJCAI (1)* 95, pp. 631–639.

Schulte, Christian, Mikael Lagerkvist, and Guido Tack (2006). *Gecode*.

Sen, Sandip and Edmund H Durfee (1995). "Unsupervised Surrogate Agents and Search Bias Change in Flexible Distributed Scheduling." In: *ICMAS*, pp. 336–343.

Shiloach, Yossi and Uzi Vishkin (1982). "An O (n2log n) parallel max-flow algorithm". In: *Journal of Algorithms* 3.2, pp. 128–146.

Simonis, Helmut (1995). "The CHIP system and its applications". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 643–646.

Sparrow, Robert (2007). "Killer robots". In: *Journal of applied philosophy* 24.1, pp. 62–77.

Stuckey, Peter J et al. (2014). "The minizinc challenge 2008–2013". In: *AI Magazine* 35.2, pp. 55–60.

Tsang, Edward (1995). "Foundations of constraint satisfaction". In: *Journal of the Operational Research Society* 46.5, pp. 666–666.

Tsuruta, Takuo and Toramatsu Shintani (2000). "Scheduling meetings using distributed valued constraint satisfaction algorithm". In: *ECAI*, pp. 383–387.

Verfaillie, Gérard, Michel Lemaître, and Thomas Schiex (1996). "Russian doll search for solving constraint optimization problems". In: *AAAI/IAAI, Vol. 1*, pp. 181–187.

Vessey, Iris and Ron Weber (1984). "Research on structured programming: An empiricist's evaluation". In: *IEEE Transactions on Software Engineering* 4, pp. 397–407.

Wahbi, Mohamed et al. (2011). "DisChoco 2: A platform for distributed constraint reasoning". In: *Proceedings of DCR* 11, pp. 112–121.

— (2013). "Nogood-based asynchronous forward checking algorithms". In: *Constraints* 18.3, pp. 404–433.

Weigel, Rainer and Boi Faltings (1999). "Compiling constraint satisfaction problems". In: *Artificial Intelligence* 115.2, pp. 257–287.

Xi, Hongwei (2000). "Imperative programming with dependent types". In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*. IEEE, pp. 375–387.

Xiong, Hejing et al. (2010). "A scheduling model for temporally constrained grid workflow for distributed simulation system on grid". In: *Kybernetes*.

Yokoo, Makoto (1995). "Asynchronous Weak-Commitment Search for Solving Large-Scale Distributed Constraint Satisfaction Problems." In: *ICMAS*, p. 467.

— (2001). "Distributed Constraint Satisfaction Problem". In: *Distributed Constraint Satisfaction*. Springer, pp. 47–54.

Yokoo, Makoto et al. (1992). "Distributed constraint satisfaction for formalizing distributed problem solving". In: *Proceedings of the 12th International Conference on Distributed Computing Systems, IEEE*. IEEE, pp. 614–621.

Yokoo, Makoto et al. (1998). "The distributed constraint satisfaction problem: Formalization and algorithms". In: *IEEE Transactions on Knowledge and Data Engineering* 10.5, pp. 673–685.

Zivan, Roie, Alon Grubshtein, and Amnon Meisels (2011). "Hybrid search for minimal perturbation in Dynamic CSPs". In: *Constraints* 16.3, pp. 228–249.

Zwitter, Andrej (2014). "Big data ethics". In: *Big Data & Society* 1.2, p. 2053951714559253.

Résumé:

Les problèmes de satisfaction des contraintes distribuées (DisCSP) représentent de nombreux problèmes distribués combinatoires. Il a prouvé sa puissance dans l'interprétation et la résolution de nombreux problèmes réels. Cette thèse présente une extension de l'état de l'art du domaine du DisCSP. Les contributions apportées sont regroupées en quatre parties principales. Dans la première partie, nous avons étudié les algorithmes DisCSP existants, préparé la plateforme DisCSP dynamique JChoc, et proposé de nouveaux algorithmes DisCSP basés sur la fusion des algorithmes existants.

La deuxième partie entame les DisCSPs avec des problèmes locaux complexes. Dans cette partie, nous avons proposé le Minimal Perturbation based Asynchronous BackTracking (MP-ABT), un algorithme considérant les problèmes locaux du DisCSP comme des problèmes de minimisation des perturbation (MPP), le Selective Sorting and Smart Nogood (SS&SN), une méthode générale applicable sur chaque algorithme DisCSP afin qu'il puisse gérer les DisCSP avec les problèmes locaux complexes, et SS&SN based ABT (ABT-SS&SN), SS&SN based AFC-ng (AFC-ng-SS&SN), l'application du SS&SN sur les algorithmes DisCSP ABT et AFC-ng.

La troisième partie s'attaque à la création d'un nouveau formalisme nommé Ethical DisCSP (E-DisCSP). Nous avons intégré le concept d'éthique dans les DisCSPs. Nous avons traité la gravité de la présence d'un agent non éthique dans une résolution DisCSP, proposé quelques méthodes pour détecter un tel agent, et nous avons suggéré des actions à appliquer afin que la résolution DisCSP soit aussi normale que possible. Tandis que la quatrième partie est une initiation à l'application de ce que nous avons réalisé sur un problème réaliste et récent qui est le Smart Grid.

***Mots clés**: L'intelligence artificielle, les problèmes de satisfaction des contraintes distribués (DisCSP), les DisCSPs avec les problèmes locaux complexes, l'éthique,* les DisCSPs éthiques (E-DisCSP), Smart Grid.

Abstract:

Distributed Constraint Satisfaction Problem (DisCSP) is a mathematical formalism solving many combinatorial distributed problems. It has proven its effectiveness in representing and solving many real problems. This thesis presents an extension of the state of the art of the DisCSP domain. Therefore, the made contributions are regrouped into four main parts. In the first part, we have studied the exiting DisCSP algorithms, prepared the dynamic DisCSP platform JChoc, and proposed new DisCSP algorithms based on the existing algorithms' merge.

The second part broaches DisCSPs with Complex local problems. We proposed the Minimal Perturbation based Asynchronous Backtracking (MP-ABT), an algorithm treating the DisCSP local problems as Minimal Perturbation Problems (MPP), the Selective Sorting and Smart Nogood (SS&SN), a general method applicable to every DisCSP algorithm so as it can solve DisCSPs with Complex local problems, and SS&SN based ABT (ABT-SS&SN) and SS&SN based AFC-ng (AFC-ng-SS&SN), the application of the SS&SN method on the DisCSP algorithms ABT and AFC-ng.

The third part approaches the creation of a new formalism named Ethical DisCSP (E-DisCSP). We integrated the ethics concept into DisCSPs. We dealt the gravity of the presence of an unethical agent into a DisCSP resolution, we proposed some methods to detect such an agent, and we suggested actions to apply on so as the DisCSP resolution is as smooth as possible. Whilst the fourth part is an initiation to the application of what we have realized on a realistic and recent problem which is the Smart Grid.

***Keywords**: Artificial Intelligence, Distributed Constraint Satisfaction Problems (DisCSP), DisCSPs with complex local problems, ethics, Ethical DisCSPs (E-DisCSP), Smart Grid.*

Année Universitaire : 2019/2020