Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

Mohamed Wahbi

UNIVERSITÉ MONTPELLIER 2
Sciences et Techniques du Languedoc
FRANCE

UNIVERSITÉ MOHAMMED V - AGDAL
Faculté des Sciences du Rabat
MAROC

Ph.D Thesis

présentée pour obtenir le diplôme de Doctorat en Informatique
de l'Université Montpellier 2 & l'Université Mohammed V-Agdal

par

Mohamed Wahbi

SPÉCIALITÉ : **Informatique**
École Doctorale Information, Structures, Systèmes- France &
Le Centre d'Etudes Doctorales en Sciences et Technologies de Rabat- Maroc

---

Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

---

Soutenue le 03 Juillet 2012, devant le jury composé de :

President
Mme. Awatef SAYAH, PES ..... Université Mohammed V-Agdal, Maroc

Reviewers
Mr. Pedro MESEGUER, Directeur de recherche ..... IIIA, Barcelona, Espagne
Mr. Mustapha BELAISSAOUI, Professeur Habilité ..... l'ENCG, Université Hassan I, Maroc

Examinator
Mr. Rémi COLETTA, Maitre de conférence ..... LIRMM, Université Montpellier 2, France

Supervisors
Mr. Christian BESSIERE, Directeur de recherche . LIRMM, Université Montpellier 2, France
Mr. El Houssine BOUYAKHF, PES..... Université Mohammed V-Agdal, Maroc

To my family

Acknowledgements

The research work presented in this thesis has been performed in the Laboratoire d'Informatique Mathématiques appliquées Intelligence Artificielle et Reconnaissance de Formes (LIMIARF), Faculty of Science, University Mohammed V-Agdal, Rabat, Morocco and the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), University Montpellier 2, France.

This thesis has been done in collaboration between University Mohammed V-Agdal, Morocco and University Montpellier 2, France under the financial support of the scholarship of the programme Averroés funded by the European Commission within the framework of Erasmus Mundus.

First and foremost, it is with immense gratitude that I acknowledge all the support, advice, and guidance of my supervisors, Professor El-Houssine Bouyakhf and Dr. Christian Bessiere. It was a real pleasure to work with them. Their truly scientist intuition has made them as a source of ideas and passions in science, which exceptionally inspire and enrich my growth as a student, a researcher and a scientist want to be. I want to thank them especially for letting me wide autonomy while providing appropriate advice. I am indebted to them more than they know and hope to keep up our collaboration in the future.

I gratefully acknowledge Professor Awatef Sayah (Faculty of sciences, University Mohammed V-Agdal, Morocco) for accepting to preside the jury of my dissertation. I am most grateful to my reviewers Professor Pedro Meseguer (Scientific Researcher, the Artificial Intelligence Research Institute (IIIA), Barcelona, Spain) and Professor Mustapha Belaissaoui (Professeur Habilité, ENCG, University Hassan I, Morocco) for their constructive comments on this thesis. I am thankful that in the midst of all their activities, they accepted to review my thesis. I would like to record my gratitude to Professor Rémi Coletta, (Maître de conférence, University Montpellier 2, France) for his thorough examination of the thesis.

Many of the works published during this thesis have been done in collaboration with so highly motivated, smart, enthusiastic, and passionate coauthors. I want to thank them for their teamwork, talent, hard work and devotion. I cannot thank my coauthors without giving my special gratefulness to Professor Redouane Ezzahir and Doctor Younes Mechqrane.

I thank the great staffs of the LIRMM and LIMIARF Laboratories for the use of facilities, consultations and moral support. The LIRMM has provided the support and equipment I have needed to produce and complete my thesis. I also want to thank my colleagues at the LIRMM and LIMIARF Laboratories for the joyful and pleasant working environment.

Especially, I would like to thank the members of the Coconut/LIRMM and IA/LIMIARF teams. I acknowledge Amine B., Fabien, Eric, Imade, Saida, Fred, Philippe, Brahim and Jaouad.

In my daily work I have been blessed with a friendly and cheerful group of fellow students. I would like to particularly thank Hajer, Younes, Mohamed, Kamel, Nawfal, Mohammed, Nabil Z., Farid, Azhar, Kaouthar, Samir, Nabil Kh., Amine M., and Hassan. It is a pleasure to express my gratitude wholeheartedly to Baslam's family for their kind hospitality during my stay in Montpellier.

Further, I am also very thankful to the professors of the department of Computer Science, University Montpellier 2, with whom I have been involved as a temporary assistant professor (Attaché Temporaire d'Enseignement et de Recherche - ATER) for providing an excellent environment to teach and develop new pedagogical techniques. I convey special acknowledgment to Professor Marianne Huchard.

Most importantly, words alone cannot express the thanks I owe to my family for believing and loving me, especially my mother who has always filled my life with generous love, and unconditional support and prayers. My thanks go also to my lovely sister and brother, my uncles and antes and all my family for their endless moral support throughout my career. To them I dedicate this thesis. Last but not the least, the one above all of us, the omnipresent God, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up and throw in the towel, thank you so much Dear Lord.

Finally, I would like to thank everybody who was important to the successful realization of thesis, as well as expressing my apology that I could not mention personally one by one.

Montpellier, July 3^{rd}, 2012 Mohamed Wahbi

Abstract

Distributed Constraint Satisfaction Problems (DisCSP) is a general framework for solving distributed problems. DisCSP have a wide range of applications in multi-agent coordination. In this thesis, we extend the state of the art in solving the DisCSPs by proposing several algorithms. Firstly, we propose the Nogood-Based Asynchronous Forward Checking (AFC-ng), an algorithm based on Asynchronous Forward Checking (AFC). However, instead of using the shortest inconsistent partial assignments, AFC-ng uses nogoods as justifications of value removals. Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. Then, we propose the Asynchronous Forward-Checking Tree (AFC-tree). In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. Next, we propose to maintain arc consistency asynchronously on the future agents instead of only maintaining forward checking. Two new synchronous search algorithms that maintain arc consistency asynchronously (MACA) are presented. After that, we developed the Agile Asynchronous Backtracking (Agile-ABT), an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing before the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Next, we present a corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). Finally, the new version of the DisChoco open-source platform for solving distributed constraint reasoning problems is described. The new version is a complete redesign of the DisChoco platform. DisChoco 2.0 is an open source Java library which aims at implementing distributed constraint reasoning algorithms.

Keywords: *Distributed Artificial Intelligence, Distributed Constraint Satisfaction (DisCSP), Distributed Solving, Maintaining Arc Consistency, Reordering, DisChoco.*

Résumé

Les problèmes de satisfaction de contraintes distribués (DisCSP) permettent de formaliser divers problèmes qui se situent dans l'intelligence artificielle distribuée. Ces problèmes consistent à trouver une combinaison cohérente des actions de plusieurs agents. Durant cette thèse nous avons apporté plusieurs contributions dans le cadre des DisCSPs. Premièrement, nous avons proposé le Nogood-Based Asynchronous Forward-Checking (AFC-ng). Dans AFC-ng, les agents utilisent les nogoods pour justifier chaque suppression d'une valeur du domaine de chaque variable. Outre l'utilisation des nogoods, plusieurs backtracks simultanés venant de différents agents vers différentes destinations sont autorisés. En deuxième lieu, nous exploitons les caractéristiques intrinsèques du réseau de contraintes pour exécuter plusieurs processus de recherche AFC-ng d'une manière asynchrone à travers chaque branche du pseudo-arborescence obtenu à partir du graphe de contraintes dans l'algorithme Asynchronous Forward-Checking Tree (AFC-tree). Puis, nous proposons deux nouveaux algorithmes de recherche synchrones basés sur le même mécanisme que notre AFC-ng. Cependant, au lieu de maintenir le forward checking sur les agents non encore instanciés, nous proposons de maintenir la consistance d'arc. Ensuite, nous proposons Agile Asynchronous Backtracking (Agile-ABT), un algorithme de changement d'ordre asynchrone qui s'affranchit des restrictions habituelles des algorithmes de backtracking asynchrone. Puis, nous avons proposé une nouvelle méthode correcte pour comparer les ordres dans ABT_DO-Retro. Cette méthode détermine l'ordre le plus pertinent en comparant les indices des agents dès que les compteurs d'une position donnée dans le timestamp sont égaux. Finalement, nous présentons une nouvelle version entièrement restructurée de la plateforme DisChoco pour résoudre les problèmes de satisfaction et d'optimisation de contraintes distribués.

Mots clefs : *L'intelligence artificielle distribuée, les problèmes de satisfaction de contraintes distribués (DisCSP), la résolution distribuée, la maintenance de la consistance d'arc, les heuristiques ordonnancement, DisChoco.*

Contents

Introduction

Constraint programming is an area in computer science that has gained increasing interest in the last four recent decades. Constraint programming is based on its powerful framework named *Constraint Satisfaction Problem* (CSP). A constraint satisfaction problem is a general framework that can formalize many real world combinatorial problems. Various problems in artificial intelligence can be naturally modeled as CSPs. Therefore, the CSP paradigm has been widely used for solving such problems. Examples of these problems can inherent from various areas related to resource allocation, scheduling, logistics and planning. Solving a constraint satisfaction problem (CSP) consists in looking for solutions to a constraint network, that is, a set of assignments of values to variables that satisfy the constraints of the problem. These constraints represent restrictions on values combinations allowed for constrained variables.

Numerous powerful algorithms were designed for solving constraint satisfaction problems. Typical systematic search algorithms try to construct a solution to a CSP by incrementally instantiating the variables of the problem. However, proving the existence of solutions or finding these solutions in CSP are NP-complete tasks. Thus, many heuristics were developed to improve the efficiency of search algorithms.

Sensor networks [**Jung *et al.*, 2001**; **Béjar *et al.*, 2005**], military unmanned aerial vehicles teams [**Jung *et al.*, 2001**], distributed scheduling problems [**Wallace and Freuder, 2002**; **Maheswaran *et al.*, 2004**], distributed resource allocation problems [**Petcu and Faltings, 2004**], log-based reconciliation [**Chong and Hamadi, 2006**], Distributed Vehicle Routing Problems [**Léauté and Faltings, 2011**], etc. are real applications of a distributed nature, that is, knowledge is distributed among several physical distributed entities. These applications can be naturally modeled and solved by a CSP process once the knowledge about the whole problem is delivered to a centralized solver. However, in such applications, gathering the whole knowledge into a centralized solver is undesirable. In general, this restriction is mainly due to privacy and/or security requirements: constraints or possible values may be strategic information that should not be revealed to others agents that can be seen as competitors. The cost or the inability of translating all information to a single format may be another reason. In addition, a distributed system provides fault tolerance, which means that if some agents disconnect, a solution might be available for the connected part. Thereby, a distributed model allowing a decentralized solving process is more adequate. The *Distributed Constraint Satisfaction Problem* (DisCSP) has such properties.

A distributed constraint satisfaction problem (DisCSP) is composed of a group of autonomous agents, where each agent has control of some elements of information about the whole problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. In order to solve a

DisCSP, agents must assign values to their variables so that all constraints are satisfied. Hence, agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with constraints between agents [Yokoo *et al.*, 1998; Yokoo, 2000a]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them to check consistency of their proposed assignments against constraints that contain variables that belong to others agents.

In solving DisCSPs, agents exchange messages about the variable assignments and conflicts of constraints. Several distributed algorithms for solving DisCSPs have been designed in the last two decades. They can be divided into two main groups: asynchronous and synchronous algorithms. The first category are algorithms in which the agents assign values to their variables in a synchronous, sequential way. The second category are algorithms in which the process of proposing values to the variables and exchanging these proposals is performed asynchronously between the agents. In the former category, agents do not have to wait for decisions of others, whereas, in general only one agent has the privilege of making a decision in the synchronous algorithms.

Contributions

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

Nogood-Based Asynchronous Forward Checking (AFC-ng) is an asynchronous algorithm based on Asynchronous Forward Checking (AFC) for solving DisCSPs. Instead of using the shortest inconsistent partial assignments AFC-ng uses nogoods as justifications of value removals. Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. Thanks to the timestamps integrated in the CPAs, the strongest CPA coming from the highest level in the agent ordering will eventually dominate all others. Interestingly, the search process with the strongest CPA will benefit from the computational effort done by the (killed) lower level processes. This is done by taking advantage from nogoods recorded when processing these lower level processes.

Asynchronous Forward-Checking Tree (AFC-tree) The main feature of the AFC-tree algorithm is using different agents to search non-intersecting parts of the search space concurrently. In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. The pseudo-tree ordering is built in a preprocessing step. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. The agents that are brothers

are committed to concurrently find the partial solutions of their variables. Therefore, AFC-tree exploits the potential speed-up of a parallel exploration in the processing of distributed problems.

Maintaining Arc Consistency Asynchronously (MACA) Instead of maintaining forward checking asynchronously on agents not yet instantiated, as is done in AFC-ng, we propose to maintain arc consistency asynchronously on these future agents. We propose two new synchronous search algorithms that *maintain arc consistency asynchronously* (MACA). The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages (***del***). Hence, whenever values are removed during a constraint propagation step, MACA-del agents notify others agents that may be affected by these removals, sending them a ***del*** message. The second algorithm, MACA-not, achieves arc consistency without any new type of message. We achieve this by storing all deletions performed by an agent on domains of its neighboring agents, and sending this information to these neighbors within the CPA message.

Agile Asynchronous Backtracking (Agile-ABT) is an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing before the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Furthermore, that agent receiving the backtrack message, called the backtracking target, is not necessarily the agent with the lowest priority within the conflicting agents in the current order. The principle of Agile-ABT is built on termination values exchanged by agents during search. A termination value is a tuple of positive integers attached to an order. Each positive integer in the tuple represents the expected current domain size of the agent in that position in the order. Orders are changed by agents without any global control so that the termination value decreases lexicographically as the search progresses. Since a domain size can never be negative, termination values cannot decrease indefinitely. An agent informs the others of a new order by sending them its new order and its new termination value. When an agent compares two contradictory orders, it keeps the order associated with the smallest termination value.

Corrigendum to "Min-domain retroactive ordering for Asynchronous Backtracking": A corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). We presented an example that shows how ABT_DO-Retro can enter an infinite loop following the natural understanding of the description given by the authors of ABT_DO-Retro. We describe the correct way for comparing time-stamps of orders. We give the proof that our method for comparing orders is correct.

DisChoco 2.0: is open-source platform for solving distributed constraint reasoning problems. The new version 2.0 is a complete redesign of the DisChoco platform. DisChoco

2.0 is not a distributed version of the centralized solver Choco [1], but it implements a model to solve distributed constraint networks with local complex problems (i.e., several variables per agent) by using Choco as local solver to each agent. The novel version we propose contains several interesting features: it is reliable and modular, it is easy to personalize and to extend, it is independent from the communication system and allows a deployment in a real distributed system as well as the simulation on a single Java Virtual Machine. DisChoco 2.0 is an open source Java library which aims at implementing distributed constraint reasoning algorithms from an abstract model of agent (already implemented in DisChoco). A single implementation of a distributed constraint reasoning algorithm can run as simulation on a single machine, or on a network of machines that are connected via the Internet or via a wireless ad-hoc network, or even on mobile phones compatible with J2ME.

Thesis Outline

Chapter 1 introduces the state of the art in the area of centralized and distributed constraint programming. We define the constraint satisfaction problem formalism (CSP) and present some academic and real examples of problems that can be modeled and solved by CSP. We then briefly present typical methods for solving centralized CSP. Next, we give preliminary definitions on the distributed constraint satisfaction problem paradigm (DisCSP). Afterwards we describe the main algorithms that have been developed in the literature to solve DisCSPs.

We present our first contribution, the Nogood-Based of Asynchronous Forward Checking (AFC-ng), in Chapter 2. Besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks to go from different agents to different destinations. We prove that AFC-ng only needs polynomial space. Correctness proofs of the AFC-ng are also given. We compare the performance of our algorithm against others well-known distributed algorithms for solving DisCSP. We present the results on random DisCSPs and instances from real benchmarks: sensor networks and distributed meeting scheduling.

In Chapter 3, we show how to extend our nogood-based Asynchronous Forward-Checking (AFC-ng) algorithm to the *Asynchronous Forward-Checking Tree (AFC-tree)* algorithm using a pseudo-tree arrangement of the constraint graph. To achieve this goal, agents are ordered a priory in a pseudo-tree such that agents in different branches of the tree do not share any constraint. AFC-tree does not address the process of ordering the agents in a pseudo-tree arrangement. Therefore, the construction of the pseudo-tree is done in a preprocessing step. We demonstrate the good properties of the Asynchronous Forward-Checking Tree. We provide a comparison of our AFC-tree to the AFC-ng on random DisCSPs and instances from real benchmarks: sensor networks and distributed meeting scheduling.

Chapter 4 presents the first attempt to maintain the arc consistency in the synchronous

1. http://choco.emn.fr/

search algorithm. Indeed, instead of using forward checking as a filtering property like AFC-ng we propose to maintain arc consistency asynchronously (MACA). Thus, we propose two new algorithms based on the same mechanism as AFC-ng that enforce arc consistency asynchronously. The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages. The second algorithm, MACA-not, achieves arc consistency without any new type of message. We provide a theoretical analysis and an experimental evaluation of the proposed approach.

Chapter 5 proposes Agile Asynchronous Backtracking algorithm (Agile-ABT), a search procedure that is able to change the ordering of agents more than previous approaches. This is done via the original notion of termination value, a vector of stamps labeling the new orders exchanged by agents during search. We first describe the concepts needed to select new orders that decrease the termination value. Next, we give the details of our algorithm and we show how agents can reorder themselves as much as they want as long as the termination value decreases as the search progresses. We also prove Agile-ABT in Chapter 5. An experimental evaluation is provided by the end of this chapter.

Chapter 6 provides a corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). We illustrate in this chapter an example that shows, if ABT_DO-Retro uses that protocol, how it can fall into an infinite loop. We present the correct method for comparing time-stamps and give the proof that our method for comparing orders is correct.

Finally, in Chapter 7, we describe our distributed constraint reasoning platform DisChoco 2.0. DisChoco is an open-source framework that provides a simple implementation of all these algorithms and obviously many other. DisChoco 2.0 then offers a complete tool for the research community for evaluating algorithms performance or being used for real applications.

Background

1

Contents

This chapter introduces the state of the art in the area of centralized and distributed constraint programming. In Section 1.1 we define the constraint satisfaction problem formalism (CSP) and present some academic and real examples of problems modeled and solved by CSPs. Typical methods for solving centralized CSP are presented in Section 1.2. Next, we give preliminary definitions on the distributed constraint satisfaction problem paradigm (DisCSP) in Section 1.3. The state of the art algorithms and heuristic for solving DisCSPs are provided in Section 1.4.

1.1 Centralized Constraint Satisfaction Problems (CSP)

Many real world combinatorial problems in artificial intelligence arising from areas related to resource allocation, scheduling, logistics and planning are solved using constraint programming. Constraint programming is based on its powerful framework named *constraint satisfaction problem* (CSP). A CSP is a general framework that involves a set of

variables and constraints. Each variable can assign a value from a domain of possible values. Constraints specify the allowed values for a set of variables. Hence, a large variety of applications can be naturally formulated as CSP. Examples of applications that have been successfully solved by constraint programming are: picture processing [**Montanari, 1974**], planning [**Stefik, 1981**], job-shop scheduling [**Fox *et al.*, 1982**], computational vision [**Mackworth, 1983**], machine design and manufacturing [**Frayman and Mittal, 1987**; **Nadel, 1990**], circuit analysis [**De Kleer and Sussman, 1980**], diagnosis [**Geffner and Pearl, 1987**], belief maintenance [**Dechter and Pearl, 1988**], automobile transmission design [**Nadel and Lin, 1991**], etc.

Solving a constraint satisfaction problem consists in looking for solutions to a constraint network, that is, a set of assignments of values to variables that satisfy the constraints of the problem. These constraints represent restrictions on values combinations allowed for constrained variables. Many powerful algorithms have been designed for solving constraint satisfaction problems. Typical systematic search algorithms try to construct a solution to a CSP by incrementally instantiating the variables of the problem.

There are two main classes of algorithms searching solutions for CSP, namely those of a look-back scheme and those of look-ahead scheme. The first category of search algorithms (look-back scheme) corresponds to search procedures checking the validity of the assignment of the current variable against the already assigned (past) variables. When the assignment of the current variable is inconsistent with assignments of past variables then an new value is tried. When no values remain then a past variable must be reassigned. Chronological backtracking (BT) [**Golomb and Baumert, 1965**], backjumping (BJ) [**Gaschnig, 1978**], graph-based backjumping (GBJ) [**Dechter, 1990**], conflict-directed backjumping (CBJ) [**Prosser, 1993**], and dynamic backtracking (DBT) [**Ginsberg, 1993**] are algorithms performing a look-back scheme.

The second category of search algorithms (look-ahead scheme) corresponds to search procedures that check forwards the assignment of the current variable. In look-ahead scheme, the not yet assigned (future) variables are made consistent, to some degree, with the assignment of the current variable. Forward checking (FC) [**Haralick and Elliott, 1980**] and maintaining arc consistency (MAC) [**Sabin and Freuder, 1994**] are algorithms that perform a look-ahead scheme.

Proving the existence of solutions or finding them in CSP are NP-complete tasks. Thereby, numerous *heuristics* were developed to improve the efficiency of solution methods. Though being various, these heuristics can be categorized into two kinds: variable ordering and value ordering heuristics. Variable ordering heuristics address the order in which the algorithm assigns the variables, whereas the value ordering heuristics establish an order on which values will be assigned to a selected variable. Many studies have been shown that the ordering of selecting variables and values dramatically affects the performance of search algorithms.

We present in the following an overview of typical methods for solving centralized CSP after defining formally a constraint satisfaction problem and given some examples of problems that can be encoded in CSPs.

1.1.1 Preliminaries

A *Constraint Satisfaction Problem - CSP* (or a constraint network [Montanari, 1974]) involves a finite set of variables, a finite set of domains determining the set of possibles values for a given variable and a finite set of constraints. Each constraint restricts the combination of values that a set of variables it involves can assign. A solution of a CSP is an assignment of values to all variables satisfying all the imposed constraints.

Definition 1.1 *A **constraint satisfaction problem (CSP)** or a **constraint network** was formally defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:*

- *$\mathcal{X}$ is a set of n **variables** $\{x_1, \ldots, x_n\}$.*
- *$\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of n current **domains**, where $D(x_i)$ is a finite set of possible values to which variable x_i may be assigned.*
- *$\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e **constraints** that specify the combinations of values (or* tuples*) allowed for the variables they involve. The variables involved in a constraint $c_k \in \mathcal{C}$ form its* scope *($\texttt{scope}(c_k) \subseteq \mathcal{X}$).*

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

The number of variables on the scope of a constraint $c_k \in \mathcal{C}$ is called the arity of the constraint c_k. Therefore, a constraint involving one variable (respectively two or n variables) is called unary (respectively *binary* or n-ary) constraint. In this thesis, we are concerned by binary constraint networks where we assume that all constraints are binary constraints (they involve two variables). A constraint in $\mathcal{C}$ between two variables x_i and x_j is then denoted by c_{ij}. c_{ij} is a subset of the Cartesian product of their domains (i.e., $c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). A direct result from this assumption is that the connectivity between the variables can be represented with a constraint graph G [Dechter, 1992].

Definition 1.2 *A binary constraint network can be represented by a **constraint graph** $G = \{X_G, E_G\}$, where vertexes represent the variables of the problem ($X_G = \mathcal{X}$) and edges (E_G) represent the constraints (i.e., $\{x_i, x_j\} \in E_G$ iff $c_{ij} \in \mathcal{C}$).*

Definition 1.3 *Two variables are **adjacent** iff they share a constraint. Formally, x_i and x_j are adjacent iff $c_{ij} \in \mathcal{C}$. If x_i and x_j are adjacent we also say that x_i and x_j are **neighbors**. The set of neighbors of a variable x_i is denoted by $\Gamma(x_i)$.*

Definition 1.4 *Given a constraint graph G, an **ordering** $\mathcal{O}$ is a mapping from the variables (vertexes of G) to the set $\{1, \ldots, n\}$. $\mathcal{O}(i)$ is the ith variable in $\mathcal{O}$.*

Solving a CSP is equivalent to find a combination of assignments of values to all variables in a way that all the constraints of the problem are satisfied.

We present in the following some typical examples of problems that can be intuitively

modeled as constraint satisfaction problems. These examples range from academic problems to real-world applications.

1.1.2 Examples of CSPs

Various problems in artificial intelligence can be naturally modeled as a constraint satisfaction problem. We present here some examples of problems that can be modeled and solved by the CSP paradigm. First, we describe the classical n-queens problem. Next, we present the graph-coloring problem. Last, we introduce the problem of meeting scheduling.

1.1.2.1 The n-queens problem

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

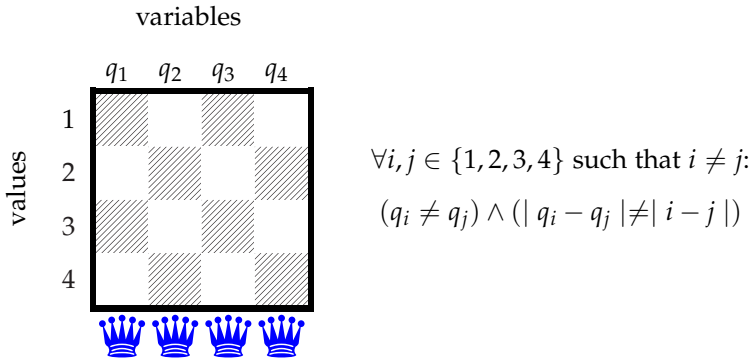


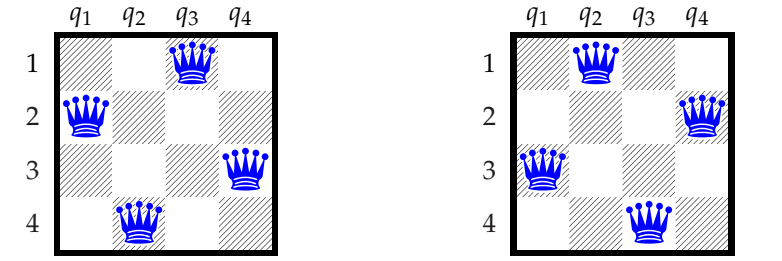
Figure 1.1 – The 4-queens problem.

In the case of 4-queens ($n = 4$), the problem can be formalized as a CSP as follows (Figure 1.1).

- $\mathcal{X} = \{q_1, q_2, q_3, q_4\}$, each variable q_i corresponds to the queen placed in the ith column.
- $\mathcal{D} = \{D(q_1), D(q_2), D(q_3), D(q_4)\}$, where $D(q_i)$=$\{1,2,3,4\}$ $\forall i \in 1..4$. The value $v \in D(q_i)$ corresponds to the row where can be placed the queen representing the ith column.
- $\mathcal{C} = \{c_{ij} : (q_i \neq q_j) \wedge (\mid q_i - q_j \mid \neq \mid i - j \mid)\ \forall\ i, j \in \{1,2,3,4\}$ and $i \neq j\}$ is the set of constraints. There exists a constraint between each pair of queens that forbids the involved queens to be placed in the same row or diagonal line.

The n-queen problem admits in the case of $n = 4$ (4-queens) two configuration as solution. We present the two possible solution in Figure 1.2. The first solution Figure 1.2(a) is $(q_1 = 2, q_2 = 4, q_3 = 1, q_4 = 3)$ where we put q_1 in the second row, q_2 in the row 4,

q_3 in the first row, and q_4 is placed in the third row. The second solution Figure 1.2(b) is $(q_1 = 3, q_2 = 1, q_3 = 4, q_4 = 2)$.



(a) $(q_1 = 2, q_2 = 4, q_3 = 1, q_4 = 3)$ (b) $(q_1 = 3, q_2 = 1, q_3 = 4, q_4 = 2)$

Figure 1.2 – The solutions for the 4-queens problem.

1.1.2.2 The Graph Coloring Problem

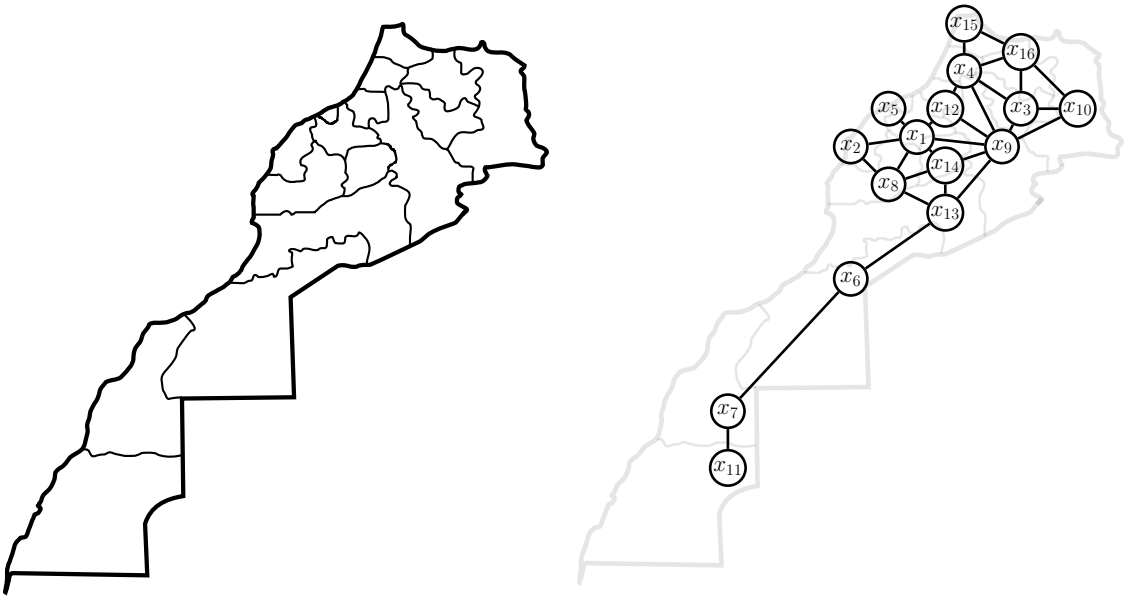
Another typical example problem is the graph coloring problem. Graph coloring is one of the most combinatorial problem studied in artificial intelligence since many real applications such as time-tabling and frequency allocation can be easily formulated as a graph coloring problem. The goal in this problem is to color all nodes of a graph so that any two adjacent vertexes should get different colors where each node has a finite number of possible colors. The Graph Coloring problem is simply formalized as a CSP. Hence, the nodes of the graph are the variables to color and the possible colors of each node/variable form its domain. There exists a constraint between each pair of adjacent variables/nodes that prohibits these variables to have the same color.

A practical application of the graph coloring problem is the problem of coloring a map (Figure 1.3). The objective in this case is to assign a color to each region so that no neighboring regions have the same color. An instance of the map-coloring problem is illustrated in Figure 1.3(a) where we present the map of Morocco with its 16 provinces. We present this map-coloring instance as a constraint graph in Figure 1.3(b). This problem can be modeled as a CSP by representing each node of the graph as a variable. The domain of each variable is defined by the possible colors. There exits a constraint between each pair neighboring regions. Therefore we get the following CSP:

- $\mathcal{X} = \{x_1,\ x_2,\ \ldots,\ x_{16}\}$.
- $\mathcal{D} = \{D(x_1),\ D(x_2),\ \ldots,\ D(x_{16})\}$, where $D(x_i) = \{red, blue, green\}$.
- $\mathcal{C} = \{c_{ij} : x_i \neq x_j \mid x_i \text{ and } x_j \text{ are neighbors}\}$.

1.1.2.3 The Meeting Scheduling Problem

The *meeting scheduling problem (MSP)* [**Sen and Durfee, 1995**; **Garrido and Sycara, 1996**; **Meisels and Lavee, 2004**] is a decision-making process that consist at scheduling several



(a) *The 16 provinces of Morocco.*

(b) *The map-coloring problem represented as a constraint graph.*

Figure 1.3 – An example of the graph-coloring problem.

meetings among various people with respect to their personal calendars. The meeting scheduling problem has been defined in many versions with different parameters (e.g, duration of meetings [**Wallace and Freuder, 2002**], preferences of agents [**Sen and Durfee, 1995**], etc). In MSP, we have a set of attendees, each with his/her own calendar (divided on time-slots), and a set of n meetings to coordinate. In general, people/participants may have several slots reserved for already filled planning in their calendars. Each meeting m_i takes place in a specified location denoted by $location(m_i)$. The proposed solution must enable the participating agents to travel among locations where their meetings will be hold. Thus, an *arrival-time* constraint is required between two meetings m_i and m_j when at least one attendee participates on both meetings. The arrival time constraint between two meetings m_i and m_j is defined in Equation 1.1:

$$\mid time(m_i) - time(m_j) \mid - duration \; > \; TravelingTime(location(m_i),\, location(m_j)). \quad (1.1)$$

The meeting scheduling problem [**Meisels and Lavee, 2004**] can be encoded in a centralized constraint satisfaction problem as follows:

- $\mathcal{X} = \{m_1, \ldots, m_n\}$ is the set of variables, each variable represents a meeting.
- $\mathcal{D} = \{D(m_1), \ldots, D(m_n)\}$ is a set of domains where $D(m_i)$ is the domain of variable/meeting (m_i). $D(m_i)$ is the intersection of time-slots from the personal calendar of all agents attending m_i (i.e., $D(m_i) = \bigcap_{A_j \in \text{ attendees of } m_i} calendar(A_j)$).
- $\mathcal{C}$ is a set of arrival-time constraints. There exists an arrival-time constraint for every pair of meetings (m_i, m_j) if there is an agent that participates in both meetings.

A simple instance of a meeting scheduling problem is illustrated in Figure 1.4. There

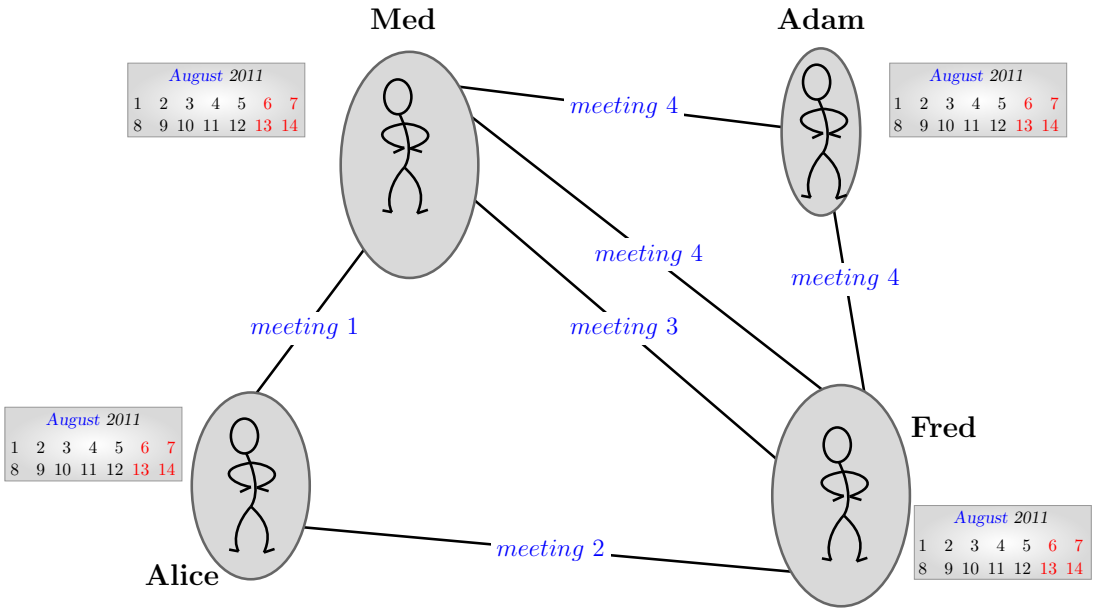


Figure 1.4 – A simple instance of the meeting scheduling problem.

are 4 attendees: *Adam*, *Alice*, *Fred* and *Med*, each having its personal calendar. There are 4 meetings to be scheduled. The first meeting (m_1) will be attended by *Alice* and *Med*. *Alice* and *Fred* will participate on the second meeting (m_2). The agents going to attend the third meeting (m_3) are *Fred* and *Med* while the last meeting (m_4) will be attended by three persons: *Adam*, *Fred* and *Med*.

The instance presented in Figure 1.4 is encoded as a centralized CSP in Figure 1.5. The nodes are the meetings/variables (m_1, m_2, m_3, m_4). The edges represent binary arrival-time constraint. Each edge is labeled by the person, attending both meetings. Thus,

- $\mathcal{X} = \{m_1, m_2, m_3, m_4\}$.
- $\mathcal{D} = \{D(m_1), D(m_2), D(m_3), D(m_4)\}$.
  - $D(m_1) = \{s \mid s \text{ is a slot in } calendar(Alice) \cap calendar(Med)\}$.
  - $D(m_2) = \{s \mid s \text{ is a slot in } calendar(Alice) \cap calendar(Fred)\}$.
  - $D(m_3) = \{s \mid s \text{ is a slot in } calendar(Adam) \cap calendar(Fred) \cap calendar(Med)\}$.
  - $D(m_4) = \{s \mid s \text{ is a slot in } calendar(Adam) \cap calendar(Fred) \cap calendar(Med)\}$.
- $\mathcal{C} = \{c_{12}, c_{13}, c_{14}, c_{23}, c_{24}, c_{34}\}$, where c_{ij} is an arrival-time constraint between m_i and m_j.

These examples show the power of the CSP paradigm to easily model different combinatorial problems arising from different issues. In the following section, we describe the main generic methods for solving a constraint satisfaction problem.

1.2 Algorithms and Techniques for Solving Centralized CSPs

In this section, we describe the basic methods for solving constraint satisfaction problems. These methods can be considered under two board approaches: constraint propagation and search. We also describe here a combination of those two approaches. In general,

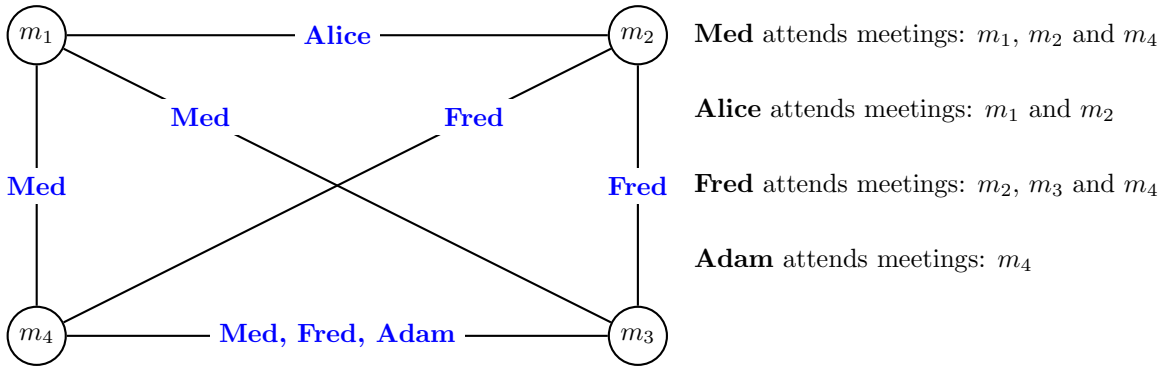


Figure 1.5 – The constraint graph of the meeting-scheduling problem.

the search algorithms explore all possible combinations of values for the variables in order to find a solution of the problem, that is, a combination of values for the variables that satisfies the constraints. However, the constraint propagation techniques are used to reduce the space of combinations that will be explored by the search process. Afterwards, we present the main heuristics used to boost the search in the centralized CSPs. We particularly summarize the main variable ordering heuristics while we briefly describe the main value ordering heuristics used in the constraint satisfaction problems.

1.2.1 Algorithms for solving centralized CSPs

Usually, algorithms for solving centralized CSPs search systematically through the possible assignments of values to variables in order to find a combination of these assignments that satisfies the constraints of the problem.

Definition 1.5 *An **assignment** of value v_i to a variable x_i is a pair (x_i, v_i) where v_i is a value from the domain of x_i (i.e., $v_i \in D(x_i)$). We often denote this assignment by $x_i = v_i$.*

Henceforth, when a variable is assigned a value from its domain, we say that the variable is assigned or instantiated.

Definition 1.6 *An **instantiation** I of a subset of variables $\{x_i, \ldots, x_k\} \subseteq \mathcal{X}$ is an ordered set of assignments $I = \{[(x_i = v_i), \ldots, (x_k = v_k)] \mid v_j \in D(x_j)\}$. The variables assigned on an instantiation $I = [(x_i = v_i), \ldots, (x_k = v_k)]$ are denoted by $\texttt{vars}(I) = \{x_i, \ldots, x_k\}$.*

Definition 1.7 *A **full instantiation** is an instantiation I that instantiates all the variables of the problem (i.e., $\texttt{vars}(I) = \mathcal{X}$) and conversely we say that an instantiation is **a partial instantiation** if it instantiates in only a part.*

Definition 1.8 *An instantiation I **satisfies** a constraint $c_{ij} \in \mathcal{C}$ if and only if the variables involved in c_{ij} (i.e., x_i and x_j) are assigned in I (i.e., $(x_i = v_i), (x_j = v_j) \in I$) and the pair (v_i, v_j) is allowed by c_{ij}. Formally, I satisfies c_{ij} iff $(x_i = v_i) \in I \wedge (x_j = v_j) \in I \wedge (v_i, v_j) \in c_{ij}$.*

Definition 1.9 *An instantiation I is **locally consistent** iff it satisfies all of the constraints whose*

scopes have no uninstantiated variables in I. I is also called a ***partial solution****. Formally, I is locally consistent iff* $\forall c_{ij} \in \mathcal{C} \mid \texttt{scope}(c_{ij}) \subseteq \texttt{vars}(I)$, *$I$ satisfies c_{ij}.*

Definition 1.10 *A* ***solution*** *to a constraint network is a full instantiation I, which is locally consistent.*

The intuitive way to search a solution for a constraint satisfaction problem is to *generate and test* all possible combinations of the variable assignments to see if it satisfies all the constraints. The first combination satisfying all the constraints is then a solution. This is the principle of the ***generate & test*** algorithm. In other words, a full instantiation is generated and then tested if it is locally consistent. In the generate & test algorithm, the consistency of an instantiation is not checked until it is full. This method drastically increases the number of combinations that will be generated. (The number of full instantiation considered by this algorithm is the size of the Cartesian product of all the variable domains). Intuitively, one can check the local consistency of instantiation as soon as its respective variables are instantiated. In fact, this is systematic search strategy of the chronological backtracking algorithm. We present the chronological backtracking in the following.

1.2.1.1 Chronological Backtracking (BT)

The chronological backtracking [Davis *et al.*, 1962; Golomb and Baumert, 1965; Bitner and Reingold, 1975] is the basic systematic search algorithm for solving CSPs. The `Backtracking` (BT) is a recursive search procedure that incrementally attempts to extend a current partial solution (a locally consistent instantiation) by assigning values to variables not yet assigned, toward a full instantiation. However, when all values of a variable are inconsistent with previously assigned variables (a *dead-end* occurs) BT backtracks to the variable immediately instantiated in order to try another alternative value for it.

Definition 1.11 *When no value is possible for a variable, a* ***dead-end*** *state occurs. We usually say that the domain of the variable is* ***wiped out*** (DWO).

Algorithm 1.1: The chronological Backtracking algorithm.

```
procedure Backtracking(I)
01. if ( isFull(I) ) then return I as solution;          /* all variables are assigned in I */
02. else
03.     select x_i in X \ vars(I) ;                      /* let x_i be an unassigned variable */
04.     foreach ( v_i ∈ D(x_i) ) do
05.         x_i ← v_i;
06.         if ( isLocallyConsistent(I ∪ {(x_i = v_i)}) ) then
07.             Backtracking(I ∪ {(x_i = v_i)});
```

The pseudo-code of the `Backtracking` (BT) algorithm is illustrated in Algorithm 1.1. The BT assigns a value to each variable in turn. When assigning a value v_i to a variable x_i, the consistency of the new assignment with values assigned thus far is checked (line 6, Algorithm 1.1). If the new assignment is consistent with previous assignments BT attempts to extend these assignments by selecting another unassigned variable (line 7). Otherwise (the

new assignment violates any of the constraints), another alternative value is tested for x_i if it is possible. If all values of a variable are inconsistent with previously assigned variables (a dead-end occurs), backtracking to the variable immediately preceding the dead-end variable takes place in order to check alternative values for this variable. By the way, either a solution is found when the last variable has been successfully assigned or BT can conclude that no solution exist if all values of the first variable are removed.

On the one hand, it is clear that we need only linear space to perform the backtracking. However, it requires time exponential in the number of variables for most nontrivial problems. On the other hand, the backtracking is clearly better than "generate & test" since a subtree from the search space is pruned whenever a partial instantiation violates a constraint. Thus, backtracking can detect early unfruitful instantiation compared to "generate & test".

Although the backtracking improves the "generate & test", it still suffer from many drawbacks. The main one is the *thrashing* problem. Thrashing is the fact that the same failure due to the same reason can be rediscovered an exponential number of times when solving the problem. Therefore, a variety of refinements of BT have been developed in order to improve it. These improvements can be classified under two main schemes: look-back methods as conflict directed backjumping or look-ahead methods such as forward checking.

1.2.1.2 Conflict-directed Backjumping (CBJ)

From the earliest works in the area of constraint programming, researchers were concerned by the trashing problem of the Backtracking, and then proposed a number of tools to avoid it. *backjumping* concept was one of the pioneer tools used for this reason. Thus, several non-chronological backtracking (intelligent backtracking) search algorithms have been designed to solve centralized CSPs. In the standard form of backtracking, each time a dead-end occurs the algorithm attempts to change the value of the most recently instantiated variable. However, backtracking chronologically to the most recently instantiated variable may not address the reason for the failure. This is no longer the case in the backjumping algorithms that identify and then *jump* directly to the responsible of the dead-end (*culprit*). Hence, the culprit variable is re-assigned if it is possible or an other jump is performed. By the way, the subtree of the search space where the thrashing may occur is pruned.

Definition 1.12 *Given a total ordering on variables $\mathcal{O}$, a constraint c_{ij} is **earlier** than c_{kl} if the latest variable in* $\texttt{scope}(c_{ij})$ *precedes the latest one in* $\texttt{scope}(c_{kl})$ *on $\mathcal{O}$.*

Example 1.1 Given the lexicographic ordering on variables $([x_1, \ldots, x_n])$, the constraint c_{25} is earlier than constraint c_{35} because x_2 precedes x_3 since x_5 belongs to both scopes (i.e., $\texttt{scope}(c_{25})$ and $\texttt{scope}(c_{35})$).

Gaschnig designed the first explicit non-chronological (*backjumping*) algorithm (BJ) in [**Gaschnig, 1978**]. BJ records for each variable x_i the *deepest* variable with which it checks

its consistency with the assignment of x_i. When a dead-end occurs on a domain of a variable x_i, BJ jumps back to the deepest variable, say x_j, to witch the consistency of x_i is checked against. However, if there are no more values remaining for x_j, BJ perform a simple backtrack to the last assigned variable before assigning x_j. **Dechter** presented in [**Dechter, 1990**; **Dechter and Frost, 2002**] the *Graph-based BackJumping* (GBJ) algorithm, a generalization of the BJ algorithm. Basically, GBJ attempts to jump back directly to the source of the failure by using only information extracted from the constraint graph. Whenever a dead-end occurs on a domain of the current variable x_i, GBJ jumps back to the most recent assigned variable (x_j) adjacent to x_i in the constraint graph. Unlike BJ, if a dead-end occurs again on a domain of x_j, GBJ jumps back to the most recent variable x_k connected to x_i or x_j. **Prosser** proposed the *Conflict-directed BackJumping* (CBJ) that rectify the bad behavior of **Gaschnig**'s algorithm in [**Prosser, 1993**].

Algorithm 1.2: The Conflict-Directed Backjumping algorithm.

```
procedure CBJ(I)
01. if ( isFull(I) ) then return I as solution;          /* all variables are assigned in I */
02. else
03.   choose x_i in X \ vars(I) ;                        /* let x_i be an unassigned variable */
04.   EMCS[i] ← ∅ ;
05.   D(x_i) ← D^0(x_i) ;
06.   foreach ( v_i ∈ D(x_i) ) do
07.     x_i ← v_i;
08.     if ( isConsistent(I ∪ (x_i = v_i)) ) then
09.       CS ← CBJ(I ∪ {(x_i = v_i)}) ;
10.       if ( x_i ∉ CS ) then return CS ;
11.       else EMCS[i] ← EMCS[i] ∪ CS \ {x_i} ;
12.     else
13.       remove v_i from D(x_i) ;
14.       let c_ij be the earliest violated constraint by (x_i = v_i);
15.       EMCS[i] ← EMCS[i] ∪ x_j ;
16.   return EMCS[i] ;
```

The pseudo-code of CBJ is illustrated in Algorithm 1.2. Instead of recording only the (*deepest* variable, CBJ records for each variable x_i the set of variables that were in conflict with some assignment of x_i. Thus, CBJ maintains a set of *earliest minimal conflict set* for each variable x_i (i.e., $EMCS[i]$) where it stores the variables belonging to the earliest violated constraints with an assignment of x_i. Whenever a variable x_i is chosen to be instantiated (line 3), CBJ initializes $EMCS[i]$ to the empty set. Next, CBJ initializes the current domain of x_i to its initial domain (line 5). Afterward, a consistent value v_i with the current search state is looked for variable x_i. If v_i is inconsistent with the current partial solution, then v_i is removed from current domain $D(x_i)$ (line 13), and x_j such that c_{ij} is the earliest violated constraint by the new assignment of x_i (i.e., $x_i = v_i$) is then added to the earliest minimal conflict set of x_i, i.e., $EMCS[i]$ (line 15). $EMCS[i]$ can be seen as the subset of the past variables in conflict with x_i. When a dead-end occurs on the domain of a variable x_i, CBJ jumps back to the last variable, say x_j, in $EMCS[i]$ (lines 16,9 and line 10). The information in $EMCS[i]$ is earned upwards to $EMCS[j]$ (line 11). Hence, CBJ performs a form of "in-

telligent backtracking" to the source of the conflict allowing the search procedure to avoid rediscovering the same failure due to the same reason.

When a dead-end occurs, the CBJ algorithm jumps back to address the culprit variable. During the backjumping process CBJ erases all assignments that were obtained since and then wastes a meaningful effort done to achieve these assignments. To overcome this drawback **Ginsberg** (**1993**) have proposed Dynamic Backtracking.

1.2.1.3 Dynamic Backtracking (DBT)

In the naive chronological of backtracking (BT), each time a dead-end occurs the algorithm attempts to change the value of the most recently instantiated variable. Intelligent backtracking algorithms were developed to avoid the trashing problem caused by the BT. Although, these algorithms identify and then jump directly to the responsible of the dead-end (*culprit*), they erase a great deal of the work performed thus far on the variables that are backjumped over. When backjumping, all variables between the culprit of the dead-end and the variable where the dead-end occurs will be re-assigned. **Ginsberg** (**1993**) proposed the *Dynamic Backtracking* algorithm (DBT) in order to keep the progress performed before the backjumping. In DBT, the assignments of non conflicting variables are preserved during the backjumping process. Thus, the assignments of all variables following the culprit are kept and the culprit variable is moved to be the last among the assigned variables.

In order to detect the culprit of the dead-end, CBJ associates a conflict set ($EMCS[i]$) to each variable (x_i). $EMCS[i]$ contains the set of the assigned variables whose assignments are in conflict with a value from the domain of x_i. In a similar way, DBT uses nogoods to justify the value elimination [**Ginsberg, 1993**]. Based on the constraints of the problem, a search procedure can infer inconsistent sets of assignments called nogoods.

Definition 1.13 *A **nogood** is a conjunction of individual assignments, which has been found inconsistent, either because the initial constraints or because searching all possible combinations.*

Example 1.2 The following nogood $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \ldots \wedge (x_k = v_k)]$ means that assignments it contains are not simultaneously allowed because they cause an inconsistency.

Definition 1.14 *A **directed nogood** ruling out value v_k from the initial domain of variable x_k is a clause of the form $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k \neq v_k$, meaning that the assignment $x_k = v_k$ is inconsistent with the assignments $x_i = v_i, x_j = v_j, \ldots$. When a nogood ($ng$) is represented as an implication, the **left hand side**, `lhs`(ng), and the **right hand side**, `rhs`(ng), are defined from the position of $\rightarrow$.*

In DBT, when a value is found to be inconsistent with previously assigned values, a directed nogood is stored as a justification of its removal. Hence, the current domain $D(x_i)$ of a variable x_i contains all values from its initial domain that are not ruled out by a stored nogood. When all values of a variable x_i are ruled out by some nogoods, a dead-end occurs, DBT resolves these nogoods producing a new nogood (*newNogood*). Let x_j be the most recent variable in the left-hand side of all these nogoods and $x_j = v_j$, that is x_j is the culprit variable in the CBJ algorithm. The `lhs`$(newNogood)$ is the conjunction of the

left-hand sides of all nogoods except $x_j = v_j$ and $\texttt{rhs}(newNogood)$ is $x_j \neq v_j$. Unlike the CBJ, DBT only removes the current assignment of x_j and keeps assignments of all variables between it an x_i since they are consistent with former assignments. Therefore, the work done when assigning these variables is preserved. The culprit variable x_j is then placed after x_i and a new assignment for it is searched since the generated nogood (*newNogood*) eliminates its current value (v_j).

Since the number of nogoods that can be generated increases monotonically, recording all of the nogoods as is done in Dependency Directed Backtracking algorithm [**Stallman and Sussman, 1977**] requires an exponential space complexity. In order to keep a polynomial space complexity, DBT stores only nogoods compatible with the current state of the search. Thus, when backtracking to x_j, DBT destroys all nogoods containing $x_j = v_j$. As a result, with this approach a variable assignment can be ruled out by at most one nogood. Since each nogood requires $O(n)$ space and there are at most nd nogoods, where n is the number of variables and d is the maximum domain size, the overall space complexity of DBT is in $O(n^2d)$.

1.2.1.4 Partial Order Dynamic Backtracking (PODB)

Instead of backtracking to the most recently assigned variable in the nogood, **Ginsberg and McAllester** proposed the *Partial Order Dynamic Backtracking* (PODB), an algorithm that offers more freedom than DBT in the selection of the variable to put on the right-hand side of the directed nogood [**Ginsberg and McAllester, 1994**]. thereby, PODB is a polynomial space algorithm that attempted to address the rigidity of dynamic backtracking.

When resolving the nogoods that lead to a dead-end, DBT always select the most among the set of inconsistent assignments recent assigned variable to be the right hand side of the generated directed nogood. However, there are clearly many different ways of representing a given nogood as an implication (directed nogood). For example, $\neg[(x_i = v_i) \wedge (x_j = v_j) \wedge \cdots \wedge (x_k = v_k)]$ is logically equivalent to $[(x_j = v_j) \wedge \cdots \wedge (x_k = v_k)] \rightarrow (x_i \neq v_i)$ meaning that the assignment $x_i = v_i$ is inconsistent with the assignments $x_j = v_j, \ldots, x_k = v_k$. Each directed nogood imposes ordering constraints, called the set of *safety conditions* for completeness [**Ginsberg and McAllester, 1994**]. Since all variables in the left hand side of a directed nogood participate in eliminating the value on its right hand side, these variable must precede the variable on the right hand side.

Definition 1.15 ***safety conditions*** *imposed by a directed nogood (ng) ruling out a value from the domain of x_j are the set of assertions of the form $x_k \prec x_j$ where x_k is a variable in the left hand side of ng (i.e., $x_k \in \texttt{vars}(\texttt{lhs}(ng))$).*

The Partial Order Dynamic Backtracking attempts to offer more freedom in the selection of the variable to put on the right-hand side of the generated directed nogood. In PODB, the only restriction to respect is that the partial order induced by the resulting directed nogood must safety the existing partial order required by the set of safety conditions, say S. In a later study, **Bliek** shows that PODB is not a generalization of DBT and then proposes the *Generalized Partial Order Dynamic Backtracking* (GPODB), a new algorithm that generalizes

both PODB and DBT [Bliek, 1998]. To achieve this, GPODB follows the same mechanism of PODB. The difference between two resides in the obtained set of safety conditions S' after generating a new directed nogood (*newNogood*). The new order has to respect the safety conditions existing in S'. While S and S' are the similar for PODB, when computing S' GPODB relaxes from S all safety conditions of the form $\texttt{rhs}(newNogood) \prec x_k$. However, both algorithms generates only directed nogoods that satisfy the already existing safety conditions in S. In the best of our knowledge, no systematic evaluation of either PODB or GPODB have been reported.

All algorithms presented previously incorporates a form of look-back scheme. Avoiding possible future conflicts may be more attractive than recovering from them. In the backtracking, backjumping and dynamic backtracking, we can not detect that an instantiation is unfruitful till all variables of the conflicting constraint are assigned. Intuitively, each time a new assignment is added to the current partial solution, one can look ahead by performing a forward check of consistency of the current partial solution.

1.2.1.5 Forward Checking (FC)

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

Algorithm 1.3: The forward checking algorithm.

```
procedure ForwardChecking(I)
01. if ( isFull(I) ) then return I as solution;          /* all variables are assigned in I */
02. else
03.    select x_i in X \ vars(I) ;                       /* let x_i be an unassigned variable */
04.    foreach ( v_i ∈ D(x_i) ) do
05.       x_i ← v_i;
06.       if ( Check-Forward(I, (x_i = v_i)) ) then
07.          ForwardChecking(I ∪ {(x_i = v_i)});
08.       else
09.          foreach ( x_j ∉ vars(I) such that ∃ c_ij ∈ C ) do restore D(x_j);
function Check-Forward(I, x_i = v_i)
10. foreach ( x_j ∉ vars(I) such that ∃ c_ij ∈ C ) do
11.    foreach ( v_j ∈ D(x_j) such that (v_i, v_j) ∉ c_ij ) do remove v_j from D(x_j) ;
12.    if ( D(x_j) = ∅ ) then return false;
13. return true;
```

The pseudo-code of FC procedure is presented in Algorithm 1.3. FC is a recursive procedure that attempts to foresee the effects of choosing an assignment on the not yet assigned variables. Each time a variable is assigned, FC checks forward the effects of this assignment on the future variables domains (`Check-Forward` call, line 6). So, all values from the domains of future variables which are inconsistent with the assigned value (v_i) of the current variable (x_i) are removed (line 11). Future variables concerned by this filtering

process are only those sharing a constraint with x_i, the current variable being instantiated (line 10). By the way, each domain of a future variable is filtered in order to keep only consistent values with past variables (variables already instantiated). Hence, FC does not need to check consistency of new assignments against already instantiated ones as opposed to chronological backtracking. The forward checking is then the easiest way to prevent assignments that guarantee later failure.

Unlike backtracking, forward checking algorithm enables to prevent assignments that guarantee later failure. This improves the performance of backtracking. However, forward checking reduces the domains of future variables checking only the constraints relating them to variables already instantiated. In addition to these constraints, one can checks also the constraints relating future variables to each other. By the way, domains of future variables may be reduced and further possible conflicts will be avoided. This is the principle of the *full* look-ahead scheme or constraint propagation. This approach is called maintaining arc consistency (MAC).

1.2.1.6 Arc-consistency (AC)

In constraint satisfaction problems, checking the existence of solutions is NP-complete. Therefore, the research community has devoted a great interest in studying the *constraint propagation* techniques. Constraint propagation techniques are filtering mechanisms that aim to improve the performance of the search process by attempting to reduce the search space. They have been widely used to simplify the search space before or during the search of solutions. Thus, constraint propagation became a central process of solving CSPs [**Bessiere, 2006**]. Historically, different kinds of constraint propagation techniques have been proposed: node consistency [**Mackworth, 1977**], arc consistency [**Mackworth, 1977**] and path consistency [**Montanari, 1974**] . The oldest and most commonly used technique for propagating constraints in literature is the *Arc Consistency*, AC.

Definition 1.16 *A value $v_i \in D(x_i)$ is consistent with c_{ij} in $D(x_j)$ iff there exists a value $v_j \in D(x_j)$ such that (v_i, v_j) is allowed by c_{ij}. Value v_j is called a **support** for v_i in $D(x_j)$.*

Let us assume the constraint graph $G = \{X_G, E_G\}$ (see Definition 1.2) associated to our constraint satisfaction problem.

Definition 1.17 *An arc $\{x_i, x_j\} \in E_G$ (constraint c_{ij}) is **arc consistent** iff $\forall\, v_i \in D(x_i), \exists\, v_j \in D(x_j)$ such that (v_i, v_j) is allowed by c_{ij} and $\forall\, v_j \in D(x_j), \exists\, v_i \in D(x_i)$ such that (v_i, v_j) is allowed by c_{ij}. A constraint network is arc consistent iff all its arcs (constraints) are arc consistent.*

A constraint network is arc consistent if and only if for any value v_i in the domain, $D(x_i)$, of a variable x_i there exist in the domain $D(x_j)$ of an adjacent variable x_j a value v_j that is compatible with v_i. Clearly, if an arc $\{x_i, x_j\}$ (i.e., a constraint c_{ij}) is not arc consistent, it can be made arc consistent by simply deleting all values from the domains of the variables in its scope for which there is not a support in the other domain. It is obvious that these deletions maintains the problem solutions since the deleted values are in no solution. The process of removing values from the domain of a variable x_i, when

making an arc $\{x_i, x_j\}$ arc consistent is called *revising* $D(x_i)$ with respect to constraint c_{ij}. A wide variety of algorithms establishing arc consistency on CSPs have been developed: AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986], AC-5 [Van Hentenryck *et al.*, 1992], AC-6 [Bessiere and Cordier, 1993; Bessiere, 1994], AC-7 [Bessiere *et al.*, 1999], AC-2001 [Bessiere and Régin, 2001], etc. The basic algorithm and the most will-known one is Mackworth's AC-3.

Algorithm 1.4: The AC-3 algorithm.

```
function Revise(x_i, x_j)
01. change ← false;
02. foreach ( v_i ∈ D(x_i) ) do
03.     if ( ∄ v_j ∈ D(x_j) such that (v_i, v_j) ∈ c_ij ) then
04.         remove v_i from D(x_i) ;
05.         change ← true;
06. return change ;
function AC-3()
07. foreach ( {x_i, x_j} ∈ E_G ) do                    /* {x_i, x_j} ∈ E_G iff ∃ c_ij ∈ C */
08.     Q ← Q ∪ {(x_i, x_j); (x_j, x_i)} ;
09. while ( Q ≠ ∅ ) do
10.     (x_i, x_j) ← Q.pop() ;                          /* Select and remove (x_i, x_j) from Q */
11.     if ( Revise(x_i, x_j) ) then
12.         if ( D(x_i) = ∅ ) then return false ;       /* The problem is unsatisfiable */
13.         else Q ← Q ∪ { (x_k, x_i) | {x_k, x_i} ∈ E_G, k ≠ i, k ≠ j } ;
14. return true ;                                       /* The problem is arc consistent */
```

We illustrate the AC-3 algorithm in Algorithm 1.4. The AC-3 algorithm maintains a queue Q [1] of arcs to render arc consistent. AC-3 algorithm will return true once the problem is made arc consistent or false if an empty domain was generated (a domain is *wiped out*) meaning that the problem is not satisfiable. Initially, Q is filled with all ordered pair of variables that participates in a constraint. Thus, for each constraint c_{ij} ($\{x_i, x_j\}$) we add to Q the ordered pair (x_i, x_j) to revise the domain of x_i and the ordered pair (x_j, x_i) the revise the domain of x_j (line 8). Next, the algorithm loops until it is guaranteed that all arcs have been made arc consistent (i.e., while Q is not empty). The ordered pair of variables are selected and removed one by one from Q to revise the domain of the first variable. Each time an ordered pair of variable (x_i, x_j) is selected and removed from Q (line 10), AC-3 calls function `Revise`(x_i, x_j) to revise the domain of x_i. When revising $D(x_i)$ with respect to an arc $\{x_i, x_j\}$ (`Revise` call, line 11), all values that are not consistent with c_{ij} are removes from $D(x_i)$ (lines 2-4). Thus, only values having a support on $D(x_j)$ are kept in $D(x_i)$. The function `Revise` returns true if the domain of variable x_i has been reduced, false otherwise (line 6). If `Revise` results in the removal of values from $D(x_i)$ it can be the case that a value for another variable x_k has lost its support on $D(x_i)$. Thus, all ordered pairs (x_k, x_i) such that $k \neq j$ are added onto Q so long as they are not already on Q in order to revise the domain of x_k. Obviously, the AC-3 algorithm will not terminate as long as there is any pair in Q. When Q is empty, we are guaranteed that all arcs have been made arc consistent. Hence, the constraint network is arc consistent. The while loop of AC-3 can

1. Other data structures as queue or stack can perfectly serve the purpose

be intuitively understood as constraint propagation (i.e., propagation the effect of value removals on other domains potentially affected by these removals).

1.2.1.7 Maintaining Arc-Consistency (MAC)

Historically, constraint propagation techniques are used in a preprocessing step to prune values before search. Thus, the search space that will be explored by the search algorithm is reduced since domains of all variables are refined. By the way, subsequent search efforts by the solution method will be reduced. Afterward, the search method can be called for searching a solution. Constraint propagation techniques are also used during search. This strategy is that used by the forward checking algorithm. Forward checking combines backtrack search with a limited form of arc consistency maintenance on the domains of future variables. Instead of performing a limited form of arc consistency, **Sabin and Freuder** proposed in [**Sabin and Freuder, 1994**] the Maintaining Arc-Consistency (MAC) algorithm that established and maintained a *full arc consistency* on the domains of future variables.

The *Maintaining Arc Consistency* (MAC) algorithm is a modern version of CS2 algorithm [**Gaschnig, 1974**]. MAC alternates the search process and constraint propagation steps as is done in forward checking [**Haralick and Elliott, 1980**]. Nevertheless, before starting the search method, MAC makes the constraint network arc consistent. In addition, when instantiating a variable x_i to a value v_i, all the other values in $D(x_i)$ are removed and the effects of these removals are propagated through the constraint network [**Sabin and Freuder, 1994**]. The maintaining arc consistency algorithm enforces arc consistency in the search process as follows. At each step of the search, a variable assignment is followed by a filtering process that corresponds to enforcing arc consistency. Therefore, MAC maintains the arc consistency each time an instantiation is added to the partial solution. In other word, whenever a value v_i is instantiated to a variable x_i, $D(x_i)$ is reduced momentarily to a single value v_i (i.e., $D(x_i) \leftarrow \{v_i\}$) and the resulting constraint network is then made arc consistent.

1.2.2 Variable Ordering Heuristics for Centralized CSP

Numerous efficient search algorithms for solving constraint satisfaction problems have been developed. The performance of these algorithms were evaluated in different studies and then shown to be powerful tools for solving CSPs. Nevertheless, since CSPs are in general NP-complete, these algorithms still exponential. Therefore, a large variety of *heuristics* were developed to improve their efficiency. That is, search algorithms solving CSPs are commonly combined with heuristics for boosting the search. The literature is rich in heuristics designed for this task. The order in which variables are assigned by a search algorithm was one of the early concern for these heuristics. The order on variables can be either static or dynamic.

1.2.2.1 Static Variable Ordering Heuristics (SVO)

The first kind of heuristics addressing the ordering of variables was based on the initial structure of the network. Thus, the order of the variables can be determined prior the search of solution. These heuristics are called Static Variable Ordering heuristics (SVO). When presenting the main search procedures (Section 1.2), we always assumed without specifying it each time, a static variable ordering. Therefore, in the previous examples we have always used the lexicographic ordering of variables. That lexicographic ordering can be simply replaced by an other ordering more appropriate to the structure of the network before starting search.

Static variable ordering heuristics (SVO) are heuristics that keep the same ordering all along the search. Hence, the ordering computed on a preprocessing step only exploit (structural) information about the initial state of the search. Examples of such SVO heuristics are:

min-width The *minimum width* heuristic [**Freuder, 1982**] chooses an ordering that minimizes the width of the constraint graph. The ***width*** of an ordering $\mathcal{O}$ is the maximum number of neighbors of any variable x_i that occur earlier than x_i under $\mathcal{O}$. The ***width*** of a constraint graph is the minimum width over all orderings of variables of that graph. Minimizing the width of the constraint graph G can be accomplished by a greedy algorithm. Hence, variables are ordered from last to first by choosing, at each step, a variable having the minimum number of neighbors (min degree) in the remaining constraint graph after deleting from the constraint graph all variables which have been already ordered.

max-degree The *maximum degree* heuristic [**Dechter and Meiri, 1989**] orders the variables in a decreasing order of their degrees in the constraint graph (i.e., the size of their neighborhood). This heuristic also aims at, without any guarantee, finding a minimum-width ordering.

max-cardinality The *maximum cardinality* heuristic [**Dechter and Meiri, 1989**] orders the variables according to the initial size of their neighborhood. ***max-cardinality*** puts in the first position of the resulting ordering an arbitrarily variable. Afterward, other variables are ordered from second to last by choosing, at each step, the most connected variable with previously ordered variables. In a particular case, ***max-cardinality*** may choose as the first variable the one that has the largest number of neighbors.

min-bandwidth The *minimum bandwidth* heuristic [**Zabih, 1990**] minimizes the bandwidth of the constraint graph. The ***bandwidth*** of an ordering is the maximum distance between any two adjacent variables in the ordering. The ***bandwidth*** of a constraint graph is the minimum bandwidth over all orderings on variables of that graph. **Zabih** claims that an ordering with a small bandwidth will reduce the need for backjumping because the culprit variable will be close to the variable where a dead-end occurs. Many heuristic procedures for finding minimum bandwidth orderings have been developed, a survey of these procedures is given in [**Chinn *et al.*, 1982**]. However, there

is currently little empirical evidence that ***min-bandwidth*** is an effective heuristic. Moreover, bandwidth minimization is NP-complete.

Another static variable ordering heuristic that tries to exploit the structural information residing in the constraint graph is presented in [**Freuder and Quinn, 1985**]. **Freuder and Quinn** (**1985**) have introduced the use of pseudo-tree arrangement of a constraint graph in order to enhance the research complexity in centralized constraint satisfaction problems.

Definition 1.18 *A **pseudo-tree** arrangement $T = (X_T, E_T)$ of a constraint graph $G = (X_G, E_G)$ is a rooted tree with the same set of vertexes as G (i.e., $X_G = X_T$) such that vertexes in different branches of T do not share any edge in G.*

The concept of *pseudo-tree* arrangement of a constraint graph has been introduced first by **Freuder and Quinn** in [**Freuder and Quinn, 1985**]. The purpose of this arrangement is to perform search in parallel on independent branches of the pseudo-tree in order to improve search in centralized CSPs. A recursive procedure for heuristically building pseudo-trees have been presented by **Freuder and Quinn** in [**Freuder and Quinn, 1985**]. The heuristic aims to select from G_X the minimal subset of vertexes named *cutset* whose removal divides G into disconnected sub-graphs. The selected *cutset* will form the first levels of the pseudo-tree while next levels are build by recursively applying the procedure to the disconnected sub-graphs obtained previously. By the way, the connected vertexes in the constraint graph G belongs to the same branch of the obtained tree. Thus, the tree obtained is a pseudo-tree arrangement of the constraint graph. Once, the pseudo-tree arrangement of the constraint graph is built, several search procedure can be performed in parallel on each branch of the pseudo-tree.

Although static variable ordering heuristics are undoubtedly cheaper since they are computed once and for all, using this kind of variable ordering heuristics does not change the worst-case complexity of the classical search algorithms. On the other hand, researchers have been expected that dynamic variable ordering heuristics (DVO) can be more efficient. DVO heuristics were expected to be potentially more powerful because they can take advantage from the information about the current search state.

1.2.2.2 Dynamic Variable Ordering Heuristics (DVO)

Instead of fixing an ordering as is done is SVO heuristics, dynamic variable ordering (DVO) heuristics determines the order of the variables as search progresses. The order of the variables may then differ from one branch of the search tree to another. It has been shown empirically for many practical problems that DVO heuristics are more effective than choosing a good static ordering [**Haralick and Elliott, 1980**; **Purdom, 1983**; **Dechter and Meiri, 1989**; **Bacchus and Van Run, 1995**; **Gent *et al.*, 1996**]. Hence, the field of constraint programming has so far mainly focused on such king of heuristics. Therefore, many dynamic variable ordering heuristics for solving constraint networks have been proposed and evaluated over the years. These heuristics are usually combined with search procedures performing some form of look ahead (see Section 1.2.1.5 and Section 1.2.1.7) in order to take into account changes on not yet instantiated (future) variables.

The guiding idea of the most DVO heuristic is to select the future variable with the smallest domain size. Henceforth, this heuristic is named ***dom***. Historically, **Golomb and Baumert** (**1965**) where the first to propose the ***dom*** heuristic. However, it was popularized when it was combined with the forward checking procedure by **Haralick and Elliott** (**1980**). ***dom*** investigates the future variables (remaining sub-problem) and provides choosing as next variable the one with the smallest remaining domain. **Haralick and Elliott** proposed ***dom*** under the rubric of an intuition called fail first principle: *"to succeed, try first where you are likely to fail"*. Moreover, they assume that *"the best search order is the one which minimizes the expected length or depth of each branch"* (p. 308). Thus, they estimate that minimizing branch length in a search procedure should also minimize search effort.

Many studies has gone into understanding the ***dom*** heuristic that is a simple but effective heuristic. Following the same principle of **Haralick and Elliott** saying that search efficiency is due to earlier failure, **Smith and Grant** (**1998**) have derived from ***dom*** new heuristics that detect failures earlier than ***dom***. Their study is based on a intuitive hypotheses saying that earlier detection of failure should lead the heuristic to lower search effort. Surprisingly, **Smith and Grant**'s experiments refuted this hypotheses contrary to their expectations. They concluded that increasing the ability to fail early in the search did not always lead to increased search efficiency. In follow on work, **Beck *et al.*** (**2005**) shown that in forward checking (Section 1.2.1.5) minimizing branch depth is associated with an increase in the branching factor. This can lead forward checking to perform badly. Nevertheless, their experiments show that minimizing branch depth in Maintained Arc Consistency (Section 1.2.1.7) reduces the search effort. Therefore, **Beck *et al.*** do not overlook the principle of trying to fail earlier in the search. They propose to redefine failing early in a such way to combine both the branching factor and the branch depth as it was suggested by **Nudel** in [**Nudel, 1983**] (for instance, minimizing the number of nodes in the failed subtrees).

In addition to the studies that has gone into understanding ***dom***, considerable research effort has been spent on improving it by suggesting numerous variants. These variants express the intuitive idea that a variable which is constrained with many future variables can also lead to a failure (a dead-end). Thus, these variants attempts to take into account the neighborhood of the variables as well as their domain size. We present in the following a set of well-known variable ordering heuristics derived from ***dom***.

dom+deg A variant of ***dom***, ***dom+deg***, have been designed in [**Frost and Dechter, 1994**] to break ties when all variables have the same initial domain size. ***dom+deg*** heuristic breaks ties by giving priority to variable with the highest *degree* (i.e., the one with the largest number of neighbors).

dom+futdeg Another variant breaking ties of ***dom*** is the ***dom+futdeg*** heuristic [**Brélaz, 1979**; **Smith, 1999**]. Originally, ***dom+futdeg*** was developed by **Brélaz** for the graph coloring problem and then applied later to CSPs. ***dom+futdeg*** chooses a variable with smallest remaining domain (***dom***), but in case of a tie, it chooses from these the variable with largest future degree, that is, the one having the largest number of neighbors in the remaining subproblem (i.e., among future variables).

dom/deg Both ***dom+deg*** and ***dom+futdeg*** use the domain size as the main criterion. The degree of the variables is considered only in case of ties. Alternatively, **Bessiere and Régin (1996)** combined ***dom*** with ***deg*** in a new heuristics called ***dom/deg***. The ***dom/deg*** do not gives priority to the domain size or degree of variables but uses them equally. This heuristic selects the variable that minimizes the ratio of current domain size to static degree. **Bessiere and Régin** have been shown that ***dom/deg*** gives good results in comparison with ***dom*** when the constraint graphs are sparse but performs badly on dense constraint graphs. They, considered a variant of this heuristic which minimizes the ratio of current domain size to future degree ***dom/futdeg***. However, they found that the performance of ***dom/futdeg*** is roughly similar to that of ***dom/deg***.

Multi-level-DVO A general formulation of dynamic variable ordering heuristics which approximates the constrainedness of variables and constraints, denoted ***Multi-level-DVO***, have been proposed in [**Bessiere *et al.*, 2001a**]. ***Multi-level-DVO*** heuristics are considered as neighborhood generalizations of ***dom*** and ***dom/deg*** and the selection function for variable x_i they suggested is as follows:

$$H_{\alpha}^{\circledcirc}(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc \alpha(x_j))}{|\ \Gamma(x_i)\ |^2}$$

where $\Gamma(x_i)$ is the set of x_i neighbors, $\alpha(x_i)$ can be any syntactical property of the variable such as ***dom*** or ***dom/deg*** and $\circledcirc \in \{+, \times\}$. Therefore, ***Multi-level-DVO*** take into account the neighborhood of variables which have shown to be quite promising. Moreover, it allows using functions to measure the weight of a given constraint.

dom/wdeg Conflict-driven variable ordering heuristics have been introduced in [**Boussemart *et al.*, 2004**]. These heuristics learn from previous failures to manage the choice on future variables. When a constraint leads to a dead-end, its weight is incremented by one. Each variable has a weighted degree, which is the sum of the weights over all constraints involving this variable. This heuristic can simply select the variable with the largest weighted degree (***wdeg***) or incorporating the domain size of variables to give the domain-over-weighted-degree heuristic (***dom/wdeg***). ***dom/wdeg*** selects among future variables the one with minimum ratio between current domain size and weighted degree. ***wdeg*** and ***dom/wdeg*** (especially ***dom/wdeg***) have been shown to perform well on a variety of problems.

In addition to the variable heuristics we presented here, other elegant dynamic heuristics have been developed for centralized CSPs in many studies [**Gent *et al.*, 1996**; **Horsch and Havens, 2000**]. However, these heuristics require extra computation and have only been tested on random problems. On other hand, it has been shown empirically that maintaining arc-consistency (MAC) combined to the ***dom/deg*** or the ***dom/wdeg*** can reduce or remove the need for backjumping on some problems [**Bessiere and Régin, 1996**; **Lecoutre *et al.*, 2004**]. Although, the variable ordering heuristics proposed are numerous, one has to notice that none of these heuristics has been proved to be efficient in every instance of the problems.

Beside different variable ordering heuristics designed to improve the efficiency of search procedure, researchers developed many Look-ahead Value Ordering (LVO) heuristics. This is because value ordering heuristic are powerful way of reducing the efforts of search algorithms [Haralick and Elliott, 1980]. Therefore the constraint programming community developed various LVO heuristics that choose which value to instantiate to the selected variable. Many designed value ordering heuristics attempt to choose next the *least constraining* values, i.e., values that are most likely to succeed. By the way, values that are expected to participate in many solutions are privileged. Minton *et al.* designed a value ordering heuristic, the ***min-conflicts***, that attempts to minimize the number of constraint violations after each step [Minton *et al.*, 1992]. Selecting first ***min-conflicts*** values maximizes the number of values available for future variables. Therefore, partial solutions that cannot be extended will be avoided. Other heuristics try to first select value maximizing the product [Ginsberg *et al.*, 1990; Geelen, 1992] or the sum of support in future domain after propagation [Frost and Dechter, 1995]. Nevertheless, However, all these heuristics are costly. Literature is rich on other LVO, to mention a few [Dechter and Pearl, 1988; Frost and Dechter, 1995; Meisels *et al.*, 1997; Vernooy and Havens, 1999; Kask *et al.*, 2004].

1.3 Distributed constraint satisfaction problems (DisCSP)

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

A Distributed Constraint Satisfaction Problem (DisCSP) is composed of a group of autonomous agents, where each agent has control of some elements of information about the whole problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints are satisfied. Hence,

agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with constraints between agents [Yokoo *et al.*, 1998; Yokoo, 2000a]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them to check consistency of their proposed assignments against constraints that contain variables that belong to other agents.

1.3.1 Preliminaries

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

Definition 1.19 *A* DisCSP *(or a* distributed constraint network*) has been formalized as a tuple* $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$*, where:*

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

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

In the following, we provide some material assumptions in context of DisCSPs. First, we assume a binary distributed constraint network where all constraints are binary constraints (they involve two variables). A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). For simplicity purposes, we consider a restricted version of DisCSPs where each agent controls exactly one variable ($p = n$). Thus, we use the terms agent and variable interchangeably and we identify the agent ID with its variable index. We assume also that each agent (A_i) knows all constraints involving its variable and its ***neighbors*** ($\Gamma(x_i)$) with whom it shares these constraints. We also assume that only the agent who is assigned a variable has control on its value and knowledge of its domain. In this thesis, we adopt the model of communication between agents presented in [Yokoo, 2000b] where it is assumed that:

- agents communicate by exchanging messages,
- the delay in delivering a message is random but finite and
- an agent can communicate with other agents if it knows their addresses.

Initially, each agent knows addresses of all its neighbors without excluding the possibility of getting addresses of other agents if it is necessary. However we discard the FIFO

assumption on communication channels between agents. Hence, we assume that communication between two agents is not necessarily generalized FIFO (aka causal order) channels [Silaghi, 2006].

Almost all distributed algorithms designed for solving distributed CSPs require a total priority order on agents. The total order on agents is denoted by $\mathcal{O}$ (see Definition 1.4). In this thesis, we present two classes of distributed algorithms with regard the agents ordering. The first category of distributed algorithms for solving DisCSPs corresponds to those using a static ordering on agents. The second category of distributed algorithms for solving DisCSPs corresponds to those performing a dynamic reordering of agents during search. For the first category of algorithms and without loss any generality, we will assume that the total order on agents is the lexicographic ordering $[A_1, A_2, \ldots, A_n]$.

For each agent $A_i \in \mathcal{A}$, an agent A_j has a ***higher priority*** than A_i if it appears before A_i in the total ordering on agents. We say that x_j precede x_i in the ordering and we denote this by $x_j \prec x_i$. Conversely, A_j has a ***lower priority*** than A_i if it appears after A_i in the total ordering on agents (i.e., $x_j \succ x_i$). Hence, the higher priority agents are those appearing before A_i in $\mathcal{O}$. Conversely, the lower priority agents are those appearing after A_i. As a results, $\mathcal{O}$ divides the neighbors of A_i, $\Gamma(x_i)$, into ***higher priority neighbors***, $\Gamma^{-}(x_i)$, and ***lower priority neighbors***, $\Gamma^{+}(x_i)$.

Instead of assuming that communication between agents is necessarily FIFO, we adopt a model where each agent (A_i) maintains a counter that is incremented whenever A_i changes its value. The current value of the counter *tags* each generated assignment.

Definition 1.20 *An **assignment** for an agent $A_i \in \mathcal{A}$ is a tuple (x_i, v_i, t_i), where v_i is a value from the domain of x_i and t_i is the tag value. When comparing two assignments, the **most up to date** is the one with the greatest tag t_i. Two sets of assignments $\{(x_{i_1}, v_{i_1}, t_{i_1}), \ldots, (x_{i_k}, v_{i_k}, t_{i_k})\}$ and $\{(x_{j_1}, v_{j_1}, t_{j_1}), \ldots, (x_{j_q}, v_{j_q}, t_{j_q})\}$ are **compatible** if every common variable is assigned the same value in both sets.*

In order to solve distributed CSPs agents exchange their proposals with other agents.

Definition 1.21 *The **AgentView** of an agent $A_i \in \mathcal{A}$ is an array containing the most up to date assignments received from other agents.*

1.3.2 Examples of DisCSPs

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

successfully modeled and solved by the DisCSP framework. We present in the following some instances of these applications.

1.3.2.1 Distributed Meeting Scheduling Problem (DisMSP)

In Section 1.1.2.3, we presented the Meeting Scheduling Problem as a centralized CSP. Nonetheless, it is a problem of a distributed nature. The *Distributed Meeting Scheduling Problem* (DisMSP) is a truly distributed problem where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [**Wallace and Freuder, 2002**; **Meisels and Lavee, 2004**]. The DisMSP involves a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location. Each agent, $A_i \in \mathcal{A}$, knows the set of the k_i among m meetings he/she must attend. It is assumed that each agent knows the traveling time between the locations where his/her meetings will be held. The traveling time between locations where two meetings m_i and m_j will be hold is denoted by $TravellingTime(m_i, m_j)$. Solving the problem consists in satisfying the following constraints: (i) all agents attending a meeting must agree on when it will occur, (ii) an agent cannot attend two meetings at same time, (iii) an agent must have enough time to travel from the location where he/she is to the location where the next meeting will be held.

Distributed Meeting Scheduling Problem (DisMSP) is encoded in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. The domain of each variable contains the $d \times h$ slots where a meeting can be scheduled such that there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. This equality constraint means that all agents attending a meeting must schedule it at the same slot (constraint (i)). There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. The arrival-time constraint between two variables m_i and m_j is defined as follows (Equation 1.2):

$$|m_i - m_j| - duration > TravellingTime(m_i, m_j) \tag{1.2}$$

where $duration$ is the duration of every meeting. This arrival-time constraint allows us to express both constraints (ii) and (iii).

We illustrate in Figure 1.6 the encoding of the instance of the meeting scheduling problem shown in Figure 1.4 in the distributed constraint network formalism. This figure shows 4 agents where each agent has a personal private calendar and a set of meetings each taking place in a specified location. Thus we get the following DisCSP:

- $\mathcal{A} = \{A_1,\ A_2,\ A_3,\ A_4\}$, each agent A_i corresponds to a real agent.
- For each agent $A_i \in \mathcal{A}$ there is a variable m_{ik}, for every meeting m_k that A_i attends, $\mathcal{X} = \{m_{11},\ m_{13},\ m_{14},\ m_{21},\ m_{22},\ m_{32},\ m_{33},\ m_{34},\ m_{44}\}$.
- $\mathcal{D} = \{D(m_{ik}) \mid m_{ik} \in \mathcal{X}\}$ where,
  - $D(m_{11}) = D(m_{13}) = D(m_{14}) = \{s \mid s \text{ is a slot in } calendar(A_1)\}$.
  - $D(m_{21}) = D(m_{22}) = \{s \mid s \text{ is a slot in } calendar(A_2)\}$.

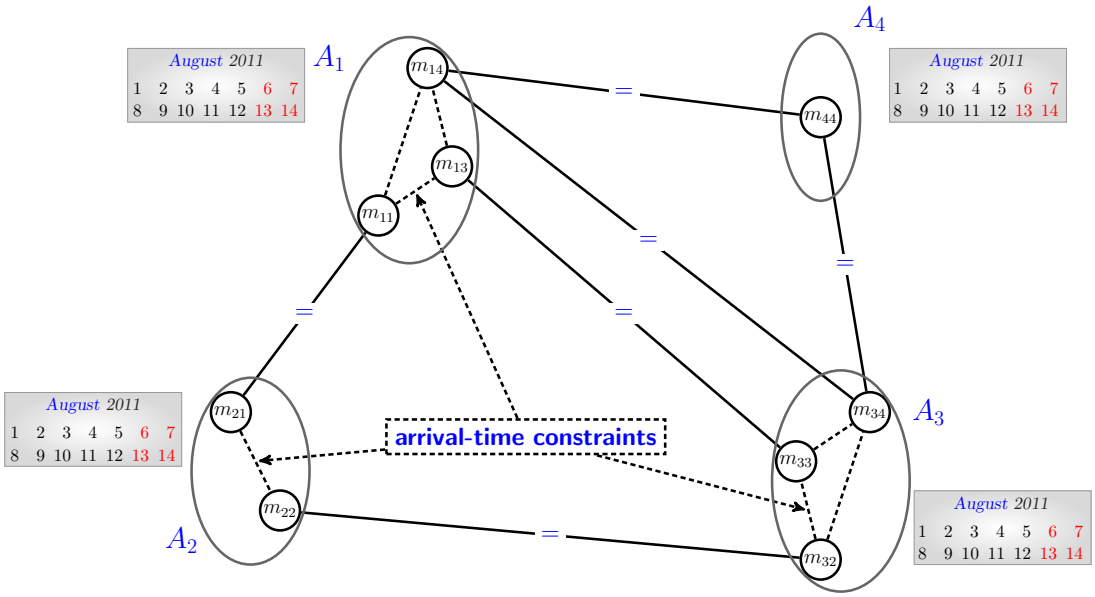


Figure 1.6 – The distributed meeting-scheduling problem modeled as DisCSP.

  - $D(m_{32}) = D(m_{33}) = D(m_{34}) = \{s \mid s \text{ is a slot in } calendar(A_3)\}$.
  - $D(m_{44}) = \{s \mid s \text{ is a slot in } calendar(A_4)\}$.
- For each agent A_i, there is a *private* arrival-time constraint (c^i_{kl}) between every pair of its local variables (m_{ik}, m_{il}). For each two agents A_i, A_j that attend the same meeting m_k there is an equality inter-agent constraint (c^{ij}_k) between the variables m_{ik} and m_{jk}, corresponding to the meeting m_k on agent A_i and A_j. Then, $\mathcal{C} = \{c^i_{kl}, c^{ij}_k\}$.

1.3.2.2 Distributed Sensor Network Problem (SensorDCSP)

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

Figure 1.7 illustrates an instance of the SensorDCSP problem. This example includes 25 sensors (blue circular disks) placed on a grid of 5×5 and 5 targets (red squares) to be tracked. Figure 1.7 illustrates the visibility constraints, that is, the set of sensors to which a target is visible. Two sensors are compatible if and only if they are in sensing range of each other.

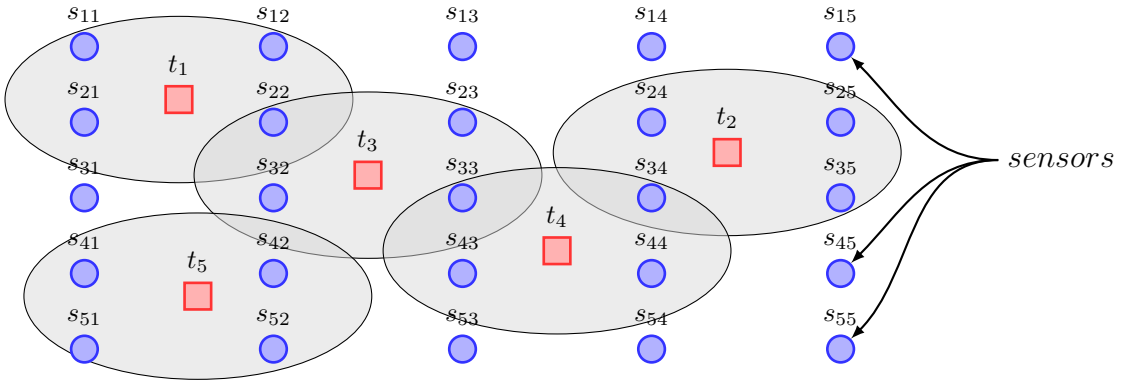


Figure 1.7 – An instance of the distributed sensor network problem.

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

- $S = \{s_1, \ldots, s_n\}$ is a set of n sensors;
- $T = \{t_1, \ldots, t_m\}$ is a set of m targets;

Each agent represents one target (i.e., $\mathcal{A} = T$). There are three variables per agent, one for each sensor that we need to allocate to the corresponding target. The domain of each variable is the set of sensors that can detect the corresponding target (the visibility constraint defines such sensors). The intra-agent constraints between the variables of one agent (target) specify that the three sensors assigned to the target must be distinct and pairwise compatible. The inter-agent constraints between the variables of different agents specify that a given sensor can be selected by at most one agent.

1.4 Methods for solving distributed CSPs

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

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

In synchronous search algorithms for solving DisCSPs, agents sequentially assign their

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

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

1.4.1 Synchronous search algorithms on DisCSPs

Synchronous Backtracking (SBT) is the simplest search algorithm for solving DisCSPs [**Yokoo, 2000b**]. SBT is a straightforward extension of the chronological Backtracking algorithm for centralized CSPs (Section 1.2.1.1). SBT requires a total order on which agents will be instantiated. Following this ordering, agents perform assignments sequentially and synchronously. When an agent receives a partial solution from its predecessor, it assigns its variable a value satisfying constraints it knows. If it succeeds in finding such value, it extends the partial solution by adding its assignment on it and passes it on to its successor. When no value is possible for its variable, then it sends a *backtracking* message to its predecessor. In SBT, only the agent holding the current partial assignment (CPA) performs an assignment or backtrack.

Zivan and Meisels (**2003**) proposed the *Synchronous Conflict-Based Backjumping* (SCBJ), a distributed version of the centralized (CBJ) algorithm [**Prosser, 1993**], see Section 1.2.1.2. While SBT performs chronological backtracking, SCBJ performs backjumping. Each agent A_i keep the conflict set (CS_i). When a wipe-out occurs on its domain, a jump is performed to the closest conflict variable in CS_i. The *backjumping* message will contain CS_i. When an agent receives a backjumping message, it discards it current value, and updates its conflict set to be the union of its old conflict-set and the one received.

Extending SBT, **Meisels and Zivan** (**2007**) proposed the Asynchronous Forward-Checking (AFC). Besides assigning variables sequentially as is done in SBT, agents in AFC perform forward checking (FC [**Haralick and Elliott, 1980**]) asynchronously. The key here is that each time an agent succeeds to extend the current partial assignment (by assigning its variable), it sends the CPA to its successor and copies of this CPA to all agents whose assignments are not yet on the CPA. When an agent receives a copy of the CPA, it performs the forward checking phase. In the forward checking phase all inconsistent values with assignments on the received CPA are removed. The forward checking operation is performed asynchronously where comes the name of the algorithm. When an agent generates an empty domain as a result of a forward checking, it informs all agents with unassigned

variables on the (inconsistent) CPA. Afterwards, one of these agents will receive the CPA and will backtrack. Thereby, only one backtrack can be generated for a given CPA.

1.4.1.1 Asynchronous Forward-Checking (AFC)

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

Definition 1.22 *A **current partial assignment**, CPA, is an ordered set of assignments* $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. *Two CPAs are **compatible** if every common variable is assigned the same value in both CPAs.*

Each CPA is associated with a counter that is updated by each agent when it succeeds in assigning its variable on the CPA. This counter, called *Step-Counter* (SC), acts as a timestamp for the CPA. In AFC algorithm, each agent stores the current assignments state of its higher priority agents on the AgentView. The AgentView of an agent $A_i \in \mathcal{A}$ has a form similar to a CPA. The AgentView contains a consistency flag, *AgentView.Consistent*, that represents whether the partial assignment it holds is consistent. The pseudo-code of AFC algorithm executed by a generic agent A_i is shown in Algorithm 1.5.

Agent A_i starts the search by calling procedure `AFC()` in which it initializes counters to 0. Next, if A_i is the initializing agent IA (the first agent in the agent ordering, $\mathcal{O}$), it initiates the search by calling procedure `Assign()` (line 2). Then, a loop considers the reception and the processing of the possible message types. Thus, agents wait for messages, and then call the procedures dealing with the relevant type of message received.

When calling procedure `Assign()` A_i tries to find an assignment, which is consistent with its AgentView. If A_i fails to find a consistent assignment, it calls procedure `Backtrack()` (line 13). If A_i succeeds, it generates a CPA from its AgentView augmented by its assignment, increments the Step Counter *SC* (lines 10-11), and then calls procedure `SendCPA(CPA)` (line 12). If the CPA includes all agents assignments (A_i is the lowest agent in the ordering, line 14), A_i reports the CPA as a solution of the problem and marks the *end* flag *true* to stop the main loop (line 15). Otherwise, A_i sends forward the CPA to every agent whose assignments are not yet on the CPA (line 17). Next agent on the ordering (successor, A_{i+1}) will receive the CPA in a ***cpa*** message and then will try to extend this CPA by assigning its variable on it (line 17). Other unassigned agents will receive the CPA, generated by A_i, in ***fc_cpa*** (line 18). Therefore, these agents will perform the forward-checking phase asynchronously to check its consistency.

A_i calls procedure `Backtrack()` when it is holding the CPA in one of two cases. Either A_i can not find a consistent assignment for its variable (line 13), or its AgentView is inconsistent and is found to be compatible with the received CPA (line 30). If A_i is the initializing agent IA, line 19, this means that the problem is unsolvable. A_i ends then the search by marking the *end* flag *true* to stop the main loop and sending a ***stp*** message to all agent informing them that search ends unsuccessfully (line 19). Other agents performing a

Algorithm 1.5: The AFC algorithm running by agent A_i.

```
procedure AFC()
01. v_i ← empty; t_i ← 0; SC ← 0; end ← false; AgentView.Consistent ← true;
02. if ( A_i = IA ) then Assign();
03. while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.       cpa      : ProcessCPA(msg);        fc_cpa : ProcessFCCPA(msg.CPA);
07.       back_cpa : ProcessCPA(msg);        not_ok : ProcessNotOk(msg.CPA);
08.       stp      : end ← true;

procedure Assign()
09. if ( D(x_i) ≠ ∅ ) then
10.    v_i ← ChooseValue(); t_i ← t_i + 1 ;
11.    CPA ← {AgentView ∪ myAssig};   CPA.SC ← AgentView.SC + 1 ;
12.    SendCPA(CPA) ;
13. else Backtrack();

procedure SendCPA(CPA)
14. if ( A_i is the last agent in O ) then
15.    end ← true;  broadcastMsg: stp(CPA)
16. else
17.    sendMsg: cpa(CPA) to A_{i+1} ;                    /* A_{i+1} is the agent next A_i */
18.    foreach ( A_k ≻ A_{i+1} ) do sendMsg: fc_cpa(CPA) to A_k ;

procedure Backtrack()
19. if ( A_i = IA ) then  end ← true;  broadcastMsg: stp() ;
20. else
21.    AgentView ← shortest inconsistent partial assignment ;
22.    AgentView.Consistent ← false;
23.    sendMsg: back_cpa(AgentView) to A_j ;     /* A_j denotes the last agent on AgentView */

procedure ProcessCPA(msg)
24. CheckConsistencyOfAgentView(msg.CPA) ;
25. if ( AgentView.Consistent ) then
26.    if ( msg.Sender ≻ x_i ) then store msg.CPA as justification of v_i removal ;
27.    else UpdateAgentView(msg.CPA) ;
28.    Assign() ;

procedure CheckConsistencyOfAgentView(CPA)
29. if ( ¬AgentView.Consistent ) then
30.    if ( AgentView ⊆ CPA ) then Backtrack() ;
31.    else AgentView.Consistent ← true ;

procedure UpdateAgentView(CPA)
32. AgentView ← CPA; AgentView.SC ← CPA.SC ;
33. foreach ( v ∈ D(x_i) such that ¬isConsistent(v, CPA ) ) do
34.    store the shortest inconsistent partial assignment as justification of v removal;

procedure ProcessFCCPA(CPA)
35. if ( CPA.SC > AgentView.SC ) then
36.    if ( ¬AgentView.Consistent ) then
37.       if ( ¬AgentView ⊆ CPA ) then AgentView.Consistent ← true ;
38.    if ( AgentView.Consistent ) then
39.       UpdateAgentView(CPA) ;
40.       if ( D(x_i) = ∅ ) then
41.          sendMsg: not_ok(CPA) to unassigned agents on AgentView ;

procedure ProcessNotOk(CPA)
42. if ( CPA ⊆ AgentView ∨ (AgentView ⊈ CPA ∧ CPA.SC > AgentView.SC) ) then
43.    AgentView ← msg.CPA ;
44.    AgentView.Consistent ← false ;
```

backtrack operation, copy to their AgentView the shortest inconsistent partial assignment (line 21) and set its flag to *false*. Next, they send the AgentView back to the agent which is the owner of the last variable in the inconsistent partial assignment (line 23).

Whenever A_i receives a ***cpa*** or a ***back_cpa*** messages, procedure `ProcessCPA()` is called. A_i then checks the consistency of its AgentView (procedure `CheckConsistencyOfAgentView` call, line 24). If the AgentView is not consistent and it is a subset of the received CPA, this means that A_i has to backtrack (line 30). If the AgentView is not consistent and not a subset of the received CPA, A_i marks its AgentView consistent by setting *AgentView.Consistent* flag to *true* (line 31). Afterwards, A_i checks the consistency of its AgentView. If it is the case, A_i calls procedure `Assign()` to assign its variable (line 28) once it removes its current value v_i storing the received *CPA* as a justification of its removal if the received message is a ***back_cpa*** message (line 26), or it updates its AgentView if the received message is a ***cpa*** message (line 27). When calling procedure `UpdateAgentView`, A_i sets its AgentView to the received CPA and the step counter of its AgentView to that associated to the received CPA (line 32). Then, A_i performs the forward-checking to remove from its domain all values inconsistent with the received CPA (lines 33-34).

Whenever a ***fc_cpa*** message is received, A_i calls procedure `ProcessFCCPA(`*msg*`)` to process it. If the *SC* associated to the received CPA is less than or equal that of the AgentView, this message is ignored since it is obsolete. Otherwise, A_i set its AgentView to be consistent if it was not consistent and it is not included in the received CPA (line 37). Afterwards, A_i checks the consistency of its AgentView. If it is the case, it calls procedure `UpdateAgentView` to perform the forward-checking (line 39). When an empty domain is generated as result of the forward-checking phase, A_i initiates a backtrack process by sending ***not_ok*** messages to all agents with unassigned variables on the (inconsistent) CPA (line 41). ***not_ok*** messages carry the shortest inconsistent partial assignment which caused the empty domain.

When an agent A_i receives the ***not_ok*** message (procedure `ProcessNotOk(`*msg*`)`), it checks the relevance of the CPA carried in the received message with its AgentView. If the received CPA is relevant, A_i replaces its AgentView by the content of the ***not_ok*** message and set it to be inconsistent (lines 43-44)

1.4.2 Asynchronous search algorithms on DisCSPs

Several distributed asynchronous algorithms for solving DisCSPs have been developed, among which Asynchronous Backtracking (ABT) is the central one.

1.4.2.1 Asynchronous Backtracking (ABT)

The first complete asynchronous search algorithm for solving DisCSPs is the *Asynchronous Backtracking* (ABT) [**Yokoo *et al.*, 1992**; **Yokoo, 2000a**; **Bessiere *et al.*, 2005**]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Agents do not have to wait for decisions of others but they are subject to a total (priority) order. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors. When an agent assigns a value to its

variable, the selected value is sent to lower priority neighbors. When no value is possible for a variable, the inconsistency is reported to higher agents in the form of a nogood (see Definition 1.13). ABT computes a solution (or detects that no solution exists) in a finite time. To be complete, ABT requires a total ordering on agents. The total ordering on agents is static.

The required total ordering on agents in ABT provides a directed acyclic graph. Constraints are then directed according to the total order among agents. Hence, a directed link between each two constrained agents is established. ABT uses this structure between agent to perform the asynchronous search. Thus, the agent from which a link departs is the value-sending agent and the agent to which the link arrives is the constraint-evaluating agent. The pseudo-code executed by a generic agent $A_i \in \mathcal{A}$ is presented in Algorithm 1.6.

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

***ok?*:** A_i informs a lower priority neighbor about its assignment.

***ngd*:** A_i informs a higher priority neighbor of a new nogood.

***adl*:** A_i requests a higher priority agent to set up a link.

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

In the main procedure `ABT()`, each agent selects a value and informs other agents (`CheckAgentView` call, line 2). Then, a loop receives and processes messages (line 3-7). `CheckAgentView` checks if the current value (v_i) is consistent with AgentView. If v_i is inconsistent with assignments of higher priority neighbors, A_i tries to select a consistent value (`ChooseValue` call, line 9). In this process, some values from $D(x_i)$ may appear as inconsistent. Thus, nogoods justifying their removal are added to the NogoodStore of A_i (line 39). When two nogoods are possible for the same value, A_i selects the best with the *Highest Possible Lowest Variable* heuristic [**Hirayama and Yokoo, 2000**; **Bessiere *et al.*, 2005**]. If a consistent value exist, it is returned and then assigned to x_i. Then, A_i notifies its new assignment to all agents in $\Gamma^+(x_i)$ through ***ok?*** messages (line 11). Otherwise, A_i has to backtrack (procedure `Backtrack()` call, line 12).

Whenever it receives an ***ok?*** message, A_i processes it by calling procedure `ProcessInfo(`*msg*`)`. The AgentView of A_i is updated (`UpdateAgentView` call, line 13) only if the received message contains an assignment more up to date than that already stored for the sender (line 16) and all nogoods becomes non compatible with the AgentView of A_i are removed (line 18). Then, a consistent value for A_i is searched after the change in the AgentView (`CheckAgentView` call, line 14).

When every value of A_i is forbidden by its NogoodStore, procedure `Backtrack()` is called. In procedure `Backtrack()`, A_i resolves its nogoods, deriving a new nogood, *newNogood* (line 19). If *newNogood* is empty, the problem has no solution. A_i broadcasts the ***stp*** messages to all agents and terminates the execution (line 20). Otherwise, the new nogood is sent in a ***ngd*** message to the agent, say A_j, owning the variable appearing in its

Algorithm 1.6: The ABT algorithm running by agent A_i.

```
procedure ABT()
01. v_i ← empty; t_i ← 0; end ← false;
02. CheckAgentView() ;
03. while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.       ok?   : ProcessInfo(msg);        ngd  : ResolveConflict(msg);
07.       adl   : AddLink(msg);            stp  : end ← true;
procedure CheckAgentView()
08. if ( ¬isConsistent(v_i, AgentView) ) then
09.    v_i ← ChooseValue() ;
10.    if ( v_i ≠ empty ) then
11.       foreach ( child ∈ Γ+(x_i) ) do sendMsg: ok?( myAssig⟨x_i, v_i, t_i⟩ ) ;
12.    else Backtrack() ;
procedure ProcessInfo(msg)
13. UpdateAgentView(msg.Assig) ;
14. CheckAgentView() ;
procedure UpdateAgentView(newAssig)
15. if ( newAssig.tag > AgentView[j].tag ) then          /* x_j ∈ newAssig */
16.    AgentView[j] ← newAssig ;
17. foreach ( ng ∈ myNogoodStore ) do
18.    if ( ¬Compatible(lhs(ng), AgentView) ) then remove(ng,myNogoodStore) ;
procedure Backtrack()
19. newNogood ← solve(myNogoodStore) ;
20. if ( newNogood = empty ) then end ← true; sendMsg: stp(system);
21. else
22.    sendMsg: ngd(newNogood) to A_j ;   /* Let x_j denote the variable on rhs(newNogood) */
23.    UpdateAgentView(x_j ← empty) ;
24.    CheckAgentView() ;
procedure ResolveConflict(msg)
25. if ( ¬Compatible(lhs(msg.Nogood), AgentView) ) then
26.    CheckAddLink(msg.Nogood);
27.    add(msg.Nogood, myNogoodStore) ;
28.    CheckAgentView();
29. else if ( rhs(msg.Nogood).Value = v_i ) then
30.    sendMsg: ok?(myAssig) to msg.Sender ;
procedure CheckAddLink(nogood)
31. foreach ( x_j ∈ lhs(nogood) \ Γ−(x_i) ) do
32.    add(x_j = v_j, AgentView) ;
33.    Γ−(x_i) ← Γ−(x_i) ∪ {x_j} ;
34.    sendMsg: adl(x_j = v_j) to A_j ;
procedure AddLink(msg)
35. add(msg.Sender, Γ+(x_i)) ;
36. if ( v_i ≠ msg.Assig.Value ) then sendMsg: ok?(myAssig) to msg.Sender ;
function ChooseValue()
37. foreach ( v ∈ D(x_i) ) do
38.    if ( isConsistent(v, AgentView ) ) then return v;
39.    else store the best nogood for v ;
40. return empty;
```

`rhs` (line 22). Then, the assignment of x_j is deleted from the AgentView (`UpdateAgentView` call, line 23). Finally, a new consistent value is selected (`CheckAgentView` call, line 24).

Whenever A_i receives a ***ngd*** message, procedure `ResolveConflict` is called. The nogood included in the ***ngd*** message is accepted only if its *lhs* is compatible with assignments on the AgentView of A_i. Next, A_i calls procedure `CheckAddLink` (line 26). In procedure, `CheckAddLink()` the assignments in the received nogood for variables not directly linked with A_i are taken to update the AgentView (line 32) and a request for a new link is sent to agents owning these variables (line 34). Next, the nogood is stored, acting as justification for removing the value on its *rhs* (line 27). A new consistent value for A_i is then searched (`CheckAgentView` call, line 28) if the current value was removed by the received nogood. If the nogood is not compatible with the AgentView, it is discarded since it is obsolete. However, if the value of x_i was correct in the received nogood, A_i re-sends its assignment to the nogood sender by an ***ok?*** message (lines 29-30).

When a link request is received, A_i calls procedure `AddLink`(msg). Then, the sender is included in $\Gamma^+(x_i)$ (line 35). Afterwards, A_i sends its assignment through an ***ok?*** message to the sender if its value is different than that included in the received *msg* (line 36).

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

ABT **(*Adding links during search*):** In ABT, presented above, new links between unrelated agents may be added during search. A link is requested by an agent when it receives a ***ngd*** message containing unrelated agents in the ordering. New links are permanent. These links are used to remove obsolete information stored by a given agent.

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

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

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

Figure 1.8 illustrates an example of Asynchronous Backtracking algorithm execution on a simple instance (Figure 1.8(a)). This instance includes three agents, each holding one variable (x_1, x_2 and x_3). There domains are respectively $\{1,2\}$, $\{2\}$ and $\{1,2\}$. This instance includes two constraints $x_1 \neq x_3$ and $x_2 \neq x_3$. In Figure 1.8(b), by receiving ***ok?*** messages from x_1 and x_2, the AgentView of x_3 will be $[x_1 = 1, x_2 = 2]$. These assignments remove values 1 and 2 from $D(x_3)$ storing two nogoods as justification of there removal

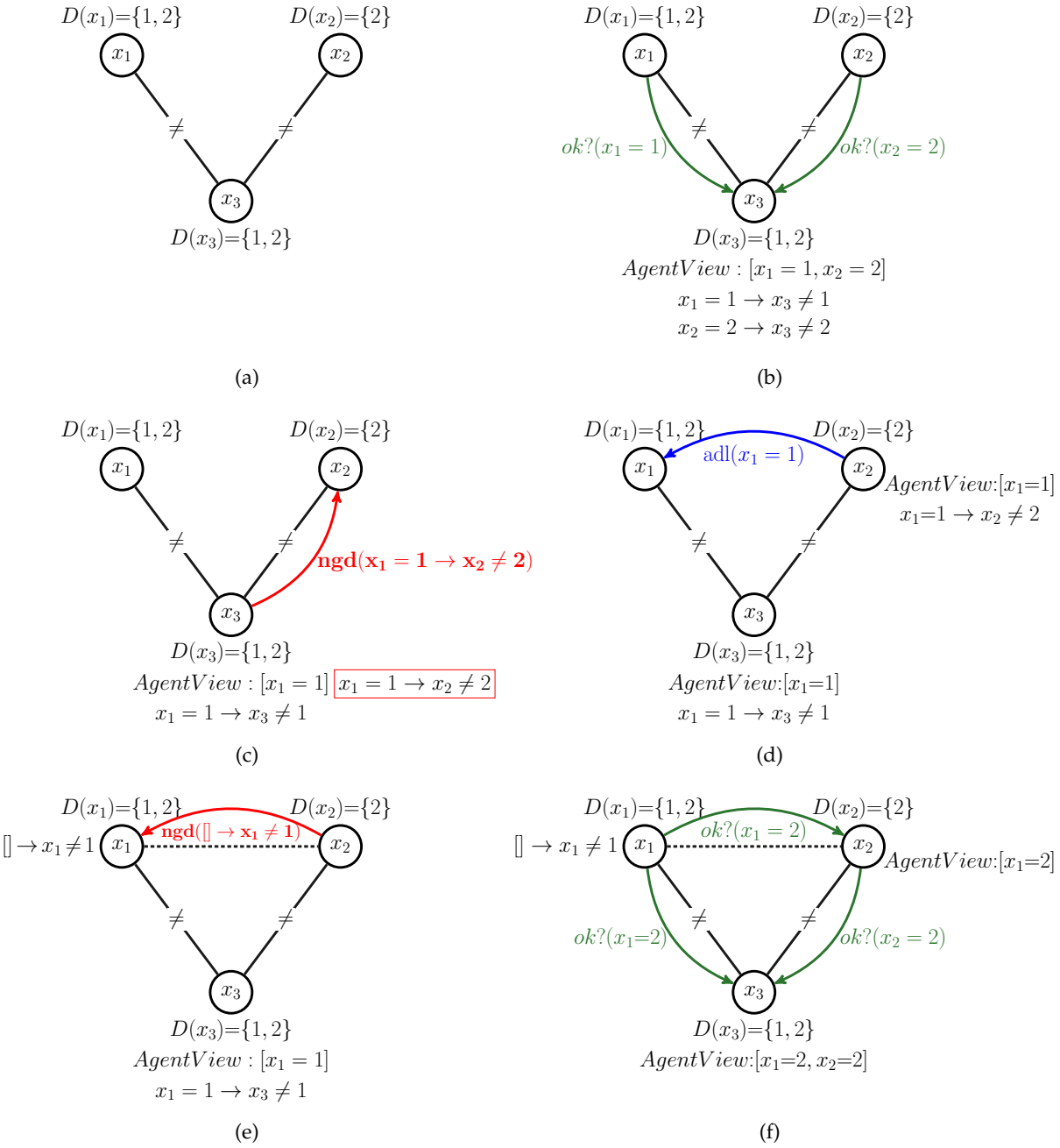


Figure 1.8 – An example of Asynchronous Backtracking execution.

(i.e., x_1=1 → $x_3 \neq 1$ respectively, x_2=2 → $x_3 \neq 2$). Since there is no possible value consistent with its AgentView, agent x_3 resolves its nogoods producing a new nogood (x_1=1 → $x_2 \neq 2$) (Figure 1.8(c)). This nogood is then sent to x_2 in ***ngd*** message. By receiving this ***ngd*** message, agent x_2 records this nogood. This nogood contains assignment of agent x_1, which is not connected to x_2 by a link. Therefore, agent x_2 requests a new link between him and x_1 by sending an ***adl*** message (Figure 1.8(d)). Agent x_2 checks whether its value is consistent with its AgentView ($[x_1 = 1]$). Since its only value 2 is removed by the nogood received from x_3, agent x_2 resolves its NogoodStore producing a new nogood, $[] \rightarrow x_1 \neq 1$. This nogood is then sent to agent x_1 (Figure 1.8(e)). This nogood will lead x_1 to change its current value to 1, and henceforth it will send its assignment on an ***ok?*** message to both

agent x_2 and x_3. Simultaneously, agent x_2 assigns its variable and then sends its assignment to its lower priority neighbor x_3. Hence, we get the situation shown in Figure 1.8(f).

1.4.3 Dynamic Ordering Heuristics on DisCSPs

In algorithms presented above for solving DisCSPs, the total ordering on agents is static. Therefore, a single mistake on the order is very penalizing. Moreover, it is known from centralized CSPs that dynamic reordering of variables during search drastically fastens the search procedure (see Section 1.2.2.2). Many attempts were made to apply this principle for improving distributed constraint algorithms.

The first reordering algorithm for DisCSP is the Asynchronous Weak Commitment (AWC) [**Yokoo, 1995**]. Asynchronous Weak Commitment (AWC) dynamically reorders agents during search by moving the sender of a nogood higher in the order than the other agents in the nogood. Whenever a wipe-out occurs on the domain of a variable x_i, the total agent ordering is revised so as to assign the highest priority to the agent x_i. While AWC was shown to outperform ABT empirically on small problems, contrary to ABT, the AWC algorithm requires an exponential space for storing all generated nogoods.

Silaghi *et al.* (**2001c**) later proposed Asynchronous Backtracking with Reordering (ABTR) an attempt to hybridize ABT with AWC. Abstract agents fulfill the reordering operation to guarantee a finite number of asynchronous reordering operations. Asynchronous Backtracking with Reordering is the first asynchronous complete algorithm with polynomial space requirements that enables a largest number of reordering heuristics in asynchronous search. However, to achieve this the position of first agent on the ordering had to be fix. A dynamic variable reordering heuristic for ABTR that exactly mimics the one employed in centralized Dynamic Backtracking [**Ginsberg, 1993**] and that requires no exchange of heuristic messages was presented in [**Silaghi, 2006**].

Zivan and Meisels (**2006a**) proposed Dynamic Ordering for Asynchronous Backtracking (ABT_DO aka ABTR). ABT_DO is a simple algorithm for dynamic ordering in Asynchronous Backtracking search. Agents choose orders dynamically and asynchronously while keeping space complexity polynomial. When an ABT_DO agent changes its assignment, it can reorder all agents with lower priority. **Zivan and Meisels** proposed three different ordering heuristics in ABT_DO. In the best of those heuristics called *Nogood-triggered heuristic*, inspired by dynamic backtracking [**Ginsberg, 1993**], the agent that generates a nogood is placed in front of all other lower priority agents.

A new kind of ordering heuristics for ABT_DO is presented in [**Zivan *et al.*, 2009**]. These heuristics, called retroactive heuristics, enable the generator of the nogood to be moved to a higher position than that of the target of the backtrack. The degree of flexibility of these retroactive heuristics depends on a parameter K. K defines the level of flexibility of the heuristic with respect to the amount of information an agent can store in its memory. Agents that detect a dead end move themselves to a higher priority position in the order. If the length of the nogood generated is not larger than K then the agent can move to any position it desires (even to the highest priority position) and all agents that are included in the nogood are required to add the nogood to their set of constraints and hold it until the

algorithm terminates. Since agents must store nogoods that are smaller than or equal to K, the space complexity of agents is exponential in K. If the size of the created nogood is larger than K, the agent that generated the nogood can move up to the place that is right after the second last agent in the nogood.

The best retroactive heuristic introduced in [**Zivan *et al.*, 2009**] is called ABT_DO-Retro-MinDom. This heuristic does not require any additional storage (i.e., $K = 0$). In this heuristic, the agent that generates a nogood is placed in the new order between the last and the second last agents in the generated nogood. However, the generator of the nogood moves to a higher priority position than the backtracking target (the agent the nogood was sent to) only if its domain is smaller than that of the agents it passes on the way up. Otherwise, the generator of the nogood is placed right after the last agent with a smaller domain between the last and the second last agents in the nogood.

1.4.4 Maintaining Arc Consistency on DisCSPs

Although, its success for solving centralized CSPs was empirically demonstrated, the Maintenance of Arc Consistency (MAC) has not yet been well investigated in distributed CSPs. **Silaghi *et al.* (2001b)** introduced the Distributed Maintaining Asynchronously Consistency for ABT, DMAC-ABT, the first algorithm able to maintain arc consistency in distributed CSPs [**Silaghi *et al.*, 2001b**]. DMAC-ABT considers consistency maintenance as a hierarchical nogood-based inference. However, the improvement obtained on ABT was minor.

Brito and Meseguer (2008) proposed ABT-uac and ABT-dac, two algorithms that connect ABT with arc consistency [**Brito and Meseguer, 2008**]. The first algorithm they propose, ABT-uac, propagates unconditionally deleted values (i.e., values removed by a nogood having an empty left-hand side) to enforce an amount of full arc consistency. The intuitive idea behind ABT-uac is since unconditionally deleted values are removed once and for all, their propagation may causes new deletions in the domains of other variables. Thus, the search effort required to solve the DisCSP can be reduced. The second algorithm they propose, ABT-dac, extends the first one in order to propagate conditionally and unconditionally deleted values using directional arc consistency. ABT-uac shows minor improvement in communication load and ABT-dac is harmful in many instances.

1.5 Summary

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

2 Nogood based Asynchronous Forward Checking (AFC-ng)

Contents

This chapter introduces our first contribution, that is, an asynchronous algorithm for solving Distributed Constraint Satisfaction Problems (DisCSPs). Our algorithm is a nogood-based version of Asynchronous Forward Checking (AFC). We call it Nogood-Based Asynchronous Forward Checking (AFC-ng). Besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks going from different agents to different destinations. We prove that AFC-ng only needs polynomial space. We compare the performance of our contribution with other DisCSP algorithms on random DisCSPs and instances from real benchmarks: sensor networks and distributed meeting scheduling. Our experiments show that AFC-ng improves on AFC.

This chapter is organized as follows. Section 2.1 introduces our algorithm by briefly recalling necessary background on the AFC algorithm. Our Nogood-Based Asynchronous Forward Checking is described in Section 2.2. Correctness proofs are given in Section 2.3. Section 2.4 presents an experimental evaluation of our proposed algorithm against other well-known distributed algorithms. Section 2.5 summarizes several related works and we conclude the chapter in Section 2.6.

2.1 Introduction

As seen in Section 1.4.1 Asynchronous Forward-Checking (AFC) incorporates the idea of the forward-checking (FC) algorithm for centralized CSP [**Haralick and Elliott, 1980**]. However, agents perform the forward checking phase asynchronously [**Meisels and Zivan, 2003**; **Meisels and Zivan, 2007**]. As in synchronous backtracking, agents assign their variables only when they hold the current partial assignment (***cpa***). The ***cpa*** is a unique message (token) that is passed from one agent to the next one in the ordering. The ***cpa*** message carries the partial assignment (CPA) that agents attempt to extend into a complete solution by assigning their variables on it. When an agent succeeds in assigning its variable on the CPA, it sends this CPA to its successor. Furthermore, copies of the CPA are sent to all agents whose assignments are not yet on the CPA. These agents perform the forward checking asynchronously in order to detect as early as possible inconsistent partial assignments. The forward-checking process is performed as follows. When an agent receives a CPA, it updates the domain of its variable, removing all values that are in conflict with assignments on the received CPA. Furthermore, the shortest CPA producing the inconsistency is stored as justification of the value deletion.

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

The manner in which the backtrack operation is performed is a major drawback of the AFC algorithm. The backtrack operation requires a lot of work from the agents. An improved backtrack method for AFC was described in Section 6 of [**Meisels and Zivan, 2007**]. Instead of just sending ***not_ok*** messages to all agents unassigned in the CPA, the agent who detects the empty domain can itself initiate a backtrack operation. It sends a backtrack message to the last agent assigned in the inconsistent CPA in addition to the ***not_ok*** messages to all agents not instantiated in the inconsistent CPA. The agent who receives a backtrack message generates (if it is possible) a new CPA that will dominate older ones thanks to a time-stamping mechanism.

We present in this chapter the Nogood-based Asynchronous Forward Checking (AFC-ng), an algorithm for solving DisCSPs based on Asynchronous Forward Checking (AFC). Instead of using the shortest inconsistent partial assignments we use nogoods as justifications of value removals. Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different des-

tinations. As a result, several CPAs could be generated simultaneously by the destination agents. Thanks to the timestamps integrated in the CPAs, the *strongest* CPA coming from the highest level in the agent ordering will eventually dominate all others. Interestingly, the search process with the strongest CPA will benefit from the computational effort done by the (killed) lower level processes. This is done by taking advantage from nogoods recorded when processing these lower level processes.

2.2 Nogood-based Asynchronous Forward Checking

The nogood-based Asynchronous Forward-Checking (AFC-ng) is based on the Asynchronous Forward Checking (AFC). AFC-ng tries to enhance the asynchronism of the forward checking phase. The two main features of AFC-ng are the following. First, it uses the nogoods as justification of value deletions. Each time an agent performs a forward-check, it revises its *initial domain*, (including values already removed by a stored nogood) in order to store the best nogoods for removed values (one nogood per value). When comparing two nogoods eliminating the same value, the nogood with the *Highest Possible Lowest Variable* involved is selected (HPLV heuristic) [**Hirayama and Yokoo, 2000**]. As a result, when an empty domain is found, the resolvent nogood contains variables as high as possible in the ordering, so that the backtrack message is sent as high as possible, thus saving unnecessary search effort [**Bessiere *et al.*, 2005**].

Second, each time an agent A_i generates an empty domain it no longer sends ***not_ok*** messages. It resolves the nogoods ruling out values from its domain, producing a new nogood *newNogood*. *newNogood* is the conjunction of the left hand sides of all nogoods stored by A_i. Then, A_i sends the resolved nogood *newNogood* in a ***ngd*** (backtrack) message to the lowest agent in *newNogood*. Hence, multiple backtracks may be performed at the same time coming from different agents having an empty domain. These backtracks are sent concurrently by these different agents to different destinations. The reassignment of the destination agents then happen simultaneously and generate several CPAs. However, the strongest CPA coming from the highest level in the agent ordering will eventually dominate all others. Agents use the timestamp (see Definition 2.1) to detect the strongest CPA. Interestingly, the search process of higher levels with stronger CPAs can use nogoods reported by the (killed) lower level processes, so that it benefits from their computational effort.

2.2.1 Description of the algorithm

In the Asynchronous Forward-Checking only the agent holding the current partial assignment, CPA (Definition 1.22) can perform an assignment or backtracks. In order to enhance the asynchronism of the forward-checking phase, unlike AFC, the nogood-based Asynchronous Forward-Checking algorithm (AFC-ng) allows simultaneous backtracks going from different agents to different destinations. The reassignments of the destination agents then happen simultaneously and generate several CPAs. For allowing agents to

simultaneously propose new CPAs, they must be able to decide which CPA to select. We propose that the priority between the CPAs is based on *timestamp*.

Definition 2.1 *A **timestamp** associated with a CPA is an ordered list of counters $[t_1, t_2, \ldots, t_i]$ where t_j is the tag of the variable x_j. When comparing two CPAs, the **strongest** one is that associated with the lexicographically greater timestamp. That is, the CPA with greatest value on the first counter on which they differ, if any, otherwise the longest one.*

Based on the timestamp associated with each CPA, now agents can detect the strongest CPA. Therefore, the strongest CPA coming from the highest level in the agent ordering will eventually dominate all others.

Each agent $A_i \in \mathcal{A}$ executes the pseudo-code shown in Algorithm 2.1. Each agent A_i stores a nogood per removed value in the NogoodStore. The other values not ruled out by a nogood form $D(x_i)$, the current domain of x_i. Moreover, A_i keeps an AgentView that stores the most up to date assignments received from higher priority agents in the agent ordering. It has a form similar to a current partial assignment CPA (see, Definition 1.22) and is initialized to the set of empty assignments $\{(x_j, empty, 0) \mid x_j \prec x_i\}$.

Agent A_i starts the search by calling procedure `AFC-ng()` in which it initializes its AgentView (line 1) by setting counters to zero (line 9). The AgentView contains a consistency flag that represents whether the partial assignment it holds is consistent. If A_i is the initializing agent IA (the first agent in the agent ordering), it initiates the search by calling procedure `Assign()` (line 2). Then, a loop considers the reception and the processing of the possible message types (lines 3-8). In AFC-ng, agents exchange the following types of messages (where A_i is the sender):

cpa A_i passes on the current partial assignment (CPA) to a lower priority agent. According to its position on the ordering, the receiver will try to extend the CPA (when it is the next agent on the ordering) or perform the forward-checking phase.

ngd A_i reports the inconsistency to a higher priority agent. The inconsistency is reported by a nogood.

stp A_i informs agents either if a solution is found or the problem is unsolvable.

When calling `Assign()` A_i tries to find an assignment, which is consistent with its AgentView. If A_i fails to find a consistent assignment, it calls procedure `Backtrack()` (line 14). If A_i succeeds, it increments its counter t_i and generates a CPA from its AgentView augmented by its assignment (line 12). Afterwards, A_i calls procedure `SendCPA(CPA)` (line 13). If the CPA includes all agents assignments (A_i is the lowest agent in the order, line 15), A_i reports the CPA as a solution of the problem and marks the *end* flag true to stop the main loop (line 16). Otherwise, A_i sends forward the CPA to every agent whose assignments are not yet on the CPA (line 17). So, the next agent on the ordering (successor) will try to extend this CPA by assigning its variable on it while other agents will perform the forward-checking phase asynchronously to check its consistency.

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA(msg)` is called (line 6). A_i checks its AgentView status. If it is not consistent and the AgentView is a subset of the received CPA, this means that A_i has already backtracked, then A_i does nothing (line 18).

Algorithm 2.1: Nogood-based AFC (AFC-ng) algorithm running by agent A_i.

```
procedure AFC-ng()
01. end ← false; AgentView.Consistent ← true; InitAgentView();
02. if ( A_i = IA ) then Assign() ;
03. while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.       cpa  : ProcessCPA(msg);
07.       ngd  : ProcessNogood(msg);
08.       stp  : end ← true; if ( msg.CPA ≠ ∅ ) then solution ← msg.CPA ;
procedure InitAgentView()
09. foreach ( x_j ≺ x_i ) do AgentView[j] ← {(x_j, empty, 0)} ;
procedure Assign()
10. if ( D(x_i) ≠ ∅ ) then
11.    v_i ← ChooseValue() ;   t_i ← t_i + 1 ;
12.    CPA ← {AgentView ∪ myAssig} ;
13.    SendCPA(CPA) ;
14. else Backtrack();
procedure SendCPA(CPA)
15. if ( size(CPA) = n ) then                          /* A_i is the last agent in O */
16.    broadcastMsg: stp(CPA) ; end ← true
17. else foreach ( x_k ≻ x_i ) do sendMsg: cpa(CPA) to A_k ;
procedure ProcessCPA(msg)
18. if ( ¬AgentView.Consistent ∧ AgentView ⊂ msg.CPA ) then return ;
19. if ( msg.CPA is stronger than AgentView ) then
20.    UpdateAgentView(msg.CPA) ; AgentView.Consistent ←true;
21.    Revise();
22.    if ( D(x_i) = ∅ ) then Backtrack() ;
23.    else CheckAssign(msg.Sender) ;
procedure CheckAssign(sender)
24. if ( A_{i-1} = sender ) then Assign() ;          /* the sender is the predecessor of A_i */
procedure Backtrack()
25. newNogood ← solve(myNogoodStore) ;
26. if ( newNogood = empty ) then broadcastMsg: stp( ∅ ); end ← true;
27. else
28.    sendMsg: ngd(newNogood) to A_j ;     /* x_j denotes the variable on rhs(newNogood) */
29.    foreach ( x_k ≻ x_j ) do AgentView[k].value ← empty ;
30.    foreach ( ng ∈ NogoodStore ) do
31.       if ( ¬Compatible(ng, AgentView) ∨ x_j ∈ ng ) then remove(ng, myNogoodStore) ;
32.    AgentView.Consistent ← false;   v_i ← empty;
procedure ProcessNogood(msg)
33. if ( Compatible(msg.Nogood, AgentView) ) then
34.    add(msg.nogood, NogoodStore) ;                  /* according to the HPLV */
35.    if ( rhs(msg.nogood).Value = v_i ) then v_i ← empty; Assign() ;
procedure Revise()
36. foreach ( v ∈ D^0(x_i) ) do
37.    if ( ¬isConsistent(v, AgentView) ) then store the best nogood for v;
procedure UpdateAgentView(CPA)
38. AgentView ← CPA ;                                  /* update values and tags */
39. foreach ( ( ng ∈ myNogoodStore ) ) do
40.    if ( (¬Compatible(ng, AgentView)) ) then remove(ng, myNogoodStore) ;
```

Otherwise, if the received CPA is stronger than its AgentView, A_i updates its AgentView and marks it consistent (lines 19-20). Procedure `UpdateAgentView(CPA)` (lines 38-40) sets the AgentView and the NogoodStore to be consistent with the received CPA. Each nogood in the NogoodStore containing a value for a variable different from that on the received CPA will be deleted (line 40). Next, A_i calls procedure `Revise()` (line 21) to store nogoods for values inconsistent with the new AgentView or to try to find a better nogood for values already having one in the NogoodStore (line 37). A nogood is better according to the *HPLV* heuristic if the lowest variable in the body (*lhs*) of the nogood is higher. If A_i generates an empty domain as a result of calling `Revise()`, it calls procedure `Backtrack()` (line 22), otherwise, A_i calls procedure `CheckAssign(`*sender*`)` to check if it has to assign its variable (line 23). In `CheckAssign(`*sender*`)`, A_i calls procedure `Assign` to try to assign its variable only if sender is the predecessor of A_i (i.e., CPA was received from the predecessor, line 24).

When every value of A_i's variable is ruled out by a nogood (line 22), the procedure `Backtrack()` is called. These nogoods are resolved by computing a new nogood *newNogood* (line 25). *newNogood* is the conjunction of the left hand sides of all nogoods stored by A_i in its NogoodStore. If the new nogood (*newNogood*) is empty, A_i terminates execution after sending a ***stp*** message to all agents in the system meaning that the problem is unsolvable (line 26). Otherwise, A_i backtracks by sending one ***ngd*** message to the agent owner of the variable in the right hand side of *newNogood*, say A_j, (line 28). The ***ngd*** message carries the generated nogood (*newNogood*). Next, A_i updates its AgentView by removing assignments of every agent that is placed after the agent A_j owner of `rhs(`*newNogood*`)` in the total order (line 29). A_i also updates its NogoodStore by removing obsolete nogoods (line 31). Obsolete nogoods are nogoods inconsistent with the AgentView or containing the assignment of x_j, i.e., the variable on the right hand side of *newNogood*, (line 31). Finally, A_i marks its AgentView as inconsistent an removes its last assignment (line 32). A_i remains in an inconsistent state until receiving a stronger CPA holding at least one agent assignment with counter higher than that in the AgentView of A_i.

When a ***ngd*** message is received by an agent A_i, it checks the validity of the received nogood (line 33). If the received nogood is consistent with the AgentView, this nogood is a valid justification for removing the value on its right hand side *rhs*. Then if the value on the *rhs* of the received nogood is already removed, A_i adds the received nogood to its NogoodStore if it is better (according to the HPLV heuristic [**Hirayama and Yokoo, 2000**]) than the current stored nogood. If the value on the *rhs* of the received nogood belongs to the current domain of x_i, A_i simply adds it to its NogoodStore. If the value on the *rhs* of the received nogood equals v_i, the current value of A_i, A_i dis-instantiates its variable and calls the procedure `Assign()` (line 35).

Whenever ***stp*** message is received, A_i marks *end* flag true to stop the main loop (line 8). If the CPA attached to the received message is empty then there is no solution. Otherwise, the solution of the problem is retrieved from the CPA.

2.3 Correctness Proofs

Theorem 2.1. *The spatial complexity of AFC-ng is polynomially bounded by $O(nd)$ per agent.*

Proof. In AFC-ng, the size of nogoods is bounded by n, the total number of variables. Now, on each agent, AFC-ng only stores one nogood per removed value. Thus, the space complexity of AFC-ng is in $O(nd)$ on each agent. □

Lemma 2.1. AFC-ng is guaranteed to terminate.

Proof. We prove by induction on the agent ordering that there will be a finite number of new generated CPAs (at most d^n, where d is the size of the initial domain and n the number of variables.), and that agents can never fall into an infinite loop for a given CPA. The base case for induction ($i = 1$) is obvious. The only messages that x_1 can receive are ***ngd*** messages. All nogoods contained in these ***ngd*** messages have an empty *lhs*. Hence, values on their *rhs* are removed once and for all from the domain of x_1. Now, x_1 only generates a new CPA when it receives a nogood ruling out its current value. Thus, the maximal number of CPAs that x_1 can generate equals the size of its initial domain (d). Suppose now that the number of CPAs that agents $x_1, \ldots, x_{i-1}$ can generate is finite (and bounded by d^{i-1}). Given such a CPA on $[x_1, \ldots, x_{i-1}]$, x_i generates new CPAs (line 12, Algorithm 2.1) only when it changes its assignment after receiving a nogood ruling out its current value v_i. Given the fact that any received nogood can include, in its *lhs*, only the assignments of higher priority agents ($[x_1, \ldots, x_{i-1}]$), this nogood will remain valid as long as the CPA on $[x_1, \ldots, x_{i-1}]$ does not change. Thus, x_i cannot regenerate a new CPA containing v_i without changing assignments on higher priority agents ($[x_1, \ldots, x_{i-1}]$). Since there are a finite number of values on the domain of variable x_i, there will be a finite number of new CPAs generated by x_i (d^i). Therefore, by induction we have that there will be a finite number of new CPAs (d^n) generated by AFC-ng.

Let cpa be the strongest CPA generated in the network and A_i be the agent that generated cpa. After a finite amount of time, all unassigned agents on cpa ($[x_{i+1}, \ldots, x_n]$) will receive cpa and thus will discard all other CPAs. Two cases occur. First case, at least one agent detects a dead-end and thus backtracks to an agent A_j included in cpa (i.e., $j \leq i$) forcing it to change its current value on cpa and to generate a new stronger CPA. Second case (no agent detects dead-end), if $i < n$, A_{i+1} generates a new stronger CPA by adding its assignment to cpa, else ($i = n$), a solution is reported. As a result, agents can never fall into an infinite loop for a given CPA and AFC-ng is thus guaranteed to terminate. □

Lemma 2.2. AFC-ng cannot infer inconsistency if a solution exists.

Proof. Whenever a stronger CPA or a ***ngd*** message is received, AFC-ng agents update their NogoodStore. Hence, for every CPA that may potentially lead to a solution, agents only store valid nogoods. In addition, every nogood resulting from a CPA is redundant with regard to the DisCSP to solve. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, the empty nogood cannot be inferred if the network is solvable. This mean that AFC-ng is able to produce all solutions. □

Theorem 2.2. *AFC-ng is correct.*

Proof. The argument for soundness is close to the one given in [Meisels and Zivan, 2007; Nguyen *et al.*, 2004]. The fact that agents only forward consistent partial solution on the CPA messages at only one place in procedure `Assign()` (line 12, Algorithm 2.1), implies that the agents receive only consistent assignments. A solution is reported by the last agent only in procedure `SendCPA(CPA)` at line 16. At this point, all agents have assigned their variables, and their assignments are consistent. Thus the AFC-ng algorithm is sound. Completeness comes from the fact that AFC-ng is able to terminate and does not report inconsistency if a solution exists (Lemma 2.1 and 2.2). □

2.4 Experimental Evaluation

In this section we experimentally compare AFC-ng to two other algorithms: AFC [Meisels and Zivan, 2007] and ABT [Yokoo *et al.*, 1998; Bessiere *et al.*, 2005]. Algorithms are evaluated on three benchmarks: uniform binary random DisCSPs, distributed sensor-target networks and distributed meeting scheduling problems. All experiments were performed on the DisChoco 2.0 platform [1] [Wahbi *et al.*, 2011], in which agents are simulated by Java threads that communicate only through message passing (see Chapter 7). All algorithms are tested on the same static agents ordering using the ***dom/deg*** heuristic [Bessiere and Régin, 1996] and the same nogood selection heuristic (*HPLV*) [Hirayama and Yokoo, 2000]. For ABT we implemented an improved version of Silaghi's solution detection [Silaghi, 2006] and counters for tagging assignments.

We evaluate the performance of the algorithms by communication load [Lynch, 1997] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution (*#msg*), including those of termination detection (system messages). Computation effort is measured by the number of non-concurrent constraint checks (*#nccs*) [Zivan and Meisels, 2006b]. *#nccs* is the metric used in distributed constraint solving to simulate the computation time.

2.4.1 Uniform binary random DisCSPs

The algorithms are tested on uniform binary random DisCSPs which are characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of agents/variables, d is the number of values in each of the domains, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of constraint graphs: sparse graphs $\langle 20, 10, 0.2, p_2 \rangle$ and dense ones $\langle 20, 10, 0.7, p_2 \rangle$. We vary the tightness from 0.1 to 0.9 by steps of 0.05. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. Thereafter, we report average over the 100 runs.

Figure 2.1 presents computational effort of AFC-ng, AFC, and ABT running on the sparse instances (p_1 =0.2). We observe that at the complexity peak, AFC is the less efficient algorithm. It is better than ABT (the second worst) only on instances to the right

1. http://www2.lirmm.fr/coconut/dischoco/

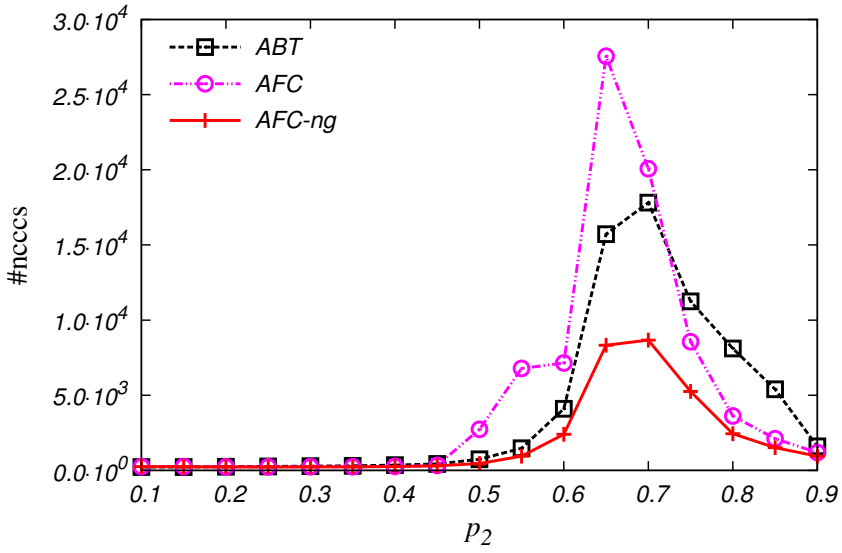


Figure 2.1 – The number of non-concurrent constraint checks (#*ncccs*) performed on sparse problems (p_1 =0.2).

of the complexity peak (over-constrained region). On the most difficult instances, AFC-ng improves the performance of standard AFC by a factor of 3.5 and outperforms ABT by a factor of 2.

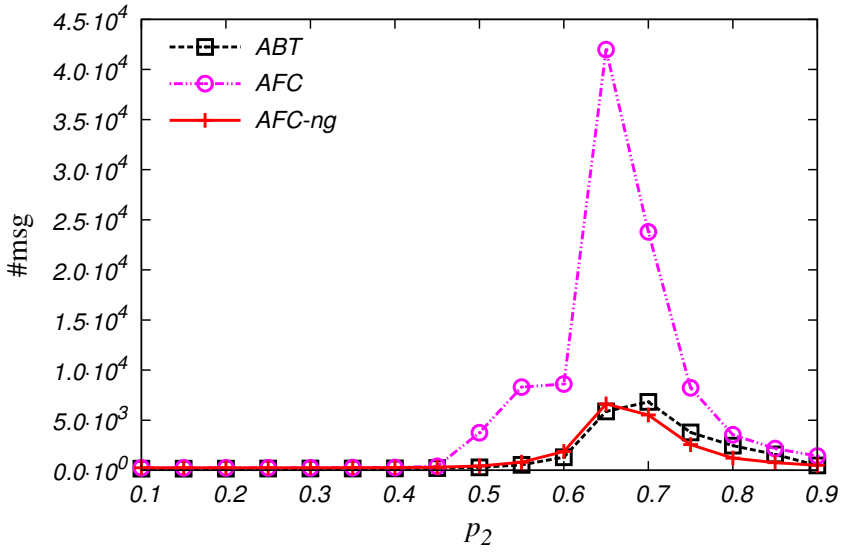


Figure 2.2 – The total number of messages sent on sparse problems (p_1 =0.2).

The total number of exchanged messages by compared algorithms on sparse problems (p_1 =0.2) is illustrated in Figure 2.2. When comparing the communication load, AFC dramatically deteriorates compared to other algorithms. AFC-ng improves AFC by a factor of 7. AFC-ng exchanges slightly fewer messages than ABT in the over-constrained area. In the complexity peak, both algorithms (ABT and AFC-ng) require almost the same number of messages.

Figure 2.3 presents the number of non-concurrent constraint checks (#*ncccs*) performed

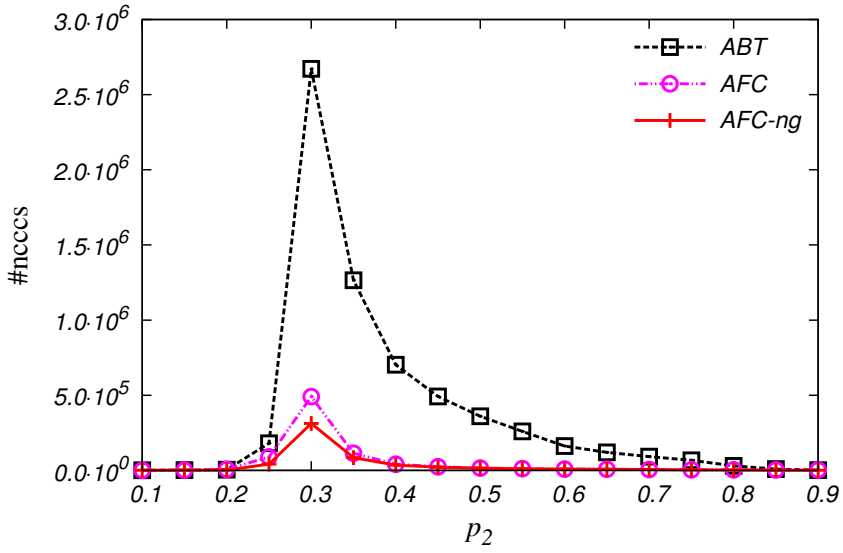


Figure 2.3 – The number of non-concurrent constraint checks (*#nccccs*) performed on dense problems (p_1 =0.7).

by compared algorithms on dense instances (p_1 =0.7). The results obtained show that ABT dramatically deteriorates compared to synchronous algorithms. This is consistent with results presented in [**Meisels and Zivan, 2007**]. Among all compared algorithms, AFC-ng is the fastest one on these dense problems.

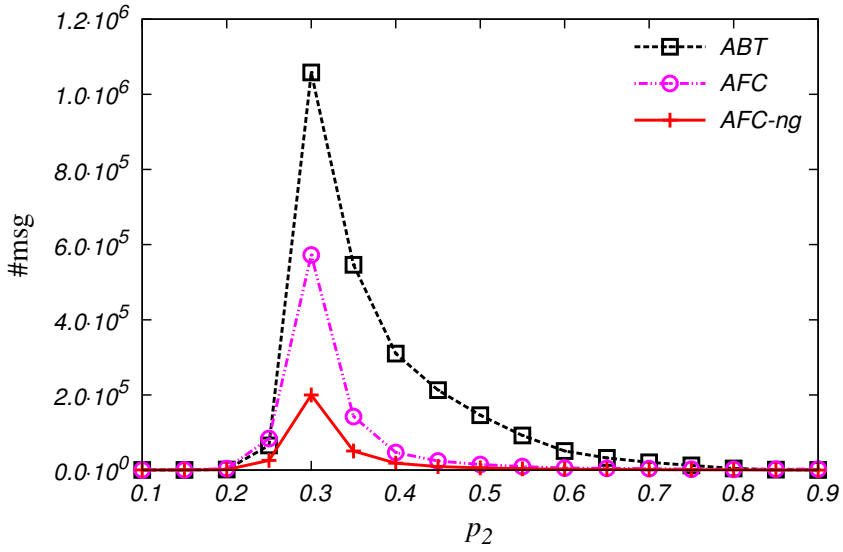


Figure 2.4 – The total number of messages sent on the dense problems (p_1 =0.7).

Regarding the number of exchanged messages (Figure 2.4), ABT is again significantly the worst. AFC requires less messages than ABT. Our AFC-ng algorithm outperforms AFC by a factor 3. Hence, our experiments on uniform random DisCSPs show that AFC-ng improves on AFC and ABT algorithms.

2.4.2 Distributed Sensor Target Problems

The *Distributed Sensor-Target Problem* (SensorDisCSP) [**Béjar *et al.*, 2005**] is a benchmark based on a real distributed problem (see Section 1.3.2.2). It consists of n sensors that track m targets. Each target must be tracked by 3 sensors. Each sensor can track at most one target. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a target is visible. The compatibility constraint defines the compatibility among sensors. In our implementation of the DisCSP algorithms, the encoding of the SensorDisCSP presented in Section 1.3.2.2 is translated to an equivalent formulation where we have three virtual agents for every real agent, each virtual agent handling a single variable.

Problems are characterized by $\langle n, m, p_c, p_v \rangle$, where n is the number of sensors, m is the number of targets, each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each target can be tracked by a fraction p_v of the sensors having the target in their sensing range. We present results for the class $\langle$ 25, 5, 0.4, $p_v \rangle$, where we vary p_v from 0.1 to 0.9 by steps of 0.05. For each pair (p_c, p_v) we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

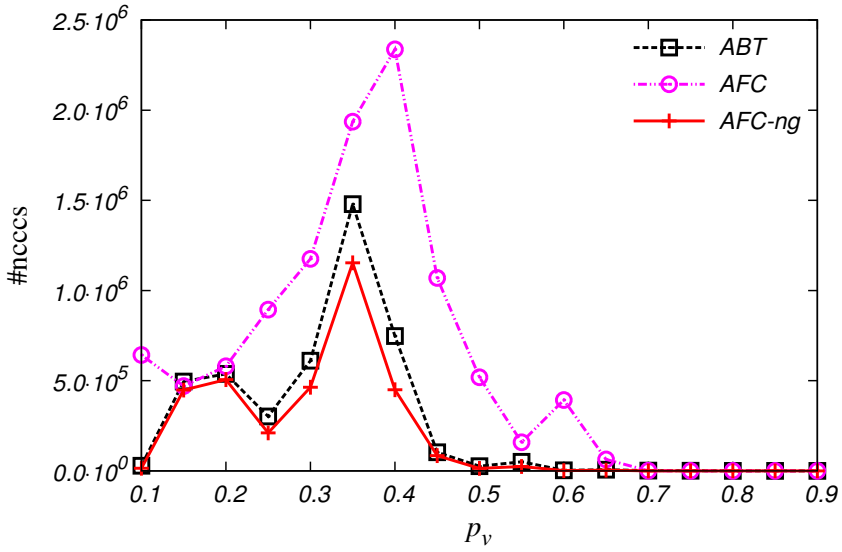


Figure 2.5 – The number non-concurrent constraint checks performed on sensor target instances where p_c =0.4.

Figure 2.5 presents the computational effort performed by AFC-ng, AFC, and ABT on sensor target problems where $\langle n$ =25, m=5, p_c =0.4$\rangle$. Our results show that ABT outperforms AFC whereas AFC-ng outperforms both. We observe that on the exceptionally hard instances (where 0.1< p_v <0.25) the improvement on the Asynchronous Backtracking is minor.

Concerning the communication load (Figure 2.6), the ranking of algorithms is similar to that on computational effort, though differences tend to be smaller between ABT and AFC-ng. AFC-ng remains the best on all problems.

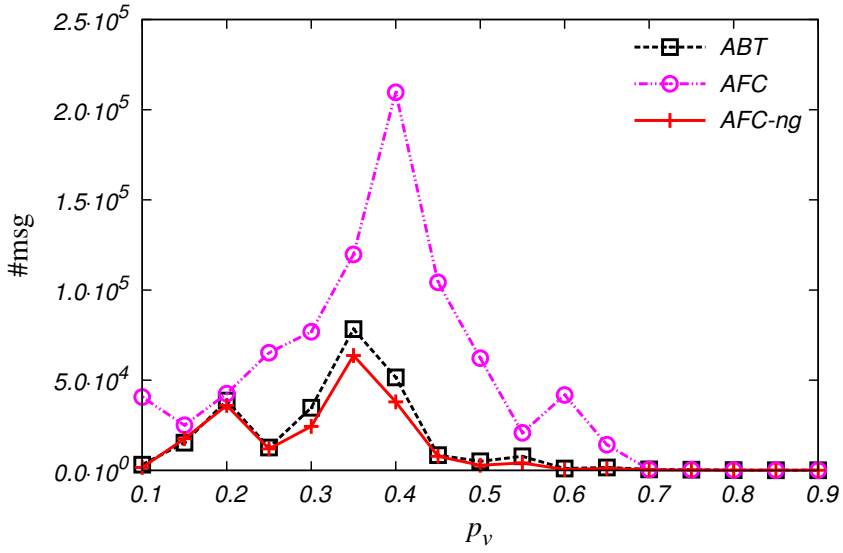


Figure 2.6 – The total number of exchanged messages on sensor target instances where p_c =0.4.

2.4.3 Distributed Meeting Scheduling Problems

The *Distributed Meeting Scheduling Problem* (DisMSP) is a truly distributed benchmark where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [**Wallace and Freuder, 2002**; **Meisels and Lavee, 2004**] (see Section 1.3.2.1). The DisMSP consists of a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location.

We encode the DisMSP in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. These k meetings are selected randomly among the m meetings. The domain of each variable contains the $d \times h$ slots where a meeting can be scheduled. A slot is one hour long, and there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. We place meetings randomly on the nodes of a uniform grid of size $g \times g$ and the traveling time between two adjacent nodes is 1 hour. Thus, the traveling time between two meetings equals the Euclidean distance between nodes representing the locations where they will be held. For varying the tightness of the arrival-time constraint we vary the size of the grid on which meetings are placed.

Problems are characterized by $\langle n, m, k, d, h, g \rangle$, where n is the number of agents, m is the number meetings, k is the number of meetings/variables per agent, d is the number of days and h is the number of hours per day, and g is the grid size. The duration of each meeting is one hour. In our implementation of the DisCSP algorithms, this encoding is translated to an equivalent formulation where we have k (number of meetings per agent) virtual agents for every real agent, each virtual agent handling a single variable. We present

results for the class $\langle$ 20, 9, 3, 2, 10, $g\rangle$ where we vary g from 2 to 22 by steps of 2. Again, for each g we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

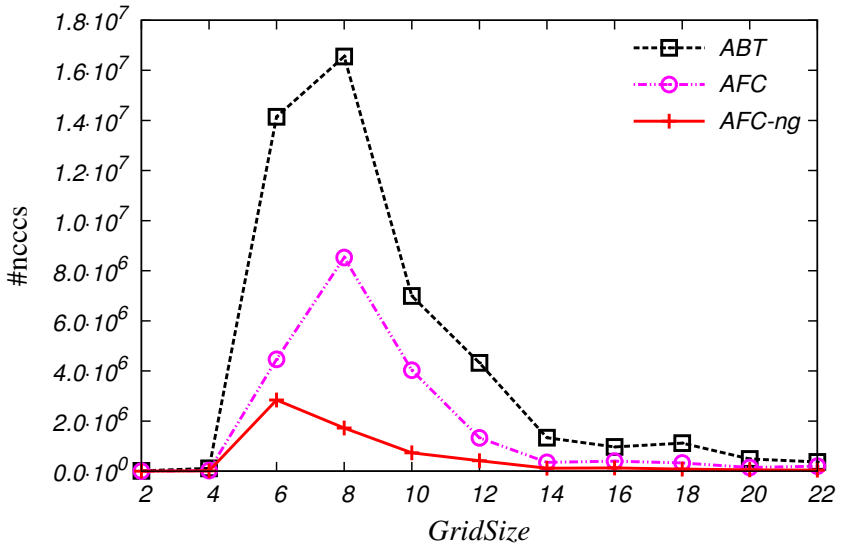


Figure 2.7 – The number of non-concurrent constraint checks performed on meeting scheduling benchmarks where the number of meeting per agent is 3.

On this class of meeting scheduling benchmarks AFC-ng continues to perform well. AFC-ng is significantly better than ABT and AFC, both for computational effort (Figure 2.7) and communication load (Figure 2.8). Concerning the computational effort, ABT is the slowest algorithm to solve such problems. AFC outperforms ABT by a factor of 2 at the peak (i.e., where the *GridSize* equals 8). However, ABT requires less messages than AFC.

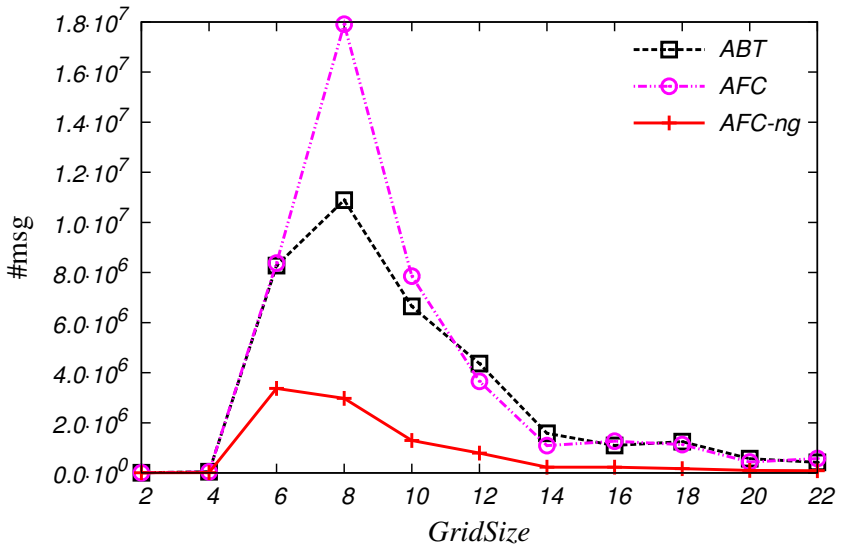


Figure 2.8 – The total number of exchanged messages on meeting scheduling benchmarks where the number of meeting per agent is 3.

2.4.4 Discussion

Table 2.1 – The percentage of messages per type exchanged by AFC to solve instances of uniform random DisCSPs where p_1=0.2.

p_2	*#msg*	*cpa*	*back_cpa*	*fc_cpa*	*not_ok*
0.55	8297	5,93%	3,76%	50,99%	38,58%
0.60	8610	4,49%	2,75%	52,46%	39,57%
0.65	41979	3,37%	1,77%	42,20%	52,60%
0.70	23797	3,00%	1,75%	43,48%	51,68%
0.75	8230	2,61%	1,53%	40,66%	54,97%

Table 2.2 – The percentage of messages per type exchanged by AFC to solve instances of uniform random DisCSPs where p_1=0.7.

p_2	*#msg*	*cpa*	*back_cpa*	*fc_cpa*	*not_ok*
0.25	83803	4,85%	2,86%	47,68%	44,54%
0.30	572493	3,61%	2,11%	43,64%	50,63%
0.35	142366	2,90%	1,69%	39,35%	56,27%
0.40	46883	2,60%	1,52%	37,77%	58,58%
0.45	24379	2,35%	1,41%	35,56%	61,52%
0.50	14797	2,14%	1,29%	33,32%	64,38%

Table 2.3 – The percentage of messages per type exchanged by AFC to solve instances of distributed sensor-target problem where p_c=0.4.

p_v	*#msg*	*cpa*	*back_cpa*	*fc_cpa*	*not_ok*
0.30	76914	23,16%	23,14%	49,50%	4,14%
0.35	119759	24,91%	24,90%	47,49%	2,66%
0.40	209650	23,55%	23,55%	47,52%	5,35%
0.45	104317	19,07%	19,06%	57,17%	4,68%

Table 2.4 – The percentage of messages per type exchanged by AFC to solve instances of distributed meeting scheduling problem where k=3.

GridSize	*#msg*	*cpa*	*back_cpa*	*fc_cpa*	*not_ok*
4	39112	2,71%	1,70%	50,41%	44,71%
6	8376151	2,19%	1,59%	49,31%	46,91%
8	17911100	2,39%	1,66%	53,88%	42,07%
10	7855300	2,30%	1,66%	52,20%	43,83%
12	3653697	1,77%	1,33%	57,19%	39,71%

We present in Tables 2.1, 2.2, 2.4 and 2.3 the percentage of messages per type exchanged by the AFC algorithm to solve instances around the complexity peak of respectively sparse random DisCSPs, dense random DisCSPs, distributed sensor-target problem where p_c=0.4 and distributed meeting scheduling problem where k=3. These tables allow us to more understand the behavior of the AFC algorithm and to explain the good performance of AFC-ng compared to AFC.

A first observation on our experiments is that AFC-ng is always better than AFC, both

in terms of exchanged messages and computational effort (#*ncccs*). A closer look at the type of exchanged messages shows that the backtrack operation in AFC requires exchanging a lot of ***not_ok*** messages (approximately 50% of the total number of messages sent by agents). This confirms the significance of using nogoods as justification of value removals and allowing several concurrent backtracks in AFC-ng. A second observation on these experiments is that ABT performs bad in dense graphs compared to synchronous algorithms.

2.5 Other Related Works

In [**Brito and Meseguer, 2004**; **Zivan and Meisels, 2003**] the performance of asynchronous (ABT), synchronous (Synchronous Conflict BackJumping, SCBJ), and hybrid approaches (ABT-Hyb) was studied. It is shown that ABT-Hyb improves over ABT and that SCBJ requires less communication effort than ABT-Hyb. Dynamic Distributed BackJumping (DDBJ) was presented in [**Nguyen *et al.*, 2004**]. It is an improved version of the basic AFC. DDBJ combines the concurrency of an asynchronous dynamic backjumping algorithm, and the computational efficiency of the AFC algorithm, coupled with the *possible conflict heuristics* of dynamic value and variable ordering. As in DDBJ, AFC-ng performs several backtracks simultaneously. However, AFC-ng should not be confused with DDBJ. DDBJ is based on dynamic ordering and requires additional messages to compute ordering heuristics. In AFC-ng, all agents that received a ***ngd*** message continue search concurrently. Once a stronger CPA is received by an agent, all nogoods already stored can be kept if consistent with that CPA.

2.6 Summary

A new complete, asynchronous algorithm is presented for solving distributed CSPs. This algorithm is based on the AFC an uses nogoods as justification of value removals. We call it nogood-based Asynchronous Forward Checking (AFC-ng). Besides its use of nogoods as justification of value removal, AFC-ng allows simultaneous backtracks going from different agents to different destinations. Thus, it enhances the asynchronism of the forward-checking phase. Our experiments show that AFC-ng improves the AFC algorithm in terms of computational effort and number of exchanged messages.

Asynchronous Forward Checking Tree (AFC-tree)

3

Contents

This chapter shows how to extend our nogood-based Asynchronous Forward-Checking (AFC-ng) algorithm to the *Asynchronous Forward-Checking Tree (AFC-tree)* algorithm using a pseudo-tree arrangement of the constraint graph. To achieve this goal, agents are ordered a priory in a pseudo-tree such that agents in different branches of the tree do not share any constraint. AFC-tree does not address the process of ordering the agents in a pseudo-tree arrangement. Therefore, the construction of the pseudo-tree is done in a preprocessing step.

This chapter is organized as follows. Section 3.1 recalls the principle of our AFC-ng algorithm. The concept of the pseudo-tree arrangement of the constraint graph is given in Section 3.2. A Distributed Depth-First Search trees construction is presented in Section 3.3. The AFC-tree is described in Section 3.4 and correctness proofs are given in Section 3.5. Section 3.6 presents an experimental evaluation of AFC-tree against AFC-ng. Section 3.7 summarizes some related works and we conclude the chapter in Section 3.8.

3.1 Introduction

We have described in Chapter 1, Synchronous Backtracking (SBT), the simplest search algorithm for solving distributed constraint satisfaction problems. Since it is a straightforward extension of the chronological algorithm for centralized CSPs, SBT performs assignments sequentially and synchronously. Thus, only the agent holding a current partial assignment (CPA) performs an assignment or backtrack [**Yokoo, 2000b**]. Researchers in distributed CSP area have devoted many effort to improve the SBT algorithm. Thus, a variety improvements have been proposed. Hence, **Zivan and Meisels** (**2003**) proposed the Synchronous Conflict-Based Backjumping (SCBJ) that performs backjumping instead of chronological backtracking as is done in SBT.

In a subsequent study, **Meisels and Zivan** proposed the Asynchronous Forward-Checking (AFC) another promising distributed search algorithm for DisCSPs [**Meisels and Zivan, 2007**]. AFC algorithm is based on the forward checking (FC) algorithm for CSPs [**Haralick and Elliott, 1980**]. The forward checking operation is performed asynchronously while the search is performed synchronously. Hence, this algorithm improves on SBT by adding to them some amount of concurrency. The concurrency arises from the fact that forward checking phase is processed concurrently by future agents. However, the manner in which the backtrack operation is performed is a major drawback of the AFC algorithm. The backtrack operation requires a lot of work from the agents.

We presented in Chapter 2, our nogood-based Asynchronous Forward-Checking (AFC-ng), a new complete and asynchronous algorithm that is based on the AFC. Besides its use of nogoods as justification of value removal, AFC-ng allows simultaneous backtracks going from different agents to different destinations. Thus, AFC-ng enhances the asynchronism of the forward-checking phase and attempts to avoid the drawbacks of the backtrack operation of the AFC algorithm. Our experiments show that AFC-ng improves the AFC algorithm in terms of computational effort and number of exchanged messages.

In this chapter, we propose another algorithm based on AFC-ng and is named Asynchronous Forward-Checking Tree (AFC-tree). The main feature of the AFC-tree algorithm is using different agents to search non-intersecting parts of the search space concurrently. In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. The pseudo-tree ordering is build in a preprocessing step. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. The agents that are brothers are committed to concurrently find the partial solutions of their variables. Therefore, AFC-tree exploits the potential speed-up of a parallel exploration in the processing of distributed problems [**Freuder and Quinn, 1985**]. A solution is found when all leaf agents succeed in extending the CPA they received. Furthermore, in AFC-tree privacy may be enhanced because communication is restricted to agents in the same branch of the pseudo-tree.

3.2 Pseudo-tree ordering

We have seen in Chapter 1 that any binary distributed constraint network (DisCSP) can be represented by a *constraint graph* $G = (X_G, E_G)$, whose vertexes represent the variables and edges represent the constraints (see, Definition 1.2). Therefore, $X_G = \mathcal{X}$ and for each constraint $c_{ij} \in \mathcal{C}$ connecting two variables x_i and x_j there exists an edge $\{x_i, x_j\} \in E_G$ linking vertexes x_i and x_j.

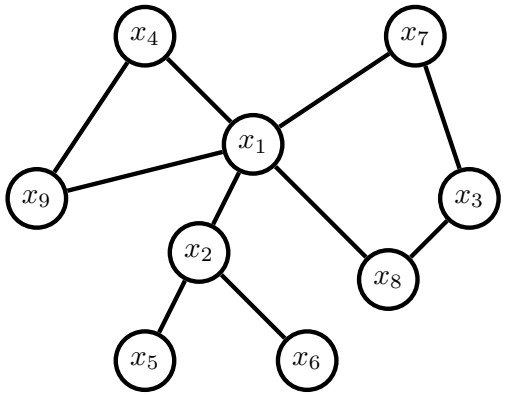


Figure 3.1 – Example of a constraint graph G.

Figure 3.1 shows an example of a constraint graph G of a problem involving 9 variables $\mathcal{X} = X_G = \{x_1, \ldots, x_9\}$ and 10 constraints $\mathcal{C} = \{c_{12}, c_{14}, c_{17}, c_{18}, c_{19}, c_{25}, c_{26}, c_{37}, c_{38}, c_{49}\}$. There are constraint between x_1 and x_2 (c_{12}), x_1 and x_4, etc.

The concept of *pseudo-tree* arrangement (see Definition 1.18) of a constraint graph has been introduced first by **Freuder and Quinn** in [**Freuder and Quinn, 1985**]. The purpose of this arrangement is to perform search in parallel on independent branches of the pseudo-tree in order to improve search in centralized constraint satisfaction problems. The aim in introducing the pseudo-tree is to boost the search by performing search in parallel on the independent branches of the pseudo-tree. Thus, variables belonging to different branches of the pseudo-tree can be instantiated independently.

An example of a pseudo-tree arrangement T of the constraint graph G (Figure 3.1) is illustrated in Figure 3.2. Notice that G and T have the same vertexes ($X_G = X_T$). However, a new (dotted) edge, $\{x_1, x_3\}$, linking x_1 to x_3 is added to T where $\{x_1, x_3\} \notin E_G$. Moreover, edges $\{x_1, x_7\}$, $\{x_1, x_8\}$ and $\{x_1, x_8\}$ belonging to the constraint graph G are not part of T. They are represented in T by dashed edges to show that constrained variables must be located in the same branch of T even if there is not an edge linking them.

From a pseudo-tree arrangement of the constraint graph we can define:

- A *branch* of the pseudo-tree is a path from the root to some leaf (e.g., $\{x_1, x_4, x_9\}$).
- A leaf is a vertex that has no child (e.g., x_9).
- The *children* of a vertex are its descendants connected to it through tree edges (e.g., $\texttt{children}(x_1) = \{x_2, x_3, x_4\}$).
- The *descendants* of a vertex x_i are vertexes belonging to the subtree rooted at x_i (e.g., $\texttt{descendants}(x_2) = \{x_5, x_6\}$ and $\texttt{descendants}(x_1) = \{\mathcal{X} \setminus x_1\}$).

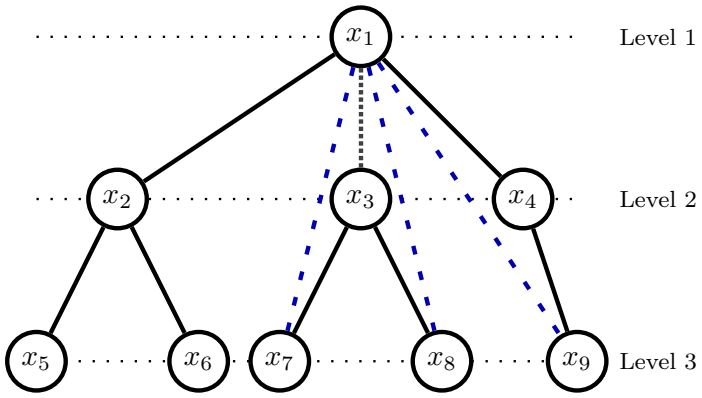


Figure 3.2 – Example of a pseudo-tree arrangement T of the constraint graph illustrated in Figure 3.1.

- The *linked descendants* of a vertex are its descendants constrained with it together with its children, (e.g., $\texttt{linkedDescendants}(x_1) = \{x_2, x_3, x_4, x_7, x_8, x_9\}$).
- The *parent* of a vertex is the ancestor connected to it through a tree edge (e.g., $\texttt{parent}(x_9) = \{x_4\}$, $\texttt{parent}(x_3) = \{x_1\}$).
- A vertex x_i is an *ancestor* of a vertex x_j if x_i is the parent of x_j or an ancestor of the parent of x_j.
- The *ancestors* of a vertex x_i is the set of agents forming the path from the root to x_i's parent (e.g., $\texttt{ancestors}(x_8) = \{x_1, x_3\}$).

3.3 Distributed Depth-First Search trees construction

The construction of the pseudo-tree can be processed by a centralized procedure. First, a *system agent* must be elected to gather information about the constraint graph. Such system agent can be chosen using a leader election algorithm like that presented in [**Abu-Amara, 1988**]. Once, all information about the constraint graph is gathered by the system agent, it can perform a centralized algorithm to build the pseudo-tree ordering (see Section 1.2.2.1). A decentralized modification of the procedure for building the pseudo-tree was introduced by **Chechetka and Sycara** in [**Chechetka and Sycara, 2005**]. This algorithm allows the distributed construction of pseudo-trees without needing to deliver any global information about the whole problem to a single process.

Whatever the method (centralized or distributed) for building the pseudo-tree, the obtained pseudo-tree may require the addition of some edges not belonging to the original constraint graph. In the example presented in Figure 3.2, a new edge linking x_1 to x_3 is added to the resulting pseudo-tree T. The structure of the pseudo-tree will be used for communication between agents. Thus, the added link between x_1 and x_3 will be used to exchange messages between them. However, in some distributed applications, the communication might be restricted to the neighboring agents (i.e., a message can be passed only locally between agents that share a constraint). The solution in such applications is to use a

depth-first search tree (DFS-tree). DFS-trees are special cases of pseudo-trees where all edges belong to the original graph.

Algorithm 3.1: The distributed depth-first search construction algorithm.

```
procedure distributedDFS()
01. Select the root via a leader election algorithm ;
02. Visited ← ∅; end ← false ;
03. if ( x_i is the elected root ) then CheckNeighbourhood() ;
04. while ( ¬end ) do
05.   msg ← getMsg();
06.   Visited ← Visited ∪ {Γ(x_i) ∩ msg.DFS)} ;
07.   if ( msg.Sender ∈ children(x_i) ) then
08.     descendants(x_i) ← descendants(x_i) ∪ msg.DFS ;
09.   else
10.     parent(x_i) ← msg.Sender ;
11.     ancestors(x_i) ← msg.DFS ;
12.   CheckNeighbourhood() ;
procedure CheckNeighbourhood()
13. if ( Γ(x_i) = Visited ) then
14.   sendMsg: token(descendants(x_i) ∪ {x_i}) to parent(x_i) ;
15.   end ← true ;
16. else
17.   select x_j in Γ(x_i) \ Visited ;
18.   children(x_i) ← children(x_i) ∪ {x_j} ;
19.   sendMsg: token(ancestors(x_i) ∪ {x_i}) to A_j ;
```

We present in Algorithm 3.1 a simple distributed algorithm for the distributed construction of the DFS-tree named `DistributedDFS` algorithm. The `DistributedDFS` is similar to the algorithm proposed by **Cheung** in [**Cheung, 1983**]. The `DistributedDFS` algorithm is a distribution of a DFS traversal of the constraint graph. Each agent maintains a set *Visited* where it stores its neighbors which are already visited (line 2). The first step is to design the root agent using a leader election algorithm (line 1). An example of leader election algorithm was presented by **Abu-Amara** in [**Abu-Amara, 1988**]. Once the root is designed, it can start the distributed construction of the DFS-tree (procedure `CheckNeighbourhood()` call, line 3). The designed root initiates the propagation of a ***token***, which is a unique message that will be circulated on the network until "visiting" all the agents of the problem.

When an agent x_i receives the ***token***, it marks all its neighbors included in the received message as visited (line 6). Next, x_i checks if the ***token*** is sent back by a child. If it is the case, x_i sets all agents belonging to the subtree rooted at message sender (i.e., its child) as its descendants (lines 7-8). Otherwise, the ***token*** is received for the first time from the parent of x_i. Thus, x_i marks the sender as its parent (line 10) and all agents contained in the ***token*** (i.e., the sender and its ancestors) as its ancestors (line 11). Afterwards, x_i calls the procedure `CheckNeighbourhood()` to check if it has to pass on the ***token*** to an unvisited neighbor or to return back the ***token*** to its parent if all its neighbors are already visited.

The procedure `CheckNeighbourhood()` checks if all neighbors are already visited (line 13). If it is the case, the agent x_i sends back the ***token*** to its parent (line 14). The ***token*** contains the set *DFS* composed by x_i and its descendants. Until this point the agent

x_i knows all its ancestors, its children and its descendants. Thus, the agent x_i terminates the execution of `DistributedDFS` (line 15). Otherwise, agent x_i chooses one of its neighbors (x_j) not yet visited and designs it as a child (lines 17-18). Afterwards, x_i passes on to x_j the ***token*** where it puts the ancestors of the child x_j (i.e., `ancestors`$(x_i) \cup \{x_i\}$) (line 19).

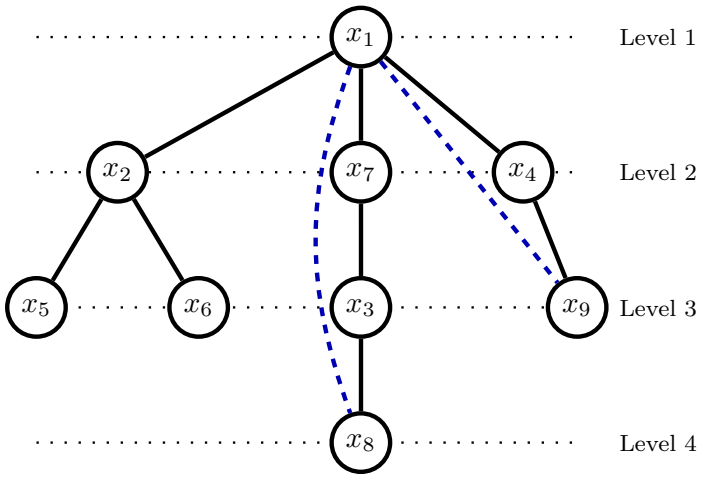


Figure 3.3 – A DFS-tree arrangement of the constraint graph in Figure 3.1.

Consider for example the constraint graph G presented in Figure 3.1. Figure 3.3 shows an example of a DFS-tree arrangement of the constraint graph G obtained by performing distributively the `DistributedDFS` algorithm. The `DistributedDFS` algorithm can be performed as follows. First, let x_1 be the elected root of the DFS-tree (i.e., the leader election algorithm elects the most connected agent). The root x_1 initiates the DFS-tree construction by calling procedure `CheckNeighbourhood()` (line 3). Then, x_1 selects from its unvisited neighbors x_2 to be its child (lines 17-18). Next, x_1 passes on the ***token*** to x_2 where it put itself to be the ancestor of the receiver (x_2) (line 19). After receiving the ***token***, x_2 updates the set of its visited neighbors (line 6) by marking x_1 (the only neighbor included in the ***token***) visited. Afterwards, x_2 sets x_1 to be its parent and puts $\{x_1\}$ to be its set of ancestors (lines 10-11). Next, x_2 calls procedure `CheckNeighbourhood()` (line 12). Until this point, x_2 has one visited neighbor (x_1) and two unvisited neighbors (x_5 and x_6). For instance, let x_2 chooses x_5 to be its child. Thus, x_2 sends the ***token*** to x_5 where it sets the DFS set to $\{x_1, x_2\}$. After receiving the ***token***, x_5 marks its single neighbor x_2 as visited (line 6), sets x_2 to be its parent (line 10), sets $\{x_1, x_2\}$ to be its ancestors ans sends the ***token*** back to x_2 where it puts itself. After receiving back the ***token*** from x_5, x_2 adds x_5 to its descendants and selects the last unvisited neighbor (x_6) to be its child and passes the ***token*** to x_6. In a similar way, x_6 returns back the ***token*** to x_2. Then, x_2 sends back the ***token*** to its parent x_1 since all its neighbors have been visited. The ***token*** contains the descendants of x_1 ($\{x_2, x_5, x_6\}$) on the subtree rooted at x_2. After receiving the ***token*** back from x_2, x_1 will select an agent from its unvisited neighbors $\{x_4, x_7, x_8, x_9\}$. Hence, the subtree rooted at x_2 where each agent knows its ancestors and its descendants is build without delivering any global information. The other subtrees respectively rooted at x_7 and x_4 are build in a similar manner. Thus, we obtain the DFS-tree shown in Figure 3.3.

3.4 The AFC-tree algorithm

The AFC-tree algorithm is based on AFC-ng performed on a pseudo-tree ordering of the constraint graph (built in a preprocessing step). Agents are prioritized according to the pseudo-tree ordering in which each agent has a single parent and various children. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves. The root initiates the search by generating a CPA, assigning its value on it, and sending ***cpa*** messages to its linked descendants. Among all agents that receive the CPA, children perform AFC-ng on the sub-problem restricted to its ancestors (agents that are assigned in the CPA) and the set of its descendants. Therefore, instead of giving the privilege of assigning to only one agent, agents who are in disjoint subtrees may assign their variables simultaneously. AFC-tree thus exploits the potential speed-up of a parallel exploration in the processing of distributed problems. The degree of asynchronism is enhanced.

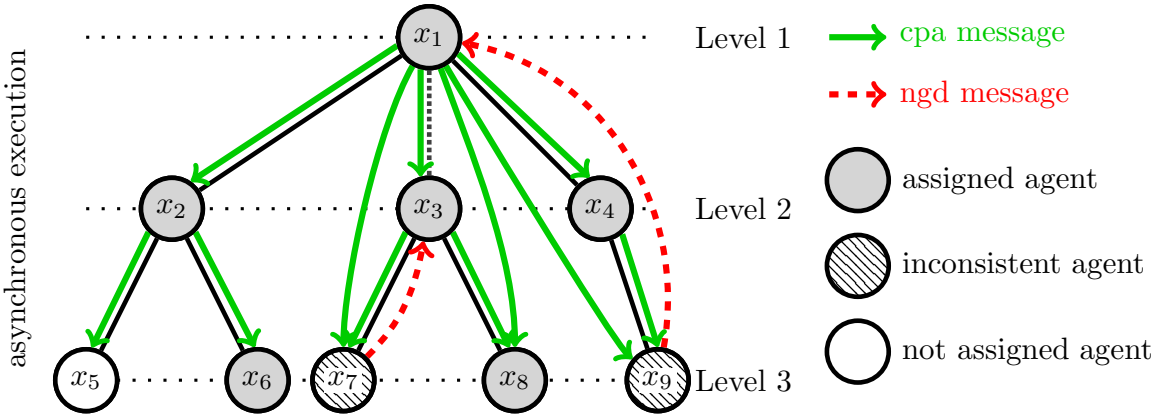


Figure 3.4 – An example of the AFC-tree execution.

An execution of AFC-tree on a sample DisCSP problem is shown in Figure 3.4. At time t_1, the root x_1 sends copies of the CPA on ***cpa*** messages to its linked descendants. Children x_2, x_3 and x_4 assign their values simultaneously in the received CPAs and then perform concurrently the AFC-tree algorithm. Agents x_7, x_8 and x_9 only perform a forward-checking. At time t_2, x_9 finds an empty domain and sends a ***ngd*** message to x_1. At the same time, other CPAs propagate down through the other paths. For instance, a CPA has propagated down from x_3 to x_7 and x_8. x_7 detects an empty domain and sends a nogood to x_3 attached on a ***ngd*** message. For the CPA that propagates on the path (x_1, x_2, x_6), x_6 successfully assigned its value and initiated a solution detection. The same thing is going to happen on the path (x_1, x_2, x_5) when x_5 (not yet instantiated) will receive the CPA from its parent x_2. When x_1 receives the ***ngd*** message from x_9, it initiates a new search process by sending a new copy of the CPA which will dominate all other CPAs where x_1 is assigned its old value. This new CPA generated by x_1 can then take advantage from efforts done by the obsolete CPAs. Consider for instance the subtree rooted at x_2. If the value of x_2 is consistent with the value of x_1 on the new CPA, all nogoods stored on the subtree rooted at x_2 are still valid and a solution is reached on the subtree without any nogood generation.

In AFC-ng, a solution is reached when the last agent in the agent ordering receives the CPA and succeeds in assigning its variable. In AFC-tree, the situation is different because a CPA can reach a leaf agent without being complete. When all agents are assigned and no constraint is violated, this state is a global solution and the network has reached quiescence, meaning that no message is traveling through it. Such a state can be detected using specialized snapshot algorithms [**Chandy and Lamport, 1985**], but AFC-tree uses a different mechanism that allows to detect solutions before quiescence. AFC-tree uses an additional type of messages called ***accept*** that informs parents of the acceptance of their CPA. Termination can be inferred earlier and the number of messages required for termination detection can be reduced. A similar technique of solution detection was used in the AAS algorithm [**Silaghi and Faltings, 2005**].

The mechanism of solution detection is as follows: whenever a leaf node succeeds in assigning its value, it sends an ***accept*** message to its parent. This message contains the CPA that was received from the parent incremented by the value-assignment of the leaf node. When a non-leaf agent A_i receives ***accept*** messages from all its children that are all consistent with each other, all consistent with A_i's AgentView and with A_i's value, A_i builds an ***accept*** message being the conjunction of all received ***accept*** messages plus A_i's value-assignment. If A_i is the root a solution is found, and A_i broadcasts this solution to all agents. Otherwise, A_i sends the built ***accept*** message to its parent.

3.4.1 Description of the algorithm

We present in Algorithm 3.2 only the procedures that are new or different from those of AFC-ng in Algorithm 2.1. In `InitAgentView()`, the AgentView of A_i is initialized to the set `ancestors`(A_i) and t_j is set to 0 for each agent x_j in `ancestors`(A_i) (line 10). The new data structure storing the received ***accept*** messages is initialized to the empty set (line 11). In `SendCPA(CPA)`, instead of sending copies of the CPA to all agents not yet instantiated on it, A_i sends copies of the CPA only to its linked descendants (`linkedDescendants`(A_i), lines 13-14). When the set `linkedDescendants`(A_i) is empty (i.e., A_i is a leaf), A_i calls the procedure `SolutionDetection()` to build and send an ***accept*** message. In `CheckAssign(`*sender*`)`, A_i assigns its value if the CPA was received from its parent (line 16) (i.e., if *sender* is the parent of A_i).

In `ProcessAccept(`*msg*`)`, when A_i receives an ***accept*** message from its *child* for the first time, or the CPA contained in the received ***accept*** message is stronger than that received before, A_i stores the content of this message (lines 17-18) and calls the `SolutionDetection` procedure (line 19).

In procedure `SolutionDetection()`, if A_i is a leaf (i.e., `children`(A_i) is empty, line 20), it sends an ***accept*** message to its parent. The ***accept*** message sent by A_i contains its AgentView incremented by its assignment (lines 20-21). If A_i is not a leaf, it calls function `BuildAccept()` to build an accept partial solution *PA* (line 23). If the returned partial solution *PA* is not empty and A_i is the root, *PA* is a solution of the problem. Then, A_i broadcasts it to other agents including the system agent and sets the *end* flag to *true* (line 25). Otherwise, A_i sends an ***accept*** message containing *PA* to its parent (line 26).

Algorithm 3.2: New lines/procedures of AFC-tree with respect to AFC-ng.

```
procedure AFC-tree()
01. end ← false; AgentView.Consistent ← true; InitAgentView() ;
02. if ( A_i = IA ) then Assign() ;
03. while ( ¬end ) do
04.     msg ← getMsg();
05.     switch ( msg.type ) do
06.         cpa    : ProcessCPA(msg);
07.         ngd    : ProcessNogood(msg);
08.         stp    : end ← true;  if ( msg.CPA ≠ ∅ ) then solution ← msg.CPA ;
09.         accept : ProcessAccept(msg);
procedure InitAgentView()
10. foreach ( A_j ∈ ancestors(A_i) ) do AgentView[j] ← {(x_j, empty, 0)} ;
11. foreach ( child ∈ children(A_i) ) do Accept[child] ← ∅ ;
procedure SendCPA(CPA)
12. if ( children(A_i) ≠ ∅ ) then
13.     foreach ( descendant ∈ linkedDescendants(A_i) ) do
14.         sendMsg: cpa(CPA) to descendant ;
15. else SolutionDetection() ;
procedure CheckAssign(sender)
16. if ( parent(A_i) = sender ) then Assign() ;
procedure ProcessAccept(msg)
17. if ( msg.CPA is stronger than Accept[msg.Sender] ) then
18.     Accept[msg.Sender] ← msg.CPA ;
19.     SolutionDetection() ;
procedure SolutionDetection()
20. if ( children(A_i) = ∅ ) then
21.     sendMsg: accept(AgentView ∪ {(x_i, x_i, t_i)}) to parent(A_i) ;
22. else
23.     PA ← BuildAccept() ;
24.     if ( PA ≠ ∅ ) then
25.         if ( A_i = root ) then broadcastMsg: stp(PA);   end ← true;
26.         else sendMsg: accept(PA) to parent(A_i) ;
function BuildAccept()
27. PA ← AgentView ∪ {(x_i, x_i, t_i)} ;
28. foreach ( child ∈ children(x_i) ) do
29.     if ( Accept[child] = ∅ ∨ ¬isConsistent(PA, Accept[child]) ) then
30.         return ∅ ;
31.     else PA ← PA ∪ Accept[child] ;
32. return PA ;
```

In function `BuildAccept`, if an accept partial solution is reached. A_i generates a partial solution PA incrementing its AgentView with its assignment (line 27). Next, A_i loops over the set of ***accept*** messages received from its children. If at least one *child* has never sent an ***accept*** message or the ***accept*** message is inconsistent with PA, then the partial solution has not yet been reached and the function returns empty (line 30). Otherwise, the partial solution PA is incremented by the ***accept*** message of *child* (line 31). Finally, the accept partial solution is returned (line 32).

3.5 Correctness Proofs

Theorem 3.1. *The spatial complexity of AFC-tree is polynomially bounded by $O(nd)$ per agent.*

Proof. In AFC-tree, the size of nogoods is bounded by h ($h \leq n$), the height of the pseudo-tree where n is the total number of variables. Now, on each agent, AFC-tree only stores one nogood per removed value. Thus, the space complexity of nogoods storage is in $O(hd)$ on each agent. AFC-tree also stores its set of descendants and ancestors, which is bounded by n on each agent. Therefore, AFC-tree has a space complexity in $O(hd + n)$. □

Theorem 3.2. *AFC-tree algorithm is correct.*

Proof. AFC-tree agents only forward consistent partial assignments (CPAs). Hence, leaf agents receive only consistent CPAs. Thus, leaf agents only send ***accept*** message holding consistent assignments to their parent. Since a parent builds an ***accept*** message only when the ***accept*** messages received from all its children are consistent with each other and all consistent with its own value, the ***accept*** message it sends contains a consistent partial solution. The root broadcasts a solution only when it can build itself such an ***accept*** message. Therefore, the solution is correct and AFC-tree is sound.

From Lemma 2.1 we deduce that the AFC-tree agent of highest priority cannot fall into an infinite loop. By induction on the level of the pseudo-tree no agent can fall in such a loop, which ensures the termination of AFC-tree. AFC-tree performs multiple AFC-ng processes on the paths of the pseudo-tree from the root to the leaves. Thus, from Lemma 2.2, AFC-tree inherits the property that an empty nogood cannot be inferred if the network is solvable. As AFC-tree s terminates, this ensures its completeness. □

3.6 Experimental Evaluation

In this section we experimentally compare AFC-tree to our AFC-ng presented previously in Chapter 2. Algorithms are evaluated on three benchmarks: uniform binary random DisCSPs, distributed sensor-target networks and distributed meeting scheduling problems. All experiments were performed on the DisChoco 2.0 platform [1] [**Wahbi *et al.*, 2011**], in which agents are simulated by Java threads that communicate only through message passing (see Chapter 7). All algorithms are tested using the same nogood selection heuristic (*HPLV*) [**Hirayama and Yokoo, 2000**].

We evaluate the performance of the algorithms by communication load [**Lynch, 1997**] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution (#*msg*), including those of termination detection for AFC-tree. Computation effort is measured by the number of non-concurrent constraint checks (#*nccs*) [**Zivan and Meisels, 2006b**]. #*nccs* is the metric used in distributed constraint solving to simulate the computation time.

1. http://www2.lirmm.fr/coconut/dischoco/

3.6.1 Uniform binary random DisCSPs

The algorithms are tested on uniform binary random DisCSPs which are characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of agents/variables, d is the number of values in each of the domains, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of constraint graphs: sparse graphs $\langle 20, 10, 0.2, p_2 \rangle$ and dense ones $\langle 20, 10, 0.7, p_2 \rangle$. We vary the tightness from 0.1 to 0.9 by steps of 0.05. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. We report average over the 100 runs.

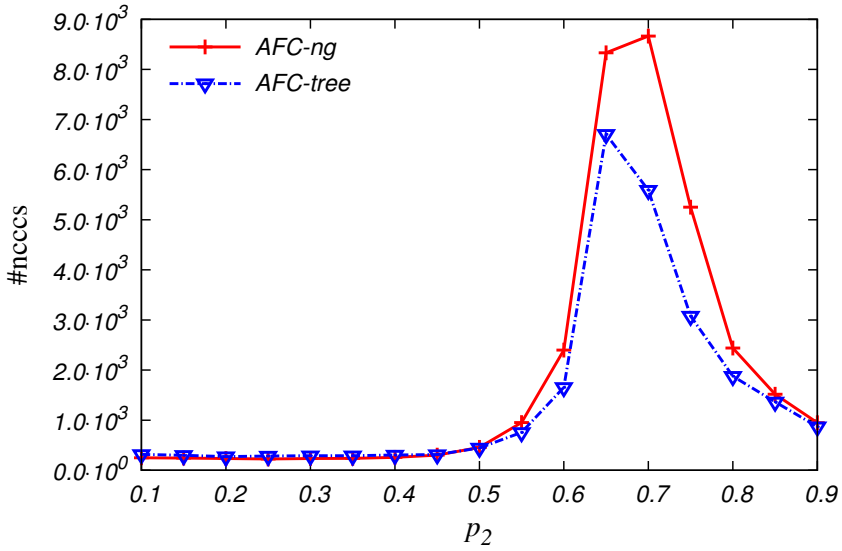


Figure 3.5 – The number of non-concurrent constraint checks (#*nccccs*) performed on sparse problems ($p_1 = 0.2$).

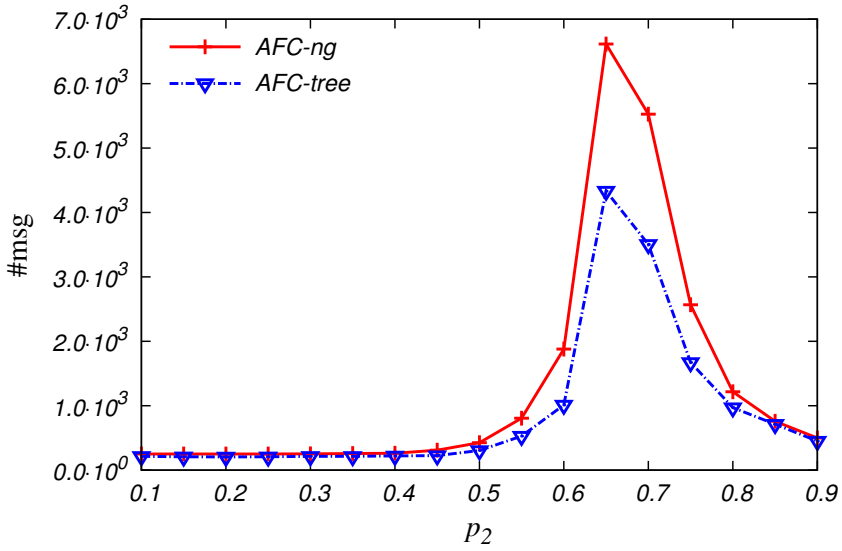


Figure 3.6 – The total number of messages sent on sparse problems ($p_1 = 0.2$).

Figures 3.5 and 3.6 present the performance of AFC-tree and AFC-ng run on the sparse instances (p_1=0.2). In terms of computational effort (Figures 3.5), we observe that at the complexity peak, AFC-tree takes advantage of the pseudo-tree arrangement to improve the speed-up of AFC-ng. Concerning communication load (Figure 3.6), AFC-tree improves on our AFC-ng algorithm. The improvement of AFC-tree over AFC-ng is approximately 30% on communication load and 35% on the number of non-concurrent constraint checks.

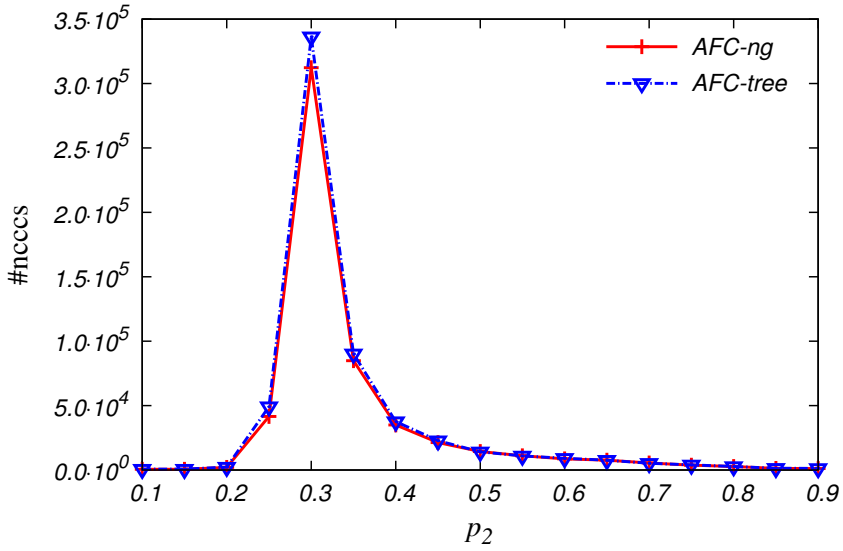


Figure 3.7 – The number of non-concurrent constraint checks (#*ncccs*) performed on the dense problems ($p_1 = 0.7$).

Figures 3.7 and 3.8 illustrates respectively the number of non-concurrent constraint checks (#*ncccs*) performed by compared algorithms and the total number of exchanged messages on the dense problems (p_1=0.7). On these dense graphs, AFC-tree behaves like

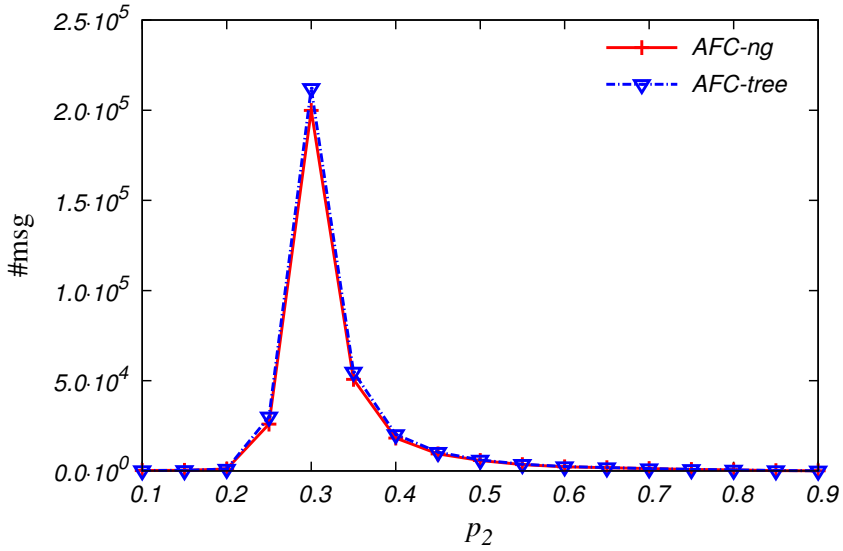


Figure 3.8 – The total number of messages sent on the dense problems ($p_1 = 0.7$).

AFC-ng with a very slight domination of AFC-ng. This is AFC-tree does not benefit from the pseudo-tree arrangement, which is like a chain-tree in such graphs.

3.6.2 Distributed Sensor Target Problems

The *Distributed Sensor-Target Problem* (SensorDisCSP) [**Béjar *et al.*, 2005**] is a benchmark based on a real distributed problem (see Section 1.3.2.2). It consists of n sensors that track m targets. Each target must be tracked by 3 sensors. Each sensor can track at most one target. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a target is visible. The compatibility constraint defines the compatibility among sensors. In our implementation of the DisCSP algorithms, the encoding of the SensorDisCSP presented in Section 1.3.2.2 is translated to an equivalent formulation where we have three virtual agents for every real agent, each virtual agent handling a single variable.

Problems are characterized by $\langle n, m, p_c, p_v \rangle$, where n is the number of sensors, m is the number of targets, each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each target can be tracked by a fraction p_v of the sensors having the target in their sensing range. We present results for the class $\langle$ 25, 5, 0.4, $p_v \rangle$, where we vary p_v from 0.1 to 0.9 by steps of 0.05. Again, for each pair (p_c, p_v) we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

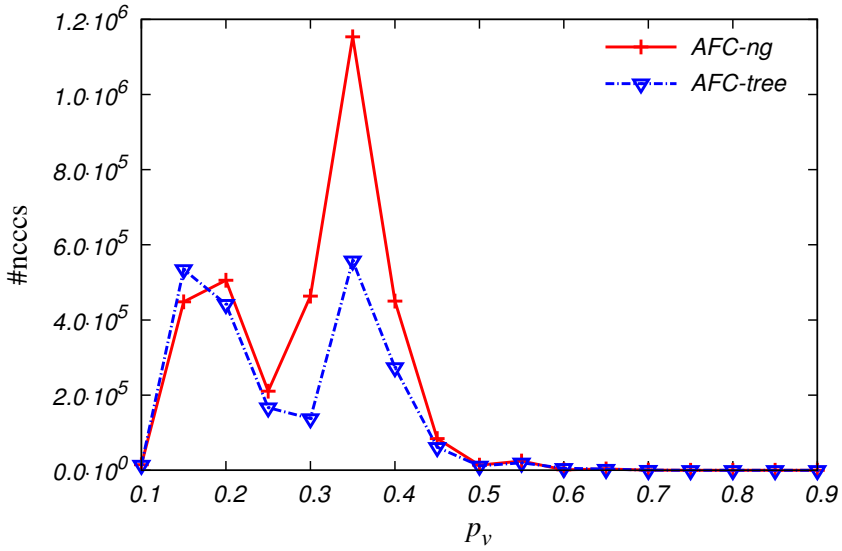


Figure 3.9 – Total number non-concurrent constraint checks performed on instances where $p_c = 0.4$.

We present the results obtained on the SensorDisCSP benchmark in Figures 3.9 and Figure 3.10. Our experiments shows that AFC-tree outperforms AFC-ng algorithm when comparing the computational effort (Figure 3.9). Concerning the communication load (Figure 3.10), the ranking of algorithms is similar to that on computational effort for the instances at the complexity peak. However it is slightly dominated by the AFC-ng on the

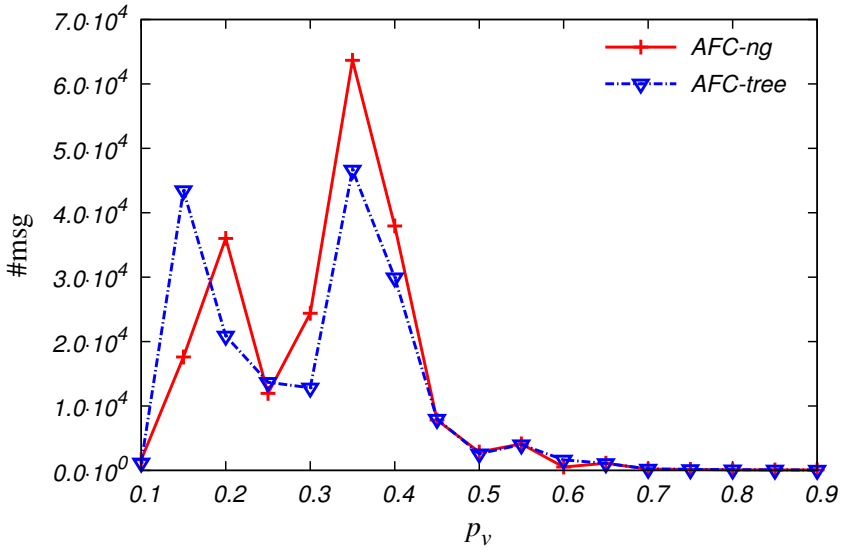


Figure 3.10 – Total number of exchanged messages on instances where $p_c = 0.4$.

exceptionally hard problems ($p_v = 1.5$). Hence, AFC-tree remains the best on all problems except for a single point ($p_v = 1.5$), where AFC-ng shows a trivial improvement.

3.6.3 Distributed Meeting Scheduling Problems

The *Distributed Meeting Scheduling Problem* (DisMSP) is a truly distributed benchmark where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [**Wallace and Freuder, 2002**; **Meisels and Lavee, 2004**] (see Section 1.3.2.1). The DisMSP consists of a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location.

We encode the DisMSP in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. These k meetings are selected randomly among the m meetings. The domain of each variable contains the $d \times h$ slots where a meeting can be scheduled. A slot is one hour long, and there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. We place meetings randomly on the nodes of a uniform grid of size $g \times g$ and the traveling time between two adjacent nodes is 1 hour. Thus, the traveling time between two meetings equals the Euclidean distance between nodes representing the locations where they will be held. For varying the tightness of the arrival-time constraint we vary the size of the grid on which meetings are placed.

Problems are characterized by $\langle n, m, k, d, h, g \rangle$, where n is the number of agents, m is the number meetings, k is the number of meetings/variables per agent, d is the number of days and h is the number of hours per day, and g is the grid size. The duration of each meeting is one hour. In our implementation of the DisCSP algorithms, this encoding is translated to an equivalent formulation where we have k (number of meetings per agent)

virtual agents for every real agent, each virtual agent handling a single variable. We present results for the class $\langle$ 20, 9, 3, 2, 10, $g\rangle$ where we vary g from 2 to 22 by steps of 2. Again, for each g we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

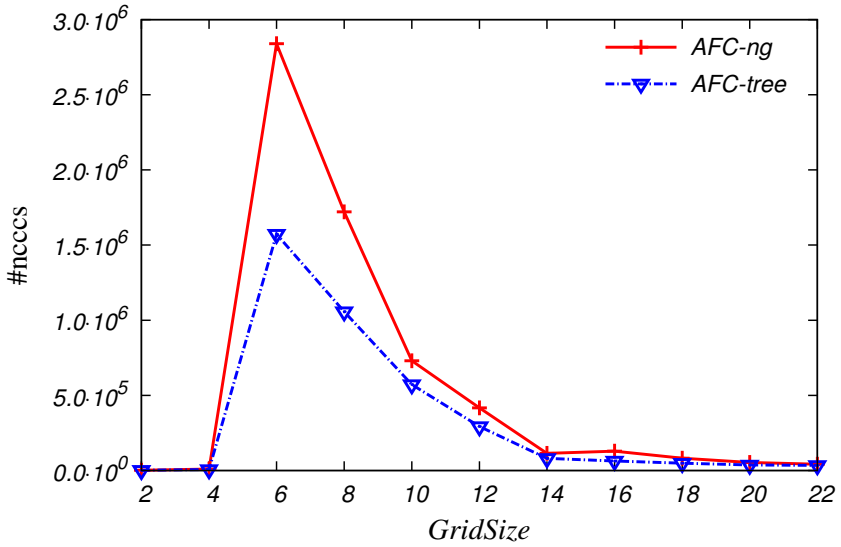


Figure 3.11 – Total number of non-concurrent constraint checks performed on meeting scheduling benchmarks where the number of meeting per agent is 3 (i.e., $k = 3$).

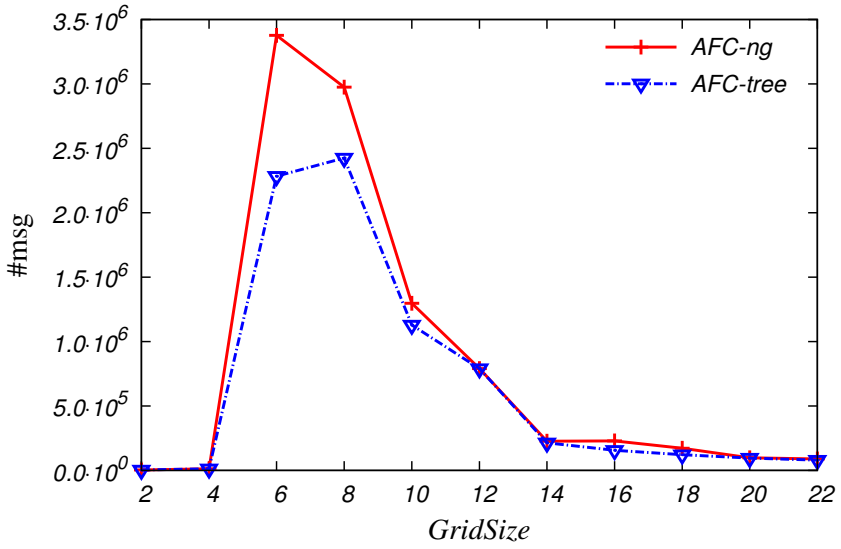


Figure 3.12 – Total number of exchanged messages on meeting scheduling benchmarks where the number of meeting per agent is 3 (i.e., $k = 3$).

On this class of meeting scheduling benchmarks AFC-tree continues to perform well compared to AFC-ng. AFC-tree is significantly better than AFC-ng both for computational effort (Figure 3.11) and communication load (Figure 3.12). The improvement on the complexity peak approximates 45% for the number of non-concurrent constraint checks. Regarding the number of exchanged messages this improvement approximates 30%.

3.6.4 Discussion

Our experiments demonstrates that AFC-tree is almost always better than or equivalent to AFC-ng both in terms of communication load and computational effort. When the graph is sparse, AFC-tree benefits from running separate search processes in disjoint problem subtrees. When agents are highly connected (dense graphs), AFC-tree runs on a chain-tree pseudo-tree and thus mimics AFC-ng.

3.7 Other Related Works

The Synchronous Backtracking (SBT) [Yokoo, 2000b] is the naive search method for solving distributed CSPs. SBT is a decentralized extension of the chronological backtracking algorithm for centralized CSPs. Although this algorithm communicates only consistent current partial assignments (CPA), it does not take advantage of parallelism, because the problem is solved sequentially and only the agent holding the current partial assignments is activated while other agents are in an idle state. Collin *et al.* (1991) proposed *Network Consistency Protocol* (NCP) a variation of the synchronous backtracking. NCP agents are prioritized using a depth-first search tree. Despite the fact that agents on the same branch act synchronously, agents having the same parent can act concurrently. Thus, instead of given the privilege to only one agent, as is done in SBT, an agent passes on the privilege of extending the CPA or backtracking to all its children concurrently.

In Interleaved Asynchronous Backtracking (IDIBT) [Hamadi, 2002], agents participate in multiple processes of asynchronous backtracking. Each agent keeps a separate AgentView for each search process in IDIBT. The number of search processes is fixed by the first agent in the ordering. The performance of concurrent asynchronous backtracking [Hamadi, 2002] was tested and found to be ineffective for more than two concurrent search processes [Hamadi, 2002].

3.8 Summary

A new complete, asynchronous algorithm, which needs polynomial space is presented. This algorithm called Asynchronous Forward-Checking Tree (AFC-tree), is based on our nogood-based Asynchronous Forward Checking (AFC-ng) and is performed on a pseudo-tree arrangement of the constraint graph. AFC-tree runs simultaneous AFC-ng processes on each branch of the pseudo-tree to exploit the parallelism inherent in the problem. Our experiments show that AFC-tree is more robust than AFC-ng. It is particularly good when the problems are sparse because it takes advantage of the pseudo-tree ordering.

4 Maintaining Arc Consistency Asynchronously in Synchronous Distributed Search

Contents

We presented in (Chapter 2) our Nogood-Based Asynchronous Forward Checking (AFC-ng). AFC-ng is an efficient and robust algorithm for solving Distributed Constraint Satisfaction Problems (DisCSPs). AFC-ng performs an asynchronous forward checking phase during synchronous search. In this chapter, we propose two new algorithms based on the same mechanism as AFC-ng. However, instead of using forward checking as a filtering property, we propose to maintain arc consistency asynchronously (MACA). The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages. The second algorithm, MACA-not, achieves arc consistency without any new type of message. We provide a theoretical analysis and an experimental evaluation of the proposed approach. Our experiments show the good performance of MACA algorithms, particularly those of MACA-not.

This chapter is organized as follows. Section 4.1 introduces the previous works for maintenance of arc consistency in DisCSPs and situates our contribution in the domain. Section 4.2 recalls the necessary background on MAC. Sections 4.3 describes the MACA-del and MACA-not algorithms. Theoretical analysis and correctness proofs are given in Section 4.4. Section 4.5 presents an experimental evaluation of our proposed algorithms against state-of-the-art other algorithms. Finally, we will conclude this chapter in Section 4.6.

4.1 Introduction

We presented in Chapter 1 many backtrack search algorithms that were developed for solving constraint satisfaction problems. Typical backtrack search algorithms try to build a solution to a CSP by interleaving variable instantiation with constraint propagation. Forward Checking (FC) [**Haralick and Elliott, 1980**] and Maintaining Arc Consistency (MAC) [**Sabin and Freuder, 1994**] are examples of such algorithms. In the 80's, FC was considered as the most efficient search algorithm. In the middle 90's, several works have empirically shown that MAC is more efficient than FC on hard and large problems [**Bessiere and Régin, 1996**; **Grant and Smith, 1996**].

Although, many studies incorporated FC successfully in distributed CSPs [**Brito and Meseguer, 2003**; **Meisels and Zivan, 2007**; **Ezzahir *et al.*, 2009**], MAC has not yet been well investigated. The only tentatives to include arc consistency maintenance in distributed algorithms were done on the Asynchronous Backtracking algorithm. **Silaghi *et al.* (2001b)** introduced the Distributed Maintaining Asynchronously Consistency for ABT, (*DMAC-ABT*), the first algorithm able to maintain arc consistency in distributed CSPs [**Silaghi *et al.*, 2001b**]. DMAC-ABT considers consistency maintenance as a hierarchical nogood-based inference. **Brito and Meseguer (2008)** proposed ABT-uac and ABT-dac, two algorithms that connect ABT with arc consistency [**Brito and Meseguer, 2008**]. ABT-uac propagates unconditionally deleted values to enforce an amount of full arc consistency. ABT-dac propagates conditionally and unconditionally deleted values using directional arc consistency. ABT-uac shows minor improvement in communication load and ABT-dac is harmful in many instances.

In this chapter, we propose two new synchronous search algorithms based on the same mechanism as AFC-ng. However, instead of maintaining forward checking asynchronously on agents not yet instantiated, we propose to maintain arc consistency asynchronously on these future agents. We call this new scheme MACA, for *maintaining arc consistency asynchronously*. As in AFC-ng, only the agent holding the current partial assignment (CPA) can perform an assignment. However, unlike AFC-ng, MACA attempts to maintain the arc consistency instead of performing only FC. The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages (***del***). Hence, whenever values are removed during a constraint propagation step, MACA-del agents notify other agents that may be affected by these removals, sending them a ***del*** message. ***del*** messages contain all removed values and the nogood justifying their removal. The second algorithm, MACA-not, achieves arc consistency without any new type of message. We achieve this by storing all deletions performed by an agent on domains of its neighboring agents, and sending this information to these neighbors within the CPA message.

4.2 Maintaining Arc Consistency

Constraint propagation is a central feature of efficiency for solving CSPs [Bessiere, 2006]. The oldest and most commonly used technique for propagating constraints is arc consistency (AC).

The Maintaining Arc Consistency (MAC) algorithm [Sabin and Freuder, 1994] alternates exploration steps and constraint propagation steps. That is, at each step of the search, a variable assignment is followed by a filtering process that corresponds to enforcing arc consistency. For implementing MAC in a distributed CSP, Each agent A_i is assumed to know all constraints in which it is involved and the agents with whom it shares a constraint (i.e., $\Gamma(x_i)$). These agents and the constraints linking them to A_i form the local constraint network of A_i, denoted by $CSP(i)$.

Definition 4.1 *The **local constraint network** $CSP(i)$ of an agent $A_i \in \mathcal{A}$ consists of all constraints involving x_i and all variables of these constraints (i.e., its neighbors).*

In order to allow agents to maintain arc consistency in distributed CSPs, our proposed approach consists in enforcing arc consistency on the local constraint network of each agent. Basically, each agent A_i stores locally copies of all variables in $CSP(i)$. We also assume that each agent knows the neighborhood it has in common with its own neighbors, without knowing the constraints relating them. That is, for each of its neighbors A_k, an agent A_i knows the list of agents A_j such that there is a constraint between x_i and x_j and a constraint between x_k and x_j.

Agent A_i stores nogoods for its removed values. They are stored in $NogoodStore[x_i]$. But in addition to nogoods stored for its own values, A_i needs to store nogoods for values removed from variables x_j in $CSP(i)$. Nogoods justifying the removals of values from $D(x_j)$ are stored in $NogoodStore[x_j]$. Hence, the NogoodStore of an agent A_i is a vector of several NogoodStores, one for each variable in $CSP(i)$.

4.3 Maintaining Arc Consistency Asynchronously

In AFC-ng, the forward checking phase aims at anticipating the backtrack. Nevertheless, we do not take advantage of the value removals caused by FC if it does not completely wipe out the domain of the variable. One can investigate these removals by enforcing arc consistency. This is motivated by the fact that the propagation of a value removal, for an agent A_i, may generate an empty domain for a variable in its local constraint network $CSP(i)$. We can then detect an earlier dead-end and then anticipate as soon as possible the backtrack operation.

In synchronous search algorithms for solving DisCSPs, agents sequentially assign their variables. Thus, agents perform the assignment of their variable only when they hold the current partial assignment, CPA. We propose an algorithm in which agents assign their variable one by one following a total ordering on agents. Hence, whenever an agent succeeds in extending the CPA by assigning its variable on it, it sends the CPA to its successor to extend it. Copies of this CPA are also sent to the other agents whose assignments are not

yet on the CPA in order to *maintain arc consistency asynchronously*. Therefore, when an agent receives a copy of the CPA, it maintains arc consistency in its local constraint network. To enforce arc consistency on all variables of the problem, agents communicate information about value removals produced locally with other agents. We propose two methods to achieve this. The first method, namely MACA-del, uses a new type of messages (***del*** messages) to share this information. The second method, namely MACA-not, includes the information about deletions generated locally within ***cpa*** messages.

4.3.1 Enforcing AC using *del* messages (MACA-del)

In MACA-del, each agent A_i maintains arc consistency on its local constraint network, $CSP(i)$ whenever a domain of a variable in $CSP(i)$ is changed. Changes can occur either on the domain of A_i or on another domain in $CSP(i)$. In MACA-del on agent A_i, only removals on $D(x_i)$ are externally shared with other agents. The propagation of the removals on $D(x_i)$ is achieved by communicating to other agents the nogoods justifying these removals. These removals and their associated nogoods are sent to neighbors via ***del*** messages.

The pseudo code of MACA-del, executed by each agent A_i, is shown in Algorithm 4.1. Agent A_i starts the search by calling procedure `MACA-del()`. In procedure `MACA-del()`, A_i calls function `Propagate()` to enforce arc consistency (line 1) in its local constraint network, i.e., $CSP(i)$. Next, if A_i is the initializing agent IA (the first agent in the agent ordering), it initiates the search by calling procedure `Assign()` (line 2). Then, a loop considers the reception and the processing of the possible message types.

When calling procedure `Assign()`, A_i tries to find an assignment which is consistent with its AgentView. If A_i fails to find a consistent assignment, it calls procedure `Backtrack()` (line 12). If A_i succeeds, it increments its counter t_i and generates a CPA from its AgentView augmented by its assignment (lines 9 and 10). Afterwards, A_i calls procedure `SendCPA(CPA)` (line 11). If the CPA includes all agents assignments (A_i is the lowest agent in the order, line 13), A_i reports the CPA as a solution of the problem and marks the *end* flag *true* to stop the main loop (line 13). Otherwise, A_i sends forward the CPA to all agents whose assignments are not yet on the CPA (line 14). So, the next agent on the ordering (successor) will try to extend this CPA by assigning its variable on it while other agents will maintain arc consistency asynchronously.

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA()` is called (line 6). The received message will be processed only when it holds a CPA stronger than the AgentView of A_i. If it is the case, A_i updates its AgentView (line 16) and then updates the NogoodStore of each variable in $CSP(i)$ to be compatible with the received CPA (line 17). Afterwards, A_i calls function `Propagate()` to enforce arc consistency on $CSP(i)$ (line 18). If arc consistency wiped out a domain in $CSP(i)$ (i.e., $CSP(i)$ is not arc consistent), A_i calls procedure `Backtrack()` (line 18). Otherwise, A_i checks if it has to assign its variable (line 19). A_i tries to assign its variable by calling procedure `Assign()` only if it received the ***cpa*** from its predecessor.

When calling function `Propagate()`, A_i restores arc consistency on its local constraint network according to the assignments on its AgentView (line 20). In our implementation

Algorithm 4.1: MACA-del algorithm running by agent A_i.

```
procedure MACA-del()
01. end ← false;  Propagate() ;
02. if ( A_i = IA ) then Assign() ;
03. while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.       cpa  : ProcessCPA(msg);        ngd  : ProcessNogood(msg);
07.       del  : ProcessDel(msg);        stp  : end ← true;
procedure Assign()
08. if ( D(x_i) ≠ ∅ ) then
09.    v_i ← ChooseValue() ; t_i ← t_i+1 ;
10.    CPA ← {AgentView ∪ (x_i, v_i, t_i)} ;
11.    SendCPA(CPA) ;
12. else Backtrack() ;
procedure SendCPA(CPA)
13. if ( size(CPA) = n ) then broadcastMsg: stp(CPA) ;  end ← true ;
14. else foreach ( x_k ≻ x_i ) do sendMsg: cpa(CPA) to A_k ;
procedure ProcessCPA(msg)
15. if ( msg.CPA is stronger than the AgentView ) then
16.    AgentView ← CPA ;
17.    Remove all nogoods incompatible with AgentView ;
18.    if ( ¬Propagate() ) then Backtrack() ;
19.    else if ( msg.sender = predecessor(A_i) ) then Assign();
function Propagate()
20. if ( ¬AC(CSP(i)) ) then return false ;
21. else if ( D(x_i) was changed ) then
22.    foreach ( x_j ∈ CSP(i) ) do
23.       nogoods ← get nogoods from NogoodStore[x_i] that are relevant to x_j ;
24.       sendMsg: del(nogoods) to A_j ;
25. return true ;
procedure ProcessDel(msg)
26. foreach ( ng ∈ msg.nogoods such that Compatible(ng, AgentView) ) do
27.    add(ng, NogoodStore[x_k]) ;                    /* A_k is the agent that sent msg */
28. if ( D(x_k) = ∅ ∧ x_i ∈ NogoodStore[x_k] ) then
29.    add(solve(NogoodStore[x_k]), NogoodStore[x_i]) ;  Assign() ;
30. else if ( D(x_k) = ∅ ∨ ¬Propagate() ) then Backtrack() ;
procedure Backtrack()
31. newNogood ← solve(NogoodStore[x_k]) ;          /* x_k is a variable such that D(x_k)=∅ */
32. if ( newNogood = empty ) then broadcastMsg: stp(∅);  end ← true ;
33. else                                  /* Let x_j be the variable on the rhs(newNogood) */
34.    sendMsg: ngd(newNogood) to A_j ;
35.    foreach ( x_l ≽ x_j ) do AgentView[x_l].Value ← empty ;
36.    Remove all nogoods incompatible with AgentView ;
procedure ProcessNogood(msg)
37. if ( Compatible(lhs(msg.nogood), AgentView) ) then
38.    add(msg.nogood, NogoodStore[x_i]) ; /* using to the HPLV Hirayama and Yokoo (2000) */
39.    if ( rhs(msg.nogood).Value = v_i ) then Assign();
40.    else if ( ¬Propagate() ) then Backtrack() ;
```

we used *AC-2001* [**Bessiere and Régin, 2001**] to enforce arc consistency but any generic AC algorithm can be used. MACA-del requires from the algorithm enforcing arc consistency to store a nogood for each removed value. When two nogoods are possible for the same value, we select the best with the *Highest Possible Lowest Variable* heuristic [**Hirayama and Yokoo, 2000**]. If enforcing arc consistency on $CSP(i)$ has failed, i.e., a domain was wiped out, the function returns $false$ (line 20). Otherwise, if the domain of x_i was changed (i.e., there are some deletions to propagate), A_i informs its constrained agents by sending them ***del*** messages that contain nogoods justifying these removals (lines 23-24). Finally, the function returns *true* (line 25). When sending a ***del*** message to a neighboring agent A_j, only nogoods such that all variables in their left hand sides have a higher priority than A_j will be communicated to A_j. Furthermore, all nogoods having the same left hand side are factorized in one single nogood whose right hand side is the set of all values removed by this left hand side.

Whenever A_i receives a ***del*** message, it adds to the NogoodStore of the sender, say A_k, (i.e., $NogoodStore[x_k]$) all nogoods compatible with the AgentView of A_i (lines 26-27). Afterward, A_i checks if the domain of x_k is wiped out (i.e., the remaining values in $D(x_k)$ are removed by nogoods that have just been received from A_k) and x_i belongs to the NogoodStore of x_k (i.e., x_i is already assigned and its current assignment is included in at least one nogood removing a value from $D(x_k)$) (line 28). If it is the case, A_i removes its current value by storing the resolved nogood from the NogoodStore of x_k (i.e., `solve(`$NogoodStore[x_k]$`)`) as justification of this removal, and then calls procedure `Assign()` to try an other value (line 29). Otherwise, when $D(x_k)$ is wiped-out (x_i is not assigned) or if a dead-end occurs when trying to enforce arc consistency, A_i has to backtrack and thus it calls procedure `Backtrack()` (line 30).

Each time a dead-end occurs on a domain of a variable x_k in $CSP(i)$ (including x_i), the procedure `Backtrack()` is called. The nogoods that generated the dead-end are resolved by computing a new nogood *newNogood* (line 31). *newNogood* is the conjunction of the left hand sides of all these nogoods stored by A_i in $NogoodStore[x_k]$. If the new nogood *newNogood* is empty, A_i terminates execution after sending a ***stp*** message to all agents in the system meaning that the problem is unsolvable (line 32). Otherwise, A_i backtracks by sending a ***ngd*** message to agent A_j, the owner of the variable on the right hand side of *newNogood* (line 34). Next, A_i updates its AgentView in order to keep only assignments of agents that are placed before A_j in the total ordering (line 35). A_i also updates the NogoodStore of all variables in $CSP(i)$ by removing nogoods incompatible with its new AgentView (line 36).

Whenever a ***ngd*** message is received, A_i checks the validity of the received nogood (line 37). If the received nogood is compatible with its AgentView, A_i adds this nogood to its NogoodStore (i.e. $NogoodStore[x_i]$, line 38). Then, A_i checks if the value on the right hand side of the received nogood equals its current value (v_i). If it is the case, A_i calls the procedure `Assign()` to try another value for its variable (line 39). Otherwise, A_i calls function `Propagate()` to restore arc consistency. When a dead-end was generated in its local constraint network, A_i calls procedure `Backtrack()` (line 40).

4.3.2 Enforcing AC without additional kind of message (MACA-not)

In the following, we show how to enforce arc consistency without additional kind of messages. In MACA-del, global consistency maintenance is achieved by communicating to constrained agents (agents in $CSP(i)$) all values pruned from $D^0(x_i)$. This may generate many ***del*** messages in the network, and then result in a communication bottleneck. In addition, many ***del*** messages may lead agents to perform more efforts to process them. In MACA-not, communicating the removals produced in $CSP(i)$ is delayed until the agent A_i wants to send a ***cpa*** message. When sending the ***cpa*** message to a lower priority agent A_k, agent A_i attaches nogoods justifying value removals from $CSP(i)$ to the ***cpa*** message. But it does not attach all of them because some variables are irrelevant to A_k (not connected to x_k by a constraint).

MACA-not shares with A_k all nogoods justifying deletions on variables not yet instantiated that share a constraint with both A_i and A_k (i.e., variables in $\{CSP(i) \cap CSP(k)\} \setminus$ `vars(`CPA`)`). Thus, when A_k receives the ***cpa*** it receives also deletions performed in $CSP(i)$ that can lead him to more arc consistency propagation.

Algorithm 4.2: New lines/procedures for MACA-not with respect to MACA-del.

```
procedure MACA-not()
01. end ← false;  Propagate() ;
02. if ( A_i = IA ) then Assign() ;
03. while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.       cpa  : ProcessCPA(msg);          ngd  : ProcessNogood(msg);
07.       stp      : end ← true;
procedure SendCPA(CPA)
08. if ( size(CPA) = n ) then broadcastMsg: stp(CPA) ;  end ← true ;
09. else
10.    foreach ( x_k ≻ x_i ) do
11.       nogoods ← ∅;
12.       foreach ( x_j ∈ {CSP(i) ∩ CSP(k)} such that x_j ≻ x_i ) do
13.          nogoods ← nogoods ∪ getNogoods(x_j) ;
14.       sendMsg: cpa(CPA, nogoods) to A_k ;
procedure ProcessCPA(msg)
15. if ( msg.CPA is stronger than the AgentView ) then
16.    AgentView ← CPA ;
17.    Remove all nogoods incompatible with AgentView ;
18.    foreach ( nogoods ∈ msg.nogoods ) do add(nogoods, NogoodStore) ;
19.    if ( ¬Propagate() ) then Backtrack() ;
20.    else if ( msg.sender = predecessor(A_i) ) then Assign();
function Propagate()
21. return AC(CSP(i)) ;
```

We present in Algorithm 4.2 the pseudo-code of MACA-not algorithm. Only procedures that are new or different from those of MACA-del in Algorithm 4.1 are presented. Function `Propagate()` no longer sends ***del*** messages, it just maintains arc consistency on $CSP(i)$ and returns *true* iff no domain was wiped out.

In procedure `SendCPA(CPA)`, when sending a ***cpa*** message to an agent A_k, A_i attaches

to the CPA the nogoods justifying the removals from the domains of variables in $CSP(i)$ constrained with A_k (lines 10-14, Algorithm 4.2).

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA()` is called (line 6). The received message will be processed only when it holds a CPA stronger than the AgentView of A_i. If it is the case, A_i updates its AgentView (line 16) and then updates the NogoodStore to be compatible with the received CPA (line 17). Next, all nogoods contained in the received message are added to the NogoodStore (line 18). Obviously, nogoods are added to the NogoodStore referring to the variable in their right hand side (i.e., ng is added to $NogoodStore[x_j]$ if x_j is the variable in `rhs(`ng`)`). Afterwards, A_i calls function `Propagate()` to restore arc consistency in $CSP(i)$ (line 19). If a domain of a variable in $CSP(i)$ wiped out, A_i calls procedure `Backtrack()` (line 19). Otherwise, A_i checks if it has to assign its variable (line 20). A_i tries to assign its variable by calling procedure `Assign()` only if it received the ***cpa*** from its predecessor.

4.4 Theoretical analysis

We demonstrate that MACA is sound, complete and terminates, with a polynomial space complexity.

Lemma 4.1. MACA is guaranteed to terminate.

Proof. (Sketch) The proof is close to the one given in Lemma 2.1, Chapter 2. It can easily be obtained by induction on the agent ordering that there will be a finite number of new generated CPAs (at most nd, where n the number of variables and d is the maximum domain size), and that agents can never fall into an infinite loop for a given CPA. □

Lemma 4.2. MACA cannot infer inconsistency if a solution exists.

Proof. Whenever a stronger ***cpa*** or a ***ngd*** message is received, MACA agents update their NogoodStores. In MACA-del, the nogoods contained in ***del*** are accepted only if they are compatible with AgentView (line 27, Algorithm 4.1). In MACA-not, the nogoods included in the ***cpa*** messages are compatible with the received CPA and they are accepted only when the CPA is stronger than AgentView (line 15, Algorithm 4.2). Hence, for every CPA that may potentially lead to a solution, agents only store valid nogoods. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, the empty nogood cannot be inferred if the network is satisfiable. □

Theorem 4.1. *MACA is correct.*

Proof. The argument for soundness is close to the one given in Theorem 2.2, Chapter 2. The fact that agents only forward consistent partial solution on the ***cpa*** messages at only one place in procedure `Assign()` (line 11, Algorithm 4.1), implies that the agents receive only consistent assignments. A solution is found by the last agent only in procedure `SendCPA(CPA)` at (line 13, Algorithm 4.1 and line 8, Algorithm 4.2). At this point, all agents have assigned their variables, and their assignments are consistent. Thus MACA is

sound. Completeness comes from the fact that MACA is able to terminate and does not report inconsistency if a solution exists (Lemma 4.1 and 4.2). □

Theorem 4.2. *MACA is polynomial in space.*

Proof. On each agent, MACA stores one nogood of size at most n per removed value in its local constraint network. The local constraint network contains at most n variables. Thus, the space complexity of MACA is in $O(n^2d)$ on each agent where d is the maximal initial domain size. □

Theorem 4.3. *MACA messages are polynomially bounded.*

Proof. The largest messages for MACA-del are ***del*** messages. In the worst-case, a ***del*** message contains a nogood for each value. Thus, the size of ***del*** messages is in $O(nd)$. In MACA-not, the largest messages are ***cpa*** messages. The worst-case is a ***cpa*** message containing a CPA and one nogood for each value of each variable in the local constraint network. Thus, the size of a ***cpa*** message is in $O(n + n^2d) = O(n^2d)$. □

4.5 Experimental Results

In this section we experimentally compare MACA algorithms to ABT-uac, ABT-dac [**Brito and Meseguer, 2008**] and AFC-ng Chapter 2. These algorithms are evaluated on uniform random binary DisCSPs. All experiments were performed on the DisChoco 2.0 platform [1] [**Wahbi *et al.*, 2011**], in which agents are simulated by Java threads that communicate only through message passing. All algorithms are tested on the same static agents ordering (lexicographic ordering) and the same nogood selection heuristic (*HPLV*) [**Hirayama and Yokoo, 2000**]. For ABT-dac we implemented an improved version of **Silaghi**'s solution detection [**Silaghi, 2006**] and counters for tagging assignments.

We evaluate the performance of the algorithms by communication load [**Lynch, 1997**] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution (#*msg*), including those of termination detection (system messages). Computation effort is measured by the number of non-concurrent constraint checks (#*ncccs*) [**Zivan and Meisels, 2006b**]. #*ncccs* is the metric used in distributed constraint solving to simulate the computation time.

The algorithms are tested on uniform random binary DisCSPs which are characterized by $\langle n, d, p_1, p_2\rangle$, where n is the number of agents/variables, d is the number of values in each of the domains, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of constraint networks: sparse networks $\langle 20, 10, 0.25, p_2\rangle$ and dense ones $\langle 20, 10, 0.7, p_2\rangle$. We vary the tightness from 0.1 to 0.9 by steps of 0.1. For each pair of fixed density and tightness (p_1, p_2) we generated 100 instances. The average over the 100 instances is reported.

1. http://www.lirmm.fr/coconut/dischoco/

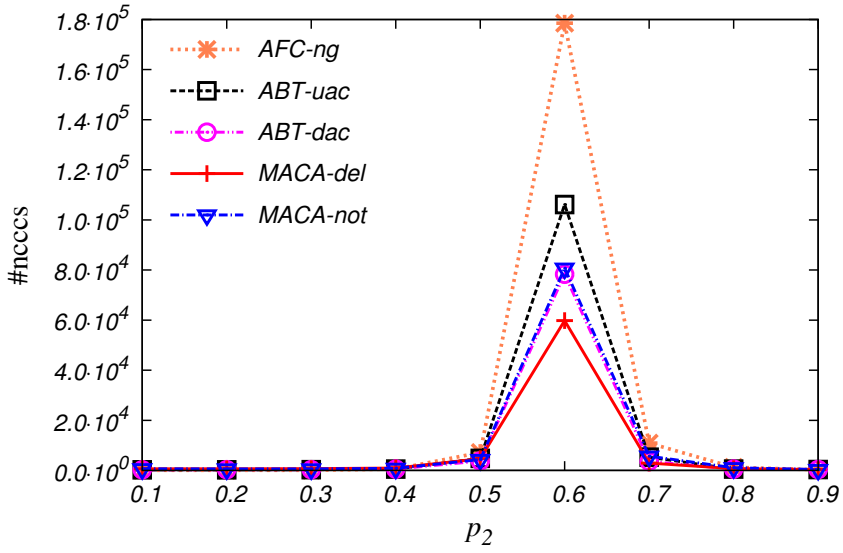


Figure 4.1 – The number of non-concurrent constraint checks (*#ncccs*) performed for solving sparse problems ($p_1 = 0.25$).

First, we present the performance of the algorithms on the sparse instances, $p_1 = 0.25$, (Figures 4.1 and 4.2). Concerning the computational effort (Figure 4.1), algorithms enforcing an amount of arc consistency are better than AFC-ng, which only enforces forward checking. Among these algorithms MACA-del is the fastest one. MACA-not behaves like ABT-dac, which is better than ABT-uac.

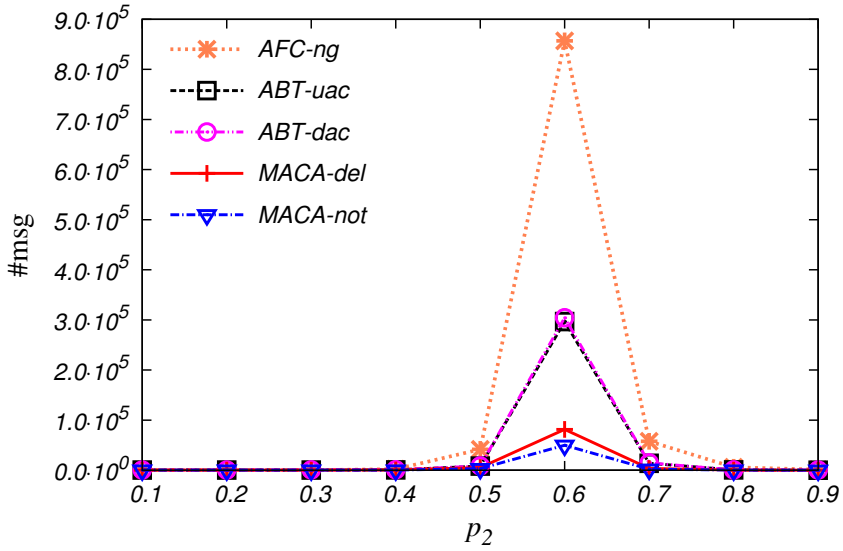


Figure 4.2 – The total number of messages sent for solving sparse problems ($p_1 = 0.25$).

Concerning the communication load (Figure 4.2), algorithms performing an amount of arc consistency improve on AFC-ng by an even larger scale than for computational effort. ABT-uac and ABT-dac require almost the same number of exchanged messages. Among the algorithms maintaining an amount of arc consistency, the algorithms with a synchronous behavior (MACA algorithms) outperform those with an asynchronous behavior (ABT-dac

and ABT-uac) by a factor of 6. It thus seems that on sparse problems, maintaining arc consistency in synchronous search algorithms provides more benefit than in asynchronous ones. MACA-not exchanges slightly fewer messages than MACA-del at the complexity peak.

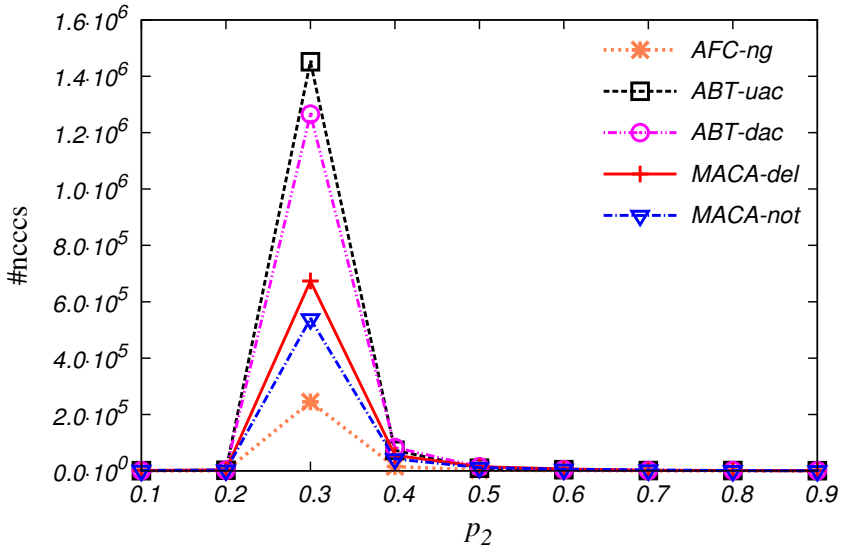


Figure 4.3 – The number of non-concurrent constraint checks (#*ncccs*) performed for solving dense problems ($p_1 = 0.7$).

In the following, we present the performance of the algorithms on the dense instances ($p_1 = 0.7$). Concerning the computational effort (Figure 4.3), the first observation is that asynchronous algorithms are less efficient than those performing assignments sequentially. Among all compared algorithms, AFC-ng is the fastest one on these dense problems. This is consistent with results on centralized CSPs where FC had a better behavior on dense problems than on sparse ones [**Bessiere and Régin, 1996**; **Grant and Smith, 1996**]. As on sparse problems, ABT-dac outperforms ABT-uac. Conversely to sparse problems, MACA-not outperforms MACA-del.

Concerning the communication load (Figure 4.4), on dense problems, asynchronous algorithms (ABT-uac and ABT-dac) require a large number of exchanged messages. MACA-del does not improve on AFC-ng because of a too large number of exchanged ***del*** messages. On these problems, MACA-not is the algorithm that requires the smallest number of messages. MACA-not improves on synchronous algorithms (AFC-ng and MACA-del) by a factor of 11 and on asynchronous algorithms (ABT-uac and ABT-dac) by a factor of 40.

4.5.1 Discussion

From these experiments we can conclude that in synchronous algorithms, maintaining arc consistency is better than maintaining forward checking in terms of computational effort when the network is sparse, and is always better in terms of communication load. We can also conclude that maintaining arc consistency in synchronous algorithms produces

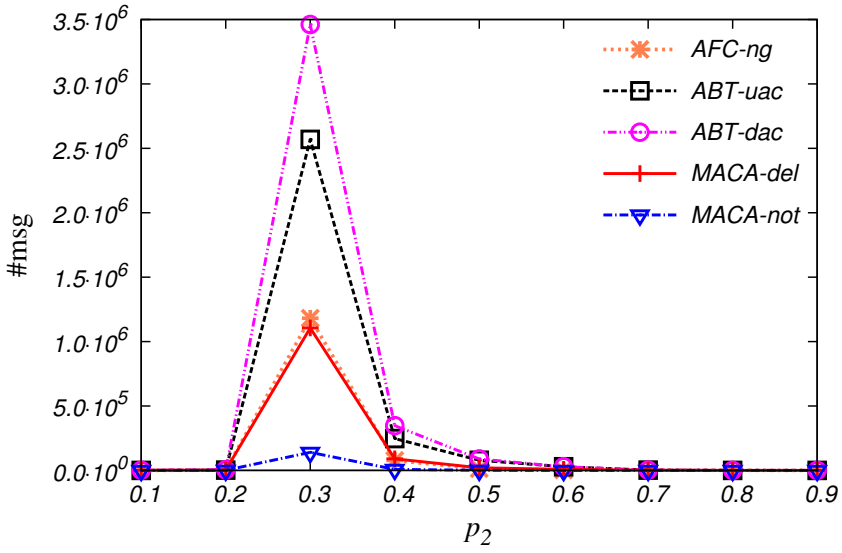


Figure 4.4 – The total number of messages sent for solving dense problems ($p_1 = 0.7$).

much larger benefits than maintaining arc consistency in asynchronous algorithms like ABT.

4.6 Summary

We have proposed two new synchronous search algorithms for solving DisCSPs. These are the first attempts to maintain arc consistency during synchronous search in DisCSPs. The first algorithm, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages. The second algorithm, MACA-not, achieves arc consistency without any new type of message. Despite the synchronicity of search, these two algorithms perform the arc consistency phase asynchronously. Our experiments show that maintaining arc consistency during synchronous search produces much larger benefits than maintaining arc consistency in asynchronous algorithms like ABT. The communication load of MACA-del can be significantly lower than that of AFC-ng, the best synchronous algorithm to date. MACA-not shows even larger improvements thanks to its more parsimonious use of messages.

Agile Asynchronous BackTracking (Agile-ABT)

5

Contents

It is known from centralized CSPs that reordering variables dynamically improves the efficiency of the search procedure. Moreover, reordering in ABT is required in various applications (e.g., security [**Silaghi *et al.*, 2001a**]). All polynomial space algorithms proposed so far to improve Asynchronous Backtracking by reordering agents during search only allow a limited amount of reordering Section 1.4.3. In this chapter, we propose Agile-ABT [**Bessiere *et al.*, 2011**], a search procedure that is able to change the ordering of agents more than previous approaches. This is done via the original notion of termination value, a vector of stamps labeling the new orders exchanged by agents during search. In Agile-ABT, agents can reorder themselves as much as they want as long as the termination value decreases as the search progresses. Agents cooperate without any global control to reduce termination values rapidly, gaining efficiency while ensuring polynomial space complexity. We compare the performance of Agile-ABT with other algorithms and our experiments show the good performance of Agile-ABT when compared to other dynamic reordering techniques.

The rest of the chapter is organized as follows. Section 5.1 introduces and situates our contribution (Agile-ABT). Section 5.2 describes the concepts needed to select new orders that decrease the termination value. We give the details of our algorithm in Section 5.3 and

we prove it in Section 5.4. An experimental evaluation is given in Section 5.5. Section 5.7 concludes the chapter.

5.1 Introduction

Several distributed algorithms for solving DisCSPs have been developed, among which Asynchronous Backtracking (ABT) is the central one [**Yokoo *et al.*, 1998**; **Bessiere *et al.*, 2005**]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. In ABT, the priority order of agents is static, and an agent tries to find an assignment satisfying the constraints with higher priority agents. When an agent sets a variable value, the selected value will not be changed unless an exhaustive search is performed by lower priority agents. Now, it is known from centralized CSPs that adapting the order of variables dynamically during search drastically fastens the search procedure. Moreover, reordering in ABT is required in various applications (e.g., security [**Silaghi *et al.*, 2001a**]).

Asynchronous Weak Commitment (AWC) dynamically reorders agents during search by moving the sender of a nogood higher in the order than the other agents in the nogood [**Yokoo, 1995**]. But AWC requires exponential space for storing nogoods. **Silaghi *et al.*** (**2001c**) tried to hybridize ABT with AWC. Abstract agents fulfill the reordering operation to guarantee a finite number of asynchronous reordering operations. In [**Silaghi, 2006**], the heuristic of the centralized dynamic backtracking [**Ginsberg, 1993**] was applied to ABT. However, in both studies, the improvement obtained on ABT was minor.

Zivan and Meisels (**2006a**) proposed another algorithm for Dynamic Ordering in Asynchronous Backtracking (ABT_DO). When an agent assigns a value to its variable, ABT_DO can reorder only lower priority agents. A new kind of ordering heuristics for ABT_DO is presented in [**Zivan *et al.*, 2009**]. These heuristics, called retroactive heuristics ABT_DO-Retro, enable the generator of the nogood to be moved to a higher position than that of the target of the backtrack. The degree of flexibility of these heuristics is dependent on the size of the nogood storage capacity, which is predefined. Agents are limited to store nogoods equal or smaller than a predefined size K. The space complexity of the agents is thus exponential in K. However, the best heuristic, ABT_DO-Retro-MinDom, proposed in [**Zivan *et al.*, 2009**] is a heuristic that does not require this exponential storage of nogoods. In ABT_DO-Retro-MinDom, the agent that generates a nogood is placed in the new order between the last and the second last agents in the nogood if its domain size is smaller than that of the agents it passes on the way up.

In this chapter, we propose Agile Asynchronous Backtracking (Agile-ABT), an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing *before* the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Furthermore, that agent receiving the backtrack message, called the backtracking *target*, is not necessarily the agent with the lowest priority within the conflicting agents in the current order. The principle of Agile-ABT is built on termina-

tion values exchanged by agents during search. A termination value is a tuple of positive integers attached to an order. Each positive integer in the tuple represents the expected current domain size of the agent in that position in the order. Orders are changed by agents without any global control so that the termination value decreases lexicographically as the search progresses. Since, a domain size can never be negative, termination values cannot decrease indefinitely. An agent informs the others of a new order by sending them its new order and its new termination value. When an agent compares two contradictory orders, it keeps the order associated with the smallest termination value.

5.2 Introductory Material

In Agile-ABT, all agents start with the same order $\mathcal{O}$. Then, agents are allowed to change the order asynchronously. Each agent $A_i \in \mathcal{A}$ stores a unique order denoted by $\mathcal{O}_i$. $\mathcal{O}_i$ is called the current order of A_i. Agents appearing before A_i in $\mathcal{O}_i$ are the higher priority agents (predecessors) denoted by $\mathcal{O}_i^-$ and conversely the lower priority agents (successors) $\mathcal{O}_i^+$ are agents appearing after A_i.

Agents can infer inconsistent sets of assignments, called nogoods. A nogood can be represented as an implication. There are clearly many different ways of representing a given nogood as an implication. For example, $\neg[(x_i{=}v_i) \wedge (x_j{=}v_j) \wedge \cdots \wedge (x_k{=}v_k)]$ is logically equivalent to $[(x_j{=}v_j) \wedge \cdots \wedge (x_k{=}v_k)] \rightarrow (x_i \neq v_i)$. When a nogood is represented as an implication, the *left hand side* (`lhs`) and the *right hand side* (`rhs`) are defined from the position of $\rightarrow$. A nogood ng is ***relevant*** with respect to an order $\mathcal{O}_i$ if all agents in `lhs`(ng) appear before `rhs`(ng) in $\mathcal{O}_i$.

The current domain of x_i is the set of values $v_i \in D^0(x_i)$ such that $x_i \neq v_i$ does not appear in any of the right hand sides of the nogoods stored by A_i. Each agent keeps only one nogood per removed value. The size of the current domain of A_i is denoted by d_i (i.e., $|D(x_i)| = d_i$). The initial domain size of a variable x_i, before any value has been pruned, is denoted by d_i^0 (i.e., $d_i^0 = |D^0(x_i)|$ and $d_i = |D(x_i)|$).

Before presenting Agile-ABT, we need to introduce new notions and to present some key subfunctions.

5.2.1 Reordering details

In order to allow agents to asynchronously propose new orders, they must be able to coherently decide which order to select. We propose that the priority between the different orders is based on *termination values*. Informally, if $\mathcal{O}_i = [A_1, \ldots, A_n]$ is the current order known by an agent A_i, then the tuple of domain sizes $[d_1, \ldots, d_n]$ is the termination value of $\mathcal{O}_i$ on A_i. To build termination values, agents need to know the current domain sizes of other agents. To this end, agents exchange *explanations*.

Definition 5.1 *An **explanation** e_j is an expression* `lhs`$(e_j) \rightarrow d_j$*, where* `lhs`(e_j) *is the conjunction of the left hand sides of all nogoods stored by A_j as justifications of value removals for x_j, and*

d_j is the number of values not pruned by nogoods in the domain of A_j. d_j is the right hand side of e_j, $\texttt{rhs}(e_j)$.

Each time an agent communicates its assignment to other agents (by sending them an ***ok?*** message, see Section 5.3), it inserts its explanation in the ***ok?*** message for allowing other agents to build their termination value.

The variables in the left hand side of an explanation e_j must precede the variable x_j in the order because the assignments of these variables have been used to determine the current domain of x_j. An explanation e_j induces ordering constraints, called *safety conditions* in [**Ginsberg and McAllester, 1994**] (see Section 1.2.1.4).

Definition 5.2 *A **safety condition** is an assertion $x_k \prec x_j$. Given an explanation e_j, $S(e_j)$ is the set of safety conditions induced by e_j, where $S(e_j)=\{(x_k \prec x_j) \mid x_k \in \texttt{lhs}(e_j)\}$.*

An explanation e_j is ***relevant*** to an order $\mathcal{O}$ if all variables in $\texttt{lhs}(e_j)$ appear before x_j in $\mathcal{O}$. Each agent A_i stores a set $\mathcal{E}_i$ of explanations sent by other agents. During search, $\mathcal{E}_i$ is updated to remove explanations that are no longer *valid*.

Definition 5.3 *An explanation e_j in $\mathcal{E}_i$ is **valid** on agent A_i if it is* relevant *to the current order $\mathcal{O}_i$ and $\texttt{lhs}(e_j)$ is* compatible *with the AgentView of A_i.*

When $\mathcal{E}_i$ contains an explanation e_j associated with A_j, A_i uses this explanation to justify the size of the current domain of A_j. Otherwise, A_i assumes that the size of the current domain of A_j is equal to its initial domain size d_j^0. The termination value depends on the order and the set of explanations.

Definition 5.4 *Let $\mathcal{E}_i$ be the set of explanations stored by A_i, $\mathcal{O}$ be an order on the agents such that every explanation in $\mathcal{E}_i$ is relevant to $\mathcal{O}$, and $\mathcal{O}(k)$ be such that $A_{\mathcal{O}(k)}$ is the kth agent in $\mathcal{O}$. The **termination value** $\texttt{TV}(\mathcal{E}_i, \mathcal{O})$ is the tuple $[tv^1, \ldots, tv^n]$, where $tv^k = \texttt{rhs}(e_{\mathcal{O}(k)})$ if $e_{\mathcal{O}(k)} \in \mathcal{E}_i$, otherwise, $tv^k = d^0_{\mathcal{O}(k)}$.*

In Agile-ABT, an order $\mathcal{O}_i$ is always associated with a termination value TV_i. When comparing two orders the ***strongest*** order is that associated with the lexicographically *smallest* termination value. In case of ties, we use the lexicographic order on agents IDs, the smaller being the stronger.

Example 5.1 Consider for instance the two orders $\mathcal{O}_1=[A_1, A_2, A_5, A_4, A_3]$ and $\mathcal{O}_2=[A_1, A_2, A_4, A_5, A_3]$. If the termination value associated with $\mathcal{O}_1$ is equal to the termination value associated with $\mathcal{O}_2$, $\mathcal{O}_2$ is stronger than $\mathcal{O}_1$ because the vector $[1, 2, 4, 5, 3]$ of IDs in $\mathcal{O}_2$ is lexicographically smaller than the vector $[1, 2, 5, 4, 3]$ of IDs in $\mathcal{O}_1$.

In the following we will show that the interest of the termination values is not limited to the role of establishing a priority between the different orders proposed by agents. We use them to provide more flexibility in the choice of the backtracking target and to speed up the search.

5.2.2 The Backtracking Target

When all the values of an agent A_i are ruled out by nogoods, these nogoods are resolved, producing a new nogood *newNogood*. *newNogood* is the conjunction of the left hand side (`lhs`) of all nogoods stored by A_i. If *newNogood* is empty, then the inconsistency is proved. Otherwise, one of the conflicting agents must change its value. In standard ABT, the backtracking target (i.e., the agent that must change its value) is the agent with the lowest priority. Agile-ABT overcomes this restriction by allowing A_i to select with great freedom the backtracking target. When a new nogood *newNogood* is produced by resolution, the only condition to choose a variable x_k as the backtracking target (i.e., the variable to put in the right hand side of *newNogood*) is to find an order $\mathcal{O}'$ such that $\texttt{TV}(up\_E_i, \mathcal{O}')$ is lexicographically smaller than the termination value associated with the current order of A_i (i.e., $\mathcal{O}_i$). up_E_i is obtained by updating $\mathcal{E}_i$ after placing x_k in $\texttt{rhs}(newNogood)$.

Function `UpdateExplanations` takes as arguments the set of explanations stored by A_i (i.e., $\mathcal{E}_i$), the generated nogood *newNogood* and the variable x_k to place in the right hand side (`rhs`) of *newNogood*. `UpdateExplanations` removes all explanations that are no longer compatible with the AgentView of A_i after placing x_k in the right hand side of *newNogood*. (The assignment of x_k will be removed from AgentView after backtracking). Next, it updates the explanation of agent A_k stored in A_i and it returns a set of (updated) explanations up_E_i.

Algorithm 5.1: Function Update Explanations.

function `UpdateExplanations` $(\mathcal{E}_i, newNogood, x_k)$
01. $up\_E_i \leftarrow \mathcal{E}_i$;
02. `SetRhs` $(newNogood, x_k)$;
03. remove each $e_j \in up\_E_i$ such that $x_k \in \texttt{lhs}(e_j)$;
04. **if (** $e_k \notin up\_E_i$ **) then**
05. $\quad e_k \leftarrow \{\emptyset \rightarrow d_k^0\}$;
06. $\quad$ `add` (e_k, up_E_i) ;
07. $e'_k \leftarrow \{[\texttt{lhs}(e_k) \cup \texttt{lhs}(newNogood)] \rightarrow \texttt{rhs}(e_k) - 1\}$;
08. replace e_k by e'_k ;
09. **return** up_E_i;

This function does not create cycles in the set of safety conditions $S(up\_E_i)$ if $S(\mathcal{E}_i)$ is acyclic. Indeed, all the explanations added or removed from $S(\mathcal{E}_i)$ to obtain $S(up\_E_i)$ contain x_k. Hence, if $S(up\_E_i)$ contains cycles, all these cycles should contain x_k. However, there does not exist any safety condition of the form $x_k \prec x_j$ in $S(up\_E_i)$ because all of these explanations have been removed in line 3. Thus, $S(up\_E_i)$ cannot be cyclic. As we will show in Section 5.3, the updates performed by A_i ensure that $S(\mathcal{E}_i)$ always remains acyclic. As a result, $S(up\_E_i)$ is acyclic as well, and it can be represented by a directed acyclic graph $\overrightarrow{G} = (X_{\overrightarrow{G}}, E_{\overrightarrow{G}})$ where $X_{\overrightarrow{G}} = \mathcal{X} = \{x_1, \ldots, x_n\}$. An edge $(x_j, x_l) \in E_{\overrightarrow{G}}$ if the safety condition $(x_j \prec x_l) \in S(up\_E_i)$, i.e., $e_l \in up\_E_i$ and $x_j \in \texttt{lhs}(e_l)$. Any topological sort of $\overrightarrow{G}$ is an order relevant to the safety conditions induced by up_E_i.

To recap, when all values of an agent A_i are ruled out by some nogoods, they are resolved, producing a new nogood (*newNogood*). In Agile-ABT, A_i can select with great

freedom the variable x_k whose value is to be changed. The only restriction to place a variable x_k in the `rhs`$(newNogood)$ is to find an order $\mathcal{O}'$ such that `TV`$(up\_E_i, \mathcal{O}')$ is lexicographically smaller than the termination value associated with the current order of A_i. Note that up_E_i being acyclic, there are always one or more topological orders that agree with $S(up\_E_i)$. In the following, we will discuss in more details how to choose the order $\mathcal{O}'$.

5.2.3 Decreasing termination values

Termination of Agile-ABT is based on the fact that the termination values associated with orders selected by agents decrease as search progresses. To speed up the search, Agile-ABT is written so that agents decrease termination values whenever they can. When an agent resolves its nogoods, it checks whether it can find a new order of agents such that the associated termination value is smaller than that of the current order. If so, the agent will replace its current order and termination value by those just computed, and will inform all other agents.

Algorithm 5.2: Function Compute Order.

```
function ComputeOrder(up_E_i)
10. G⃗ = (X_G⃗, E_G⃗) is the acyclic directed graph associated to up_E_i ;
11. p ← 1 ;
12. O is an array of length n ;
13. while ( G⃗ ≠ ∅ ) do
14.     Roots ← {x_j ∈ X_G⃗ | x_j has no incoming edges} ;
15.     O(p) ← x_j such that d_j = min{d_k | x_k ∈ Roots} ;
16.     remove x_j from G⃗ ;                    /* with all outgoing edges from x_j */
17.     p ← p + 1 ;
18. return O ;
```

Assume that after resolving its nogoods, an agent A_i, decides to place x_k in the right hand side of the nogood ($newNogood$) produced by the resolution and let $\mathcal{E}_i =$ `UpdateExplanations`$(\mathcal{E}_i, newNogood, x_k)$. The function `ComputeOrder` takes as parameter the set up_E_i and returns an order $\mathcal{O}$ relevant to the partial ordering induced by up_E_i. Let $\overrightarrow{G}$ be the acyclic directed graph associated with up_E_i. The function `ComputeOrder` works by determining, at each iteration p, the set $Roots$ of vertexes that have no predecessor (line 14). As we aim at minimizing the termination value, function `ComputeOrder` selects the vertex x_j in $Roots$ that has the smallest domain size (line 15). This vertex is placed at the pth position. Finally, p is incremented after removing x_j and all outgoing edges from x_j from $\overrightarrow{G}$ (lines 16-17).

Having proposed an algorithm that determines an order with small termination value for a given backtracking target x_k, one needs to know how to choose this variable to obtain an order decreasing more the termination value. The function `ChooseVariableOrder` iterates through all variables x_k included in the nogood, computes a new order and termination value with x_k as the target (lines 21-23), and stores the target and the associated order if it

Algorithm 5.3: Function Choose Variable Ordering.

```
function ChooseVariableOrder(E_i, newNogood)
19. O' ← O_i;  TV' ← TV_i;  E' ← nil;  x' ← nil ;
20. foreach ( x_k ∈ newNogood ) do
21.     up_E_i ← UpdateExplanations(E_i, newNogood, x_k) ;
22.     up_O ← ComputeOrder(up_E_i) ;
23.     up_TV ← TV(up_E_i, up_O ) ;
24.     if ( up_TV is smaller than TV' ) then
25.         x' ← x_k;
26.         O' ← up_O;
27.         TV' ← up_TV;
28.         E' ← up_E_i ;
29. return ⟨x', O', TV', E'⟩;
```

is the strongest order found so far (lines 24-28). Finally, the information corresponding to the strongest order is returned.

5.3 The Algorithm

Each agent, say A_i, keeps some amount of local information about the global search, namely an AgentView, a NogoodStore, a set of explanations ($\mathcal{E}_i$), a current order ($\mathcal{O}_i$) and a termination value (TV_i). Agile-ABT allows the following types of messages (where A_i is the sender):

ok?: The ***ok?*** message is sent by A_i to lower agents to ask whether a chosen value is acceptable. Besides the chosen value, the ***ok?*** message contains an explanation e_i which communicates the current domain size of A_i. An ***ok?*** message also contains the current order $\mathcal{O}_i$ and the current termination value TV_i stored by A_i.

ngd: The ***ngd*** message is sent by A_i when all its values are ruled out by its NogoodStore. This message contains a nogood, as well as $\mathcal{O}_i$ and TV_i.

order: The ***order*** message is sent to propose a new order. This message includes the order $\mathcal{O}_i$ proposed by A_i accompanied by the termination value TV_i.

Agile-ABT (Algorithm 5.4-5.5) is executed on every agent A_i. After initialization, each agent assigns a value and informs lower priority agents of its decision (`CheckAgentView` call, line 31) by sending ***ok?*** messages. Then, a loop considers the reception of the possible message types. If no message is traveling through the network, the state of quiescence is detected by a specialized algorithm [**Chandy and Lamport, 1985**], and a global solution is announced. The solution is given by the current variables' assignments.

When an agent A_i receives a message (of any type), it checks if the order included in the received message is stronger than its current order $\mathcal{O}_i$ (`CheckOrder` call, lines 37, 41 and 43). If it is the case, A_i replaces $\mathcal{O}_i$ and TV_i by those newly received (line 52). The nogoods and explanations that are no longer relevant to $\mathcal{O}_i$ are removed to ensure that $S(\mathcal{E}_i)$ remains acyclic (line 55).

If the message was an ***ok?*** message, the AgentView of A_i is updated to include the new assignments (`UpdateAgentView` call, line 38). Beside the assignment of the sender, A_i

Algorithm 5.4: The Agile-ABT algorithm executed by an agent A_i (Part 1).

procedure `Agile-ABT()`
30. $t_i \leftarrow 0$; $TV_i \leftarrow [\infty, \infty, \ldots, \infty]$; $end \leftarrow$ **false**; $v_i \leftarrow empty$;
31. `CheckAgentView()` ;
32. **while (** $\neg end$ **) do**
33. $msg \leftarrow$ `getMsg()`;
34. **switch (** $msg.type$ **) do**
35. ***ok?*** : `ProcessInfo(`msg`)`; ***ngd*** : `ResolveConflict(`msg`)`;
36. ***order*** : `ProcessOrder(`msg`)`; ***stp*** : $end \leftarrow$ **true**;

procedure `ProcessInfo(`msg`)`
37. `CheckOrder(`$msg.Order, msg.TV$`)` ;
38. `UpdateAgentView(`$msg.Assig \cup$ `lhs(`$msg.Exp$`))` ;
39. **if (** $msg.Exp$ *is* ***valid*** **) then** `add(`$msg.Exp, \mathcal{E}_i$`)`;
40. `CheckAgentView()` ;

procedure `ProcessOrder(`msg`)`
41. `CheckOrder(`$msg.Order, msg.TV$`)` ;
42. `CheckAgentView()` ;

procedure `ResolveConflict(`msg`)`
43. `CheckOrder(`$msg.Order, msg.TV$`)` ;
44. `UpdateAgentView(`$msg.Assig \cup$ `lhs(`$msg.Nogood$`))` ;
45. **if (** `Compatible(`$msg.Nogood, AgentView \cup myAssig$`)` **) then**
46. **if (** `Relevant(`$msg.Nogood, \mathcal{O}_i$`)` **) then**
47. `add(`$msg.Nogood, NogoodStore$`)` ;
48. $v_i \leftarrow empty$;
49. `CheckAgentView()` ;
50. **else if (** `rhs(`$msg.Nogood$`)` $= v_i$ **) then**
51. `sendMsg:` ***ok?***$(myAssig, e_i, \mathcal{O}_i, TV_i)$ **to** $msg.Sender$

procedure `CheckOrder(`$\mathcal{O}', TV'$`)`
52. **if (** $\mathcal{O}'$is stronger than $\mathcal{O}_i$ **) then**
53. $\mathcal{O}_i \leftarrow \mathcal{O}'$;
54. $TV_i \leftarrow TV'$;
55. remove nogoods and explanations non relevant to $\mathcal{O}_i$;

procedure `CheckAgentView()`
56. **if (** `(¬isConsistent(`$v_i, AgentView$`)` **) then**
57. $v_i \leftarrow$ `ChooseValue()` ;
58. **if (** v_i **) then**
59. **foreach (** $x_k \succ x_i$ **) do**
60. `sendMsg:` ***ok?***$(myAssig, e_i, \mathcal{O}_i, TV_i)$ **to** A_k ;
61. **else** `Backtrack()` ;
62. **else if (** $\mathcal{O}_i$ *was modified* **) then**
63. **foreach (** $x_k \succ x_i$ **) do**
64. `sendMsg:` ***ok?***$(myAssig, e_i, \mathcal{O}_i, TV_i)$ **to** A_k ;

procedure `UpdateAgentView(`$Assignments$`)`
65. **foreach (** $x_j \in Assignments$ **) do**
66. **if (** $Assignments[j].tag > AgentView[j].tag$ **) then**
67. $AgentView[j] \leftarrow Assignments[j]$;
68. **foreach (** $ng \in NogoodStore$ ***such that*** `¬Compatible(lhs(`ng`), `$AgentView$`)` **) do**
69. `remove(`$ng, myNogoodStore$`)` ;
70. **foreach (** $e_j \in \mathcal{E}_i$ ***such that*** `¬Compatible(lhs(`e_j`), `$AgentView$`)` **) do**
71. `remove(`$e_j, \mathcal{E}_i$`)` ;

also takes newer assignments contained in the left hand side of the explanation included in the received ***ok?*** message to update its AgentView. Afterwards, the nogoods and the explanations that are no longer compatible with AgentView are removed (`UpdateAgentView`, lines 68-71). Then, if the explanation in the received message is valid, A_i updates the set of explanations by storing the newly received explanation. Next, A_i calls the procedure `CheckAgentView` (line 40).

When receiving an ***order*** message, A_i processes the new order (`CheckOrder`) and calls `CheckAgentView` (line 42).

When A_i receives a ***ngd*** message, it calls `CheckOrder` and `UpdateAgentView` (lines 43 and 44). The nogood contained in the message is accepted if it is compatible with the AgentView and the assignment of x_i and relevant to the current order of A_i. Otherwise, the nogood is discarded and an ***ok?*** message is sent to the sender as in ABT (lines 50 and 51). When the nogood is accepted, it is stored, acting as justification for removing the current value of A_i (line 47). A new value consistent with the AgentView is searched (`CheckAgentView` call, line 49).

The procedure `CheckAgentView` checks if the current value v_i is consistent with the AgentView. If v_i is consistent, A_i checks if $\mathcal{O}_i$ was modified (line 62). If so, A_i must send its assignment to lower priority agents through ***ok?*** messages. If v_i is not consistent with its AgentView, A_i tries to find a consistent value (`ChooseValue` call, line 57). In this process, some values of A_i may appear as inconsistent. In this case, the nogoods justifying their removal are added to the NogoodStore (line 92 of function `ChooseValue()`). If a new

Algorithm 5.5: The Agile-ABT algorithm executed by an agent A_i (Part 2).

```
procedure Backtrack()
72.  newNogood ← solve(NogoodStore) ;
73.  if ( newNogood = empty ) then
74.     end ← true;
75.     sendMsg: stp( ) to system agent ;
76.  ⟨x_k, O', TV', E'⟩ ← ChooseVariableOrder(E_i, newNogood) ;
77.  if ( TV' is smaller than TV_i ) then
78.     TV_i ← TV' ;
79.     O_i ← O' ;
80.     E_i ← E' ;
81.     SetRhs(newNogood, x_k) ;
82.     sendMsg: ngd(newNogood, O_i, TV_i) to A_k ;
83.     remove e_k from E_i ;
84.     broadcastMsg: order(O_i, TV_i) ;
85.  else
86.     SetRhs(newNogood, x_k) ;          /* x_k is the lower agent in newNogood */
87.     sendMsg: ngd(newNogood, O_i, TV_i) to A_k ;
88.  UpdateAgentView (x_k ← unknown) ;
89.  CheckAgentView() ;
function ChooseValue()
90.  foreach ( v ∈ D(x_i) ) do
91.     if ( isConsistent(v, AgentView ) ) then return v;
92.     else store the best nogood for v ;
93.  return empty;
```

consistent value is found, an explanation e_i is built and the new assignment is notified to the lower priority agents of A_i through ***ok?*** messages (line 60). Otherwise, every value of A_i is forbidden by the NogoodStore and A_i has to backtrack (`Backtrack` call, line 61).

In procedure `Backtrack()`, A_i resolves its nogoods, deriving a new nogood (*newNogood*). If *newNogood* is empty, the problem has no solution. A_i terminates execution after sending a ***stp*** message (lines 74-75). Otherwise, one of the agents included in *newNogood* must change its value. The function `ChooseVariableOrder` selects the variable to be changed (x_k) and a new order ($\mathcal{O}'$) such that the new termination value TV' is as small as possible. If TV' is smaller than that stored by A_i, the current order and the current termination value are replaced by $\mathcal{O}'$ and TV' and A_i updates its explanations by that returned by `ChooseVariableOrder` (lines 78-80). Then, a ***ngd*** message is sent to the agent A_k owner of x_k (line 82). e_k is removed from $\mathcal{E}_i$ since A_k will probably change its explanation after receiving the nogood (line 83). Afterwards, A_i sends an ***order*** message to all other agents (line 84). When TV' is not smaller than the current termination value, A_i cannot propose a new order and the variable to be changed (x_k) is the variable that has the lowest priority according to the current order of A_i (lines 86 and 87). Next, the assignment of x_k (the target of the backtrack) is removed from the AgentView of A_i (line 88). Finally, the search is continued by calling the procedure `CheckAgentView` (line 89).

5.4 Correctness and complexity

In this section we demonstrate that Agile-ABT is sound, complete and terminates, and that its space complexity is polynomially bounded.

Theorem 5.1. *The spatial complexity of Agile-ABT is polynomial.*

Proof. The size of nogoods, explanations, termination values, and orderings, is bounded by n, the total number of variables. Now, on each agent, Agile-ABT only stores one nogood per value, one explanation per agent, one termination value and one ordering. Thus, the space complexity of Agile-ABT is in $O(nd + n^2 + n + n) = O(nd + n^2)$ on each agent. □

Theorem 5.2. *The algorithm Agile-ABT is sound.*

Proof. Let us assume that the state of quiescence is reached. The order (say $\mathcal{O}$) known by all agents is the same because when an agent proposes a new order, it sends it to all other agents. Obviously, $\mathcal{O}$ is the strongest order that has ever been calculated by agents. Also, the state of quiescence implies that every pair of constrained agents satisfies the constraint between them. To prove this, assume that there exist some constraints that are not satisfied. This implies that there are at least two agents A_i and A_k that do not satisfy the constraint between them (i.e., c_{ik}). Let A_i be the agent which has the highest priority between the two agents according to $\mathcal{O}$. Let v_i be the current value of A_i when the state of quiescence is reached (i.e., v_i is the most up to date assignment of A_i) and let M be the last ***ok?*** message sent by A_i before the state of quiescence is reached. Clearly, M contains v_i, otherwise, A_i would have sent another ***ok?*** message when it chose v_i. Moreover, when M was sent, A_i already knew the order $\mathcal{O}$, otherwise A_i would have sent another ***ok?*** message when

it received (or generated) $\mathcal{O}$. A_i sent M to all its successors according to $\mathcal{O}$ (including A_k). The only case where A_k can forget v_i after receiving it is the case where A_k derives a nogood proving that v_i is not feasible. In this case, A_k should send a nogood message to A_i. If the nogood message is accepted by A_i, A_i must send an ***ok?*** message to its successors (and therefore M is not the last one). Similarly, if the nogood message is discarded, A_i have to re-send an ***ok?*** message to A_k (and therefore M is not the last one). So the state of quiescence implies that A_k knows both $\mathcal{O}$ and v_i. Thus, the state of quiescence implies that the current value of A_k is consistent with v_i, otherwise A_k would send at least a message and our quiescence assumption would be broken. □

Theorem 5.3. *The algorithm Agile-ABT is complete.*

Proof. All nogoods are generated by logical inferences from existing constraints. Therefore, an empty nogood cannot be inferred if a solution exists. □

The proof of termination is built on Lemma 5.1 and 5.2.

Lemma 5.1. For any agent A_i, while a solution is not found and the inconsistency of the problem is not proved, the termination value stored by A_i decreases after a finite amount of time.

Proof. Let $TV_i = [tv^1, \ldots, tv^n]$ be the current termination value of A_i. Assume that A_i reaches a state where it cannot improve its termination value. If another agent succeeds in generating a termination value smaller than TV_i, Lemma 5.1 holds since A_i will receive the new termination value. Now assume that Agile-ABT reaches a state σ where no agent can generate a termination value smaller than TV_i. We show that Agile-ABT will exit σ after a finite amount of time. Let t be the time when Agile-ABT reaches the state σ. After a finite time δt, the termination value of each agent $A_{j\in\{1,\ldots,n\}}$ will be equal to TV_i, either because A_j has generated itself a termination value equal to TV_i or because A_j has received TV_i in an order message. Let $\mathcal{O}$ be the lexicographically smallest order among the current orders of all agents at time $t + \delta t$. The termination value associated with $\mathcal{O}$ is equal to TV_i. While Agile-ABT is getting stuck in σ, no agent will be able to propose an order stronger than $\mathcal{O}$ because no agent is allowed to generate a new order with the same termination value as the one stored (line 77, Algorithm 5.5). Thus, after a finite time $\delta' t$, all agents will receive $\mathcal{O}$. They will take it as their current order and Agile-ABT will behave as ABT, which is known to be complete and to terminate.

We know that $d^0_{\mathcal{O}(1)} - tv^1$ values have been removed once and for all from the domain of the variable $x_{\mathcal{O}(1)}$ (i.e., $d^0_{\mathcal{O}(1)} - tv^1$ nogoods with empty `lhs` have been sent to $A_{\mathcal{O}(1)}$). Otherwise, the generator of $\mathcal{O}$ could not have put $A_{\mathcal{O}(1)}$ in the first position. Thus, the domain size of $x_{\mathcal{O}(1)}$ cannot be greater than tv^1 ($d_{\mathcal{O}(1)} \leq tv^1$). After a finite amount of time, if a solution is not found and the inconsistency of the problem is not proved, a nogood–with an empty left hand side – will be sent to $A_{\mathcal{O}(1)}$ which will cause it to replace its assignment and to reduce its current domain size ($d'_{\mathcal{O}(1)} = d_{\mathcal{O}(1)} - 1$). The new assignment and the new current domain size of $A_{\mathcal{O}(1)}$ will be sent to the $(n-1)$ lower priority agents. After receiving this message, we are sure that any generator of a new

nogood (say A_k) will improve the termination value. Indeed, when A_k resolves its nogoods, it computes a new order such that its termination value is minimal. At worst, A_k can propose a new order where $A_{\mathcal{O}(1)}$ keeps its position. Even in this case the new termination value $TV'_k = [d'_{\mathcal{O}(1)}, \dots]$ is lexicographically smaller than $TV_i = [tv^1, \dots]$ because $d'_{\mathcal{O}(1)} = d_{\mathcal{O}(1)} - 1 \leq tv^1 - 1$. After a finite amount of time, all agents (A_i included) will receive TV'_k. This will cause A_i to update its termination value and to exit the state σ. This completes the proof. □

Lemma 5.2. Let $TV = [tv^1, \dots, tv^n]$ be the termination value associated with the current order of any agent. We have $tv^j \geq 0, \forall j \in 1..n$

Proof. Let A_i be the agent that generated TV. We first prove that A_i never stores an explanation with a `rhs` smaller than 1. An explanation e_k stored by A_i was either sent by A_k or generated when calling `ChooseVariableOrder`. If e_k was sent by A_k, we have $\texttt{rhs}(e_k) \geq 1$ because the size of the current domain of any agent is always greater than or equal to 1. If e_k was computed by `ChooseVariableOrder`, the only case where $\texttt{rhs}(e_k)$ is made smaller than the right hand side of the previous explanation stored for A_k by A_i is in (line 7 of `UpdateExplanations`. This happens when x_k is selected to be the backtracking target (lines 21 and 28 of `ChooseVariableOrder`) and in such a case, the explanation e_k is removed just after sending the nogood message to A_k (line 83, Algorithm 5.5, of `Backtrack()`). Hence, A_i never stores an explanation with a `rhs` equal to zero.

We now prove that it is impossible that A_i generated TV with $tv^j < 0$ for some j. From the point of view of A_i, tv^j is the size of the current domain of $A_{\mathcal{O}(j)}$. If A_i does not store any explanation for $A_{\mathcal{O}(j)}$ at the time it computes TV, A_i assumes that tv^j is equal to $d^0_{\mathcal{O}(j)} \geq 1$. Otherwise, tv^j is equal to $\texttt{rhs}(e_{\mathcal{O}(j)})$, where $e_{\mathcal{O}(j)}$ was either already stored by A_i or generated when calling `ChooseVariableOrder`. Now, we know that every explanation e_k stored by A_i has $\texttt{rhs}(e_k) \geq 1$ and we know that `ChooseVariableOrder` cannot generate an explanation e'_k with $\texttt{rhs}(e'_k) < \texttt{rhs}(e_k) - 1$, where e_k was the explanation stored by A_i (line 7 of `UpdateExplanations`). Therefore, we are guaranteed that TV is such that $tv^j \geq 0, \forall j \in 1..n$. □

Theorem 5.4. *The algorithm Agile-ABT terminates.*

Proof. The termination value of any agent decreases lexicographically and does not stay infinitely unchanged (Lemma 5.1). A termination value $[tv^1, \dots, tv^n]$ cannot decrease infinitely because $\forall i \in \{1, \dots, n\}$, we have $tv^i \geq 0$ (Lemma 5.2). Hence the theorem. □

5.5 Experimental Results

We compared Agile-ABT to ABT, ABT_DO, and ABT_DO-Retro (ABT_DO with retroactive heuristics). All experiments were performed on the DisChoco 2.0 [**Wahbi *et al.*, 2011**] platform [1], in which agents are simulated by Java threads that communicate only through message passing. We evaluate the performance of the algorithms by communication load

1. http://www.lirmm.fr/coconut/dischoco/

and computation effort. Communication load is measured by the total number of messages exchanged among agents during algorithm execution (#*msg*), including termination detection (system messages). Computation effort is measured by an adaptation of the number of non-concurrent constraint checks (generic number of non-concurrent constraint checks #*gncccs* [**Zivan and Meisels, 2006b**]).

For ABT, we implemented the standard version where we use counters for tagging assignments. For ABT_DO [**Zivan and Meisels, 2006a**], we implemented the best version, using the *nogood-triggered* heuristic where the receiver of a nogood moves the sender to be in front of all other lower priority agents (denoted by ABT_DO-ng). For ABT_DO with retroactive heuristics [**Zivan *et al.*, 2009**], we implemented the best version, in which a nogood generator moves itself to be in a higher position between the last and the second last agents in the generated nogood [2]. However, it moves before an agent only if its current domain is smaller than the domain of that agent (denoted by ABT_DO-Retro-MinDom).

5.5.1 Uniform binary random DisCSPs

The algorithms are tested on uniform binary random DisCSPs characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of agents/variables, d the number of values per variable, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of problems: sparse problems $\langle 20, 10, 0.2, p_2 \rangle$ and dense problems $\langle 20, 10, 0.7, p_2 \rangle$. We vary the tightness p_2 from 0.1 to 0.9 by steps of 0.1. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. We report average over the 100 runs.

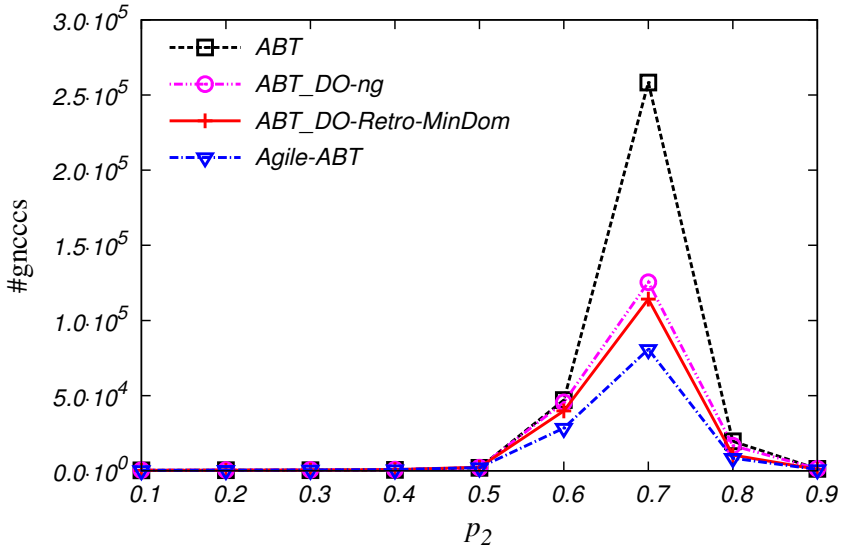


Figure 5.1 – The generic number of non-concurrent constraint checks (#*gncccs*) performed for solving dense problems (p_1=0.2).

2. There are some discrepancies between the results reported in [**Zivan *et al.*, 2009**] and our version. This is due to a bug that we fixed to ensure that ABT_DO-ng and ABT_DO-Retro-MinDom actually terminate [**Mechqrane *et al.*, 2012**], see Chapter 6.

Figures 3.5 and 3.6 present the performance of the algorithms on the sparse instances (p_1=0.2). In term of computational effort, #*gncccs* (Figure 3.5), ABT is the less efficient algorithm. ABT_DO-ng improves ABT by a large scale and ABT_DO-Retro-MinDom is more efficient than ABT_DO-ng. These findings are similar to those reported in [**Zivan *et al.*, 2009**]. Agile-ABT outperforms all these algorithms, suggesting that on sparse problems, the more sophisticated the algorithm is, the better it is.

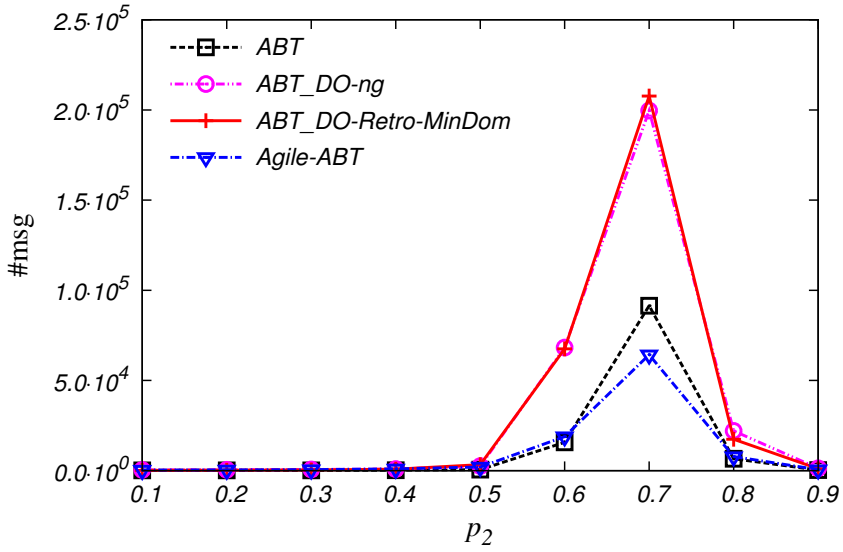


Figure 5.2 – The total number of messages sent for solving dense problems (p_1=0.2).

Regarding the number of exchanged messages, #*msg* (Figure 5.2), the picture is a bit different. ABT_DO-ng and ABT_DO-Retro-MinDom require a number of messages substantially larger than ABT algorithm. Agile-ABT is the algorithm that requires the smallest

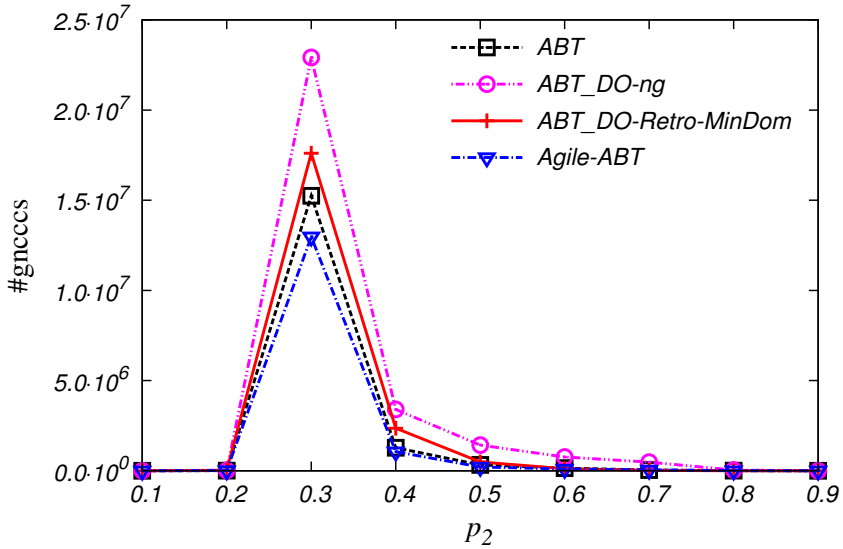


Figure 5.3 – The generic number of non-concurrent constraint checks (#*gncccs*) performed for solving dense problems (p_1=0.7).

number of messages. This is not only because Agile-ABT terminates faster than the other algorithms (see *#gncccs*). Agile-ABT is more parsimonious than ABT_DO algorithms in proposing new orders. Termination values seem to focus changes on those which will pay off.

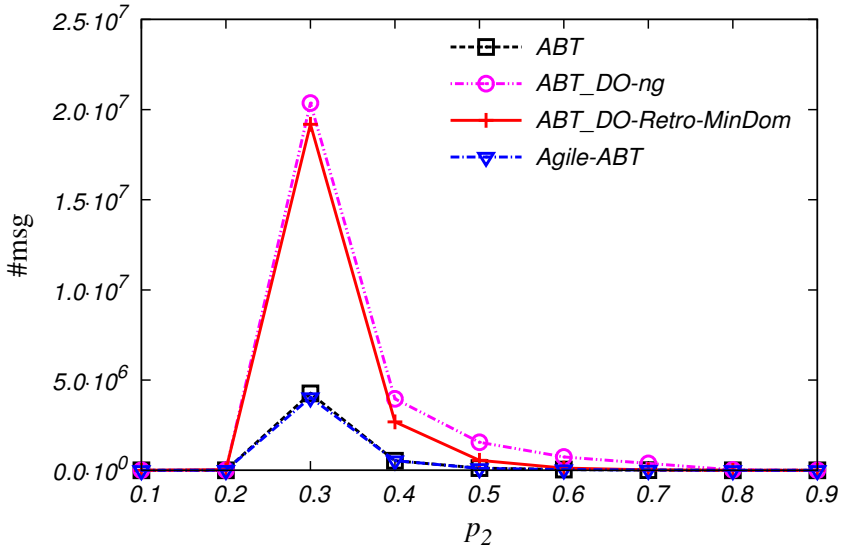


Figure 5.4 – The total number of messages sent for solving dense problems (p_1=0.7).

Figures 5.3 and 5.4 illustrate the performance of the algorithms on the dense instances (p_1=0.7). Some differences appear compared to sparse problems. Concerning *#gncccs* (Figure 5.3), ABT_DO algorithms deteriorate compared to ABT. However, Agile-ABT still outperforms all these algorithms. Regarding communication load, *#msg* (Figure 5.4), ABT_DO-ng and ABT_DO-Retro-MinDom show the same bad performance as in sparse problems. Agile-ABT shows similar communication load as ABT. This confirms its good behavior observed on sparse problems.

5.5.2 Distributed Sensor Target Problems

The *Distributed Sensor-Target Problem* (SensorDisCSP) [**Béjar *et al.*, 2005**] is a benchmark based on a real distributed problem (see Section 1.3.2.2). It consists of n sensors that track m targets. Each target must be tracked by 3 sensors. Each sensor can track at most one target. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a target is visible. The compatibility constraint defines the compatibility among sensors. In our implementation of the DisCSP algorithms, the encoding of the SensorDisCSP presented in Section 1.3.2.2 is translated to an equivalent formulation where we have three virtual agents for every real agent, each virtual agent handling a single variable.

Problems are characterized by $\langle n, m, p_c, p_v \rangle$, where n is the number of sensors, m is the number of targets, each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each target can be tracked by a fraction p_v of the sensors having the target in their sensing range. We present results for the class $\langle$ 25, 5, 0.4, $p_v \rangle$, where we

vary p_v from 0.1 to 0.9 by steps of 0.1. Again, for each p_v we generated 25 instances, solved 4 times each and averaged over the 100 runs. The results are shown in Figures 5.5 and 5.6.

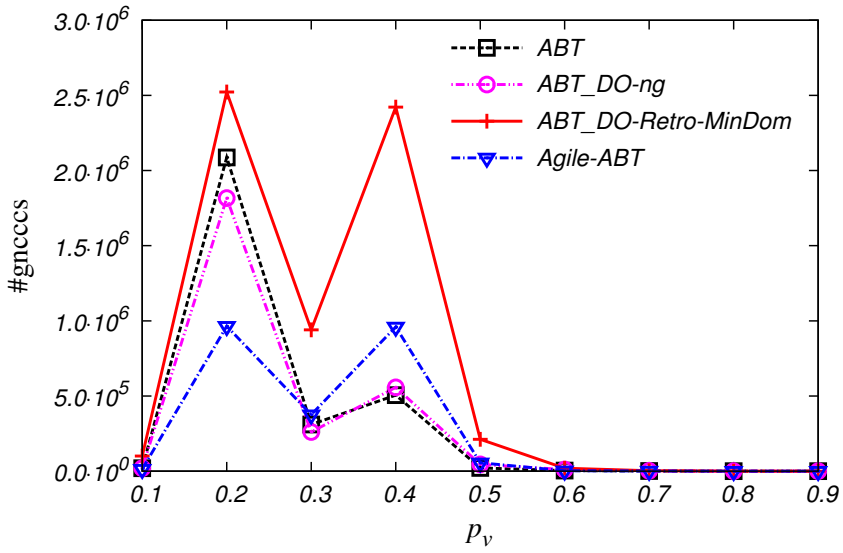


Figure 5.5 – The generic number non-concurrent constraint checks performed on instances where p_c=0.4.

When comparing the speed-up of algorithms (Figure 5.5), Agile-ABT is slightly dominated by ABT and ABT_DO-ng in the interval [0.3 0.5], while outside of this interval, Agile-ABT outperforms all the algorithms. Nonetheless, the performance of ABT and ABT_DO-ng dramatically deteriorate in the interval [0.1 0.3]. Concerning communication load (Figure 5.6), as opposed to other dynamic ordering algorithm, Agile-ABT is always better than or as good as standard ABT.

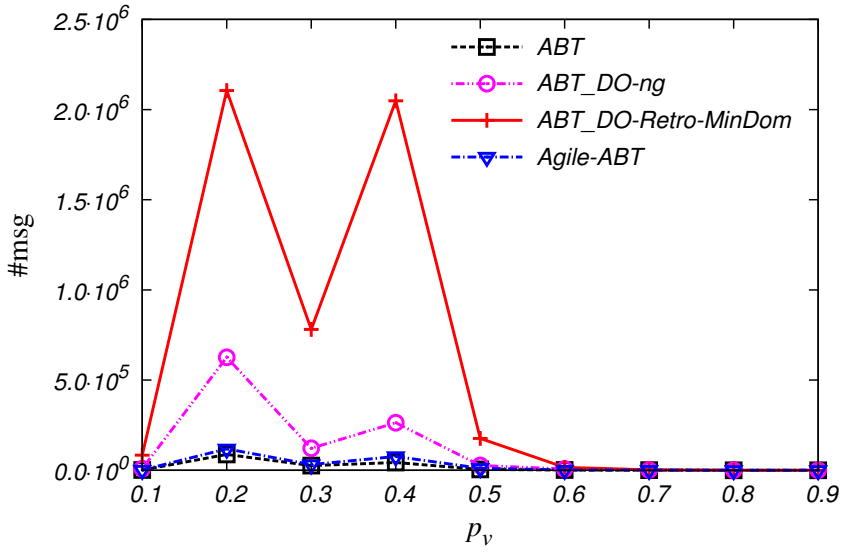
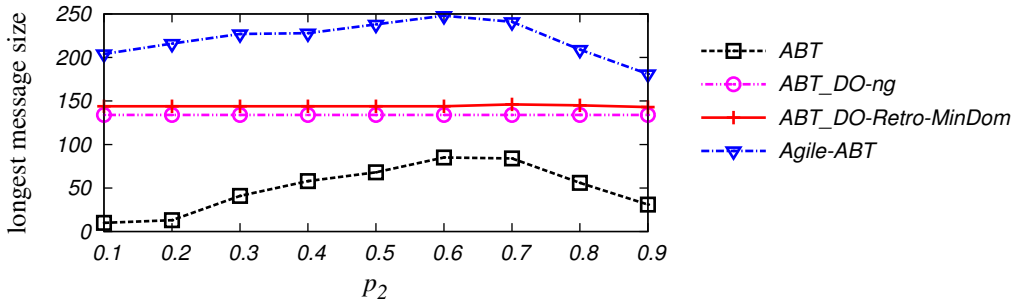


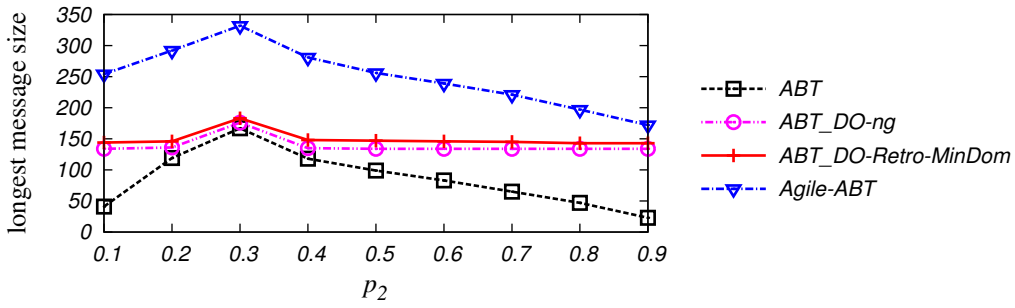
Figure 5.6 – Total number of exchanged messages on instances where p_c=0.4.

5.5.3 Discussion

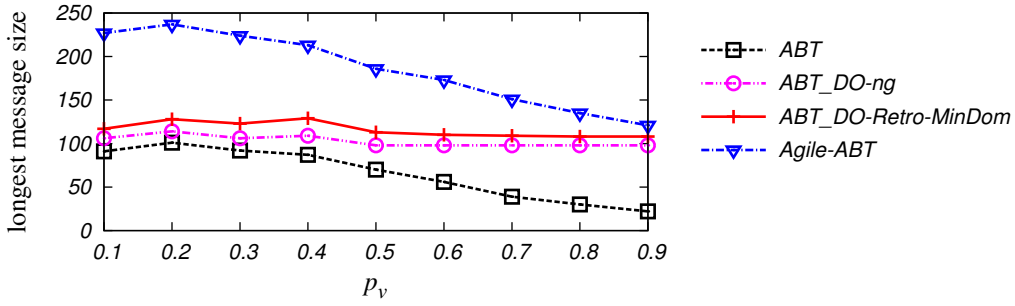
From the experiments above we can conclude that Agile-ABT outperforms other algorithms in term of computation effort (#*gncccs*) when solving random DisCSP problem. On structured problems (SensorDCSP), our results suggest that Agile-ABT is more robust than other algorithms whose performance is sensitive to the type of problems solved. Concerning communication load (#*msg*), Agile-ABT is more robust than other versions of ABT with dynamic agent ordering. As opposed to them, it is always better than or as good as standard ABT on difficult problems.



(a) *sparse random problems* (p_1=0.2)



(b) *dense random problems* (p_1=0.7)



(c) *Sensor Target problems where* $p_c = 0.4$

Figure 5.7 – Maximum message size in bytes.

At first sight, Agile-ABT seems to need less messages than other algorithms but these messages are longer than messages sent by other algorithms. One could object that for Agile-ABT, counting the number of exchanged messages is biased. However, counting the number of exchanged messages would be biased only if #*msg* was smaller than the number of *physically* exchanged messages (going out from the network card). Now, in our experiments, they are the same.

The International Organization for Standardization (ISO) has designed the Open Systems Interconnection (OSI) model to standardize networking. TCP and UDP are the principal Transport Layer protocols using OSI model. The internet protocols IPv4 (http://tools.ietf.org/html/rfc791) and IPv6 (http://tools.ietf.org/html/rfc2460) specify the minimum datagram size that we are guaranteed to send without fragmentation of a message (in one physical message). This is 568 bytes for IPv4 and 1,272 bytes for IPv6 when using either TCP or UDP (UDP is 8 bytes less than TCP, see RFC-768 – http://tools.ietf.org/html/rfc768).

Figure 5.7 shows the size of the longest message sent by each algorithm on our random and sensor problems. It is clear that Agile-ABT requires lengthy messages compared to other algorithms. However, the longest message sent is always less than 568 bytes (in the worst case it is less than 350, see Figure 5.7(b)).

5.6 Related Works

In [**Ginsberg and McAllester, 1994**], **Ginsberg and McAllester** proposed Partial Order Dynamic Backtracking (PODB), a polynomial space algorithm for centralized CSP that attempted to address the rigidity of dynamic backtracking. The *Generalized Partial Order Dynamic Backtracking* (GPODB), an algorithm that generalizes both PODB [**Ginsberg and McAllester, 1994**] and the Dynamic Backtracking DBT [**Ginsberg, 1993**] was proposed in [**Bliek, 1998**]. GPODB maintains a set of ordering constraints (aka. safety conditions) on the variables. These ordering constraints imply only a partial order on the variables. This provides flexibility in the reordering of variables in a nogood. Agile-ABT has some similarities with GPODB because Agile-ABT also maintains a set of safety conditions (induced by explanations). However, the set of safety conditions maintained by Agile-ABT allows more total orderings than the set of safety conditions maintained by GPODB. In addition, whenever a new nogood is generated by GPODB, the target of this nogood must be selected such that the safety conditions induced by the new nogood satisfy all existing safety conditions. On the contrary, Agile-ABT allows discarding explanations, and thus, relaxing some of the safety conditions. These two points give Agile-ABT more flexibility in choosing the backtracking target.

5.7 Summary

We have proposed Agile-ABT, an algorithm that is able to change the ordering of agents more agilely than all previous approaches. Thanks to the original concept of termination value, Agile-ABT is able to choose a backtracking target that is not necessarily the agent with the current lowest priority within the conflicting agents. Furthermore, the ordering of agents appearing before the backtracking target can be changed. These interesting features are unusual for an algorithm with polynomial space complexity. Our experiments confirm the significance of these features.

6 Corrigendum to "Min-domain retroactive ordering for Asynchronous Backtracking"

The asynchronous backtracking algorithm with dynamic ordering, ABT_DO, have been proposed in [Zivan and Meisels, 2006a]. ABT_DO allows changing the order of agents during distributed asynchronous search. In ABT_DO, when an agent assigns a value to its variable, it can reorder lower priority agents. Retroactive heuristics called ABT_DO-Retro which allow more flexibility in the selection of new orders were introduced in [Zivan *et al.*, 2009]. Unfortunately, the description of the time-stamping protocol used to compare orders in ABT_DO-Retro may lead to an implementation in which ABT_DO-Retro may not terminate. In this chapter, we give an example that shows how ABT_DO-Retro can enter in an infinite loop if it uses this protocol and we propose a new correct way for comparing time-stamps [Mechqrane *et al.*, 2012].

This chapter is organized as follows. Section 6.1 introduces the retroactive heuristics of the asynchronous backtracking algorithm with dynamic ordering (ABT_DO-Retro). We describe in Section 6.2 the natural understanding of the protocol used for comparing time-stamps in ABT_DO with retroactive heuristics (ABT_DO-Retro). Section 6.3 illustrates an example that shows, if ABT_DO-Retro uses that protocol, how it can fall into an infinite loop. We describe the correct method for comparing time-stamps and give the proof that our method for comparing orders is correct in Section 6.4.

6.1 Introduction

Zivan and Meisels (2006a) proposed the asynchronous backtracking algorithm with dynamic ordering, ABT_DO, in [Zivan and Meisels, 2006a]. In ABT_DO, when an agent assigns a value to its variable, it can reorder lower priority agents. Each agent in ABT_DO holds a current order (that is, a vector of agent IDs) and a vector of counters (one counter attached to each agent ID). The vector of counters attached to agent IDs forms a time-stamp. Initially, all time-stamp counters are set to zero and all agents start with the same

order. Each agent that proposes a new order increments its counter by one and sets to zero counters of all lower priority agents (the counters of higher priority agents are not modified). When comparing two orders, the strongest is the one with the lexicographically *larger* time-stamp. In other words, the strongest order is the one for which the first different counter is larger. The most successful ordering heuristic found in [Zivan and Meisels, 2006a] was the *nogood-triggered* heuristic in which an agent that receives a nogood moves the nogood generator to be right after it in the order.

A new type of ordering heuristics for ABT_DO is presented in [Zivan *et al.*, 2009]. These heuristics, called retroactive heuristics (ABT_DO-Retro), enable the generator of the nogood to propose a new order in which it moves itself to a higher priority position than that of the target of the backtrack. The degree of flexibility of these heuristics depends on a parameter K. Agents that detect a dead end are moved to a higher priority position in the order. If the length of the created nogood is larger than K, they can be moved up to the place that is right after the second last agent in the nogood. If the length of the created nogood is smaller than or equal to K, the sending agent can be moved to a position before all the participants in the nogood and the nogood is sent and saved by all of the participants in the nogood. Since agents must store nogoods that are smaller than or equal to K, the space complexity of agents is exponential in K.

Recent attempts to implement the ABT_DO-Retro algorithm proposed in [Zivan *et al.*, 2009] have revealed a specific detail of the algorithm that concerns its time-stamping protocol. The natural understanding of the description given in [Zivan *et al.*, 2009] of the time-stamping protocol used to compare orders in ABT_DO-Retro can affect the correctness of the algorithm. In this chapter we address this protocol by describing the undesired outcome of this protocol and propose an alternative deterministic method that ensures the outcome expected in [Zivan *et al.*, 2009].

6.2 Background

The degree of flexibility of the retroactive heuristics mentioned above depends on a parameter K. K defines the level of flexibility of the heuristic with respect to the amount of information an agent can store in its memory. Agents that detect a dead end move themselves to a higher priority position in the order. If the length of the nogood created is not larger than K then the agent can move to any position it desires (even to the highest priority position) and all agents that are included in the nogood are required to add the nogood to their set of constraints and hold it until the algorithm terminates. If the size of the created nogood is larger than K, the agent that created the nogood can move up to the place that is right after the second last agent in the nogood. Since agents must store nogoods that are smaller than or equal to K, the space complexity of agents is exponential in K.

The best retroactive heuristic introduced in [Zivan *et al.*, 2009] is called ABT_DO-Retro-MinDom. This heuristic does not require any additional storage (i.e., $K = 0$). In this heuristic, the agent that generates a nogood is placed in the new order between the last

and the second last agents in the generated nogood. However, the generator of the nogood moves to a higher priority position than the backtracking target (the agent the nogood was sent to) only if its domain is smaller than that of the agents it passes on the way up. Otherwise, the generator of the nogood is placed right after the last agent with a smaller domain between the last and the second last agents in the nogood.

In asynchronous backtracking algorithms with dynamic ordering, agents propose new orders asynchronously. Hence, one must enable agents to coherently decide which of two different orders is the stronger. To this end, as it has been explained in [**Zivan and Meisels, 2006a**] and recalled in [**Zivan *et al.*, 2009**], each agent in ABT_DO holds a *counter vector* (one counter attached to each position in the order). The counter vector and the indexes of the agents currently in these positions form a time-stamp. Initially, all counters are set to zero and all agents are aware of the initial order. Each agent that proposes a new order increments the counter attached to its position in the current order and sets to zero counters of all lower priority positions (the counters of higher priority positions are not modified). The strongest order is determined by a lexicographic comparison of counter vectors combined with the agent indexes. However, the rules for reordering agents in ABT_DO imply that the strongest order is always the one for which the first different counter is larger.

In ABT_DO-Retro agents can be moved to a position that is higher than that of the target of the backtrack. This new feature makes it possible to generate two contradictory orders that have the same time stamp. To address this additional issue, the description given by the authors was limited to two sentences: *"The most relevant order is determined lexicographically. Ties which could not have been generated in standard ABT_DO, are broken using the agents indexes"* [quoted from [**Zivan *et al.*, 2009**], page 190, Theorem 1].

The natural understanding of this description is that the strongest order is the one associated with the lexicographically greater counter vector, and when the counter vectors are equal the lexicographic order on the indexes of agents breaks the tie by preferring the one with smaller vector of indexes. We will refer to this natural interpretation as method m_1. Let us illustrate method m_1 via an example. Consider two orders $\mathcal{O}_1$=$[A_1, A_3, A_2, A_4, A_5]$ and $\mathcal{O}_2$=$[A_1, A_2, A_3, A_4, A_5]$ where the counter vector associated with $\mathcal{O}_1$ equals $\mathcal{V}_1 = [2, 4, 2, 2, 0]$ and the counter vector associated with $\mathcal{O}_2$ equals $\mathcal{V}_2 = [2, 4, 2, 1, 0]$. Since in m_1 the strongest order is determined by comparing lexicographically the counter vectors, in this example $\mathcal{O}_1$ is considered stronger than $\mathcal{O}_2$. In Section 6.3 of this chapter, we show that method m_1 may lead ABT_DO-Retro to fall in an infinite loop when $K = 0$.

The right way to compare orders is to compare their counter vectors, one position at a time from left to right, until they differ on a position (preferring the order with greater counter) or they are equal on that position but the indexes of the agents in that position differ (preferring the smaller index). We will refer to this method as m_2. Consider again the two orders $\mathcal{O}_1$ and $\mathcal{O}_2$ and associated counter vectors defined above. The counter at the first position equals 2 on both counter vectors and the index of the first agent in $\mathcal{O}_1$ (i.e., A_1) is the same as in $\mathcal{O}_2$, the counter at the second position equals 4 on both counter vectors,however the index of the second agent in $\mathcal{O}_2$ (i.e., A_2) is smaller than the index of

the second agent in $\mathcal{O}_1$ (i.e., A_3). Hence, in this case $\mathcal{O}_2$ is considered stronger than $\mathcal{O}_1$. (Note that according to m_1, $\mathcal{O}_1$ is stronger than $\mathcal{O}_2$.) In Section 6.4 of this chapter, we give the proof that method m_2 for comparing orders is correct.

6.3 ABT_DO-Retro May Not Terminate

In this section we show that ABT_DO-Retro may not terminate when using m_1 and when $K = 0$. We illustrate this on ABT_DO-Retro-MinDom as described in [**Zivan *et al.*, 2009**] as it is an example of ABT_DO-Retro where $K = 0$. Consider a DisCSP with 5 agents $\{A_1, A_2, A_3, A_4, A_5\}$ and domains $D(x_1)$=$D(x_5)$=$\{1,2,3,4,5\}$, $D(x_2)$=$D(x_3)$=$D(x_4)$=$\{6,7\}$. We assume that, initially, all agents store the same order $\mathcal{O}_1 = [A_1, A_5, A_4, A_2, A_3]$ with associated counter vector $\mathcal{V}_1 = [0,0,0,0,0]$. The constraints are:
$c_{12} : (x_1, x_2) \notin \{(1,6),(1,7)\}$;
$c_{13} : (x_1, x_3) \notin \{(2,6),(2,7)\}$;
$c_{14} : (x_1, x_4) \notin \{(1,6),(1,7)\}$;
$c_{24} : (x_2, x_4) \notin \{(6,6),(7,7)\}$.
$c_{35} : (x_3, x_5) \notin \{(7,5)\}$.

In the following we give a possible execution of ABT_DO-Retro-MinDom (Figure 6.1).

$\mathcal{O}_1 = [\, A_1 , A_5 , A_4 , A_2 , A_3 \,] \quad \mathcal{V}_1 = [\, 0 , 0 , 0 , 0 , 0 \,]$
$\mathcal{O}_2 = [\, A_4 , A_1 , A_5 , A_2 , A_3 \,] \quad \mathcal{V}_2 = [\, 1 , 0 , 0 , 0 , 0 \,]$
$\mathcal{O}_3 = [\, A_2 , A_1 , A_5 , A_4 , A_3 \,] \quad \mathcal{V}_3 = [\, 1 , 0 , 0 , 0 , 0 \,]$
$\mathcal{O}_4 = [\, A_4 , A_3 , A_1 , A_5 , A_2 \,] \quad \mathcal{V}_4 = [\, 1 , 1 , 0 , 0 , 0 \,]$

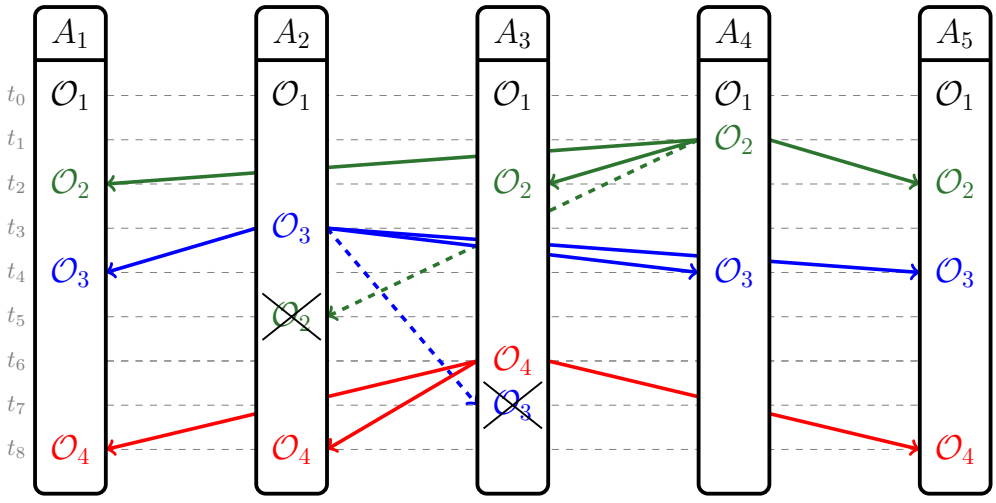


Figure 6.1 – The schema of exchanging ***order*** messages by ABT_DO-Retro

t_0: All agents assign the first value in their domains to their variables and send ***ok?*** messages to their neighbors.

t_1: A_4 receives the first ***ok?*** $(x_1 = 1)$ message sent by A_1 and generates a nogood ng_1 : $\neg(x_1 = 1)$. Then, it proposes a new order $\mathcal{O}_2 = [A_4, A_1, A_5, A_2, A_3]$ with $\mathcal{V}_2 =$

$[1, 0, 0, 0, 0]$. Afterwards, it assigns the value 6 to its variable and sends ***ok?*** $(x_4 = 6)$ message to all its neighbors (including A_2).

t_2: A_3 receives $\mathcal{O}_2 = [A_4, A_1, A_5, A_2, A_3]$ and deletes $\mathcal{O}_1$ since $\mathcal{O}_2$ is stronger; A_1 receives the nogood sent by A_4, it replaces its assignment to 2 and sends an ***ok?*** $(x_1 = 2)$ message to all its neighbors.

t_3: A_2 has not yet received $\mathcal{O}_2$ and the new assignment of A_1. A_2 generates a new nogood $ng_2 : \neg(x_1 = 1)$ and proposes a new order $\mathcal{O}_3 = [A_2, A_1, A_5, A_4, A_3]$ with $\mathcal{V}_3 = [1, 0, 0, 0, 0]$; Afterwards, it assigns the value 6 to its variable and sends ***ok?*** $(x_2 = 6)$ message to all its neighbors (including A_4).

t_4: A_4 receives the new assignment of A_2 (i.e., $x_2 = 6$) and $\mathcal{O}_3 = [A_2, A_1, A_5, A_4, A_3]$. Afterwards, it discards $\mathcal{O}_2$ since $\mathcal{O}_3$ is stronger; Then, A_4 tries to satisfy c_{24} because A_2 has a higher priority according to $\mathcal{O}_3$. Hence, A_4 replaces its current assignment (i.e., $x_4 = 6$) by $x_4 = 7$ and sends an ***ok?*** $(x_4 = 7)$ message to all its neighbors (including A_2).

t_5: When receiving $\mathcal{O}_2$, A_2 discards it because its current order is stronger;

t_6: After receiving the new assignment of A_1 (i.e., $x_1 = 2$) and before receiving $\mathcal{O}_3 = [A_2, A_1, A_5, A_4, A_3]$, A_3 generates a nogood $ng_3 : \neg(x_1 = 2)$ and proposes a new order $\mathcal{O}_4 = [A_4, A_3, A_1, A_5, A_2]$ with $\mathcal{V}_4 = [1, 1, 0, 0, 0]$; The order $\mathcal{O}_4$ is stronger according to m_1 than $\mathcal{O}_3$. Since in ABT_DO, an agent sends the new order only to lower priority agents, A_3 will not send $\mathcal{O}_4$ to A_4 because it is a higher priority agent.

t_7: A_3 receives $\mathcal{O}_3$ and then discards it because it is obsolete;

t_8: A_2 receives $\mathcal{O}_4$ but it has not yet received the new assignment of A_4. Then, it tries to satisfy c_{24} because A_4 has a higher priority according to its current order $\mathcal{O}_4$. Hence, A_2 replaces its current assignment (i.e., $x_2 = 6$) by $x_2 = 7$ and sends an ***ok?*** $(x_2 = 7)$ message to all its neighbors (including A_4).

t_9: A_2 receives the ***ok?*** $(x_4 = 7)$ message sent by A_4 in t_4 and changes its current value (i.e., $x_2 = 7$) by $x_2 = 6$. Then, A_2 sends an ***ok?*** $(x_2 = 6)$ message to all its neighbors (including A_4). At the same time, A_4 receives ***ok?*** $(x_2 = 7)$ sent by A_2 in t_8. A_4 changes its current value (i.e., $x_4 = 7$) by $x_4 = 6$. Then, A_4 sends an ***ok?*** $(x_4 = 6)$ message to all its neighbors (including A_2).

t_{10}: A_2 receives the ***ok?*** $(x_4 = 6)$ message sent by A_4 in t_9 and changes its current value (i.e., $x_2 = 6$) by $x_2 = 7$. Then, A_2 sends an ***ok?*** $(x_2 = 7)$ message to all its neighbors (including A_4). At the same moment, A_4 receives ***ok?*** $(x_2 = 6)$ sent by A_2 in t_9. A_4 changes its current value (i.e., $x_4 = 6$) by $x_4 = 7$. Then, A_4 sends an ***ok?*** $(x_4 = 7)$ message to all its neighbors (including A_2).

t_{11}: We come back to the situation we were facing at time t_9, and therefore ABT_DO-Retro-MinDom may fall in an infinite loop when using method m_1.

6.4 The Right Way to Compare Orders

Let us formally define the second method, m_2, for comparing orders in which we compare the indexes of agents as soon as the counters in a position are equal on both counter vectors associated with the orders being compared. Given any order $\mathcal{O}$, we denote by $\mathcal{O}(i)$ the index of the agent located in the ith position in $\mathcal{O}$ and by $\mathcal{V}(i)$ the counter in the ith position in the counter vector $\mathcal{V}$ associated to order $\mathcal{O}$. An order $\mathcal{O}_1$ with counter vector $\mathcal{V}_1$ is stronger than an order $\mathcal{O}_2$ with counter vector $\mathcal{V}_2$ if and only if there exists a position $i, 1 \leq i \leq n$, such that for all $1 \leq j < i$, $\mathcal{V}_1(j) = \mathcal{V}_2(j)$ and $\mathcal{O}_1(j) = \mathcal{O}_2(j)$, and $\mathcal{V}_1(i) > \mathcal{V}_2(i)$ or $\mathcal{V}_1(i) = \mathcal{V}_2(i)$ and $\mathcal{O}_1(i) < \mathcal{O}_1(i)$.

In our correctness proof for the use of m_2 in ABT_DO-Retro we use the following notations. The initial order known by all agents is denoted by $\mathcal{O}_{init}$. Each agent, A_i, stores a current order, $\mathcal{O}_i$, with an associated counter vector, $\mathcal{V}_i$. Each counter vector $\mathcal{V}_i$ consists of n counters $\mathcal{V}_i(1), \ldots, \mathcal{V}_i(n)$ such that $\mathcal{V}_i = [\mathcal{V}_i(1), \ldots, \mathcal{V}_i(n)]$. When $\mathcal{V}_i$ is the counter vector associated with an order $\mathcal{O}_i$, we denote by $\mathcal{V}_i(k)$ the value of the kth counter in the counter vector stored by the agent A_i. We define ρ to be equal to $\max\{\mathcal{V}_i(1) \mid i \in 1..n\}$. The value of ρ evolves during the search so that it always corresponds to the value of the largest counter among all the first counters stored by agents.

Let K be the parameter defining the degree of flexibility of the retroactive heuristics (see Section 6.1). Next we show that the ABT_DO-Retro algorithm is correct when using m_2 and with $K = 0$. The proof that the algorithm is correct when $K \neq 0$ can be found in [**Zivan *et al.*, 2009**].

To prove the correctness of ABT_DO-Retro we use induction on the number of agents. For a single agent the order is static therefore the correctness of standard ABT implies the correctness of ABT_DO-Retro. Assume ABT_DO-Retro is correct for every DisCSP with $n-1$ agents. We show in the following that ABT_DO-Retro is correct for every DisCSP with n agents. To this end we first prove the following lemmas.

Lemma 6.1. Given enough time, if the value of ρ does not change, the highest priority agent in all orders stored by all agents will be the same.

Proof. Assume the system reaches a state σ where the value of ρ no longer increases. Let $\mathcal{O}_i$ be the order that, when generated, caused the system to enter state σ. Inevitably, we have $\mathcal{V}_i(1) = \rho$. Assume that $\mathcal{O}_i \neq \mathcal{O}_{init}$ and let A_i be the agent that generated $\mathcal{O}_i$. The agent A_i is necessarily the highest priority agent in the new order $\mathcal{O}_i$ because, the only possibility for the generator of a new order to change the position of the highest priority agent is to put itself in the first position in the new order. Thus, $\mathcal{O}_i$ is sent by A_i to all other agents because A_i must send $\mathcal{O}_i$ to all agents that have a lower priority than itself. So after a finite time all agents will be aware of $\mathcal{O}_i$. This is also true if $\mathcal{O}_i = \mathcal{O}_{init}$. Now, by assumption the value of ρ no longer increases. As a result, the only way for another agent to generate an order $\mathcal{O}'$ such that the highest priority agents in $\mathcal{O}_i$ and $\mathcal{O}'$ are different (i.e., $\mathcal{O}'(1) \neq \mathcal{O}_i(1)$) is to put itself in first position in $\mathcal{O}'$ and to do that *before* it has received $\mathcal{O}_i$ (otherwise $\mathcal{O}'$ would increase ρ). Therefore, the time passed from the moment the system entered state σ until a new order $\mathcal{O}'$ was generated is finite. Let $\mathcal{O}_j$ be the strongest such order (i.e., $\mathcal{O}'$) and let

A_j be the agent that generated $\mathcal{O}_j$. That is, A_j is the agent with smallest index among those who generated such an order $\mathcal{O}'$. The agent A_j will send $\mathcal{O}_j$ to all other agents and $\mathcal{O}_j$ will be accepted by all other agents after a finite amount of time. Once an agent has accepted $\mathcal{O}_j$, all orders that may be generated by this agent do not reorder the highest priority agent otherwise ρ would increase. □

Lemma 6.2. If the algorithm is correct for $n-1$ agents then it terminates for n agents.

Proof. If during the search ρ continues to increase, this means that some of the agents continue to send new orders in which they put themselves in first position. Hence, the nogoods they generate when proposing the new orders are necessarily unary (i.e., they have an empty left-hand side) because in ABT_DO-Retro, when the parameter K is zero the nogood sender cannot put itself in a higher priority position than the second last in the nogood. Suppose $ng_0 = \neg(x_i = v_i)$ is one of these nogoods, sent by an agent A_j. After a finite amount of time, agent A_i, the owner of x_i, will receive ng_0. Three cases can occur. First case, A_i still has value v_i in its domain. So the value v_i is pruned once and for all from $D(x_i)$ thanks to ng_0. Second case, A_i has already received a nogood equivalent to ng_0 from another agent. Here, v_i no longer belongs to $D(x_i)$. When changing its value, A_i has sent an ***ok?*** message with its new value v_i'. If A_i and A_j were neighbors, this ***ok?*** message has been sent to A_j. If A_i and A_j were not neighbors when A_i changed its value to v_i', this ***ok?*** message was sent by A_i to A_j after A_j requested to add a link between them at the moment it generated ng_0. Thanks to the assumption that messages are always delivered in a finite amount of time, we know that A_j will receive the ***ok?*** message containing v_i' a finite amount of time after it sent ng_0. Thus, A_j will not be able to send forever nogoods about a value v_i pruned once and for all from $D(x_i)$. Third case, A_i already stores a nogood with a non empty left-hand side discarding v_i. Notice that although A_j moves to the highest priority position, A_i may be of lower priority, i.e., there can be agents with higher priority than A_i according to the current order that are not included in ng_0. Thanks to the standard *highest possible lowest variable involved* [**Hirayama and Yokoo, 2000**; **Bessiere *et al.*, 2005**] heuristic for selecting nogoods in ABT algorithms, we are guaranteed that the nogood with empty left-hand side ng_0 will replace the other existing nogood and v_i will be permanently pruned from $D(x_i)$. Thus, in all three cases, every time ρ increases, we know that an agent has moved to the first position in the order, and a value was definitively pruned a finite amount of time before or after. There is a bounded number of values in the network. Thus, ρ cannot increase forever. Now, if ρ stops increasing, then after a finite amount of time the highest priority agent in all orders stored by all agents will be the same (Lemma 6.1). Since the algorithm is correct for $n-1$ agents, after each assignment of the highest priority agent, the rest of the agents will either reach an idle state,[1] generate an empty nogood indicating that there is no solution, or generate a unary nogood, which is sent to the highest priority agent. Since the number of values in the system is finite, the third option, which is the only one that does not imply immediate termination, cannot occur forever. □

1. As proved in Lemma 6.3, this indicates that a solution was found.

Lemma 6.3. If the algorithm is correct for $n-1$ agents then it is sound for n agents.

Proof. Let $\mathcal{O}'$ be the strongest order generated before reaching the state of quiescence and let $\mathcal{O}$ be the strongest order generated such that $\mathcal{V}(1) = \mathcal{V}'(1)$ (and such that $\mathcal{O}$ has changed the position of the first agent –assuming $\mathcal{O} \neq \mathcal{O}_{init}$). Given the rules for reordering agents, the agent that generated $\mathcal{O}$ has necessarily put himself first because it has modified $\mathcal{V}(1)$ and thus also the position of the highest agent. So it has sent $\mathcal{O}$ to all other agents. When reaching the state of quiescence, we know that no order $\mathcal{O}_j$ with $\mathcal{O}_j(1) \neq \mathcal{O}(1)$ has been generated because this would break the assumption that $\mathcal{O}$ is the strongest order where the position of the first agent has been changed. Hence, at the state of quiescence, every agent A_i stores an order $\mathcal{O}_i$ such that $\mathcal{O}_i(1) = \mathcal{O}(1)$. (This is also true if $\mathcal{O} = \mathcal{O}_{init}$.) Let us consider the DisCSP P composed of the $n-1$ lower priority agents according to $\mathcal{O}$. Since the algorithm is correct for $n-1$ agents, the state of quiescence means that a solution was found for P. Also, since all agents in P are aware that $\mathcal{O}(1)$ is the agent with the highest priority, the state of quiescence also implies that all constraints that involve $\mathcal{O}(1)$ have been successfully tested by agents in P, otherwise at least one agent in P would try to change its value and send an ***ok?*** or ***ngd*** message. Therefore, the state of quiescence implies that a solution was found. □

Lemma 6.4. The algorithm is complete

Proof. All nogoods are generated by logical inferences from existing constraints. Thus, an empty nogood cannot be inferred if a solution exists. □

Following Lemma 6.2, 6.3 and 6.4 we obtain the correctness of the main theorem in this chapter.

Theorem 6.1. *The ABT_DO-Retro algorithm with $K = 0$ is correct when using the m_2 method for selecting the strongest order.*

6.5 Summary

We proposed in this chapter a corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). We presented an example that shows how ABT_DO-Retro can enter an infinite loop following the natural understanding of the description given by the authors of ABT_DO-Retro. We described the correct way for comparing time-stamps of orders. We gave the proof that our method for comparing orders is correct.

DisChoco 2.0

7

Distributed constraint reasoning is a powerful concept to model and solve naturally distributed constraint satisfaction/optimization problems. However, there are very few open-source tools dedicated to solve such problems: DisChoco, DCOPolis and FRODO. A distributed constraint reasoning platform must have some important features: It should be reliable and modular in order to be easy to personalize and extend, be independent of the communication system, allow the simulation of agents on a single virtual machine, make it easy for deployment on a real distributed framework, and allow agents with a local complex problems. This paper presents DisChoco 2.0, a complete redesign of the DisChoco platform that guarantees these features and that can deal both with distributed constraint satisfaction problems and with distributed constraint optimization problems.

This chapter is organized as follows. Section 7.2 presents the global architecture of DisChoco 2.0. In Section 7.3, we show how a user can define her problem and solve it using the DisChoco 2.0 platform. Section 7.4 shows the different benchmarks available in DisChoco and how researchers in the DCR field can use them for evaluating algorithms performance. We conclude the paper in Section 7.5.

7.1 Introduction

Distributed Constraint Reasoning (DCR) is a framework for solving various problems arising in Distributed Artificial Intelligence. In DCR, a problem is expressed as a Distributed Constraint Network (DCN). A DCN is composed of a group of autonomous agents where each agent has control of some elements of information about the problem, that is, variables and constraints. Each agent own its local constraint network. Variables in different agents are connected by constraints. Agents try to find a local solution (locally consistent assignment) and communicate it with other agents using a DCR protocol to check its consistency against constraints with variables owned by other agents [**Yokoo *et al.*, 1998**; **Yokoo, 2000a**].

A DCN offers an elegant way for modeling many everyday combinatorial problems that are distributed by nature (e.g., distributed resource allocation [**Petcu and Faltings, 2004**], distributed meeting scheduling [**Wallace and Freuder, 2002**], sensor networks [**Béjar *et al.*, 2005**]). Several algorithms for solving this kind of problems have been devel-

oped. Asynchronous Backtracking (ABT [Yokoo *et al.*, 1992], ABT-Family [Bessiere *et al.*, 2005]), Asynchronous Forward Checking (AFC) [Meisels and Zivan, 2007] and Nogood-based Asynchronous Forward-Checking (AFC-ng) [Wahbi *et al.*, 2012] were developed to solve Distributed Constraint Satisfaction Problems (DisCSP). Asynchronous Distributed constraints OPTimization (Adopt) [Modi *et al.*, 2005], Asynchronous Forward-Bounding (AFB) [Gershman *et al.*, 2009], Asynchronous Branch-and-Bound (Adopt-BnB) [Yeoh *et al.*, 2008] and Dynamic backtracking for distributed constraint optimization (DyBop) [Ezzahir *et al.*, 2008a] were developed to solve Distributed Constraint Optimization Problems (DCOP).

Programming DCR algorithms is a difficult task because the programmer must explicitly juggle between many very different concerns, including centralized programming, parallel programming, asynchronous and concurrent management of distributed structures and others. In addition, there are very few open-source tools for solving DCR problems: DisChoco, DCOPolis [Sultanik *et al.*, 2008] and FRODO [Léauté *et al.*, 2009]. Researchers in DCR are concerned with developing new algorithms, and comparing their performance with existing algorithms. Open-source platforms are essential tools to integrate and test new ideas without having the burden to reimplement from scratch an ad-hoc solver. For this reason a DCR platform should have the following features:

- be reliable and modular, so it is easy to personalize and extend;
- be independent from the communication system;
- allow the simulation of multi-agent systems on a single machine;
- make it easy to implement a real distributed framework;
- allow the design of agents with local constraint networks.

In this paper we present DisChoco 2.0,[1] a completely redesigned platform that guarantees the features above. DisChoco 2.0 allows to represent both DisCSPs and DCOPs, as opposed to other platforms. DisChoco 2.0 is not a distributed version of the centralized solver Choco, but it implements a model to solve DCN with local complex problems (i.e., several variables per agent) by using Choco[2] as local solver to each agent. DisChoco 2.0 is an open source Java library which aims at implementing DCR algorithms from an abstract model of agent (already implemented in DisChoco). A single implementation of a DCR algorithm can run as simulation on a single machine, or on a network of machines that are connected via the Internet or via a wireless ad-hoc network, or even on mobile phones compatible with J2ME.

7.2 Architecture

In order to reduce the time of development and therefore the cost of the design we choose a components approach allowing pre-developed components to be reused. This components approach is based on two principles:

- Each component is developed independently;
- An application is an assemblage of particular components.

1. http://www.lirmm.fr/coconut/dischoco/
2. http://choco.emn.fr/

Figure 7.1 shows the general structure of DisChoco kernel. It shows a modular architecture with a clear separation between the modules used, which makes the platform easily maintainable and extensible.

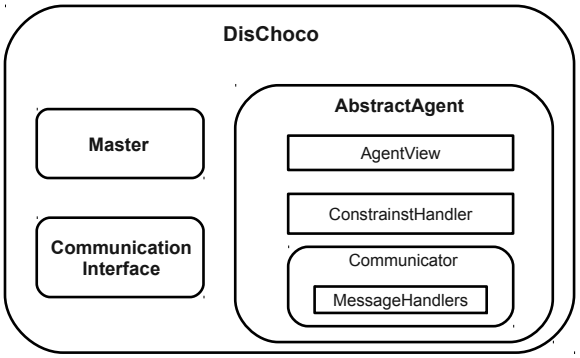


Figure 7.1 – Architecture of DisChoco kernel.

The kernel of DisChoco consists of an abstract model of an agent and several components namely the communicator, messages handlers, constraints handler, the Agent View (AgentView), a Master who controls the global search (i.e., send messages to launch and to stop the search, etc.) and a communication interface.

7.2.1 Communication System

Thanks to independence between the kernel of DisChoco and the communication system that will be used (Figure 7.2), DisChoco enables both: the simulation on one machine and the full deployment on a real network. This is done independently of the type of network, which can be a traditional wired network or an ad-hoc wireless network.

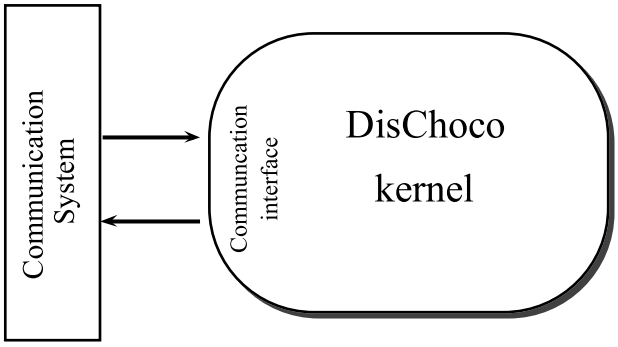


Figure 7.2 – Independence between the kernel of DisChoco and the communication system.

Instead of rewriting a new system of communication between DisChoco agents we adopted the component approach. Thus a communication component pre-developed can be used as a communication system if it satisfies a criterion of tolerance to failure. This allows us to use only the identifiers of agents (IDs) to achieve communication between agents. Thus when agent A_i wants to send a message to the agent A_j, it only attaches its ID (i) and the ID (j) of the recipient. It is the communication interface that will deal with

mapping between the IDs and IP addresses of agents (we assume that an agent identifier is unique).

In the case of a simulation on a single Java Virtual Machine agents are simulated by Java threads. Communication among agents is done using an Asynchronous Message Delay Simulator (MailerAMDS) [**Zivan and Meisels, 2006b**; **Ezzahir *et al.*, 2007**]. MailerAMDS is a simulator that models the asynchronous delays of messages. Then, agents IDs are sufficient for communication. In the case of a network of Java Virtual Machines, we have used SACI[3] (Simple Agent Communication Infrastructure) as communication system. The validity of this choice has not yet been validated by an in depth analysis. Future work will be devoted to testing a set of communication systems on different types of networks.

7.2.2 Event Management

DisChoco performs constraint propagation via events on variables and events on constraints, as in Choco. These events are generated by changes on variables, and managing them is one of the main tasks of a constraint solver. In a distributed system there are some other events that must be exploited. These events correspond to a reception of a message, changing the state of an agent (wait, idle and stop) or to changes on the AgentView.

The AgentView of a DisChoco agent consists of external variables (copy of other agents variables). Whenever an event occurs on one of these external variables, some external constraints can be awakened and so added to the queue of constraints that will be propagated. Using a queue of constraints to be propagated allows to only process constraints concerned by changes on the AgentView instead of browsing the list of all constraints. To this end, the DisChoco user can use methods offered by the constraints handler (*ConstraintsHandler*).

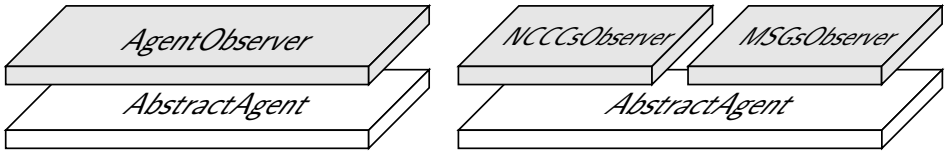
Detecting the termination of a distributed algorithm is not a trivial task. It strongly depends on statements of agents. To make the implementation of a termination detection algorithm easy, we introduced in the DisChoco platform a mechanism that generates events for changes on the statements of an agent during its execution. A module for detecting termination is implemented under each agent as a listener of events on statements changes. When the agent state changes, the termination detector receives the event, recognizes the type of the new state and executes methods corresponding to termination detection.

The events corresponding to an incoming message are managed in DisChoco in a manner different from the standard method. Each agent has a Boolean object that is set to false as long as the inbox of the agent is empty. When a message has arrived to the inbox, the agent is notified by the change of this Boolean object to true. The agent can use methods available in the communicator module to dispatch the received message to its corresponding handler.

7.2.3 Observers in layers

DisChoco provides a Java interface (*AgentObserver*) that allows the user to track operations of a DCR algorithm during its execution. This interface defines two main functions:

3. http://www.lti.pcs.usp.br/saci/



(a) AgentObserver *implemented as Layer* (b) *Metrics implemented as observers*

Figure 7.3 – Layer model for observers.

whenSendMessage and *whenReceivedMessage*. The class *AbstractAgent* provides a list of observers and functions to add one or several observers. Thus, when we want to implement an application using DisChoco, we can use *AgentObserver* to develop a specific observer. This model is shown in Figure 7.3(a).

When developing new algorithms, an important task is to compare their performance to other existing algorithms. There are several metrics for measuring performance of DCR algorithms: non-concurrent constraint checks (#*ncccs* [**Meisels *et al.*, 2002**]), equivalent non-concurrent constraint checks (#*encccs* [**Chechetka and Sycara, 2006**]), number of exchanged messages (#*msg* [**Lynch, 1997**]), degree of privacy loss[**Brito *et al.*, 2009**], etc. DisChoco simply uses *AgentObserver* to implement these metrics as shown in Figure 7.3(b). The user can enable metrics when she needs them or disable some or all these metrics. The user can develop her specific metric or her methods for collecting statistics by implementing *AgentObserver*.

7.3 Using DisChoco 2.0

Figure 7.4 presents a definition of a distributed problem named (*Hello DisChoco*) using the Java code. In this problem there are 3 agents $\mathcal{A} = \{A_1, A_2, A_3\}$ where each agent controls exactly one variable. The domain of A_1 and A_2 contains two values $D_1 = D_2 = \{1, 2\}$ and that of A_3 contains one value $D_3 = \{2\}$. There are two constraints of *difference*: the first constraint is between A_1 and A_2 and the second one is between A_2 and A_3. After defining our problem we can configure our solver. Thus, the problem can be solved using a specified implemented protocol (ABT for example).

For DisChoco inputs we choose to use a XML format called *XDisCSP* derived from the famous format XCSP 2.1.[4] Figure 7.5 shows an example of representation of the problem defined above in the *XDisCSP* format. Each variable has a unique ID, which is the concatenation of the ID of it owner agent and index of the variable in the agent. This is necessary when defining constraints (scope of constraints). For constraints, we used two types of constraints: TKC for Totally Known Constraint and PKC for Partially Known Constraint [**Brito *et al.*, 2009**]. Constraints can be defined in extension or as a Boolean function. Different types of constraints are predefined: equal to $eq(x, y)$, different from $ne(x, y)$, greater than or equal $ge(x, y)$, greater than $gt(x, y)$, less than or equal $le(x, y)$, less than $lt(x, y)$,etc.

According to this format we can model DisCSPs and DCOPs. Once a distributed con-

4. http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

```java
AbstractMaster master = Protocols.getMaster(Protocols.ABT);
DisProblem disCSP = new DisProblem("Hello DisChoco", master);
SimpleAgent[] agents = new SimpleAgent[3];
IntVar[] variables = new IntVar[3];
// Make agents
agents[0] = (SimpleAgent) disCSP.makeAgent("A1", "");
agents[1] = (SimpleAgent) disCSP.makeAgent("A2", "");
agents[2] = (SimpleAgent) disCSP.makeAgent("A3", "");
// Make one single variable for each agent
variables[0] = agents[0].makeInternalVar(new int[] {1, 2}); // x1
variables[1] = agents[1].makeInternalVar(new int[] {1, 2}); // x2
variables[2] = agents[2].makeInternalVar(new int[] {2}); // x3
// Make two constraints, we must to create external var on each agent
// But each agent must known its constraints
// x1!=x2
agents[0].neqY(agents[0].makeExternalVar(variables[1]));
agents[1].neqY(agents[1].makeExternalVar(variables[0]));
// x2!=x3
agents[1].neqY(agents[1].makeExternalVar(variables[2]));
agents[2].neqY(agents[2].makeExternalVar(variables[1]));
// Make a simulator to resolve the problem
DisCPSolver solver = new DisSolverSimulator(disCSP);
solver.setCentralizedAO(new LexicographicAO());
solver.addNCCCMetric();
solver.addCommunicationMetric();
solver.solve();
System.out.println("Problem : " + disCSP.getProblemName());
System.out.println("Solution of the problem using " + disCSP.master.getClass());
System.out.println("-------------------------------------------------------------");
System.out.println(solver.getGlobalSolution());
System.out.println("-------------------------------------------------------------");
System.out.println("Statistics :");
System.out.println(solver.getStatistics());
```

Figure 7.4 – Definition of a distributed problem using Java code.

```xml
<instance>
  <presentation name="Hello DisChoco" model="Simple" maxConstraintArity="2" format="XDisCSP 1.0" />
  <agents nbAgents="3">
    <agent name="A1" id="1" description="Agent 1" />
    <agent name="A2" id="2" description="Agent 2" />
    <agent name="A3" id="3" description="Agent 3" />
  </agents>
  <domains nbDomains="2">
    <domain name="D1" nbValues="2">1 2</domain>
    <domain name="D2" nbValues="1">2</domain>
  </domains>
  <variables nbVariables="3">
    <variable agent="A1" name="X1.0" id="0" domain="D1" description="Variable x_1" />
    <variable agent="A2" name="X2.0" id="0" domain="D1" description="Variable x_2" />
    <variable agent="A3" name="X3.0" id="0" domain="D2" description="Variable x_3" />
  </variables>
  <predicates nbPredicates="1">
    <predicate name="P0">
      <parameters>int x int y</parameters>
      <expression>
        <functional>ne(x,y)</functional>
      </expression>
    </predicate>
  </predicates>
  <constraints nbConstraints="2">
    <constraint name="C1" model="TKC" arity="2" scope="X1.0 X2.0" reference="P0">
      <parameters>X1.0 X2.0</parameters>
    </constraint>
    <constraint name="C2" model="TKC" arity="2" scope="X2.0 X3.0" reference="P0">
      <parameters>X2.0 X3.0</parameters>
    </constraint>
  </constraints>
</instance>
```

Figure 7.5 – Definition of the *Hello DisChoco* problem in XDisCSP 1.0 format.

straint network problem is expressed in the *XDisCSP* format, we can solve it using one of the protocols developed on the platform. The algorithms currently implemented in DisChoco 2.0 are: ABT [Yokoo *et al.*, 1992; Bessiere *et al.*, 2005], ABT-Hyb [Brito and Meseguer, 2004], ABT-dac [Brito and Meseguer, 2008], AFC [Meisels and Zivan, 2007], AFC-ng [Ezzahir *et al.*, 2009], AFC-tree [Wahbi *et al.*, 2012], DBA [Yokoo and Hirayama, 1995] and DisFC [Brito *et al.*, 2009] in the class of DisCSPs with simple agents. In the class of DisCSPs where agents have local complex problems, ABT-cf [Ezzahir *et al.*, 2008b] was implemented. For DCOPs, the algorithms that are implemented in DisChoco 2.0 are: Adopt [Modi *et al.*, 2005], BnB-Adopt [Yeoh *et al.*, 2008] and AFB [Gershman *et al.*, 2009]. For solving a problem, we can use a simple command line:

```
1 java -cp dischoco.jar dischoco.simulation.Run protocol problem.xml
```

The Graphical User Interface (GUI) of DisChoco allows to visualize the constraint graph. Hence, the user can analyses the structure of the problem to be solved. This also helps to debug the algorithms. An example of the visualization is shown in Figure 7.6.

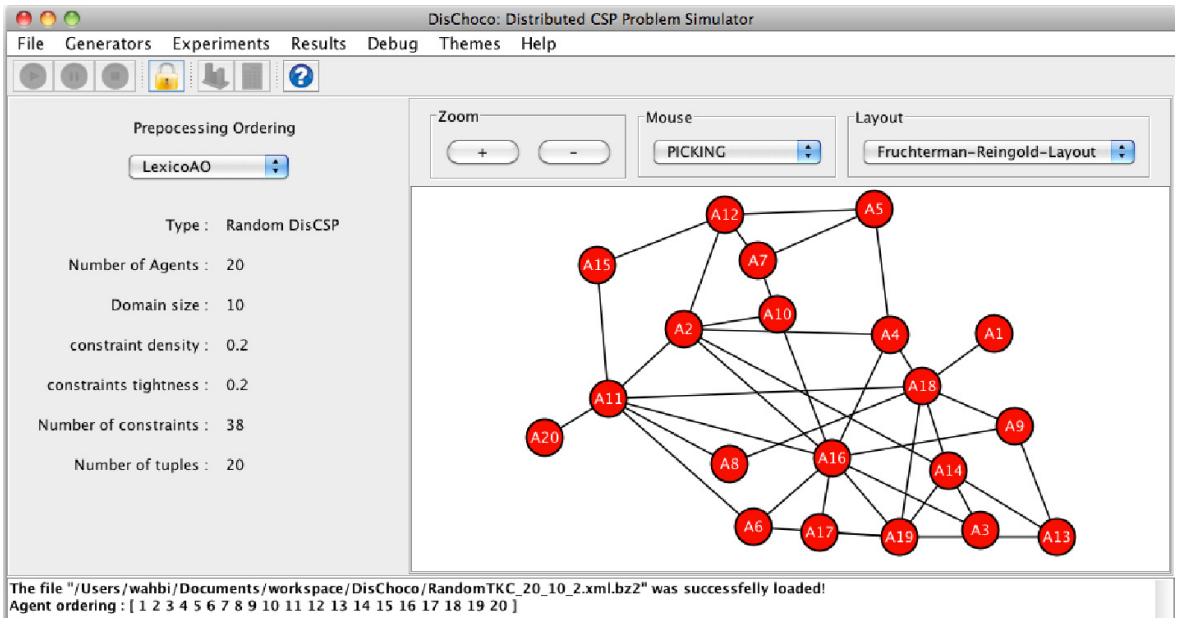


Figure 7.6 – Visualization of the structure of the distributed constraint graph.

7.4 Experimentations

In addition to its good properties (reliable and modular), DisChoco provides several other facilities, especially for performing experimentation. The first facility is in the generation of benchmark problems. DisChoco offers a library of generators for distributed constraint satisfaction/optimization problems (e.g., random binary DisCSPs using model B, random binary DisCSPs with complex local problems, distributed graph coloring, distributed meeting scheduling, sensor networks, distributed N-queens, etc. These generators

allow the user to test her algorithms on various types of problems ranging from purely random problems to real world problems.

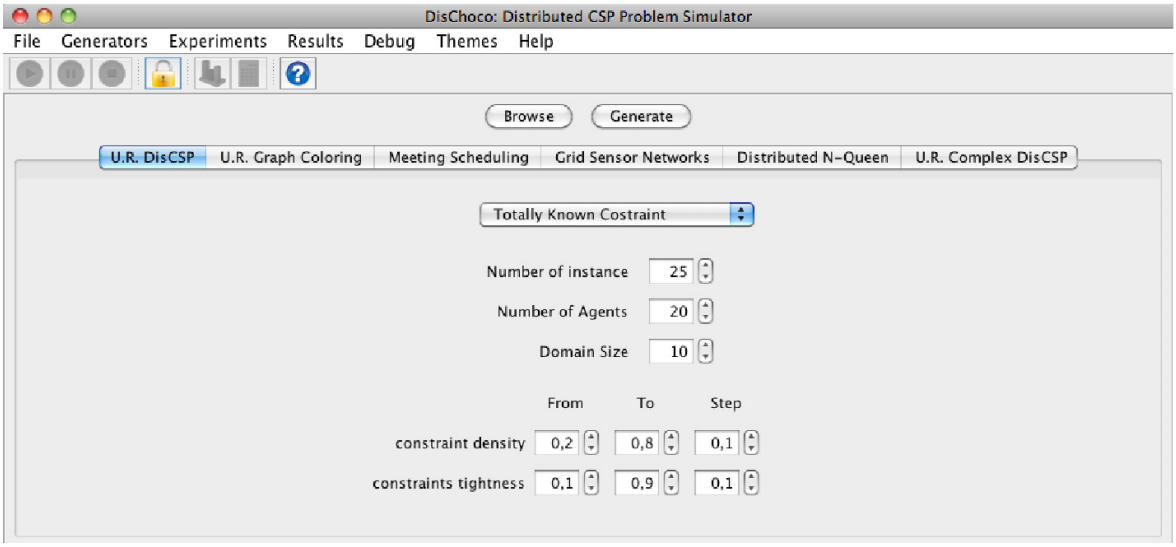


Figure 7.7 – A screenshot of the graphical user interface showing generators in DisChoco.

DisChoco is equipped with a GUI for manipulating all above generators. A screenshot of the GUI of DisChoco shows various generators implemented on DisChoco (Figure 7.7). Once the instances have been generated, a XML configuration file is created to collect the instances. The generated instances are organized in a specific manner for each kind of problems generator in a directory indicated by the user. The configuration file can also contain details related to the configuration of the communicator and the list of algorithms to be compared. It will be used for launching experiments. After all these configurations have been set, the user can launch the experiments either on the GUI mode or on the command mode.

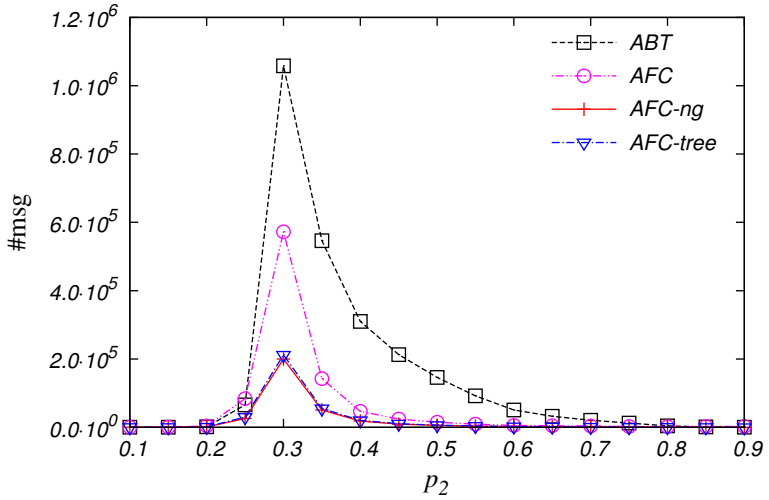


Figure 7.8 – Total number of exchanged messages on dense graph ⟨n=20, d=10, p_1=0.7, p_2⟩.

DisChoco is also equipped with a complete manager of results. The user does not have to worry about organizing and plotting results. All this is offered by DisChoco that automatically generates *gnuplot* plots of the requested measures. The user can also handle all results and compare algorithms using the GUI of DisChoco. Figure 7.8 shows an example of plot generated from experimentations on some algorithms implemented in DisChoco.

7.5 Conclusion

In this chapter, we have presented the new version 2.0 of the DisChoco platform for solving DCR problems. This version contains several interesting features: it is reliable and modular, it is easy to personalize and to extend, it is independent from the communication system and allows a deployment in a real distributed system as well as the simulation on a single Java Virtual Machine.

Conclusions and perspectives

In this thesis, we have addressed the Distributed Constraint Satisfaction Problem (DisCSP) framework. We proposed several complete distributed search algorithms and reordering heuristics for DisCSPs. We have given a complete evaluation of the efficiency of our contributions against the existing approaches in literature. Our results show that they improve the current state-of-the-art.

Once we defined the constraint satisfaction problem formalism (CSP) and presented some examples of academical and real combinatorial problems that can be modeled as CSP, we reported the main existent algorithms and heuristics used for solving CSPs. Next, we formally defined the distributed constraint satisfaction problem (DisCSP) paradigm. We have illustrated how some instances of real world applications in multi-agent coordination can be encoded in DisCSP. We introduced the meeting scheduling problem in its distributed form where agents may solve the problem, thanks to the DisCSP, without delivering their personal information to a centralized agent. We described a real distributed resource allocation application, that is, the distributed sensor network problem, and formalize it as a distributed CSP. These two problems have been used as benchmarks when comparing the algorithms proposed in this thesis. We have also described the state of the art algorithms and heuristics for solving DisCSP.

Numerous contributions have been proposed in this thesis. Our first contribution is the Nogood-Based Asynchronous Forward Checking (AFC-ng). AFC-ng is an algorithm based on Asynchronous Forward Checking (AFC). AFC incorporates the idea of the forward-checking in a synchronous search procedure. However, agents perform the forward checking phase asynchronously. AFC-ng is the nogood-based version of the AFC. Hence, instead of using the shortest inconsistent partial assignments, AFC-ng uses nogoods as justifications of value removals. Moreover, unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. AFC-ng tries to enhance the asynchronism of the forward checking phase.

In order to enhance the asynchronism in our nogood-based Asynchronous Forward-Checking (AFC-ng) algorithm, we extended it to the Asynchronous Forward-Checking Tree (AFC-tree). The main feature of the AFC-tree algorithm is using different agents to search non-intersecting parts of the search space concurrently. In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. The pseudo-tree ordering is built in a preprocessing step. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. The agents that are brothers are committed to concurrently find the partial solutions of their

variables. Therefore, AFC-tree exploits the potential speed-up of a parallel exploration in the processing of distributed problems.

Since our experiments show that our AFC-ng is one of the most efficient and robust algorithm for solving DisCSP, we proposed two new algorithms based on the same mechanism as AFC-ng to maintain arc consistency in synchronous search procedure. Thereby, instead of using forward checking as a filtering property, we maintain arc consistency asynchronously (MACA). The first algorithm we proposed enforces arc consistency thanks to an additional type of messages, deletion messages. This algorithm is called MACA-del. The second algorithm, that we called MACA-not, achieves arc consistency without any new type of message.

In our contributions mentioned above, the agents assign values to their variables in a sequential way. These contributions can be classified under the category of synchronous algorithms. The other category of algorithms for solving DisCSPs are algorithms in which the process of proposing values to the variables and exchanging these proposals is performed asynchronously between the agents. In the last category, we proposed Agile Asynchronous Backtracking (Agile-ABT), an asynchronous dynamic ordering algorithm that is able to change the ordering of agents more agilely than all previous approaches. Thanks to the original concept of termination value, Agile-ABT is able to choose a backtracking target that is not necessarily the agent with the current lowest priority within the conflicting agents. Furthermore, the ordering of agents appearing before the backtracking target can be changed. These interesting features are unusual for an algorithm with polynomial space complexity.

In this thesis, we proposed a corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). We presented an example that shows how ABT_DO-Retro can enter an infinite loop following the natural understanding of the description given by the authors of ABT_DO-Retro. We described the correct way for comparing time-stamps of orders. We gave the proof that our method for comparing orders is correct.

Finally, we have presented the new version of the DisChoco platform for solving DCR problems, DisChoco 2.0. This version contains several interesting features: it is reliable and modular, it is easy to personalize and to extend, it is independent from the communication system and allows a deployment in a real distributed system as well as the simulation on a single Java Virtual Machine. DisChoco 2.0 is an open source Java library which aims at implementing distributed constraint reasoning algorithms from an abstract model of agent (already implemented in DisChoco). A single implementation of a distributed constraint reasoning algorithm can run as simulation on a single machine, or on a network of machines. All algorithms proposed in this thesis were implemented and tested using this platform.

Conclusions

From the works presented in this thesis we can conclude that:

- Using nogoods as justification of value removals is significant in distributed search algorithms for solving DisCSPs.
- Allowing several concurrent backtracks in a synchronous search improves the efficiency.
- Enhancing the asynchronism in the synchronous search that propagates only consistent partial assignments is beneficial.
- The asynchronous algorithms performs bad in dense graphs compared to synchronous algorithms.
- In synchronous algorithms, maintaining arc consistency is better than maintaining forward checking in terms of computational effort when the network is sparse, and is always better in terms of communication load.
- Maintaining arc consistency during synchronous search produces much larger benefits than maintaining arc consistency in asynchronous algorithms like ABT.
- Our experiments confirm the significance of reordering the agents appearing before the backtracking target.

Future Works

In future research, testing some reordering heuristics for the nogood-based Asynchronous Forward-Checking (AFC-ng) and Maintaining Arc Consistency Asynchronously (MACA) algorithms could be very interesting. For the Asynchronous Forward-Checking Tree (AFC-tree), an intuitive improvement will be to maintain the arc consistency instead of using forward checking as filtering property. In the AFC-tree the agents are prioritized using a pseudo-tree arrangement of the constraint graph. There exist in literature various heuristics to build the pseudo-trees from constraint graphs. These heuristics aim to produce shallow or deep pseudo-trees. We believe that testing AFC-tree on different kinds of pseudo-trees will allow more understanding of its good performance.

In Agile Asynchronous Backtracking (Agile-ABT), we successfully combined the timestamps of orders with the ***dom*** heuristic in the termination value concept. We can test other heuristics in the termination values. The ***dom/deg*** and the ***dom/wdeg*** could be very promising heuristics to incorporate in the distributed search. Our goal is to integrate these heuristics in the Agile-ABT.

In Agile-ABT, when updating the set of the stored explanations, some explanations can be deleted. Unfortunately, this process lead to lose the information about the domain sizes received from other agents. Thus, other data structures (i.e., nogoods) allowing more accurate definition of domain sizes should be tested in Agile-ABT. However, sharing the whole nogoods will increase the size of exchanged messages. Thereby, another improvement could be to try to minimize the size of exchanged messages.

Minimizing the size of exchanged messages on MACA algorithms is also required in order to use it for solving applications where the message size is limited. Testing different kind of consistency maintenance in MACA algorithms may also help to improve their efficiency.

We aim also at enhancing the DisChoco platform by the implementation of other Distributed Constraint Reasoning (DCR) algorithms and to enrich the graphical user interface to make it easier to use for researchers from the DCR field. Another direction of improvement is to allow DisChoco to support other types of constraints that match as much as possible the needs of real applications. The modularity of DisChoco will allow us to look for other types of system communication. Finally, for a complete validation, it would be interesting to test DisChoco on a real distributed system.

Bibliography

[**Abu-Amara, 1988**] Hosame H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37:449–453, April 1988. (Cited on pages 33, 64, and 65.)

[**Bacchus and Van Run, 1995**] Fahiem Bacchus, and Paul Van Run. Dynamic variable ordering in CSPs. In *Proceeding of the First International Conference on Principles and Practice of Constraint Programming*, CP'95, pages 258–275, 1995. (Cited on page 25.)

[**Beck *et al.*, 2005**] J. Christopher Beck, Patrick Prosser, and Richard J. Wallace. Trying again to fail-first. In *Proceedings of the 2004 joint ERCIM/CoLOGNET international conference on Recent Advances in Constraints*, CSCLP'04, pages 41–55, Berlin, Heidelberg, 2005. Springer-Verlag. (Cited on page 26.)

[**Béjar *et al.*, 2005**] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman, and Magda Valls. Sensor networks and distributed csp: communication, computation and complexity. *Artificial Intelligence*, 161:117–147, 2005. (Cited on pages 1, 28, 30, 32, 33, 55, 73, 103, and 115.)

[**Bessiere and Cordier, 1993**] Christian Bessiere, and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the eleventh national conference on Artificial intelligence*, AAAI'93, pages 108–113. AAAI Press, 1993. (Cited on page 22.)

[**Bessiere and Régin, 1996**] Christian Bessiere, and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, CP'96, pages 61–75, 1996. (Cited on pages 27, 52, 78, and 87.)

[**Bessiere and Régin, 2001**] Christian Bessiere, and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01, pages 309–315, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. (Cited on pages 22 and 82.)

[**Bessiere *et al.*, 1999**] Christian Bessiere, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, jan 1999. (Cited on page 22.)

[**Bessiere *et al.*, 2001a**] Christian Bessiere, Assef Chmeiss, and Lakhdar Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, CP'01, pages 565–569, London, UK, UK, 2001. Springer-Verlag. (Cited on page 27.)

[**Bessiere *et al.*, 2001b**] Christian Bessiere, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In *Proceeding of Workshop on Distributed Constraint Reasoning, IJCAI'01*, Seattle, Washington, USA, August 4 2001. (Cited on page 33.)

[**Bessiere *et al.*, 2005**] Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161:7–24, 2005. (Cited on pages 33, 37, 38, 40, 47, 52, 90, 113, 116, and 121.)

[**Bessiere *et al.*, 2011**] Christian Bessiere, El-Houssine Bouyakhf, Younes Mechqrane, and Mohamed Wahbi. Agile Asynchronous Backtracking for Distributed Constraint Satisfaction Problems. In *Proceedings of 23rd IEEE International Conference on Tools with Artificial Intelligence*, ICTAI'11, pages 777–784, Boca Raton, Florida, USA, November 2011. IEEE. (Cited on page 89.)

[**Bessiere, 1994**] Christian Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, January 1994. (Cited on page 22.)

[**Bessiere, 2006**] Christian Bessiere. Chapter 3 constraint propagation. In Rossi Francesca, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006. (Cited on pages 21 and 79.)

[**Bitner and Reingold, 1975**] James R. Bitner, and Edward M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–656, nov 1975. (Cited on page 15.)

[**Bliek, 1998**] Christian Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI'98/IAAI'98, pages 319–325, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. (Cited on pages 19, 20, and 106.)

[**Boussemart *et al.*, 2004**] Fréd'eric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, ECAI'04, pages 146–150, 2004. (Cited on page 27.)

[**Brélaz, 1979**] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, apr 1979. (Cited on page 26.)

[**Brito and Meseguer, 2003**] Ismel Brito, and Pedro Meseguer. Distributed forward checking. In *Proceeding of 9th International Conference on Principles and Practice of Constraint Programming*, CP'03, pages 801–806, Ireland, 2003. (Cited on pages 33 and 78.)

[**Brito and Meseguer, 2004**] Ismel Brito, and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSP. In *Proceeding of the fifth Workshop on Distributed Constraint Reasoning at the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, DCR'04, pages 80–94. Toronto, Canada, September 2004. (Cited on pages 33, 59, and 121.)

[**Brito and Meseguer, 2008**] Ismel Brito, and Pedro Meseguer. Connecting abt with arc consistency. In *CP*, pages 387–401, 2008. (Cited on pages 43, 78, 85, and 121.)

[**Brito *et al.*, 2009**] Ismel Brito, Amnon Meisels, Pedro Meseguer, and Roie Zivan. Distributed constraint satisfaction with partially known constraints. *Constraints*, 14:199–234, June 2009. (Cited on pages 119 and 121.)

[**Chandy and Lamport, 1985**] K. Mani Chandy, and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985. (Cited on pages 68 and 95.)

[**Chechetka and Sycara, 2005**] Anton Chechetka, and Katia Sycara. A decentralized variable ordering method for distributed constraint optimization. Technical Report CMU-RI-TR-05-18, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005. (Cited on page 64.)

[**Chechetka and Sycara, 2006**] Anton Chechetka, and Katia Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS'06, pages 1427–1429, Hakodate, Japan, 2006. (Cited on page 119.)

[**Cheung, 1983**] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE transaction on software engineering*, 9(4):504–512, 1983. (Cited on page 65.)

[**Chinn *et al.*, 1982**] P. Z. Chinn, J. Chvátalová, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices-a survey. *Journal of Graph Theory*, 6(3):223–254, 1982. (Cited on page 24.)

[**Chong and Hamadi, 2006**] Yek Loong Chong, and Youssef Hamadi. Distributed log-based reconciliation. In *Proceedings of the 17th European Conference on Artificial Intelligence*, ECAI'06, pages 108–112, 2006. (Cited on pages 1, 28, and 30.)

[**Collin *et al.*, 1991**] Zeev Collin, Rina Dechter, and Shmuel Katz. On the feasibility of distributed constraint satisfaction. In *IJCAI*, pages 318–324, 1991. (Cited on page 76.)

[**Davis *et al.*, 1962**] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, jul 1962. (Cited on page 15.)

[**De Kleer and Sussman, 1980**] Johan De Kleer, and Gerald Jay Sussman. Propagation of constraints applied to circuit synthesis. *International Journal of Circuit Theory and Applications*, 8(2):127–144, 1980. (Cited on page 8.)

[**Dechter and Frost, 2002**] Rina Dechter, and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, apr 2002. (Cited on page 17.)

[**Dechter and Meiri, 1989**] Rina Dechter, and Itay Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1*, IJCAI'89, pages 271–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. (Cited on pages 24 and 25.)

[**Dechter and Pearl, 1988**] Rina Dechter, and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988. (Cited on pages 8 and 28.)

[**Dechter, 1990**] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, jan 1990. (Cited on pages 8 and 17.)

[**Dechter, 1992**] Rina Dechter. Constraint networks (survey). *In S. C. Shapiro (Eds.), Encyclopedia of Artificial Intelligence*, 1:276–285, 1992. (Cited on page 9.)

[**Ezzahir *et al.*, 2007**] Redouane Ezzahir, Christian Bessiere, Mustapha Belaissaoui, and El Houssine Bouyakhf. DisChoco: a platform for distributed constraint programming. In *Proceedings of the IJCAI'07 workshop on Distributed Constraint Reasoning*, pages 16–21, Hyderabad, India, January 8 2007. (Cited on page 118.)

[**Ezzahir *et al.*, 2008a**] Redouane Ezzahir, Christian Bessiere, Imade Benelallam, El Houssine Bouyakhf, and Mustapha Belaissaoui. Dynamic backtracking for distributed constraint optimization. In *Proceeding of the 18th European Conference on Artificial Intelligence*, ECAI'08, pages 901–902, Amsterdam, The Netherlands, 2008. IOS Press. (Cited on page 116.)

[**Ezzahir *et al.*, 2008b**] Redouane Ezzahir, Christian Bessiere, El Houssine Bouyakhf, and Mustapha Belaissaoui. Asynchronous backtracking with compilation formulation for handling complex local problems. *ICGST International Journal on Artificial Intelligence and Machine Learning, AIML*, 8:45–53, 2008. (Cited on page 121.)

[**Ezzahir *et al.*, 2009**] Redouane Ezzahir, Christian Bessiere, Mohamed Wahbi, Imade Benelallam, and El-Houssine Bouyakhf. Asynchronous inter-level forward-checking

for discsps. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 304–318, 2009. (Cited on pages 33, 78, and 121.)

[**Fox *et al.*, 1982**] Mark S. Fox, Bradley P. Allen, and Gary Strohm. Job-shop scheduling: An investigation in constraint-directed reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, AAAI'82, pages 155–158, 1982. (Cited on page 8.)

[**Frayman and Mittal, 1987**] Felix Frayman, and Sanjay Mittal. Cossack: A constraint-based expert system for configuration tasks. In *Knowledge-Based Expert Systems in Engineering, Planning and Design*, pages 144–166, 1987. (Cited on page 8.)

[**Freuder and Quinn, 1985**] Eugene C. Freuder, and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *In IJCAI 1985*, pages 1076–1078, 1985. (Cited on pages 25, 62, and 63.)

[**Freuder, 1982**] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of The ACM*, 29:24–32, 1982. (Cited on page 24.)

[**Frost and Dechter, 1994**] Daniel Frost, and Rina Dechter. In search of the best constraint satisfaction search. In *Proceeding of Twelfth National Conference of Artificial Intelligence*, AAAI'94, pages 301–306, 1994. (Cited on page 26.)

[**Frost and Dechter, 1995**] Daniel Frost, and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, IJCAI'95, pages 572–578, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. (Cited on page 28.)

[**Garrido and Sycara, 1996**] Leonardo Garrido, and Katia Sycara. Multiagent meeting scheduling: Preliminary experimental results. In *Proceedings of the Second International Conference on Multiagent Systems*, ICMAS'96, pages 95–102, 1996. (Cited on page 11.)

[**Gaschnig, 1974**] John Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of the Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974. (Cited on page 23.)

[**Gaschnig, 1978**] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*,, pages 268–277, 1978. (Cited on pages 8, 16, and 17.)

[**Geelen, 1992**] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI'92, pages 31–35, New York, NY, USA, 1992. John Wiley & Sons, Inc. (Cited on page 28.)

[**Geffner and Pearl, 1987**] Hector Geffner, and Judea Pearl. An improved constraint-propagation algorithm for diagnosis. In *Proceedings of the 10th international joint conference on Artificial intelligence - Volume 2*, IJCAI'87, pages 1105–1111, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc. (Cited on page 8.)

[**Gent *et al.*, 1996**] Ian P. Gent, Ewan MacIntyre, Patrick Presser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, CP'96, pages 179–193, 1996. (Cited on pages 25 and 27.)

[**Gershman *et al.*, 2009**] A. Gershman, Amnon Meisels, and Roie Zivan. Asynchronous forward bounding for distributed cops. *Journal of Artificial Intelligence Research*, 34:61–88, 2009. (Cited on pages 116 and 121.)

[**Ginsberg and McAllester, 1994**] Matthew L. Ginsberg, and David A. McAllester. GSAT and dynamic backtracking. In *KR*, pages 226–237, 1994. (Cited on pages 19, 92, and 106.)

[**Ginsberg *et al.*, 1990**] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI'90, pages 210–215. AAAI Press, 1990. (Cited on page 28.)

[**Ginsberg, 1993**] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993. (Cited on pages 8, 18, 42, 90, and 106.)

[**Golomb and Baumert, 1965**] Solomon W. Golomb, and Leonard D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12:516–524, oct 1965. (Cited on pages 8, 15, and 26.)

[**Grant and Smith, 1996**] Stuart A. Grant, and Barbara M. Smith. The phase transition behaviour of maintaining arc consistency. In *Proceedings of ECAI'96*, pages 175–179, 1996. (Cited on pages 78 and 87.)

[**Hamadi *et al.*, 1998**] Youssef Hamadi, Christian Bessiere, and Joël Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of the European Conference on Artificial Intelligence*, ECAI'98, pages 219–223, Brighton, UK, 1998. (Cited on pages 33 and 40.)

[**Hamadi, 2002**] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal of Artificial Intelligence Tools*, 11:167–188, 2002. (Cited on page 76.)

[**Haralick and Elliott, 1979**] Robert M. Haralick, and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international joint conference on Artificial intelligence*, IJCAI'79, pages 356–364, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. (Cited on page 20.)

[**Haralick and Elliott, 1980**] Robert M. Haralick, and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980. (Cited on pages 8, 20, 23, 25, 26, 28, 34, 46, 62, and 78.)

[**Hirayama and Yokoo, 2000**] Katsutoshi Hirayama, and Makoto Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, ICDCS'00, pages 169–177, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on pages 38, 47, 50, 52, 70, 81, 82, 85, and 113.)

[**Horsch and Havens, 2000**] Michael C. Horsch, and William S. Havens. An empirical study of probabilistic arc consistency as a variable ordering heuristic. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP'00, pages 525–530, London, UK, UK, 2000. Springer-Verlag. (Cited on page 27.)

[**Jung *et al.*, 2001**] Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. Argumentation as distributed constraint satisfaction: applications and results. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS'01, pages 324–331, 2001. (Cited on pages 1, 28, 30, and 32.)

[**Kask *et al.*, 2004**] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of 10th International Conference on Constraint Programming*, CP'04, pages 317–331, 2004. (Cited on page 28.)

[**Léauté and Faltings, 2011**] Thomas Léauté, and Boi Faltings. Coordinating logistics operations with privacy guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, IJCAI'11, pages 2482–2487, July 16–22 2011. (Cited on pages 1 and 28.)

[**Léauté *et al.*, 2009**] T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In *Proceedings of the IJCAI'09 workshop on Distributed Constraint Reasoning*, pages 160–164, Pasadena, California, USA, 2009. (Cited on page 116.)

[**Lecoutre *et al.*, 2004**] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '04, pages 549–557, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 27.)

[**Lynch, 1997**] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997. (Cited on pages 52, 70, 85, and 119.)

[**Mackworth, 1977**] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. (Cited on pages 21 and 22.)

[**Mackworth, 1983**] Alan K. Mackworth. On seeing things, again. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI'83, pages 1187–1191, 1983. (Cited on page 8.)

[**Maheswaran *et al.*, 2004**] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'04, 2004. (Cited on pages 1, 28, and 30.)

[**Mechqrane *et al.*, 2012**] Younes Mechqrane, Mohamed Wahbi, Christian Bessiere, El-Houssine Bouyakhf, Amnon Meisels, and Roie Zivan. Corrigendum to "Min-Domain Retroactive Ordering for Asynchronous Backtracking". *Constraints*, 17:348–355, 2012. (Cited on pages 101 and 107.)

[**Meisels and Lavee, 2004**] Amnon Meisels, and Oz Lavee. Using additional information in DisCSP search. In *Proceeding of 5th workshop on distributed constraints reasoning*, DCR'04, 2004. (Cited on pages 11, 12, 31, 56, and 74.)

[**Meisels and Razgon, 2002**] Amnon Meisels, and I. Razgon. Distributed forward-checking with conflict-based backjumping and dynamic ordering. In *Proceeding of CoSolv workshop, CP02*, Ithaca, NY, 2002. (Cited on page 33.)

[**Meisels and Zivan, 2003**] Amnon Meisels, and Roie Zivan. Asynchronous forward-checking for distributed CSPs. In W. Zhang, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003. (Cited on pages 33 and 46.)

[**Meisels and Zivan, 2007**] Amnon Meisels, and Roie Zivan. Asynchronous forward-checking for DisCSPs. *Constraints*, 12(1):131–150, 2007. (Cited on pages 34, 46, 52, 54, 62, 78, 116, and 121.)

[**Meisels *et al.*, 1997**] Amnon Meisels, Solomon Eyal Shimony, and Gadi Solotorevsky. Bayes networks for estimating the number of solutions to a csp. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, AAAI'97/IAAI'97, pages 179–184. AAAI Press, 1997. (Cited on page 28.)

[**Meisels *et al.*, 2002**] Amnon Meisels, I. Razgon, E. Kaplansky, and Roie Zivan. Comparing performance of distributed constraints processing algorithms. In *Proceeding of AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, 2002. (Cited on page 119.)

[**Minton *et al.*, 1992**] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, December 1992. (Cited on page 28.)

[**Modi *et al.*, 2005**] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005. (Cited on pages 116 and 121.)

[**Mohr and Henderson, 1986**] Roger Mohr, and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, Mar 1986. (Cited on page 22.)

[**Montanari, 1974**] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(0):95–132, 1974. (Cited on pages 8, 9, and 21.)

[**Nadel and Lin, 1991**] Bernard A. Nadel, and Jiang Lin. Automobile transmission design as a constraint satisfaction problem: modelling the kinematic level. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, 5:137–171, 1991. (Cited on page 8.)

[**Nadel, 1990**] Bernard A. Nadel. *Some Applications of the Constraint Satisfaction Problem*. Number 8 in CSC (Wayne State University, Department of Computer Science). Wayne State University, Department of Computer Science, 1990. (Cited on page 8.)

[**Nguyen *et al.*, 2004**] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic distributed backjumping. In *Proceeding of 5th workshop on DCR'04*, Toronto, 2004. (Cited on pages 52 and 59.)

[**Nudel, 1983**] Bernard Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21(1-2):135–178, mar 1983. (Cited on page 26.)

[**Petcu and Faltings, 2004**] Adrian Petcu, and Boi Faltings. A value ordering heuristic for distributed resource allocation. In *Proceeding of Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, CSCLP'04, pages 86–97, Feb 2004. (Cited on pages 1, 28, 30, and 115.)

[**Prosser *et al.*, 1992**] Patrick Prosser, Chris Conway, and Claude Muller. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83, oct 1992. (Cited on page 30.)

[**Prosser, 1993**] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993. (Cited on pages 8, 17, and 34.)

[**Purdom, 1983**] Paul Walton Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21(1-2):117–133, mar 1983. (Cited on page 25.)

[**Sabin and Freuder, 1994**] Daniel Sabin, and Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *CP'94*, pages 10–20, 1994. (Cited on pages 8, 23, 78, and 79.)

[**Sen and Durfee, 1995**] Sandip Sen, and Edmund H Durfee. Unsupervised surrogate agents and search bias change in flexible distributed scheduling. In *Proceedings of the First International Conference on MultiAgent Systems*, ICMAS'95, pages 336–343, 1995. (Cited on pages 11 and 12.)

[**Silaghi and Faltings, 2005**] Marius-Calin Silaghi, and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:25–53, 2005. (Cited on pages 33 and 68.)

[**Silaghi *et al.*, 2001a**] Marius-Calin Silaghi, Djamila Sam-Haroud, M. Calisti, and Boi Faltings. Generalized english auctions by relaxation in dynamic distributed CSPs with private constraints. In *Proceedings of the IJCAI'01 workshop on Distributed Constraint Reasoning*, DCR'11, pages 45–54, 2001. (Cited on pages 89 and 90.)

[**Silaghi *et al.*, 2001b**] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Consistency maintenance for abt. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, CP'01, pages 271–285, Paphos, Cyprus, 2001. (Cited on pages 43 and 78.)

[**Silaghi *et al.*, 2001c**] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report LIA-REPORT-2001-008, 2001. Technical report. (Cited on pages 42 and 90.)

[**Silaghi *et al.*, 2001d**] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Polynomial space and complete multiply asynchronous search with abstractions. In *Proceedings of the IJCAI'2001 Workshop on Distributed Constraint Reasoning*, DCR'11, pages 17–32, 2001. (Cited on page 40.)

[**Silaghi, 2006**] Marius-Calin Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. In *DCR workshop, AAMAS-06*, 2006. (Cited on pages 30, 42, 52, 85, and 90.)

[**Smith and Grant, 1998**] Barbara M. Smith, and Stuart A. Grant. Trying harder to fail first. In *ECAI*, pages 249–253, 1998. (Cited on page 26.)

[**Smith, 1999**] Barbara M. Smith. The Brélaz heuristic and optimal static orderings. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, CP'99, pages 405–418, London, UK, UK, 1999. Springer-Verlag. (Cited on page 26.)

[**Stallman and Sussman, 1977**] Richard M. Stallman, and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977. (Cited on page 19.)

[**Stefik, 1981**] Mark Stefik. Planning with constraints (molgen: Part 1). *Artificial Intelligence*, 16(2):111–139, 1981. (Cited on page 8.)

[**Sultanik *et al.*, 2008**] Evan A. Sultanik, Robert N. Lass, and William C. Regli. Dcopolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, AAMAS'08, pages 1667–1668, Estoril, Portugal, 2008. (Cited on page 116.)

[**Van Hentenryck *et al.*, 1992**] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, oct 1992. (Cited on page 22.)

[**Vernooy and Havens, 1999**] Matt Vernooy, and William S. Havens. An examination of probabilistic value-ordering heuristics. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence*, AI'99, pages 340–352, London, UK, UK, 1999. Springer-Verlag. (Cited on page 28.)

[**Wahbi *et al.*, 2011**] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El-Houssine Bouyakhf. DisChoco 2: A platform for distributed constraint reasoning. In *Proceedings of the IJCAI'11 workshop on Distributed Constraint Reasoning*, pages 112–121, Barcelona, Catalonia, Spain, 2011. (Cited on pages 52, 70, 85, and 100.)

[**Wahbi *et al.*, 2012**] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. Nogood-Based Asynchronous Forward-Checking Algorithms. Technical report, LIRMM, April 2012. (Cited on pages 116 and 121.)

[**Wallace and Freuder, 2002**] Richard J. Wallace, and Eugene C. Freuder. Constraint-based multi-agent meeting scheduling: effects of agent heterogeneity on performance and privacy loss. In *Proceeding of the 3rd workshop on distributed constrait reasoning*, DCR'02, pages 176–182, 2002. (Cited on pages 1, 12, 28, 30, 31, 56, 74, and 115.)

[**Yeoh *et al.*, 2008**] W. Yeoh, A. Felner, and S. Koenig. Bnb-adopt: an asynchronous branch-and-bound dcop algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, AAMAS'08, pages 591–598, Estoril, Portugal, 2008. (Cited on pages 116 and 121.)

[**Yokoo and Hirayama, 1995**] Makoto Yokoo, and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*. MIT Press, 1995. (Cited on pages 33 and 121.)

[**Yokoo *et al.*, 1992**] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th IEEE Int'l Conf. Distributed Computing Systems*, pages 614–621, 1992. (Cited on pages 33, 37, 116, and 121.)

[**Yokoo *et al.*, 1998**] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, September 1998. (Cited on pages 2, 29, 33, 40, 52, 90, and 115.)

[**Yokoo, 1995**] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceeding of CP*, pages 88–102, 1995. (Cited on pages 33, 42, and 90.)

[**Yokoo, 2000a**] Makoto Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000. (Cited on pages 2, 29, 37, and 115.)

[**Yokoo, 2000b**] Makoto Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer-Verlag, London, UK, 2000. (Cited on pages 29, 34, 62, and 76.)

[**Zabih, 1990**] Ramin Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the eighth National conference on Artificial intelligence*, volume 1 of *AAAI'90*, pages 46–51. AAAI Press, 1990. (Cited on page 24.)

[**Zivan and Meisels, 2003**] Roie Zivan, and Amnon Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proceedings of the First European Workshop on Multi-Agent Systems*, EUMAS'03, 2003. (Cited on pages 34, 59, and 62.)

[**Zivan and Meisels, 2006a**] Roie Zivan, and Amnon Meisels. Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints*, 11(2-3):179–197, 2006. (Cited on pages 42, 90, 101, 107, 108, and 109.)

[**Zivan and Meisels, 2006b**] Roie Zivan, and Amnon Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006. (Cited on pages 52, 70, 85, 101, and 118.)

[**Zivan *et al.*, 2009**] Roie Zivan, Moshe Zazone, and Amnon Meisels. Min-domain retroactive ordering for asynchronous backtracking. *Constraints*, 14(2):177–198, 2009. (Cited on pages 42, 43, 90, 101, 102, 107, 108, 109, 110, and 112.)

List of Figures

List of Tables

List of algorithms

Algorithmes de résolution et heuristiques d'ordonnancement pour les problèmes de satisfaction de contraintes distribués

Les problèmes de satisfaction de contraintes distribués (DisCSP) permettent de formaliser divers problèmes qui se situent dans l'intelligence artificielle distribuée. Ces problèmes consistent à trouver une combinaison cohérente des actions de plusieurs agents. Durant cette thèse nous avons apporté plusieurs contributions dans le cadre des DisCSPs. Premièrement, nous avons proposé le Nogood-Based Asynchronous Forward-Checking (AFC-ng). Dans AFC-ng, les agents utilisent les nogoods pour justifier chaque suppression d'une valeur du domaine de chaque variable. Outre l'utilisation des nogoods, plusieurs backtracks simultanés venant de différents agents vers différentes destinations sont autorisés. En deuxième lieu, nous exploitons les caractéristiques intrinsèques du réseau de contraintes pour exécuter plusieurs processus de recherche AFC-ng d'une manière asynchrone à travers chaque branche du pseudo-arborescence obtenu à partir du graphe de contraintes dans l'algorithme Asynchronous Forward-Checking Tree (AFC-tree). Puis, nous proposons deux nouveaux algorithmes de recherche synchrones basés sur le même mécanisme que notre AFC-ng. Cependant, au lieu de maintenir le forward checking sur les agents non encore instanciés, nous proposons de maintenir la consistance d'arc. Ensuite, nous proposons Agile Asynchronous Backtracking (Agile-ABT), un algorithme de changement d'ordre asynchrone qui s'affranchit des restrictions habituelles des algorithmes de backtracking asynchrone. Puis, nous avons proposé une nouvelle méthode correcte pour comparer les ordres dans ABT_DO-Retro. Cette méthode détermine l'ordre le plus pertinent en comparant les indices des agents dès que les compteurs d'une position donnée dans le timestamp sont égaux. Finalement, nous présentons une nouvelle version entièrement restructurée de la plateforme DisChoco pour résoudre les problèmes de satisfaction et d'optimisation de contraintes distribués.

Mots-clés *L'intelligence artificielle distribuée, les problèmes de satisfaction de contraintes distribués (DisCSP), la résolution distribuée, la maintenance de la consistance d'arc, les heuristiques ordonnancement, DisChoco.*

Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems

Distributed Constraint Satisfaction Problems (DisCSP) is a general framework for solving distributed problems. DisCSP have a wide range of applications in multi-agent coordination. In this thesis, we extend the state of the art in solving the DisCSPs by proposing several algorithms. Firstly, we propose the Nogood-Based Asynchronous Forward Checking (AFC-ng), an algorithm based on Asynchronous Forward Checking (AFC). However, instead of using the shortest inconsistent partial assignments, AFC-ng uses nogoods as justifications of value removals. Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. Then, we propose the Asynchronous Forward-Checking Tree (AFC-tree). In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. Next, we propose to maintain arc consistency asynchronously on the future agents instead of only maintaining forward checking. Two new synchronous search algorithms that maintain arc consistency asynchronously (MACA) are presented. After that, we developed the Agile Asynchronous Backtracking (Agile-ABT), an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing before the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Next, we present a corrigendum of the protocol designed for establishing the priority between orders in the asynchronous backtracking algorithm with dynamic ordering using retroactive heuristics (ABT_DO-Retro). Finally, the new version of the DisChoco open-source platform for solving distributed constraint reasoning problems is described. The new version is a complete redesign of the DisChoco platform. DisChoco 2.0 is an open source Java library which aims at implementing distributed constraint reasoning algorithms.

Keywords *Distributed Artificial Intelligence, Distributed Constraint Satisfaction (DisCSP), Distributed Solving, Maintaining Arc Consistency, Reordering, DisChoco.*

LIRMM — 161, rue Ada — 34095 Montpellier cedex 5 — France