MONOGRAFIES DE L'INSTITUT D´INVESTIGACIÓ EN INTEL·LIGÈNCIA ARTIFICIAL

Distributed Constraint Satisfaction

Ismel Brito Rodríguez

Consell Superior d´Investigacions Científiques

MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 30

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

Distributed Constraint Satisfaction

Ismel Brito Rodríguez

Foreword by Pedro Meseguer

2007 Consell Superior d'Investigacions Científiques
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Pedro Meseguer
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Ismel Brito Rodríguez
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

NIPO: 653-07-093-5
ISBN: 978-84-00-08572-8
Dip. Legal: B.50669-2007

Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

A mi padres, que hacen suyas todas mis ilusiones.

A mi familia y amigos que esperan, con sosiego, encontrarse en esta dedicatoria.

'All children, except one, grow up.'

Peter Pan by James Matthew Barrie

Contents

List of Figures

Foreword

Distributed constraint satisfaction (*DisCSP*) is a new model of constraint reasoning, where problem elements are distributed among agents and cannot be grouped into a central one for some reasons. A solution, as in the classical case, must satisfy all constraints. In a distributed setting, a solution can be achieved by message passing among agents.

This book deals with algorithmic approaches for *DisCSP* solving. The different types of distributed algorithms, synchronous, asynchronous and hybrids, are analyzed in the *DisCSP* context. Taking the pioneering work of Makoto Yokoo on asynchronous backtracking as reference algorithm, a number of extensions from a common core are presented here. Among them, we emphasize the new algorithm which does not add new links among agents during the solving process. The extension to non-binary constraints and its practical implementation are also addressed.

Good part of this work is devoted to privacy, one of the main motivations for distributed constraint reasoning. A new model of partially known constraints is proposed, to express those constraint problems which appear to be naturally distributed. In this model, new algorithms to preserve privacy are proposed. Looking further to protect privacy, lies are allowed among agents, providing they will tell the truth in finite time afterwards. Unsurprisingly, enforcing privacy causes efficiency in the solving process to decrease.

In addition to the conventional evaluation on distributed random problems, this work considers two applications: distributed meeting scheduling and distributed stable matching. Since these problems have been widely studied in their centralized versions, when possible centralized approaches have been extended to the distributed case, and compared with generic distributed approaches, looking for the best solving strategy for these kind of problems.

Bellaterra, June 2007

Pedro Meseguer
Researcher of IIIA-CSIC

Acknowledgements

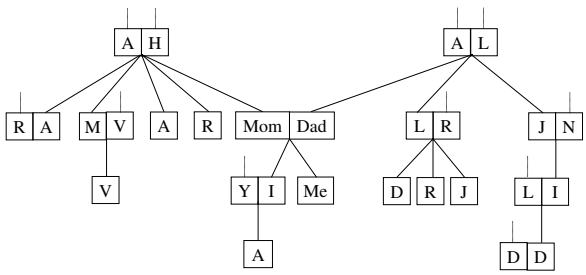
The last five years at the IIIA have been wonderful; I have enjoyed the support and space needed to grow intellectually and to be able to write this thesis. My colleagues at the Institute have been of great help on this journey, which started on 2/22/2002. I also thank the administrative staff for their support. Above all others, I would like to thank the Director of the Institute, Francesc Esteva.

This thesis owes, in great measure, its existence to Pedro Meseguer, my thesis Advisor. Pedro has been a father figure for me, with all the ups and downs that this may entail. His help has been so instrumental that simply thanking him in words would seem petty and yet that is the only manner in which I can give form to my thankfulness.

I would like to devote my sincere thanks to Javier for giving me a hand which kept me from working at McDonald.

I would like to extend my gratitude to my colleagues with whom I have co-authored papers (Christian Bessiere, Arnold Maestre, Amnon Meisels and Roie Zivan) for their valuable assistance and contributions in the development of this thesis.

Of course, I am grateful to my parents, brother and four grandparents because they have done everything to make my life easier. There is a saying that says that a tree planted in good soil grows tall and strong. I know that my strength comes not from the soil, but from the people that share this tree with me. This is a good time to thank them.



It will be an unforgivable mistake to not thank the following people: Keith, Dasi+J.P., and Wilber (for all the good and bad times we have had and will have together); Yordy, Nancy+Villa, Dayma±Luis, Mario+Lipsy, Héctor, Mily+Ale... (for always being close to me); Mario+Teresa/Dani+Ana (for trying to make me a bit more Spanish/Catalonian).

Ismel Brito Rodriguez
Terrassa, Barcelona
May 10, 2007

Abstract

In recent years, the Artificial Intelligence community has shown an increasing interest in solving distributed problems using the agents paradigm. When multiple agents in a shared environment pursue a common goal, there are usually constraints among the possible actions of these agents. Finding a consistent combination of actions that satisfies agents' constraints can be seen as a Distributed Constraint Satisfaction problem (*DisCSP*). Various application problems in multi-agent systems can be formalized as *DisCSPs*.

This thesis is dedicated to the development of distributed complete algorithms for solving *DisCSP*. In it, we study three types of algorithms: synchronous, asynchronous and hybrid. We evaluate the proposed algorithms in two dimensions: efficiency and privacy. Regarding efficiency, we propose new distributed algorithms which mainly are faster and consume less network resources than state-of-the-art algorithms. Regarding privacy, we propose novel algorithms to enforce the privacy of the local information held by agents without using cryptographic tools. The main ideas that we have developed in this thesis are:

Synchronous Algorithms: The use of variable reordering heuristics for constraint satisfaction problems has been shown to be a powerful strategy in order to improve efficiency. Inspired in this idea, we present two approximations of the popular minimum-domain heuristic for dynamic variable reordering.

Asynchronous Algorithms: We present a basic kernel for grouping asynchronous backtracking algorithms. By implementing the condition for termination in this kernel, we obtain four asynchronous algorithms. One of these algorithms does not add links between agents not sharing constraints, which can be useful for solving problems where privacy is the main concern.

Hybrid Algorithms: We present a novel algorithm which combines synchronous and asynchronous elements. This algorithm outperforms the reference asynchronous algorithm.

Non-binary Constraints: Although most of state-of-the-art methods for *DisCSP* assume that every constraint involves two variables, they can be extended to handle constraints involving more than two variables. We present new versions of existing algorithms to deal with non-binary constraints, including the addition of redundant constraint projections.

Assignment Privacy: We propose an asynchronous algorithm that allows agents to maintain their variable assignments private during problem resolution.

Constrain Privacy: We present the Partially Known Constraint model (*PKC*), a new *DisCSP* model in which constraints are kept private and are

only partially known to agents. We propose two algorithms for solving *DisCSP* expressed under the *PKC* model. Both algorithms also consider assignment privacy.

Enforcing Privacy with Lies: We present a novel algorithm that further enforces constraint privacy. This is algorithm is based on the idea that agents may lie. It requires a single extra condition: if an agent lies, it has to tell the truth in finite time afterwards.

Applications: We consider naturally distributed constraint problems which have a clear motivation to be tried with distributed techniques: *Meeting Scheduling* and *Stable Matching problems*. For each these problems we present distributed versions. Regarding *Stable Matching problems*, we consider two well-known problems: *Stable Marriage* and *Stable Roommates* problems. We propose ways to solve these problems while keeping personal preferences private.

All proposed algorithms in this thesis have been implemented, evaluated and formally proven to be correct, complete and terminate.

Chapter 1

Introduction

This thesis addresses search algorithms for Distributed Constraint Satisfaction problems. Distributed Constraint Satisfaction is a novel research topic related to two well-known research areas: Constraint Satisfaction and Distributed Algorithms. Constraint Satisfaction is the process of finding a solution to a Constraint Satisfaction problem (*CSP*). A *CSP* consists of a set of *variables*, each one taking a value in *finite domain*. Values are related by *constraints* that impose restriction to the values that variables can take. A *CSP solution* is an evaluation of these variables that satisfies all constraints.

A Distributed Constraint Satisfaction problem (*DisCSP*) is, thus, a *CSP* whose variables and/or constraints are geographically distributed among communicating agents[1] and cannot be solved by using the centralized approach.[2] Analogously to *CSP*, a *DisCSP solution* is an assignment of values to variables which satisfies every constraint. In an algorithm for *DisCSP*, agents cooperate and exchange messages in order to find a solution.

A *distributed algorithm* can be classified in two main classes: *synchronous* and *asynchronous*. In between these two classes are *hybrid* algorithms, which combine elements of both algorithm type. In general, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active agent. In an asynchronous algorithm every agent is active at any time, and they do not have to wait for any event. In a hybrid algorithm, an agent may be required to wait for some special event, but not for every event. In this thesis we consider these three different classes of distributed algorithms for *DisCSP*.

Search is one of the common approaches for *CSP* as well as for *DisCSP*. A *search algorithm* consists in searching a solution within the search space defined by all possible solutions to the problem. A search method is *complete* if the

[1] By agent we mean a software and/or hardware component capable of acting exactingly in order to accomplish tasks on behalf of its user [Nwana and Ndumu, 1997].

[2] By *centralized* we mean single processor, as opposed to *distributed*.

exploration of the search space is systematic and it is conducted until a solution is found or the absence of a solution is proven.

1.1 Motivation

Distributed Constraint Satisfaction is motivated by the existence of naturally distributed *CSP*, for which it is impossible or undesirable to gather the whole problem knowledge into a single agent and to solve it using the centralized approach. There are several reasons for that. The cost of collecting all information into a single agent could be taxing. This includes not only communication costs, but also the cost of translating the problem knowledge into a common format, which could be prohibitive for some applications. Furthermore, gathering all information into a single agent implies that this agent knows every detail about the problem, which could be undesirable for security or privacy reasons [Yokoo et al., 2002].

Distributed *CSPs* can be found in many real domains such as scheduling, planning and as part of many coordination processes in multi-agent systems. This thesis contributes to the development of *DisCSP* solving methods according to two issues: efficiency and privacy.

Distributed Constraint Satisfaction is an NP-complete task and hence all *DisCSP* algorithms have, in the worst case, exponential time in the number of variable of the problem. In such context, the development of algorithms to solve *DisCSPs* as efficient as possible is necessary. This intractability of *DisCSP* has motivated the part of our work. We present new approaches and heuristics in order to improve the efficiency of some state-of-the-art algorithms.

As mentioned above privacy is one of the main motivations to solve distributed *CSP* in a distributed form. *Privacy* in *DisCSP* is concerned with the desire of agents to conceal their information about the problem. Most of the existing algorithms for *DisCSP* were conceived without taking into account privacy issues (their agents give constraints and exchange value freely). This makes them to be unsuitable to solve naturally distributed problems where privacy is the main concern. Examples of this kind of problems are: Stable Marriage and Stable Roommate problems [Gusfield and Irving, 1989] where agents often want to keep their personal preferences private. There exists two main approaches: those that use cryptographic techniques [Silaghi and Mitra, 2004, Yokoo et al., 2005, Nissim and Zivan, 2005] and those that enforce privacy by different strategies but excluding cryptography [Silaghi, 2002, Brito and Meseguer, 2003, Brito and Meseguer, 2005b, Zivan and Meisels, 2005a]. For many applications the use of secure algorithms based on cryptographic methods is costly and difficult to implement. In our work we are concerned with *DisCSP* algorithms that leak less information that existing ones. We present new *DisCSP* algorithms that achieve a higher degree of privacy than existing approaches.

1.2 Scope and Orientation

The boundaries of this work are established by the following decisions:

- *Practical Application.* This work is oriented to further extend the applicability of *DisCSP* algorithms to solve naturally distributed problems. Our contributions are new approaches and heuristics that improve the performance of existing algorithms according to two issues: efficiency, in terms of computation cost, and privacy.

- *General Distributed Constraint Solving.* Except for the problems considered in Part "Applications", all algorithms and ideas that we propose in this thesis are applicable for solving any *DisCSP* whose agents hold only one variable. The extension of proposed approaches and heuristics for naturally handling multi-variable variants could be part of further research.

- *Complete Search.* A different approach to avoid the computationally intractability of Distributed Constraint Satisfaction uses incomplete search schemas, and they do not guarantee to find a solution to a solvable instance. In our work, however, we are concerned with complete search algorithms with polynomial space complexity.

- *Empirical evaluation.* Because of the practical orientation of our work and the intractability of *DisCSP*, the assessment of our contributions is mainly supported by empirical methods. In our experiments, we have used a set of benchmarks widely used in the *CSP* community.

1.3 Contributions

In the following we give the main contributions of this thesis:

- *A Family of Asynchronous Backtracking Algorithms.* We present a basic kernel for grouping asynchronous backtracking algorithms. We believe that this characterization of asynchronous backtracking will help better understand these non-trivial mechanisms. By implementing the condition for termination in this kernel, we obtain four asynchronous algorithms. One of these approaches does not add links between agents not sharing constraints, which can be useful for solving problems where privacy is the main concern. These ideas are gathered in:

  – C. Bessière, A. Maestre, I. Brito, P. Meseguer. *Asynchronous Backtracking without Adding Links: a New Member to ABT Family.* **Artificial Intelligence**. Volume 161, Issues 1-2. pp. 7-24. January, 2005.

- *Synchronous Backtracking.* The use of variable reordering heuristics for constraint satisfaction problems has been shown to be a powerful strategy

in order to improve efficiency. Inspired in this idea, we present two approximations of the popular minimum-domain heuristic for dynamic variable reordering.

  - I. Brito. *Synchronous, Asynchronous and Hybrid Algorithms for DisCSP*. Proceeding of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-2004). **Lecture Notes in Computer Science**, Volume 3258, p. 791, Jan 2004.

- *Hybrid Algorithms.* We present a novel algorithm which combines synchronous and asynchronous elements. This algorithm outperforms the reference asynchronous backtracking algorithm. A comparison about synchronous, asynchronous and hybrid algorithms as well as variable reordering heuristics for synchronous ones can be found in:

  - I. Brito, P. Meseguer. *Synchronous, Asynchronous and Hybrid Algorithms for DisCSP*. **Fifth International Workshop on Distributed Constraint Reasoning at the Tenth International Conference on Principles and Practice of Constraint Programming (CP-2004)**. Toronto, Canada. September, 2004.
  - I. Brito, F. Herrero, P. Meseguer. *On the Evaluation of DisCSP Algorithms*. **Fifth International Workshop on Distributed Constraint Reasoning at the Tenth International Conference on Principles and Practice of Constraint Programming (CP-2004)**. Toronto, Canada. September, 2004.

- *Non-binary Constraints.* Although most of state-of-the-art methods for *DisCSP* assume that every constraint involves two variables, they can be extended to handle constraints involving more than two variables. We present new versions of existing algorithms to deal with non-binary constraints, including the addition of redundant constraint projections.

  This ideas were published in:

  - I. Brito, P. Meseguer. *Asynchronous Backtracking Algorithms for Non-binary DisCSP*. **Workshop on Distributed Constraint Satisfaction Problems at the 17th European Conference on Artificial Intelligence (ECAI-2006)**, Riva del Garda, 2006.

- *Assignment Privacy.* We propose an asynchronous algorithm that allows agents to maintain their variable assignments private during problem resolution. This algorithm is based on the idea that agents exchange sets of consistent values instead of their own assignments.

- *Constraint Privacy.* We present the Partially Known Constraint model (*PKC*), a new *DisCSP* model in which constraints are kept private and are only partially known to agents. We propose two algorithms to solve *DisCSPs* expressed under the *PKC* model. These algorithms also preserve

agents' assignments. The ideas related to assignment and/or constraint can be found in:

  - I. Brito, P. Meseguer. *Distributed Forward Checking*. Proceeding of the Night International Conference on Principles and Practice of Constraint Programming, CP-2003. **Lecture Notes in Computer Science**, Volume 2833, pp. 801 - 806, November 2003.

- *Enforcing Privacy with Lies*. We present a novel algorithm to further enforce constraint privacy. This is algorithm is based on the idea that agents may lie. It requires a single extra condition: if an agent lies, it has to tell the truth in finite time afterwards.

  - I. Brito, P. Meseguer. *Distributed Forward Checking May Lie for Privacy*, **Recent Advances in Constraints, Lecture Note in Artificial Intelligence**. Volume 4651, 2007.

- *Applications*. We consider naturally distributed problems which have a clear motivation to be tried with distributed techniques. We examine some *DisCSP* algorithms for solving several versions of two well-known Stable Marriage problems [Gusfield and Irving, 1989]: Stable Marriage and Stable Roommates. We propose a way to resolve these problems while keeping personal preference private. Four publications develop privacy issues related to several versions of the Stable Matching:

  - I. Brito, P. Meseguer. *The Distributed Stable Marriage Problem with Ties and Incomplete Lists*. **Workshop on Distributed Constraint Satisfaction Problems at the 17th European Conference on Artificial Intelligence (ECAI-2006)**, Riva del Garda, 2006.
  - I. Brito, P. Meseguer. *Distributed Stable Matching Problems with Ties and Incomplete Lists*. Proceeding of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-2006). **Lecture Notes in Computer Science**, Volume 4204, pp. 675-680, Nantes, 2006.
  - I. Brito, P. Meseguer. *The Distributed Stable Marriage Problem*. **Sixth International Workshop on Distributed Constraint Reasoning at the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-2005)**. Edinburgh, Scotland, 30 July - 5 August, 2005.
  - I. Brito, P. Meseguer. *Distributed Stable Matching Problems*. Proceeding of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP-2005). **Lecture Notes in Computer Science**. Volume 3709. pp. 152-166. Sitges, Spain. September, 2005.

1.4 Thesis Structure

This document is divided in five Parts and one Appendix. Parts II, III and IV contain the contributions of our work.

- *Part I Background*: This part includes two Chapters. Chapter 2 and 3 are an overview of Constraint Satisfaction and Distributed Constraint Satisfaction frameworks, respectively. In both, we formally define the corresponding problems and briefly describe the main solving algorithms that will useful to understand the following Chapters.

- *Part II Approaches*: This part has four chapters. In Chapter 4, we study synchronous backtracking algorithms and present two approaches for variable ordering. In Chapter 5 we consider four asynchronous backtracking algorithms. In Chapter 6 we present a novel hybrid algorithm, which we evaluate against synchronous and asynchronous approaches. In Chapter 7 we present several distributed algorithms to deal with non-binary constraints and consider the idea of add constraints projections in order to speed up the search.

- *Part III Privacy in DisCSP*: This part contains two chapters. In Chapter 8 we study three types of privacy in *DisCSP*: domain, assignment and constraint privacy. We propose three asynchronous algorithms for enforcing assignment and/or constraint privacy. In Chapter 9, we further enforce constraint privacy in of the algorithms studied in Chapter 8 by allowing agents to lie.

- *Part IV Applications*: In this part we resolve two problems with privacy requirements: Meeting Scheduling (Chapter 10) and Stable Matching problems (Chapter 11).

- *Part V Conclusions and Appendix*: This part contain two Chapters: Chapter 12 where we present the conclusions and further research of our work and Appendix A, where we analyze centralized and distributed specialized algorithms to Stable Matching problems with privacy requirements.

1.5 Abbreviations

The abbreviations used in this thesis are summarized next:

CSP	Constraint Satisfaction Problem
DisCSP	Distributed Constraint Satisfaction Problem
BT	Chronological Backtracking algorithm
DepDB	Dependency-directed Backtracking algorithm
GBJ	Graph-based Backjumping algorithm
CBJ	Conflict-based Backjumping algorithm
FC	Forward Checking algorithm
GLS	Generic Local Search algorithm
BO	Break-out algorithm
MAC	Maintaining Arc-Consistency algorithm
DisMS	Distributed Meeting Scheduling problem
SensorDCSP	Sensor-mobile problem
SBT	Synchronous Chronological Backtracking algorithm
SCBJ	Synchronous Conflict Backjumping algorithm
ABT	Asynchronous Backtracking algorithm
AWC	Asynchronous Weak-commitment algorithm
DIBT	Asynchronous Graph-based Backjumping algorithm
AAS	Asynchronous Aggregation Search
DMAC-ABT	Distributed Maintaining Asynchronous Consistency algorithm
ABT-DO	Dynamic Ordering Asynchronous Backtracking algorithm
ConBT	Concurrent Backtracking Search
ConDB	Concurrent Dynamic Backtracking
DisBO	Distributed Break-out algorithm
amd1	An Approach of the Minimum-Domain heuristic
amd2	An Approach of the Minimum-Domain heuristic
$SCBJ_{amd1}$	*SCBJ* plus *amd1*
$SCBJ_{amd2}$	*SCBJ* plus *amd2*
ABT_{kernel}	Kernel of *ABT* family
ABT_{all}	*ABT* with all potentially new links added in advance
ABT_{temp}	*ABT* with temporary new links
ABT_{not}	*ABT* without adding new links
ABT_{hyb}	*ABT*-like algorithm with synchronous and asynchronous elements
ABT_{proj}	Non-binary *ABT* with constraint projections
$SCBJ_{proj}$	Non-binary *SCBJ* with constraint projections
ABT_1	*ABT* with the single phase strategy
ABT_2	*ABT* with the two-phase strategy
$DisFC_1$	*DisFC* with the single phase strategy
$DisFC_2$	*DisFC* with the two-phase strategy
$DisFC_{lies}$	$DisFC_1$ with lies
SM	Stable Marriage problem
EGS	Extended Gale-Shapley algorithm
SMI	Stable Marriage problem with Incomplete Lists
DisEGS	Distributed Extended Gale-Shapley algorithm
DisSM	Distributed Stable Marriage problem
DisSMI	Distributed Stable Marriage problem with Incomplete Lists

SR	Stable Roommates problem
SRI	Stable Roommates problem with Incomplete Lists
DisSR	Distributed Stable Roommates problem
DisSRI	Distributed Stable Roommates problem with Incomplete Lists
SMT	Stable Marriage problem with Ties
SMTI	Stable Marriage problem with Ties and Incomplete Lists
DisSMT	Distributed Stable Marriage problem with Ties
DisSMTI	Distributed Stable Marriage problem with Ties and Incomplete Lists
SRT	Stable Roommates problem with Ties
SRTI	Stable Roommates problem with Ties and Incomplete Lists
DisSRT	Distributed Stable Roommates problem with Ties
DisSRTI	Distributed Stable Roommates problem with Ties and Incomplete Lists
DisFC-SM	Distributed Forward Checking algorithm for finding Stable Matchings
MS	Meeting Scheduling problem
DisMS	Distributed Meeting Scheduling problem

Part I

Background

Chapter 2

Constraint Satisfaction

Constraint Satisfaction is a useful framework for modeling and solving many combinatorial problems. It is central in multiple Artificial Intelligence (AI) problems, especially in many areas related to scheduling, logistics and planning. In this chapter, we formally define the Constraint Satisfaction problem, in short *CSP*, (Section 2.1) and provide some examples (Section 2.2). We also discuss the main generic methods for *CSPs* (Section 2.3).

2.1 What is a Constraint Satisfaction Problem?

A Constraint Satisfaction Problem (*CSP*) involves a finite set of variables, each one taking a value in a finite domain. Values are related by constraints that impose restrictions on the value combinations that subsets of variables can take. The following is the formal definition of a finite *CSP*.

Definition 2.1.1. A finite Constraint Satisfaction Problem is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables;
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a collection of finite and discrete domains; $D(x_i)$ is the initial set of possible values for x_i;
- $\mathcal{C}$ is a set of constraints among variables. A constraint c_i in the ordered set of variables $var(c_i) = (x_{i_1}, \ldots, x_{i_{r(i)}})$ specifies the relation $rel(c_i)$ of the *permitted* combinations of values for the variables in $var(c_i)$. An element of $rel(c_i)$ is a tuple $(v_{i_1}, \ldots, v_{i_{r(i)}})$, $v_i \in D(x_i)$. The arity of c_i is the number of variables involved by c_i, that is, $|var(c_i)|$.

In the above definition, constraints are defined explicitly because they are given by the sets of permitted combinations of values. Conversely, a constraint can be defined implicity, in one of the following ways:

- mathematical expressions (e.g. $x_1 < x_2$)
- computing procedures (e.g. $isPrime?(x_1)$)
- a specific sematic (e.g. the all-different constraint, `all-diff`$(x_1, x_2, \ldots)$ which is equivalent to saying that the variables it involves must take different values)

Unless otherwise specified, in this document we assume that constraints are defined explicitly.

According to the arity, constraints can be classified in three types. The simplest type is the *unary constraint*, which restricts the values of a single variable. Every unary constraint can be eliminated by simply preprocessing the domain of the corresponding variable to remove any value that violates the constraint. A *binary constraints* just relates two variables. A binary constraint among variables x_i and x_j will be denoted by c_{ij}. A *non-binary constraint* includes three or more variables.

Definition 2.1.2. A binary *CSP* is a problem whose constraints are either unary or binary constraints.

A *CSP* can represented as a *constraint graph*. In the constraint graph associated to a binary *CSP*, a node represents a variable and a link between nodes represents a constraint between the corresponding variables.

Definition 2.1.3. A non-binary *CSP* is a problem with at least one non-binary constraint.

It is well known that a non-binary *CSP* can be translated into an equivalent binary *CSP*. Two general methods are known: the dual problem method [Dechter and Pearl, 1989] and the hidden variable method [Dechter, 1990]. However, both methods require the addition of new variables with exponentially large domains, which is usually seen as a serious drawback.

Solving a *CSP* is equivalent to assigning a value to each variable such that all constraints are satisfied. This is, a *CSP solution* is an assignment of values to variables which satisfies every constraint. In general, solving a *CSP* is NP-complete[1]. This property directly results from transforming *CSPs* to SAT, that is a satisfiability problem for propositional formulas in conjunctive normal form [Garey and Johnson, 1979]. In the next section we present some examples of *CSPs*.

2.2 Examples of *CSPs*

Constraint Satisfaction can be used to model and solve several problems such as: frequency assignment [Box, 1978], belief

[1]The complexity theory is clearly presented in [Garey and Johnson, 1979]. There, one can find the definitions of P, NP, NP-complete, NP-hard and P-space problems.

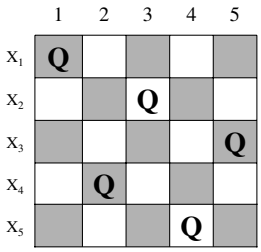


Figure 2.1: A solution to the 5-queens problem.

maintenance [Dechter and Dechter, 1988], planning and scheduling [Kautz and Selman, 1992], diagnostic reasoning [Geffner and Pearl, 1987]. Next, we describe two classical *CSP* examples: the n-queens and the graph-coloring problems.

2.2.1 N-queens Problem

A large number of problems can be modeled as a *CSP*. A typical example of *CSP* is the n-queens problem. In this problem, the objective is to place n queens on an $n \times n$ chessboard such that queens do not attack each other. One can formalize, for example, the 5-queens problem as a *CSP* as follows. There are 5 variables x_1, x_2,... ,x_5; each one corresponds to the position of a queen in each row. The domain of the variables is $\{1, 2, 3, 4, 5\}$. There exits a constraint between each pair of queens that forbids involved queens to be placed in the same column or diagonal line. A solution is a combination of values such that every constraint is satisfied. Figure 2.1 shows a solution for the 5-queens problem. The letter "Q" represent the positions of the queens for each row.

2.2.2 Graph-coloring Problem

Another example of *CSP* is the graph-coloring problem. In this problem, the goal is to color the nodes of a graph so that there is not a pair of linked nodes painted with the same color. Each node has a finite number of possible colors. 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 that this variable can take. There exits a constraint between each pair of linked variables that prohibits these variables to have the same color.

A practical application of the graph-coloring problem is the problem of coloring a map. In this situation, the goal is to color regions so that no two neighboring regions have the same color. Figure 2.2 shows an example of the map-coloring problem. In the following, we describe the basic solving methods for *CSP*.

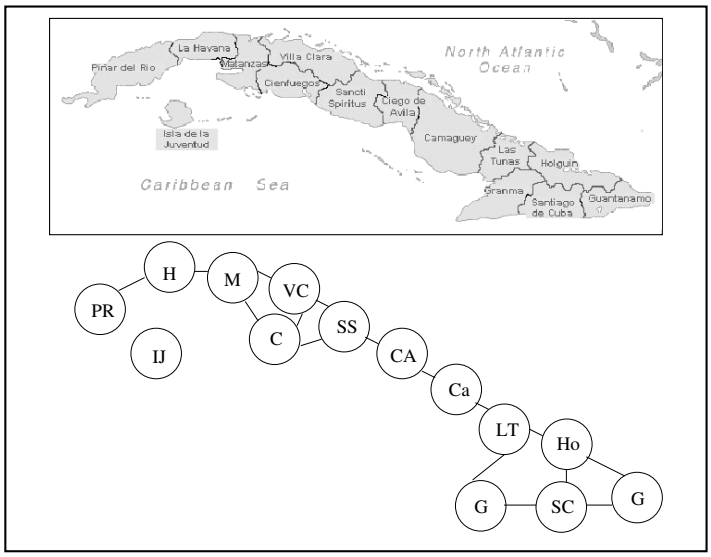


Figure 2.2: An example of the map-coloring problem. At the top, the provinces of Cuba. Coloring this map can be viewed as a *CSP*. The goal is to assign a color to each region so that no neighboring regions have the same color. At the bottom, the map-coloring problem represented as a constraint graph.

2.3 Algorithms for Solving *CSPs*

Existing algorithms for solving *CSP* can be divided into three types: *search*, *inference* and *hybrid* approaches. A search algorithm, broadly speaking, is an algorithm that returns a solution to the problem, usually after evaluating a number of possible solutions. A search algorithm can be *complete* or *incomplete*.

Definition 2.3.1. An algorithm is complete if and only if it guarantees to find a solution, if one exists, or prove that the problem is unsolvable, otherwise.

Definition 2.3.2. An algorithm is incomplete when it cannot guarantee that if it ends without finding a solution then the problem is unsolvable.

Inference methods translate a *CSP* P into a new one P' which is equivalent to the original (that is, if P is satisfiable, then P' has the same solutions as P, otherwise P' is also unsatisfiable). Presumably, P' is easier to solve than P. Inference methods also can be complete or incomplete. A complete inference method always finds a solution or detects unsatisfiability for any *CSP*. In contrast, an incomplete inference method may terminate without solving the problem. In that case, the additional use of a search algorithm is required. Hybrid algorithms result from combining search algorithms and incomplete inference methods.

2.3.1 Complete Search

The simplest complete algorithm is based on the generate-and-test paradigm. The algorithm tests, one by one, all the candidate solutions in the search space of the *CSP* until finding a solution or exhausting all of them.

Definition 2.3.3. The search space of a *CSP* is defined by the elements in the Cartesian product of the domains of the variables. Each one of these elements in search space is also called *state* and constitutes an hypothetical solution for the *CSP*.

This algorithm is very simple, and it has been proven inefficient. In contrast, the tree-search procedures with backtracking try to discard multiple non-solutions at once. A backtracking algorithm keeps a subset of variables and their values. This subset is called a *partial solution* since all variables within the subset satisfy every constraint. Initially, the partial solution is the empty set. The partial solution is expanded by adding new variables one by one, until all variables are included.

When a variable cannot find a consistent value with the previously assigned variables in the partial solution, backtracking is performed. *Backtracking* is the operation in which the current value of one of the variables that appears in the partial solution must be changed. After backtracking, the partial solution is updated and the search is resumed. There exits several backtracking-based algorithms. They differ in which variable, from the partial solution, is selected to change its value. Overviews of the classic backtracking-based algorithms appear in [Tang, 1993].

The simplest backtracking algorithm is *Chronological Backtracking* (BT) [Bitner and Reingold, 1975]. In BT, when a variable's domain becomes empty because all its values are inconsistent with previously assigned variables, the value of most recently variable added to the partial solution is changed. The partial solution that causes backtracking it is considered a *nogood*.

Definition 2.3.4. A nogood is an incompatible assignment of values to a subset of variables.

If the algorithm records all discovered nogoods it may avoid repeating elements of the search space that are not solutions to the problem because the same reasons. Chronological Backtracking appears in Figure 2.3. BT ends when the partial solution cannot be extended either because of all variables are included or because each value of the first variable is prohibited by a nogood. In the former case, the partial solution constitutes a solution to the problem while in the latter case, the problem has no a solution. BT is correct (i.e. a solution reported by the algorithm is a true solution), complete (i.e. if there is a solution, the algorithm finds it) and terminates.

Figure 2.4 presents one example of the execution of BT for solving the 4-queens problem. Initially, the partial solution is the empty set. The algorithm starts by assigning to x_1 the first value in x_1's domain. The assignment $x_1 = 1$ is added to the partial solution. Then, the algorithm ties values 1 and 2 (which are

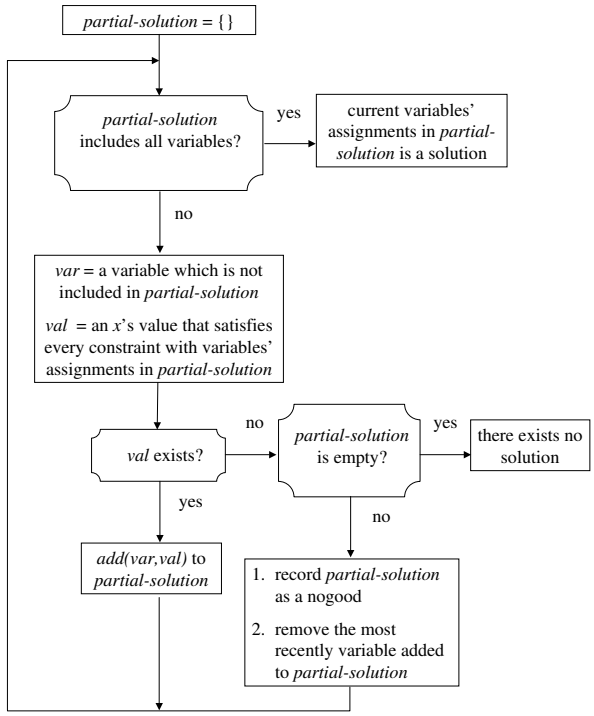


Figure 2.3: The Chronological Backtracking algorithm

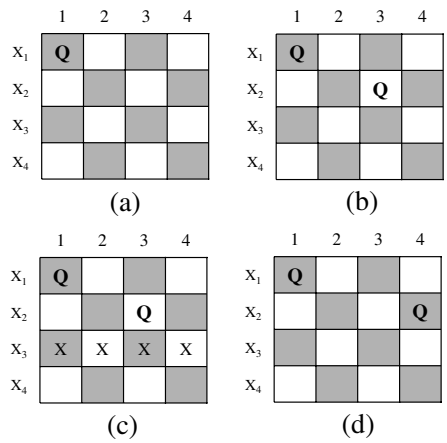


Figure 2.4: Example of the execution of the Chronological Backtracking algorithm

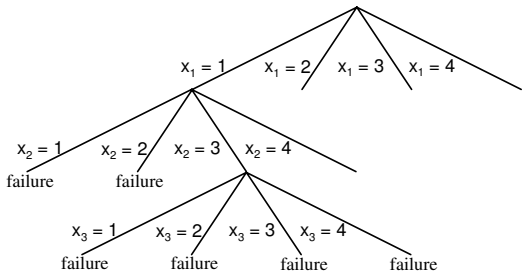


Figure 2.5: Tree Traversal of BT for example in Figure 2.4.

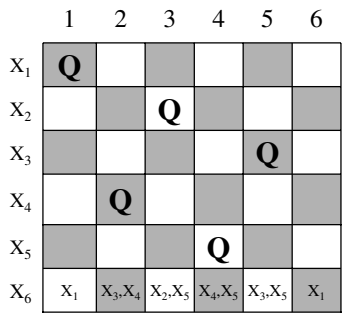


Figure 2.6: Example of the execution of the Dependency-directed algorithm

inconsistent with x_1s value) and finds that value 3 for the x_2 is valid with respect to the constraints with the variables in the partial solution (i.e. variable x_1). Variable x_2 is assigned to the third value and $x_2 = 3$ is included to the partial solution. This causes x_3 to not have a valid value. In this situation, x_2 has to change its value since it is the last variable added to the partial solution. Then, the algorithm assigns to variable x_2 the fourth value and adds this assignment to the partial solution. The algorithm continues until finding a solution.

Backtracking-based algorithms do a depth-first traversal on a search tree, which is defined by the problem variables and their domains. In the search tree corresponding to a *CSP*, each tree level corresponds to a variable and different tree nodes correspond to different assignment alternatives. Figure 2.5 shows three traversal done by BT for solving the previous example.

Another backtracking algorithm is *Dependency-directed Backtracking* [Stallman and Sussman, 1977], in short *DepDB*. The idea is to identify the variables causing failure upon backtracking. When a variable x_j cannot find a consistent value with the variables' values in the partial solution, the algorithm

backtracks not to the previous assigned variable like in *BT*, but to a variable x_i that may allow x_j to have at least one valid value.

Figure 2.6 shows an example of the algorithm execution in solving the 6-queens problem. Let us assume that the algorithm has assigned the first 5 variables in the ordering x_1, x_2, x_3, x_4 and x_5. The values of these variables are presented by shaded circles. When the algorithm tries to find a valid value for x_6, it discovers that no value exists; this causes the algorithm to backtrack. Variables that forbid each value of x_6 (i.e. culpable variables) appear in the sixth row. Since no value of x_6 becomes valid if x_5 changes its value, backtracking to x_5 is useless. Therefore, the algorithm backtracks to x_4 instead of to x_5.

There exits a trade-off between the amount of useless backtracking saved by *DepDB* and the computation cost and memory required to identify the culprit variable of a failure. In order to avoid having exponential memory, the *Graph-based Backjumping* (*GBJ*) algorithm [Dechter and Pearl, 1988] and the *Conflict-based Backjumping* (*CBJ*) [Prosser, 1993] keep a reduced set of nogoods, using polynomial amount of memory.

Various heuristics have been studied to improve efficiency. It has been showed that the ordering of selecting variables and values affects the performance of backtracking algorithms [Haralick and Elliot, 1980]. The *first-fail principle* is commonly used in variable ordering heuristics. The idea is to select first variables that are more likely to fail. The *minimum-domain heuristic* follows this principle. Each time the algorithm has to select a variable to be assigned, it selects the variable with the least number of valid values in its domain.

The *min-conflict* heuristic is a beneficial strategy for ordering value domains [Minton et al., 1992]. The idea is to select first the value which is more consistent with domains of unassigned variables.

2.3.2 Incomplete Search

In the worst-case scenario, complete algorithms must consider all hypothetical solutions of a *CSP* before deciding its satisfiability. This may cause those algorithms to be slow when solving very large *CSPs* (i.e. problems with large number of variables and large domains). For solving this kind of problems, incomplete search methods can be a suitable option.

Local search is a common form of incomplete search. Typically, in most local search algorithms, the concept *neighborhood* refers to the set of states considered as neighbors of a given state. Well-known local search methods, such as *GSAT* [Selman et al., 1992], *Min-Conflict* [Minton et al., 1990], *Hill-Climbing*, *Random-Walk*, and *Tabu-Search* [Glover, 1986], [Bartak, 1988], can be seen as specializations the generic local search algorithm *The Generic Local Search* [Silaghi, 2002], *GLS* in short. In the literature, these algorithms also appear as *Iterative Improvement* algorithms.

The generic local search algorithm considers one state at a time. The initial state can be generated randomly or based on some prior knowledge about the problem to solve. Whether the state in turn constitutes a flawed solution, the

```
function GLS()
s ← random valuation of variables;
N ∈ neighborhood(s);
while true do
  if ¬ restartRule(s, N) then
    c ← selection(s, N);
  else
    c ← alternative-selection(s, N);
  end if
  s ← c;
  if(solution(s)) return s;
  if(abandon()) return failure;
end while
```

Figure 2.7: The Generic Local Search algorithm.

```
procedure Breakout()
until current-state is solution do
  if current-state is not a local-minimum then
    make any local change in current-state that reduces the total cost;
  else increase weights of all current conflicts;
  end if
end until
```

Figure 2.8: The Break-out algorithm.

algorithm moves to a neighboring state that improves the current state. Usually, the improvement is done by changing small parts of the current state, for instance, by changing the value of one variable in the current assignment. Occasionally, it may happen that state in turn does not have a state that outperforms it in its neighborhood. In this situation, the current state is called *local-optimum*. When the algorithm is trapped in a local-optimum, it uses an alternative strategy for selecting the next state that allows it to escape from the local-optimum. *GLS* terminates when a real solution is found or when a termination condition is reached. *GLS* appears in Figure 2.7.

A typical local search method for *CSP* is the *Break-out* algorithm [Morris, 1993, Yokoo, 2001]. Figure 2.8 shows the Break-out algorithm (*BO*) presented in [Yokoo, 2001]. The algorithm assumes that also, original constraints of the problem and the inconsistencies found during the search, are represented as nogoods (i.e. a subset of conflicting variable values). Every conflict has a weight associated to it, which, initially is equal to 1. The evaluation of a state is defined as the addition of the weights of all conflicts that appears in the state. When trapped in a local-minimum, the algorithm increases the weights of all conflicts in the current state by 1. This causes the evaluation of the current state to become larger than those of the neighboring states.

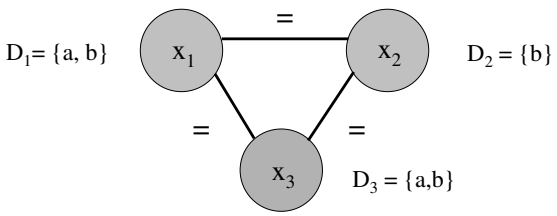


Figure 2.9: Example of an arc-inconsistency *CSP*.

2.3.3 Inference Algorithms

Incomplete Inference

Although backtracking algorithms outperform the generate-and-test method, they may still need to explore a large part of the search space before detecting the satisfiability of difficult problems. According to [Gaschnig, 1979], one of the justifications for this poor performance is that in different parts of the search space, the search fails because the same reasons, i.e. the same subset of inconsistent assignments. This phenomenon is called *thrashing*. Algorithms can reduce thrashing if they store all discovered nogoods and consider them as new constraints for the problem.

Alternatively, one can reduce thrashing by combining a backtracking-based algorithm with a *constraint propagation method*, in which the implications of a constraint on one variable are propagated to the rest of the variables. Some constraint propagation methods have the ability to discard some values that do not belong to any solution from the variable domains, while others may help to discover subsets of inconsistent assignments, i.e. nogoods. Within the related literature, constraint propagation methods are also called *filtering algorithms*.

The simplest cause of thrashing concerns unary constraints and is referred to as *node-inconsistency* [Mackworth, 1977]. If the domain D_i of a variable x_i contains a value a that does not satisfy the unary constraint on x_i, then no solution can have the assignment $x_i = a$. Therefore, thrashing because node inconsistency can be eliminated by simply removing those values from the domains D_i of each variable x_i that do not satisfy unary constraint C_i. After removing all those values, the problem becomes *node-consistent*.

Another possible source of thrashing is referred to as the lack of *arc-consistency* [Mackworth, 1977]. Here "arc" refers to a *directed* arc in the constraint graph, such as the arc from x_1 to x_2 in the constraint graph illustrated in Figure 2.9. In the following, the term $arc(x_i, x_j)$ represents the arc that goes from variable x_i to x_j. We say that $arc(x_i, x_j)$ is *arc-consistent* if each value of x_i in D_i is consistent with at least one value of x_j in D_j. The concept of arc-consistency is *directional*, that is, if $arc(x_i, x_j)$ is arc-consistent then it does

```
function REVISE(x_i, x_j)
delete ← false;
for each value a ∈ D_i do
  if no exits b ∈ D_j such as (a, b) is consistent then
    remove a from D_i;
    delete ← TRUE;
  end if
end for
return delete;
```

Figure 2.10: The Revise function for achieving arc-consistency in a given arc.

```
procedure AC-3()
Q ← {(x_i, x_j); such as x_i ≠ x_j};
while Q is not empty do
  arc(x_i, x_j) ← any arc from Q;
  remove arc(x_i, x_j) from Q;
  if REVISE(arc(x_i, x_j)) then
    for each arc from any variable x_k to the variable x_i
      Q ← Q ∪ {arc(x_k, x_i)}, such as x_k ≠ x_j and x_k ≠ x_i;
    end for
  end if
end while
```

Figure 2.11: The AC-3 algorithm for achieving arc-consistency in a *CSP*.

not imply that $arc(x_j, x_i)$ is also arc-consistent. An arc is arc-inconsistent if it is not arc-consistent.

Definition 2.3.5. A *CSP* is arc-consistent when every arc in the constraint graph is arc-consistent. Conversely, a *CSP* is arc-inconsistent if at least one arc is arc-inconsistent.

Figure 2.9 illustrates an example of an arc-inconsistent *CSP*. In this example, there are three variables, x_1, x_2 and x_3 with domains D_1, D_2, D_3, respectively. There is a binary equality constraint between each pair of variables. $arc(x_1, x_2)$ is arc-inconsistent because the value a in the domain of x_1 is not valid for the unique value in the D_2, the domain of x_2. Similarly, $arc(x_3, x_2)$ is arc-inconsistent because the value a of x_3 is not consistent with any value in the domain of x_2. In contrast, $arc(x_1, x_3)$ and $arc(x_3, x_1)$ are arc-consistent since each value of x_1 in D_1 is consistent with at least one value of x_3 in D_3 and viceversa. $arc(x_2, x_1)$ and $arc(x_2, x_3)$ are also arc-consistent since the unique value in x_2 is also included in the domains of x_1 and x_3.

Clearly, $arc(x_i, x_j)$ can be made arc-consistent by simply deleting those values from the domain D_i of x_i that are not consistent with any values for x_j in D_j. This is the idea of the function `REVISE`, which appears in Figure 2.10. It was taken from [Mackworth, 1977].

In order to make a *CSP* arc-consistent, it is not enough simply to invoke once the above function for each arc in the constraint graph corresponding to

the problem. When the domain of x_i is reduced by the application of the procedure `REVISE`, all the previously checked arcs of the form $arc(x_k, x_i)$ must be checked again. Otherwise, the arc-consistency property of those arcs cannot be guaranteed because some values of the domain of a variable x_k may no longer be compatible with no remaining values of the x_i's domain. [Mackworth, 1977] presents different versions of full algorithms for achieving arc-consistency. One of this algorithm is `AC-3`, which appears in Figure 2.11. Basically, this algorithm keeps in a queue all the arcs that have to be checked for inconsistency. The algorithm processes one arc at a time. The arc in turn $arc(x_i, x_j)$ is deleted from the queue and checked; if at least one value of the domain of x_i is removed, then, every arc of the form $arc(x_k, x_i)$ has to be inserted in the queue. `AC-3` ends when the queue is empty. The time and space complexities of `AC-3` are $\mathbf{O}(ed^3)$ and $\mathbf{O}(e + nd)$, respectively. In the equations, d is the maximum domain size and e is the number of constraints.

The term *k-consistency* refers to stronger degrees of consistency.

Definition 2.3.6. A *CSP* is k-consistent if, for any set of variables x_1, ..., x_{k-1}, and for any set of consistent values for these variables, any other variable x_k will have at least one value consistent with the assignment of the previous $k - 1$ variables.

The aforementioned concepts of node-consistency and arc-consistency, are equivalent to 1-consistency and 2-consistency, respectively. A *CSP* is strongly k consistent if it is j consistent for all $j \leq k$.

In [Freuder, 1988], Freuder presents an algorithm to make a *CSP* strongly k consistent, for $k > 2$. From the definition of strong k-consistency, it is straightforward to prove that, if a *CSP* with n variables is n-consistent, then one can find a solution without any backtracking. Nevertheless, the algorithm for making a *CSP* (with n variables) strongly n-consistent might require, in the worst case, exponential time in n.

For some kinds of *CSPs*, however, it is possible to ensure a backtracking-free search with a degree of consistency smaller than n. If a *CSP* is strongly k consistent, and $k > w$, where w is the width of the *CSP*, then a search ordering exists that is backtracking free [Freuder, 1988]. In this property the term "width of the *CSP*" is related to an *ordered CSP*, i.e. a *CSP* whose variables have been ordered linearly. The *width of an ordered CSP* is the maximum of the width of all variables. The *width of a variable* in a ordered *CSP* is defined by the number of constraints from the variable to preceding variables in the ordering.

Even when the application of a constraint propagation method on a certain problem results in a backtracking-free search, this is no good if we spend more time propagating constraints than we would have spent applying a backtracking algorithm.

Incomplete Inference plus Backtracking-based Algorithms

When combining with a backtracking-based algorithm, a constraint propagation method for achieving arc-consistency can either be invoked in a preprocessing

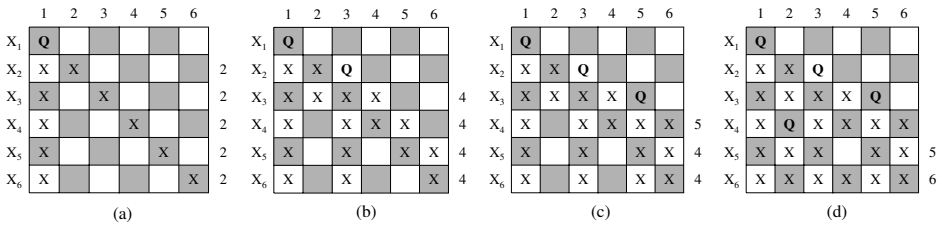


Figure 2.12: Example of the execution of the Forward Checking algorithm

step or after each assignment performed by the backtracking algorithm. In the latter case, the algorithm is known as *Maintaining Arc-Consistency* (*MAC*) algorithm [Sabin and Freuder, 1994]. In general, after establishing arc-consistency, the variable domains can have: (1) no values; (2) exactly one value; or (3) more than one value. When at least the domain of a variable becomes empty, this means that the problem is unsolvable. When the domain of each variable contains exactly one value, these values constitute a solution to the problem. If at least one variable has multiple values in its domain, then the search is required to find a solution or discover that no solution exits.

Unlike *MAC*, Forward Checking (*FC*) [Haralick and Elliot, 1980] is a backtracking-based algorithm which makes a limited amount of arc-consistency after each assignment. *FC* works like *BT*, except that it propagates the effect of each assignment by pruning from the domains of unassigned variables those are inconsistent with the assignments. Figure 2.12 provides an example of the execution of *FC* when solving the 6-queens problem. Let assume that variable are assigned following the lexicographical ordering: x_1, x_2, x_3, x_4, x_5, x_6. Then, *FC* starts by assigning the value 1 to x_1 (i.e. the first queen is placed in the first position of the first row). This assignment is added to the partial solution. Note this cause each unassigned variable to have 2 invalid values in its domain (Figure 2.12 (a)). Then, *FC* selects x_2 as the next variable to be added to the partial solution. x_2 is assigned to the value 3 ($x_2 = 3$). After this assignment, variable x_6 has 3 invalid values while the rest of the variables have 4 invalid values (Figure 2.12 (b)). *FC* searches for consistent value for x_3. Then, *FC* assigns the value 5 to x_3 ($x_3 = 5$). In this case, the numbers of invalid values are 5, 4, 4 for x_4, x_5, x_6, respectively (Figure 2.12 (c)). *FC* assigns the second value to x_4 ($x_4 = 2$) and adds this assignment to the partial solution. This produces invalid values for the unassigned variables x_5 and x_6 is 5 and 6, respectively (Figure 2.12 (d)). Since no value is valid for x_6, *FC* can backtrack to x_4 before determining a values for x_5. *FC* removes x_4's assignment from the partial solution. For this problem, *FC* continues the search until finding a solution.

Complete Inference

A method for complete inference is *variable elimination*. It consists on removing one variable of the network, joining all constraints mentioning it and producing a single constraint that summarizes their effect but it does not mention the variable. Interestingly, the new network is equivalent in the set of remaining variables to the previous one. The new network can be processed in the same way, eliminating one of its variables, and producing an equivalent (in the set of remaining variables) network with two less variables than the original one. This process is repeated until every variable has been eliminated. The resulting network is trivially solved, and its solution is passed back to generate a global solution. The algorithm that perform this process is *adaptive consistency* (*ADC*) [Dechter and Pearl, 1987], which can be seen as an instance of a more general algorithm called *bucket elimination* (*BE*) [Dechter, 1999].

The complexity of *ADC* is $O(n(2d)^{w*+1})$ in time and $O(nd^{w*})$ in space, where $w*$ is the *induced width* along the static variable ordering (required by *ADC*). Intuitively, $w*$ is the largest arity of intermediate constraints that are computed and stored by *ADC*. It holds that $w* < n$.

2.4 Summary

We introduced in this chapter the Constraint Satisfaction framework. We have given a formal definition for the Constraint Satisfaction problem and presented some sample problems that can be modeled as *CSP*. We have also provided an overview of the main algorithms, that appear in the related literature, for solving *CSP*.

Chapter 3

Distributed Constraint Satisfaction

In recent years, an increasing interest has arisen about problems where information is distributed among different computers. If this information cannot be centralized in a single computer, the classical *CSP* model is inadequate for these problems, because it assumes centralized solving. Distributed Constraint Satisfaction (*DisCSP*) consider constraint problems, where knowledge (i.e., domains, variables and constraints) is distributed among communicating agents and cannot be centralized for different reasons (i.e. prohibitive costs of constraint translation or security/privacy issues).

This chapter gives an overview of existing research *DisCSP*. First, we formally introduce the Distributed Constraint Satisfaction problem (Section 3.1) and provide some examples (Section 3.2). Then, we describe a set of solving methods for *DisCSP* (Section 3.3). We also discuss some issues to evaluate (Section 3.4) and implement (Section 3.5) *DisCSP* solving algorithms.

3.1 What is a Distributed Constraint Satisfaction Problem?

A distributed *CSP* (*DisCSP*) is a *CSP* whose variables, domains and constraints are distributed among automated agents. There exist two distributed models for representing *DisCSPs*: the *variable-based model* [Yokoo et al., 1992] and the *constraint-based model* [Silaghi et al., 2000].

Definition 3.1.1. A variable-based model is a distributed model in which each variable belongs to one agent and constraints are shared between agents.

Definition 3.1.2. A constraint-based model is a distributed model in which each constraint belongs to one agent and shared variables in two constraints not belonging to the same agent are duplicated.

In this thesis, we focus on problems that are modeled using the variable-based model, which is the most frequently found in the related literature. According to the variable-based model, the formal definition of a finite *DisCSP* is as follows.

Definition 3.1.3. A finite *DisCSP* is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are the same as in *CSP* (see Definition 2.1.1 in Chapter 2), and

- $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents,
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent.

Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjointed subsets, $\mathcal{C}_{intra} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_{ij} is known by the agent owner of x_i and x_j, but it is unknown by the other agents. Usually, an inter-agent constraint c_{ij} is known by the agents $\phi(x_i)$ and $\phi(x_j)$ [Yokoo et al., 1998, Hamadi et al., 1998].

Similar to *CSP*, a solution of a *DisCSP* is an assignment of values to variables satisfying every constraint (although *DisCSP* literature focuses mainly on solving inter-agent constraints). Distributed *CSPs* are solved by the collective and coordinated action of agents $\mathcal{A}$. We assume the following communication model ([Yokoo, 2001]):

- agents communicate by sending messages;
- an agent can send messages to other agents if it knows their addresses;
- the delay in delivering a message is finite, though random;
- for a given pair of agents, messages are delivered in the ordering they were sent.

From now on, we identify the agent number with its variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$) in *DisCSPs* where every agent owns exactly one variable. In this situation, all constraints are inter-agent constraints, so $\mathcal{C} = \mathcal{C}_{inter}$ and $\mathcal{C}_{intra} = \emptyset$. We use the terms "variables" and "agents" interchangeably.

Next, we discuss some examples of *DisCSP*.

3.2 Example of *DisCSPs*

Here, we describe three examples for *DisCSP*: the Distributed n-queens problem, the Distributed n-pieces m-chessboard problem and the Distributed Sensor-Mobile problem.

3.2.1 Distributed n-queens Problem

The *n-queens* problem is a frequent example in Constraint Satisfaction. In the distributed version of this problem, queens are represented by autonomous agents [Yokoo et al., 1998]. Similar to the n-queens problem (see Chapter 2, Section 2.2.1), the distributed n-queens problem consists of n queens which must be located in a $n \times n$ chessboard in such a way that no queen attacks any other.

We model this problem as a *DisCSP* by having the same number of agents and queens. In this formulation, each queen is represented by an agent which holds a variable. Besides, each agent is associated to one row of the $n \times n$ chessboard. The variable held by an agent corresponds to the position of the queen in the row. There exits an inter-agent constraint between each pair of variables. Every constraint is explicitly defined by the set of allowed combinations of positions where the involved two queens can be located, i.e. those combinations of positions where the two queens are not placed in the same column or diagonal (by construction, two queens cannot be at the same row).

3.2.2 Distributed n-pieces m-chessboard Problem

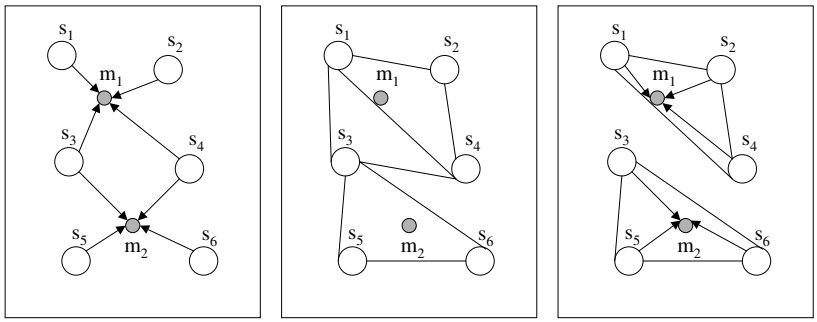
The *n-pieces m-chessboard* problem is an extension of the n-queens problem [Brito and Meseguer, 2003]. This problem consists of n chess pieces and a $m \times m$ chessboard and the goal is to put all pieces on the chessboard in such a way that no piece attacks any other. Similar to the distributed n-queens problem, in the distributed version of *n-pieces m-chessboard*, pieces are represented by autonomous agents.

One can formalize the distributed *n-pieces m-chessboard* problem as a *DisCSP* as follows. Every chess piece is represented by an agent, which holds a variable. The domains of each variable contain $m \times m$ values, each one corresponds to a position of the $m \times m$ chessboard. Analogous to the distributed n-queens problem, there exits a constraint between each pair of variables. Constraints depend on the way the involved chess pieces attack one another. Each constraint enumerates the set of combinations of positions where the two involved chess pieces can be located without attacking each other.

3.2.3 Distributed Sensor-Mobile Problem

Inspired by a real distributed resource allocation problem, [Fernández et al., 2002] introduces the Distributed Sensor-Mobile problem (*SensorDCSP*). It consists of a set of sensors $\{s_1, s_2, ..., s_n\}$ and set of mobiles $\{m_1, m_2, ...m_k\}$. Sensors are required to cooperate for tracking mobiles. Each mobile must be tracked by 3 sensors. Each sensor can track at most one mobile. A solution is an assignment of three distinct sensors to each mobile. This assignment must satisfy two sets of constraints: visibility and compatibility constraints.

Figure 3.1 presents an example of *SensorDCSP*. This example includes 6 sensors (s_1, s_2, s_3, s_4, s_5 and s_6) and 3 mobiles (m_1, m_2 and m_3). This figure



(a) Visibility constraints (b) Compatibility constraints (c) A solution

Figure 3.1: An instance of the sensor-mobile problem. (a) Visibility constraints. (b) Compatibility constraints. (c) A solution for the problem.

includes the visibility constraints (visibility graph (a)), compatibility constrains (compatibility graph (b)) and also a possible solution for this instance (given in graph (c)). One mobile is visible for a sensor if and only if there exits a directed arc between them in the visibility graph. Two sensors are compatible if and only if they are linked by an arc in the compatibility graph. In the solution, the three sensors assigned to each mobile are the sensors that form a triangle where the mobile is inside.

In general, finding a solution for the sensor-mobile problem is NP-complete [Fernández et al., 2002]. Note that this problem can be easily reduced to the problem of partitioning a graph into cliques of size three, which has been proven to be NP-complete [Kirkpatrick and Hell, 1983]. In contrast, instances of this problem, for which every pair of sensors is compatible, can be solved in polynomial time [Fernández et al., 2002].

One can formulate *SensorDCSP* as a *DisCSP*, as follows. Each agent represents one mobile. Each agent includes three variables, one for each sensor that is required to track the corresponding mobile. The domain of a variable is the set of compatible sensors. There is a binary constraint between each pair of variables in the same agent. These intra-constraints must guarantee that sensors assigned to a mobile are different but compatible. There exits a binary constraint between the variables of different agents. These inter-constraints make every sensor be selected by at most one agent.

Furthermore, we present two other examples of problems that can be modeled as *DisCSPs* in Part IV Applications. These problems are the Distributed Meeting Scheduling and Distributed Stable Matching problems, which will be studied in Chapter 10 and Chapter 11, respectively.

3.3 Algorithms for Solving *DisCSP*

The most trivial algorithm for solving a *DisCSP* is to gather all the information about the problem (variables, domains and constraints) into a single agent. Then, this agent solves the problem using a centralized algorithm (Chapter 2). In some cases, however, this approach is not convenient. The cost of collecting the whole problem into a single agent could be very high. Furthermore, in some applications, agents may desire to keep their local information as private as possible. In some other problems, agents may desire to participate actively during the solving process. For example, an agent may want to decide dynamically which value is more suitable for it, at any given time.

A distributed algorithm for *DisCSP* does not consider the centralized solving approach. In a distributed algorithm, all agents cooperate for finding a globally consistent solution. The solution involves assignments of all agents to all their variables. Agents exchange messages containing information about their assignments which allow them to check the consistency of assignments with respect to the problem constraints.

Depending on the model we assume about the timing of events in the distributed system, we obtain different types of algorithms. In [Lynch, 1997], three timing models are considered, which are informally described as follows:

1. *The synchronous model.* "This is the simplest model to describe, to program and to reason about. We assume that components (agents) take steps simultaneously, that is, that execution proceeds in synchronous rounds."

2. *The asynchronous model.* "We assume that separate components (agents) take steps in arbitrary ordering, at arbitrary relative speeds."

3. *The partially synchronous model.* "We assume some restrictions on the relative timing of events, but execution is not completely lock-step as it is in the synchronous model."

These three timing models generate three types of algorithms for solving *DisCSP*. Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active agent. In an asynchronous algorithm every agent is active at any time, and they do not have to wait for any event. A partially synchronous algorithm is in between these two types. An agent running a partially synchronous algorithm may be required to wait for some special event, but not for every event.

To solve a *DisCSP* instance, the three types of algorithms differ in their functionality and efficiency. Considering functionality, asynchronous algorithms are the most general and portable, because they impose no assumptions on the timing of computation steps. Usually, they are more robust and offer more privacy than the other two types. Regarding efficiency, defined generally as the

amount of resources required to compute a solution, there is some debate as to which type of algorithm is more efficient. Literature concentrates on asynchronous algorithms for solving *DisCSP*. In following subsections, we describe several existent synchronous and asynchronous algorithms for *DisCSP*. Only a few partially synchronous algorithms have been studied. In Chapter 6, we present an algorithm that is basically asynchronous although it requires that some agents to synchronize their actions under certain conditions.

3.3.1 Synchronous Search

The simplest backtracking algorithm for *DisCSP* is derived from Chronological Backtracking algorithm (*BT*) (Section 2.3, Chapter 2). *BT* searches for a solution by continuously trying to make the extension of the current partial solution (which does not involve all the variables) into a total one.

The distributed version of this algorithm for *DisCSP* is called *SBT*. This algorithm was first presented in [Yokoo et al., 1992]. In this work, it is assumed that agents only have one variable. *SBT* requires a static instantiation ordering of agents. Agents exchange two kinds of messages: **ok?** and **ngd**. Following the static ordering, agents try to extend a partial solution into a total one by adding consistent assignments for unassigned variables. Initially, the partial solution is empty. The first agent in the ordering assigns any value to its variable and inserts this assignment to the partial solution, which is sent, via an **ok?** message, to the second agent in the ordering. In general, when a variable receives an **ok?** message from the preceding agent in the ordering, it tries to assign a consistent value to its variable according to the constraints with the previously assigned variables. If such a value exits, the agent includes this valuation in the partial solution and sends the partial solution to the next agent in the ordering. Otherwise, the agent sends a **ngd** message to the preceding agent in the ordering, which causes the receiver to search for a new consistent value.

DisBT terminates either because all variables are included in the partial solution or because every value for the variable of the first agent has been discarded. In the former, the partial solution constitutes a solution for the problem, while in the latter, the problem is unsatisfiable. *SBT* is correct, complete and terminates.

The Conflict Backjumping (*CBJ*) algorithm is another *CSP* method that can be easily implemented in distributed settings in a synchronous form. As seen in Section 2.3, Chapter 2, *CBJ* improves the performance of *BT*. In *CBJ*, when a variable cannot find a consistent value with its constraints and the preceding variable assignments, the algorithm backtracks not to the previously assigned variable, but to the closest preceding culprit.

The term *SCBJ* represents the distributed version of *CBJ*. This algorithm was introduced in [Zivan and Meisels, 2003]. *SCBJ* requires a total ordering among agents and uses the same kinds of messages as *SBT*: **ok?**, **ngd**. The only difference between *SBT* and *SCBJ* occurs when an agent does not have a valid value for one of its variables. In this situation, the *SBT* agent sends a **ngd** message to the previous agent in the ordering while the *SCBJ* agent sends

a **ngd** message to *the closest culprit agent*. For an agent j, the closest culprit agent is the agent nearest to j, that appears before j in the ordering and holds a variable which forbids at least one value in j's domain. The **ngd** message includes a nogood, that is the set of previously assigned variables which causes the sender not to have consistent values in its domain.

Analogous to *SBT*, *SCBJ* is correct, complete and terminates. *SCBJ* is experimentally compared with other novel synchronous methods in Chapter 4.

3.3.2 Asynchronous Search

Complete Algorithms

The pioneering works of Yokoo and colleagues propose two of the most famous asynchronous algorithms for *DisCSP*: the Asynchronous Backtracking (ABT) and the Asynchronous Weak-Commitment search (AWC) algorithm [Yokoo et al., 1992, Yokoo, 1995, Yokoo et al., 1998]. Both methods assume the variable-based model.

These algorithms also require a total ordering among agents, which is static for ABT and dynamic for WCS. This ordering defines the priority of agents. Agents that appear first in the ordering have higher priority, while agents that appear last in the ordering have lower priority.

ABT and AWC assume that every constraint is directed following the total ordering among agents. For each constraint, the lowest priority agent involved in the constraint is the agent that evaluates the constraint and, therefore, is called *the constraint-evaluating agent*. The rest of agents in the constraint, that is, the agents involved in the constraint with higher priority than the constraint-evaluating agent, are called *the value-sending agents* because they send their assignments to the constraint-evaluating agent, which will check the consistency of the constraint. For each constraint, one link exits from each value-sending agent to the constraint-evaluating agent.

The neighbors of an agent A_i refer to the set of agents that share constraints with A_i. The *higher priority agents* of an agent A_i is the set of agents that are neighbors of A_i and appear before A_i in the ordering. Conversely, the *lower priority agents* of agent A_i is the set of agents that are neighbors of A_i and appear after A_i in the ordering.

ABT and AWC use three kinds of messages to solve a problem: **ok?**, **ngd** and **addl**. They initially assume links between each agent and its neighbors. In both algorithms, agents start the search by assigning initial values to their variables. Each time an agent assigns a new value to one of its variables, it uses an **ok?** to inform its lower priority agents of the new assignment. Each agent stores the last received assignments from its higher priority agents in the agent view. When an agent does not find a value for one of its variables that is consistent with the assignments in the agent view, the agent backtracks. Both ABT and AWC have different ways of performing backtracking.

In the ABT algorithm presented in [Yokoo et al., 1992], when an agent cannot find a consistent value for one of its variables, it finds all the *minimal nogoods*

of its agent view, that is, every nogood that does not contain another nogood. Then, the agent sends one **ngd** message for every of those minimal nogoods. Each nogood is sent to the agent with the lowest priority among those that appear in the nogood that the message contains. After sending these messages, the agent forgets the assignments of those agents to which it sent a **ngd** message.

A **ngd** message causes the recipient agent to record the received nogood as a new constraint and to try to find a new value consistent with the agent view and with all recorded nogoods. To simplify the computation of all the minimal nogoods that are in the agent view of an agent, [Yokoo et al., 1998] propose to use the whole agent view of the agent as a single non-minimal nogood. This nogood is sent in a **ngd** message to the closest agent in the agent view.

Because as agents act concurrently and asynchronously, a **ngd** message may be obsolete when received by an agent. In this situation, the agent ignores the message and sends again its current value to the message sender. If an agent A_j receives a **ngd** message, it may be that the received nogood includes a variable x_i belonging to A_i which does not share a constraint with A_j. In this case, A_j will send an **addl** message to A_i informing to A_i that, each time x_i changes its value A_j must be informed via an **ok?** message.

In *ABT*, the priority ordering of agents is determined at the beginning and it is maintained statically during the execution of the algorithm. This ordering identifies the agents that act as constraint-evaluating agents and the agents that act as value-sending agents. The static ordering has the drawback that any value proposed by a value-sending agent will not be changed unless an exhaustive search is performed by the constraint-evaluating agent, which could be costly in large-scale problems.

AWC can be seen as a modification of *ABT*, working with a dynamic priority ordering of agents. In *AWC*, in addition to the variable's assignment, every **ok?** message also includes the priority value of the sender agent. When the current assignment is not consistent with the agent view, the agent selects a new consistent assignment that minimizes the number of constraint violations with lower priority agents. If an agent cannot find a consistent value with its agent view, it generates a new nogood, it sends the **ngd** message to all its neighbors and increases its priority one unit over the maximal priority of its neighbors. Then, it finds a consistent value with the assignments of higher priority agents and informs its neighbors via **ok?** messages. If no new nogood can be generated, the agent waits for the next message.

In both algorithms, nogoods are exchanged and stored by agents. This may lead to extra message passing and extra memory usage. *ABT* and *AWC* are correct, complete and terminate [Yokoo, 2001]. However, the completeness of *AWC* may be affected if all nogoods are not stored. This causes *AWC* to have an exponential-space complexity.

The original *ABT* presented by Yokoo and colleagues has been developed further and studied in number of other works [Yokoo et al., 1998, Hamadi et al., 1998, Bessière et al., 2001, Bessière et al., 2005]. A new version of *ABT* which does this without adding new links is one of the contributions of

this thesis. This algorithm is presented in Chapter 5.

Next, we mention some other asynchronous and complete algorithms that have been proposed to solve *DisCSP*:

- *DIBT* is distributed asynchronous backtracking algorithm which performs Graph-based Backjumping without nogood storage [Hamadi et al., 1998, Hamadi, 1999]. In its original formulation, *DIBT* is not complete [Yokoo, 2000, Bessière et al., 2001]. A revised version of this algorithm is discussed in Chapter 5.

- The Distributed Maintaining Asynchronously Consistency for *ABT* (*DMAC-ABT*), presented in [Silaghi et al., 2001a], is a complete protocol for maintaining asynchronously consistency. *DMAC-ABT* is a generic algorithm that can be easily integrated into more complex versions of *ABT*.

- The Asynchronous Aggregation Search (*AAS*) assumes the constraint-based model [Silaghi et al., 2000, Silaghi and Faltings, 2005]. The algorithm and its later versions can be seen as generalizations of *ABT*. They are based on the exchange of sets of partial solutions among agents. Unlike the variable-based model, where variables are distributed among agents, *AAS* considers the dual case where constraints are controlled by a single agent. The use of *AAS* may cause that the problem to be solved has to be translated into a new one. This transformation could be inadequate in many naturally distributed problems where the initial problem structure must remain unchanged. This approach is especially suitable for problems with arithmetic constraints, where variables are common to multiple agents.

- Following the idea of *AWC*, other works propose new algorithms which allow agents to change dynamically the priority ordering during search. Alternative dynamic variable orderings for *DisCSP* are investigated in [Armstrong and Durfee, 1997]. [Silaghi et al., 2001b] say that a large number of reordering operations in an asynchronous algorithm may be costly. They suggest performing only a finite number of reordering operations. However, the experimental results presented in the work show minor improvements to static ordering *ABT*. More recently, [Zivan and Meisels, 2005b] propose a generic method for dynamic ordering in asynchronous backtracking (*ABT-DO*). Agents in the algorithm choose orders dynamically and asynchronously. At the same time, each agent acts according to the most updated ordering it knows. An array of counters represents each suggested ordering of its priority with respect to others orders. Each time an agent replaces the assignment of its variables, it may propose a new ordering. In this algorithm, an agent can propose only orderings which affect to agents with lower priority than itself. Thus, the first agent in the ordering will never be reordered. The combination of *ABT-DO* and heuristic inspired by the idea used for dynamic backtracking in *CSPs* [Ginsberg, 1993] was found to be very effective.

- Recent works propose new approaches to *DisCSP*. In the Concurrent Backtracking Search (*ConBT*) several search processes asynchronously scan disjoined parts of the search space. The search that each process performs is completely synchronous and when an agent cannot find a consistent value, it backtracks following a chronological ordering [Zivan and Meisels, 2004a]. In contrast to *ConBT*, in the Concurrent Dynamic Backtracking (*ConDB*) when an agent cannot find a consistent value, it backtracks to the closest agents involved in the the conflict [Zivan and Meisels, 2004b].

Incomplete Algorithms

Considering asynchronous and incomplete approaches for *DisCSP*, the Distributed Break-out algorithm (*DisBO*) [Yokoo, 2001] is the method that has been studied the most in the literature. The algorithm is inspired on the Break-out algorithm (*BO*) for *CSP*. Similar to the original Break-out algorithm, *DisBO* assumes that the original constraints of the problem and the inconsistency found during the search, are represented as nogoods (i.e. a subset of conflicting variable values). Every conflict has a weight associated, which initially is equal to 1.

In *BO*, a *flawed solution* containing some constraint violations is revised by local changes until all constraints are satisfied. The evaluation of the flawed solution is defined by the sum of the weights of all conflicts that appears in it. The algorithm moves from one flawed solution to another one if the new one has better evaluation than the preceding one.

DisBO agents exchange two kinds of messages among them: **ok?** and **improve**. The former is used by an agent to exchange the current valuation of its variables while the latter is used to reveal the maximal improvement it will get if it changes the assignments of its variables. In contrast to *BO*, in *DisBO* more than one variable may change its value in the flawed solution at a time, which allows the algorithm to take advantage of parallelism. Each agent communicates only with its neighboring agents, that is, the agents that share a constraint with them. Neighboring agents exchange values of possible improvements, and only the agent that can maximally improve the evaluation value may change its value. Note that it is possible for two non-neighboring agents to change its assignments concurrently.

Since agents only process local information, they cannot detect when the whole system is trapped in a local-minimum like in *BO*. Alternatively, *DisBO* agents work with the concept of *quasi-local-minimum*, which is a weaker condition than a local-minimum but can be detected via local communications. An agent is said to be in a quasi-local-minimum if the agent is violating some constraint and the possible improvement of it and all its neighboring agents is 0. When an agent is trapped in a quasi-local-minimum, it increases by 1 the weights of all constraint violations which help the system to jump out of a possible real local minimum.

3.4 Comparing Algorithmic Performance

Some debate has happened about the parameters for measuring the performance evaluation of the algorithms [Meisels et al., 2002]. We consider that solving distributed problems requires search effort from individual agents, plus the usage of a global network that implements message passing. The search effort can be measured as the total number of constraint checks performed by the set of agents during one execution, or as the CPU time that an agent requires to complete such execution (network communication time is not included in this measure). The network usage is measured as the total number of messages exchanged during the resolution of a problem. The cost in time of exchanging one message is usually higher than the cost of performing a constraint check, although the exact relation depends on the implementation and the network. Because of that, we tend to consider better algorithms those which exchange less messages.

In synchronous algorithms, the computation effort is measured by the total number of constraint checks (cc), and the global communication effort is evaluated by the total number of messages exchanged among agents (msg) [Lynch, 1997]. In asynchronous algorithms, search effort is measured by the number of "non-concurrent constraint checks" ($nccc$), which was defined in [Meisels et al., 2002], following Lamport's logic clocks [Lamport, 1978]. Each agent has a counter for its own number of constraint checks. The number of non-concurrent constraint checks is computed by attaching to each message the current counter of the constraint checks of the sending agent. When an agent receives a message, it updates its counter to the higher value between its own counter and the counter attached to the received message. When the algorithm terminates, the highest value among all the agent counters is taken as the number of concurrent constraint checks. Informally, this number approximates the longest sequence of constraint checks not performed concurrently. As for synchronous search, we evaluate the global communication effort as the total number of messages exchanged among agents (msg).

Note that, in synchronous algorithms, all constraint checks are performed sequentially. Therefore, $nccc = cc$. In this thesis we use the term $nccc$ to refer to the computation cost in both kinds of algorithms.

3.4.1 Random Binary *DisCSP*

Random binary problems are commonly used as sample problems to evaluate the performance of methods for solving *CSP* and *DisCSP*. A *binary random CSP* class is characterized by $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m the number of values per variable, p_1 the network *connectivity* defined as the ratio of existent constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected [Smith, 1994]. Similarly, a binary random *DisCSP* is defined by $\langle n, m, p_1, p_2 \rangle$, where each element has the same meaning as for random binary *CSP*. In addition, each variable is assigned to one agent.

3.5 Simulator

Ideally, to evaluate a new algorithm one should have n dedicated processors connected to a common network on which tests would be done. However, this setting is often not available in most of our labs. Even if there is a number of computers available, the workload of each computer and the load of the communication network are out of the control of the experimenter, and these aspects have a significant impact on the efficiency of the algorithms. Because of that, we consider that simulation on a single computer is a suitable alternative in order to make most of the experimentation in *DisCSP* algorithms. After that, some algorithms can be tested in a real setting, assuming the resources needed to perform a field test. In the following, we consider the different options for *DisCSP* algorithms when are evaluated by simulation on a single computer.

Usually, *DisCSP* algorithms are described in terms of agents. An agent is an autonomous entity that contains a part of the problem, is able to perform its own reasoning process and to communicate with other agents. In a multi-task computer (for instance, a desktop with Linux operating system (OS)), a direct option is to implement each agent as a different task, all having the same priority. The OS scheduler is in charge of activating / deactivating the agents, that take control of the CPU as any other task in the system. Communication among agents is performed using a standard task communication facilities (usually implemented using disk storage). This approach is relatively simple to implement but presents some drawbacks. First, it depends on the OS, so results obtained in computers with different OS may not be directly comparable. Second, even using the same computer and the same implementation, it is difficult to reproduce exactly the same results when repeating the same experiments. There are some sublet factors (such as the mail server, the network load, the disk storage) which change between executions and are out of the experimenter's control. Because of that, the exact reproduction of previous results is almost impossible with this approach.

To overcome this obstacle, an alternative is to use a simulator that offers the same facilities as the OS, but allows one complete control. This simulator allows agents to execute, perform the scheduling among agents and provide communication facilities. With this approach, results are reproducible, the same experiment generates the same results (providing random elements are initialized with the same seed).

The first simulator of this kind appears in the seminal work of Yokoo [Yokoo et al., 1992, Yokoo et al., 1998]. Each agent keeps its own clock, which is incremented at each cycle of computation. One cycle for an agent consists of reading all its incoming messages, processing them and writing all messages generated as answers. It is assumed that a message sent at time t is available to the receiver at time $t + 1$. This means a kind of synchronicity in the activation of agents, which is somehow contradictory with the evaluation of asynchronous procedures.

Another scheduling policy is to activate agents randomly: a random number between 1 and n determines the identifier of the agent to activate. When this

agent terminates, the same process selects the next agent to activate. This approach seems to be more adequate to evaluate asynchronous procedures. In this work, every algorithm is designed and implemented following this approach.

3.6 Summary

In this chapter we have discussed the basic issues of Distributed Constraint Satisfaction. We formally define the problem and present problems that can be modeled as *DisCSPs*. We also describe the main existent methods for solving *DisCSP*. At the end of the chapter, we discuss some issues that have to be considered when evaluating and implementing *DisCSP* procedures.

Part II

Approaches

Chapter 4

Synchronous Backtracking

Synchronous procedures can be directly derived from constraint algorithms in centralized search when extended to distributed environments. Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of the agents are waiting.[1] When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active one. One of the drawbacks of this algorithms is its low tolerance to failure: if the active agent crashes, the algorithm crashes because the rest of agents will be waiting for receiving the privilege that possibly will never arrive.

In this chapter, we describe two distributed versions of existing algorithms for *CSP*: the Synchronous Backtracking (*SBT*) algorithm (Section 4.1) and the Synchronous Conflict-based Backjumping (*SCBJ*) algorithm (Section 4.2). In addition, we introduce two approaches for dynamic variable reordering in synchronous backtracking algorithms (Section 4.3). Empirically, we compare the results of *SCBJ* with and without variable reordering on the n-queens problem and on random instances (Section 4.4).

4.1 Synchronous Backtracking

As seen in Chapter 2, the basic backtracking algorithm (*BT*) for *CSP* was presented in [Bitner and Reingold, 1975]. A value assignment to a subset of variables that satisfies all the constraints within the subset is constructed. This subset is called a *partial solution*. Initially, a partial solution is an empty set. In *SB* a partial solution is expanded by adding consistently new variables one by one, until it becomes a *complete solution* (which involves all the variables) or until a new variable cannot find a consistent value. When for one variable, no

[1] Except for special topological arrangements of the constraint graph. [Dechter and Pearl, 1988] proposes a synchronous algorithm where several agents are active concurrently.

value satisfies all the constraints with the variables in the partial solution, then a *backtracking* is performed. That is, the value of most recently added variable to the partial solution is changed. The algorithm finishes when a complete solution is found or when the first variable cannot find a compatible value (which means the problem is unsolvable).

In [Yokoo et al., 1998], BT was modified to yield synchronous backtracking (SBT) algorithm for *DisCSP*. SBT requires a instantiation ordering among agents. Basically, agents exchange two types of messages to find a solution: **ok?** and **ngd**. Each **ok?** message contains the assignments of the preceding agents. An agent sends a **ngd** message if it cannot find a consistent value. After receiving any of these messages, recipient agent becomes the active agent. When an agent receives a partial solution from the preceding agent, it instantiates its variable based on the constraints that it knows. If the agent finds such a value, it appends this assignment to the partial solution, which is passed to the next agent. If no instantiation of its variable can satisfy the constraints, then it sends a backtracking message to the previous agent.

Regarding efficiency, the major drawback of SBT is that the problem is solved sequentially. In particular, after trying all the values without finding any consistent, the SBT agent sends a **ngd** to the previous variable in the ordering, which is not necessarily the culprit variable.

4.2 Synchronous *CBJ*

The Synchronous Conflict-based Backjumping ($SCBJ$) algorithm [Zivan and Meisels, 2003] is a distributed version of the centralized Conflict-based Backjumping (CBJ) algorithm [Prosser, 1993]. $SCBJ$ performs non-chronological backtracking, that is, an agent does not necessarily backtrack to the previous agent in the total order. Each agent keeps the *conflict-set* (CS), formed by the assigned variables which are inconsistent with some value of the agent variable. Let $self$ be a generic agent. When $self$ discovers that all the values of its domain are forbidden by the assignments of previously assigned variables, $self$ backtracks directly to the agent which include the closest conflicting variable in CS_{self}, say x_i, and sends $CS_{self} - \{x_i\}$ to be added to CS_i. Like SBT, $SCBJ$ exchanges **ok?** and **ngd** messages, which are processed as follows (in the following $self$ is the receiver agent):

- **ok?***(partial-solution).* $self$ receives the partial solution, assigns its variable consistently, selects the next variable and sends the new partial solution to it in an **ok?** message. If $self$ has no consistent value, $self$ sends a **ngd** message to the agent containing the closest conflicting variable in CS_{self}.

- **ngd***(conflict-set).* $self$ has to change its value, because *sender* has no value consistent with the partial solution. The current value of $self$ is discarded, and the new conflict-set of $self$ is the union of its old conflict-set and the received one. After this, $self$ behaves as after receiving an **ok?** message.

After receiving any of these messages, $self$ becomes the active agent. $self$ passes the privilege to other agent sending to it an **ok?** or a **ngd** message. The search ends unsuccessfully when any agent encounters an empty domain and its CS is empty. Otherwise, a solution is found when the last agent is reached and it has a consistent value for its variable.
Since the activation of the message is caused by the reception of a message, it may happen that the active agent crashes. In this situation, the whole system also crashes.

4.3 Heuristics for Dynamic Variable and Value Ordering

As seen in Chapter 2, the *first-fail principle* is commonly used for guiding the selection of variables for assignment [Haralick and Elliot, 1980]. A common and effective heuristic is to select the variable with minimum-domain size (md). When we use md in an algorithm, after any assignment, the domain of each unassigned variable is updated discarding inconsistent values.

In a distributed setting, an agent does not have enough information about the whole problem. An interesting question for synchronous search is how to implement dynamic variable reordering, and in particular md.

If $self$ is the active agent, $self$ knows the values assigned to previous variables, because it has received the current partial solution. But $self$ does not know the constraints between previous variables and unassigned ones. Therefore, $self$ cannot compute the minimum domain heuristic exactly, unless it uses extra messages. Alternatively, we propose to estimate the variable with minimum domain by using one of the following approximations [Brito and Meseguer, 2004, Brito, 2004]:

- *amd1.* Each agent computes the interval $[min_i, max_i]$ of the minimum and maximum number of inconsistent values in the domain of every unassigned variable x_i with the partial solution. This interval is included in the **ok?** message. Then, the next variable to be assigned is chosen as follows:
  - if there is x_i such that $min_i \geq min\{d, max_j\}, \forall x_j$ unassigned, selects x_i (where d is the domain size);
  - otherwise, selects the variable with maximum max_j.
- *amd2.* This approach only computes the current domains of the unassigned variables after **ngd** messages. When $self$ sends a **ngd** message to x_j, instead of sending it directly to x_j it goes chronologically. Each intermediate variable recognizes that it is not its destination, and it includes the current size of its domain in the message. This messages ends in x_j and after assigning it, the minimum domain heuristic without considering the effect of x_j's assignment can be applied on the subset of intermediate variables. It causes some extra messages, but its benefits pay-off.

lex	SCBJ		$SCBJ_{amd1}$		$SCBJ_{amd2}$	
n	*nccc*	*msg*	*nccc*	*msg*	*nccc*	*msg*
10	1,612	170	2,142	127	1,181	141
15	31,761	2,231	31,540	1,951	9,048	695
20	6,518,652	306,337	960,467	46,685	21,684	1,212
25	1,771,192	70,336	25,533,029	1,015,628	138,591	6,718
rand	SCBJ		$SCBJ_{amd1}$		$SCBJ_{amd2}$	
n	*nccc*	*msg*	*nccc*	*msg*	*nccc*	*msg*
10	965	91	1,742	81	881	103
15	4,120	247	7,697	241	2,978	223
20	19,532	921	20,661	596	8,523	482
25	21,372	746	31,849	586	18,165	815
min	SCBJ		$SCBJ_{amd1}$		$SCBJ_{amd2}$	
n	*nccc*	*msg*	*nccc*	*msg*	*nccc*	*msg*
10	2,800	204	2,239	138	2,240	162
15	35,339	2,210	16,735	916	18,694	1,244
20	215,816	10,765	235,540	11,636	29,089	1,214
25	19,949,074	791,089	3,617,532	144,083	56,143	1,869

Table 4.1: Results for distributed n-queens with *lex*, *rand* and *min* value ordering.

Regarding dynamic value ordering, we consider the *min-conflict* heuristic [Minton et al., 1992], which prefers values that are more consistent with domains of unassigned variables. An exact computation requires extra messages. To avoid this, we propose an approximation, which consists of computing the heuristic assuming each agent knows the initial domains of the rest of agents. It is worth noting that if the variable ordering is static, the value ordering heuristic becomes static as well, and the value ordering can be computed in a preprocessing step, before the search starts.

4.4 Experimental Results

In this section, we evaluate *SCBJ* algorithms for solving the distributed n-queens problem and instances of the random binary problems. Regarding dynamic variable ordering, we consider three approaches for variable reordering: (i) *SCBJ* plus a lexicographical and static variable ordering (the original *SCBJ*), (ii) *SCBJ* plus *adm1*, in short, $SCBJ_{amd1}$ and (iii) *SCBJ* plus *adm2*, in short, $SCBJ_{amd2}$. *amd1* and *adm2* are the two dynamic variable reordering approaches described the in previous section. Regarding dynamic value ordering, we consider three approaches for the three algorithms above: (i) *lex*, a lexicographical and static value ordering, (ii) *rand*, in which values are chosen in random ordering; and (iii) *min*, the approach for dynamic value ordering based on the min-conflict heuristic seen in the previous section.

4.4.1 Distributed n-queens Problem

As seen in Chapter 3, the distributed n-queens problem is the classical n-queens problem seen in Chapter 2 (locate n queens in an $n \times n$ chessboard such that no pair of queens are attacking each other) where each queen is held by an agent.

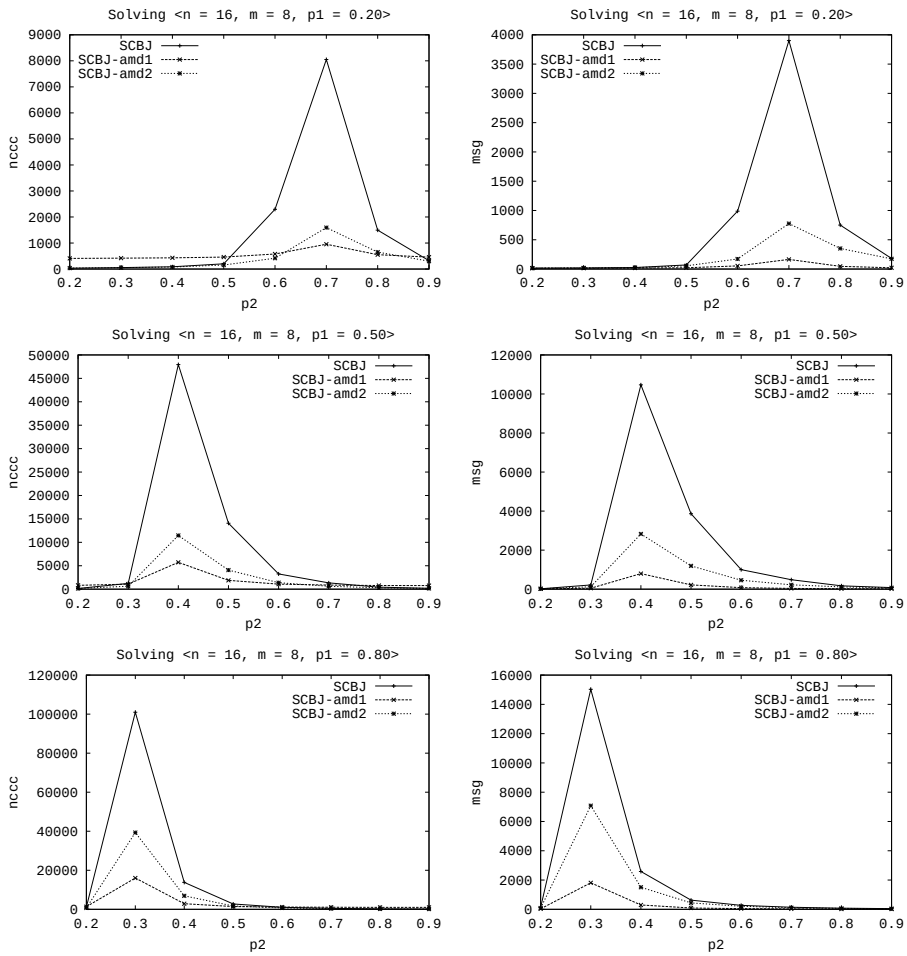


Figure 4.1: Constraint checks and number of messages for *SCBJ*, $SCBJ_{amd1}$, $SCBJ_{amd1}$ on binary random *DisCSP*.

We measure the efficiency of any algorithm by the number of constraint checks (*nccc*) and the number of messages (*msg*), which represent, respectively, the computation and communication efforts that the algorithm consumes to solve a problem.

We consider four instances of the n-queens problem: $n = 10, 15, 20, 25$. Table 4.1 shows the results in terms of *nccc* and *msg*, averaged over 100 executions with different random seeds (ties are broken randomly).

We observe that the random value ordering provides the best performance for every algorithm and every dimension tested. Considering *SCBJ*, approximating minimum domains heuristic has opposite results: *amd1* requires more *nccc* although uses less messages; *amd2* improves on *nccc* but requires more messages

min	*SCBJ*		$SCBJ_{amd1}$		$SCBJ_{amd2}$	
p_1	*nccc*	*msg*	*nccc*	*msg*	*nccc*	*msg*
0.20	7,100	3,277	907	153	1,811	687
0.50	44,024	9,367	5,637	783	11,677	2,669
0.80	102,153	15,111	16,206	1,843	40,449	7,142

Table 4.2: Results near of the pick of difficulty on binary random classes $\langle n = 16, m = 8\rangle$ with *min-conflict* value ordering.

for $n = 10, 25$. In addition, *amd2* produces more stable results, specially when the value ordering heuristic is not good (*lex* and *min*).

4.4.2 Random Binary *DisCSP*

We have tested the proposed algorithms on random instances with 16 agents and 8 values per agent, considering three connectivity classes: sparse (p_1=0.2), medium (p_1=0.5) and dense (p_1=0.8). In Figure 4.1, we report results averaged over 100 executions for *SCBJ*, $SCBJ_{amd1}$, $SCBJ_{amd2}$. In this case value are chosen following the lexicographical static order. Considering *SCBJ*, approximating minimum domains heuristic is always beneficial in terms of *nccc* and *msg*. When using $SCBJ_{amd1}$, the baseline of constraint checks is not zero. This is due to the heuristic computation done as a preprocessing step. Consistently, in the three classes tested, *amd1* provides better results than *amd2*, both in terms of checks and messages.

We also considered the min-conflict value ordering heuristic. Table 4.2 reports the results for solving the hardest instances (near to the pick of difficulty). $SCBJ_{amd1}$ is the algorithm which requires the less number of *nccc* and *msg*. However, both approaches for dynamic variable reordering result more efficient than *SCBJ* in terms of *nccc* and *msg*.

We also compared *SCBJ*, $SCBJ_{amd1}$, $SCBJ_{amd2}$ with respect to the random value ordering. This results are omitted because they show a minor but consistent improvement of all the algorithms with the respect to the lexicographical value ordering approach. The relative ranking of algorithms obtained with random value ordering remains.

4.5 Summary

In this chapter we have studied the performance of the *SCBJ* on n-queens instances and on random *DisCSP*. We proposed two approximations of the minimum domain heuristic for dynamic variable selection. Similar to the effect of using dynamic variables and values reordering for solving *CSP*, empirically, we show the benefic of using these techniques in synchronous backtracking algorithms for solving *DisCSP*.

Chapter 5

Asynchronous Backtracking

Note: A large part of the work described in this chapter is the result of a collaboration with Christian Bessière and Arnold Maestre (LIRMM, Montpelier, France). The discussion of the kernel for ABT algorithms and the theoretical results associated thereof are mainly due to them, however we can say that we have made substantial contributions to the remaining parts of this chapter.

In asynchronous search, all agents are active at any time, having a high degree of parallelism. However, the information that any agent knows about other agents is less updated than in synchronous procedures. Generally, asynchronous algorithms are more difficult to understand and implement than synchronous ones. This chapter is devoted to the study of asynchronous backtracking algorithms.

The structure of this chapter is as follows. We first describe the well known Asynchronous Backtracking (ABT) algorithm from Yokoo and colleagues (Section 5.1). We propose a basic kernel for grouping asynchronous backtracking algorithms (Section 5.2). This kernel is correct, but it does not guarantee termination. By implementing the condition for termination in this kernel we obtain four algorithms (Section 5.3). We also present some new ideas to improving the performance of asynchronous backtracking algorithms (Section 5.4). We present experimental results of algorithms on random *DisCSP* (Section 5.5).

5.1 Asynchronous Backtracking Algorithm

Asynchronous backtracking (ABT) [Yokoo et al., 1992, Yokoo et al., 1998] was a pioneering algorithm used to solve *DisCSP*, its first version dating from 1992. ABT is executed autonomously and asynchronously by each agent in the network. Each agent makes its own decisions, informs other agents about them, and no agent has to wait for the others' decisions. The algorithm computes a global consistent solution (or detects that no solution exists) in finite time; its correctness and completeness have been demonstrated. ABT requires constraints to be directed. A constraint causes a directed link between the two

constrained agents: the value-sending agent, from which the link originates, and the constraint-evaluating agent, at which the link finalizes. To make the network cycle-free, there is a total order among agents, which is followed by the directed links.

In the following, we present a description of how ABT works. To follow we recall the notion of nogood as inconsistent assignment of values to variables (presented in Definition 2.3.4, Chapter 2).

Definition 5.1.1. A directed nogood for the value c of variable x_k is $x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$, meaning that the assignment of c to x_k is inconsistent with respect to the assignments of $a, b, \ldots$ to $x_i, x_j, \ldots$. This nogood is a justification of c removal, as long as the values $a, b, \ldots$ are assigned to variables $x_i, x_j, \ldots$. The left-hand (`lhs`) and right-hand sides (`rhs`) of a nogood are defined from the position of $\Rightarrow$.

Each agent keeps its own agent view and nogood store. Considering a generic agent $self$, $self$'s agent view is the set of values that it believes to be assigned to agents connected to $self$ by incoming links. The nogood store keeps nogoods received by $self$ as justifications of inconsistent values. ABT agents exchange three types of messages: **ok?** (assignments), **ngd** (nogoods) and **adl** (link request). When $self$ makes an assignment, it informs those agents connected to it by outcoming links. $self$ always accepts new assignments, updating its agent view accordingly. When $self$ receives a nogood, it is accepted if it is consistent with $self$'s agent view, otherwise it is discarded as obsolete. An accepted nogood is added to $self$'s nogood store to justify the deletion of the value it targets. When $self$ cannot take any value consistent with its agent view, because of the original constraints or because of the received nogoods, new nogoods are generated by resolution of its nogood store and each one is sent to the closest agent involved, causing backtracking. If $self$ receives a nogood mentioning another agent not connected with it, $self$ is required to add a link from that agent to $self$. From this point on, a link from the other agent to $self$ will exist. The search terminates once quiescence is achieved, meaning that a solution has been found, or when the empty nogood is generated, meaning that the problem is unsolvable.

Figure 5.1 presents an example of ABT execution. The problem has three agents, x_1, x_2, x_3, with variable domains {1, 2, 3}, {2}, {1, 2}, respectively, and constraints $x_1 \neq x_3$ and $x_2 \neq x_3$. First, each variable takes on a value. In the Figure, the underlined value in each domain represents the current assignment. In Figure 5.1(a), the agent view of x_3 will be $\{(x_1 = 1), (x_2 = 2)\}$ after receiving **ok?** messages from x_1 and x_2. Those messages generate two nogoods that are stored in the nogood store of x_3: $\{(x_1 = 1) \wedge (x_3 \neq 1)\}$ and $\{(x_2 = 2) \wedge (x_3 \neq 2)\}$. Then, a consistent value for x_3 does not exist. Agent x_3 resolves the nogood store obtaining a new nogood (i.e. $\{(x_1 = 1) \wedge (x_2 \neq 2)\}$). Then, agent x_3, sends a nogood message with this new nogood to agent x_2, the lowest priority agent involved in it (Figure 5.1(b)). When receiving this nogood message, agent x_2 records it. This nogood contains agent x_1, which is not connected with x_2 by a link. Therefore, a new link must be added between x_1 and x_2. Agent

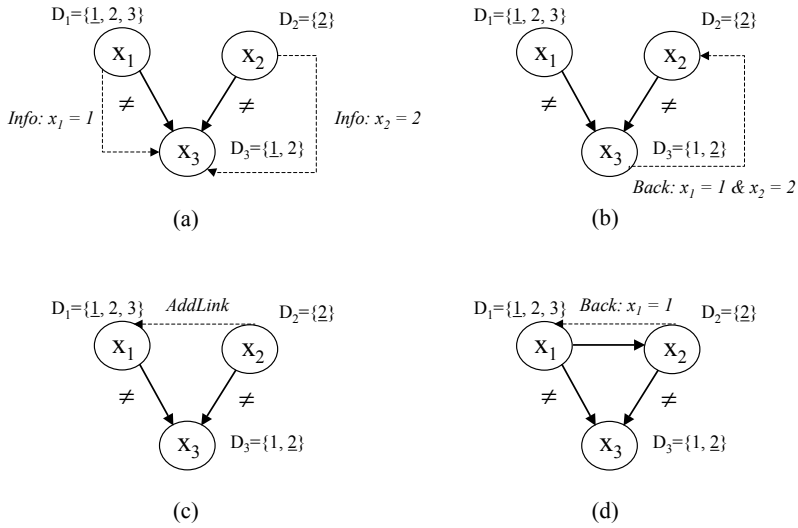


Figure 5.1: Example of ABT execution.

x_2 requests that x_1 sends x_1's value to x_2, and adds $x_1 = 1$ to its agent view (Figure 5.1(c)). Agent x_2 checks whether its value is consistent with the agent view. Since the nogood received from agent x_3 it is not obsolete (it is compatible with its assignment $x_2 = 2$ and its agent view $\{x_1 = 1\}$), the assignment $x_2 = 2$ is inconsistent because of this nogood. The agent view $\{(x_1 = 1)\}$ of x_2 constitutes a nogood because x_2 has no other possible values. There is only one agent in this nogood, i.e., agent x_1, so agent x_2 sends a nogood message to agent x_1 (Figure 5.1(d)). By receiving this nogood message, agent x_1 records this nogood. Agent x_1 checks whether its value is consistent with the agent view. Agent x_1 changes its values taking the value 2 and informs to agents x_2 and x_3. The current value of agent x_2 is consistent with the new value of agent x_1. When agent x_3 receives the **ok?** message from agent x_1, it updates its agent view with $x_1 = 2$, and discards the nogood $\{(x_1 = 1) \wedge (x_2 \neq 2)\}$ as obsolete. Agent x_3 checks whether its current value is consistent with its agent view. Finally, agent x_3 takes on the value 2, and the search ends due to the fact that current assignments constitute a solution for the problem.

5.2 The Unifying Kernel

In the following, we describe ABT_{kernel}, a generic algorithm for variable-based *DisCSP*. This algorithm is correct but it may fail to terminate. We also identify the condition to assure termination.

5.2.1 The ABT_{kernel} algorithm

The ABT_{kernel} algorithm requires, like ABT, that constraints are directed — from the value-sending agent to the constraint-evaluating agent— forming a directed acyclic graph (in short, DAG). Agents are statically ordered in agreement with their constraint orientation. Agent i has higher priority than agent j if i appears before j in the total ordering. Take for instance, a generic agent $self$, $\Gamma^-(self)$ is the set of agents constrained with $self$ appearing above $self$ in the ordering. Conversely, $\Gamma^+(self)$ is the set of agents constrained with $self$ appearing below $self$ in the ordering.

In ABT_{kernel}, nogoods can have been received from lower priority agents, or derived from constraints with higher priority agents. ABT_{kernel} takes the following options with respect to nogoods,

1. *One nogood per removed value.* Each agent keeps only one nogood per removed value. This option, also taken in some version of ABT, assures a polynomial space complexity.

2. *Nogood resolution.* When all values of a variable x_k are ruled out by some nogoods, these nogoods are resolved by computing a new nogood $newNogood$ as follows. Let x_j be the closest variable (in the total order) to x_k in the left-hand side of the nogoods, with value b. `lhs`($newNogood$) is the conjunction of the left-hand sides of all nogoods for values of x_k with the exception of x_j. `rhs`($newNogood$) is $x_j \neq b$. $newNogood$ is sent to x_j. Agent k removes nogoods with x_j in their left-hand side from its nogood store.

Each agent keeps its agent view and a nogood store, which must be consistent. The agent view of $self$ is the set of values it believes are assigned to $\Gamma^-(self)$ agents. Agents exchange assignments and nogoods until a solution is found or inconsistency is detected. A message msg can be of the following types ($sender$ is the sending agent and $self$ is the receiver),

- **ok?**: informs $self$ that $sender$ has made a new assignment $msg.Assig$;
- **ngd**: informs $self$ that $sender$ has found a nogood $msg.Nogood$ as a cause of inconsistency requiring $self$ not to take `rhs`($msg.Nogood$);
- **stp**: informs $self$ that no solution exists and stops the procedure.

ABT_{kernel} appears in Figure 5.2. In the main procedure `ABT`$_{kernel}$, each agent selects a value and informs other agents (`CheckAgentView` call, line 2). Then, a loop receives and processes messages (lines 3-8).

ok? messages are processed by `ProcessInfo` and they are always accepted. After receiving an **ok?** message, the agent view of $self$ is updated to include the new assignment, and any nogood inconsistent with the agent view is removed (`Update` call, line 1). Then, after the change in the agent view, a consistent value for $self$ is searched (`CheckAgentView` call, line 2). `CheckAgentView` checks if

```
procedure ABT_kernel()
1 myValue ← empty; end ← false;
2 CheckAgentView();
3 while (¬end) do
4   msg ← getMsg();
5   switch(msg.type)
6     ok?    : ProcessInfo(msg);
7     ngd    : ResolveConflict(msg);
8     stp    : end ← true;

procedure CheckAgentView(msg)
1 if ¬consistent(myValue, myAgentView) then
2   myValue ← ChooseValue();
3   if (myValue) then for each child ∈ Γ+(self) do sendMsg:ok?(child, myValue);
4   else Backtrack();

procedure ProcessInfo(msg)
1 Update(myAgentView, msg.Assig);
2 CheckAgentView();

procedure ResolveConflict(msg)
1 if Coherent(msg.Nogood, Γ−(self) ∪ {self}) then
2   for each assig ∈ lhs(msg.Nogood) \ Γ−(self) do Update(myAgentView, assig);
3   add(msg.Nogood, myNogoodStore); myValue ← empty;
4   CheckAgentView();
5 else if msg.sender ∈ Γ+(self) ∧ Coherent(msg.Nogood, self) then
          SendMsg:ok?(msg.sender, myValue);

procedure Backtrack()
1 newNogood ← solve(myNogoodStore);
2 if (newNogood = empty) then
3   end ← true; sendMsg:stp(system);
4 else
5   sendMsg:ngd(newNogood);
6   Update(myAgentView,rhs(newNogood) ← unknown);
7   CheckAgentView();

function ChooseValue()
1 for each v ∈ D(self) not eliminated by myNogoodStore do
2   if consistent(v, myAgentView) then return (v);
3   else add(x_j = val_j ⇒ self ≠ v, myNogoodStore); /*v is inconsistent with x_j's value*/
4 return (empty);

procedure Update(myAgentView, newAssig)
1 add(newAssig, myAgentView);
2 for each ng ∈ myNogoodStore do
3   if ¬Coherent(lhs(ng), myAgentView) then remove(ng, myNogoodStore);

function Coherent(nogood, agents)
1 for each var ∈ nogood ∪ agents do
2   if nogood[var] ≠ myAgentView[var] then return false;
3 return true;
```

Figure 5.2: The ABT_{kernel} algorithm for asynchronous backtracking search.

the current value of $self$ is still consistent (line 1). If not, it tries to select a consistent value (`ChooseValue` call, line 2). In this process, some values of $self$ may appear as inconsistent. The nogoods justifying their removal are added to the nogood store (line 3 of `ChooseValue`). If a new consistent value is found, this new assignment is notified to all agents in $\Gamma^+(self)$ through **ok?** messages (line 3). Otherwise, $self$ has to backtrack (`Backtrack` call, line 4). `Backtrack` generates a new nogood by the resolution of existent nogoods for the values of $self$ (line 1). If the new nogood is empty, a **stp** message is sent to the agent *system* and the process stops (lines 2-3). Otherwise, the new nogood is sent in a **ngd** message to the agent appearing in its `rhs` (line 5). The value of this agent is deleted from the agent view (`Update` call, line 6) and a new consistent value is selected (`CheckAgentView` call, line 7).

ngd messages are processed by `ResolveConflict`. A **ngd** message coming from *sender* is accepted if its nogood has the same assignments as $\Gamma^-(self) \cup \{self\}$ (line 1). In this case, the assignments in the nogood for variables not directly related to $self$ are taken to update the agent view (`Update` call, line 2). The nogood is stored, acting as justification for removing the current value of $self$ (line 3). A new consistent value for $self$ is searched (`CheckAgentView` call, line 4). If the message is not accepted, it is considered obsolete. Then, if the value of $self$ was correct in the received nogood, $self$ resends its value to *sender* via an **ok?** message (line 5) because *sender* has forgotten the value of $self$ when sending the **ngd** message (line 6 of `Backtrack`). If not, $self$ does nothing because there is an **ok?** message traveling from $self$ towards *sender* that has not arrived yet.

A **stp** message means that the empty nogood has been derived, so the problem has no solution and the process must stop.

Eventually, the system can stabilize in a state in which each agent has a value and no constraint is violated. This state is a global solution and the network has reached *quiescence*, which is to say that no message is traveling through the network. Such a state can be detected using specialized snapshot algorithms [Chandy and Lamport, 1985]. If no solution exists, an empty nogood will be generated.

5.2.2 Theoretical Results

ABT_{kernel} has the following formal properties.

Proposition 5.2.1. *ABT_{kernel} is correct.*

Proof. If a solution is claimed, we have to prove that all agents satisfy their constraints. Let us assume quiescence in the network. If the current assignments are not a solution, there exists at least one violated constraint, i.e., an agent still unsatisfied with its current assignment. In this case, at least one message has been sent from the unsatisfied agent to the nearest culprit. This message is either not obsolete, in which case the recipient will change its value and break our quiescence assumption by sending a message, or obsolete, which means that

some other message has not yet reached its destination and again breaks our assumption. □

Proposition 5.2.2. *ABT_{kernel} cannot infer inconsistency if a solution exists.*

Proof. Every nogood resulting from an **ok?** message is redundant with regard to the *DisCSP* to solve. Since all additional nogoods are generated by logical inference when an agent does not encounter a consistent value for its variable, the empty nogood cannot be inferred if the *DisCSP* is satisfiable. □

In spite of these good properties, ABT_{kernel} may fail to terminate. The problem lies in the obsolescence of nogoods. The way nogoods are generated guarantees that every variable appearing in the nogood is above $self$ in the ordering. But nothing ensures that those variables are in $\Gamma^-(self)$. This leads us to the following observation.

Lemma 5.2.3. *ABT_{kernel} may store obsolete information.*

Proof. Since a nogood may contain an unrelated agent u above $self$ in the ordering, it cannot be locally checked for obsolescence as u will not send its new value to $self$. Thus, an agent can end up storing indefinitely information that is no longer updated. □

Worse, the agent may use that information to prune a value. If there is a solution including this value, it will be missed. Since ABT_{kernel} cannot infer inconsistency if a solution exists, it will not terminate.

Lemma 5.2.4. *Storing obsolete information, ABT_{kernel} agents may fall into an infinite loop.*

Proof. Let i be an agent keeping a nogood about an unrelated agent u above i in the ordering, i.e. $x_u = a \Rightarrow x_i \neq c$. Suppose this nogood is now obsolete since x_u changed its value, and c is the only value of x_i in a solution. x_i will try all other values in its domain, find them unfeasible and generate a backtrack message. When this message will reach u, it will be discarded as obsolete, and i will continue looping on the same subdomain, sending backtrack messages which are doomed to be dropped by u. The solution will never be detected. □

Proposition 5.2.5. *ABT_{kernel} may fail to terminate.*

Proof. The proof flows naturally from lemma 5.2.3 and 5.2.4. □

If we eliminate obsolete information in finite time, that means that crucial values will not stay deleted forever. At least some of the backtracking messages will be processed, and will thus delete a value on some agent above $self$ in the ordering.

Lemma 5.2.6. *The first agent in the ordering can never fall into an infinite loop.*

Proof. Every variable in a nogood received by $self$ is above $self$ in the ordering. So, if agent 1 receives a nogood, it has an empty left-hand side. Thus, it will never become obsolete. □

Lemma 5.2.7. *If the first $k-1$ agents in the ordering are not trapped in an infinite loop and obsolete information disappears in finite time, agent k cannot fall into an infinite loop.*

Proof. Suppose agent k is looping. Since we assume that no obsolete information can last forever, some of the backtracking messages sent by k will be seen as relevant, and will lead to value deletions. Since no agent among $1, \ldots, k-1$ is supposed to be in an infinite loop, they can accept only a finite number of relevant backtracking messages. Thus, they will either stabilize, in which case k will exit its so-called infinite loop as soon as the obsolete data are deleted, or generate an empty nogood, which will also stop the entire system. Therefore, k is not in an infinite loop. □

Proposition 5.2.8. *Removing obsolete information in finite time, ABT_{kernel} will terminate.*

Proof. By recurrence, lemma 5.2.6 and lemma 5.2.7 show that none of our agents can fall into an infinite loop. So ABT_{kernel} terminates if obsolete information is erased in finite time. □

Therefore, complete algorithms based on ABT_{kernel} should be able to discard obsolete nogoods. If a nogood becomes obsolete, it may survive in the network only for a limited period of time.

5.3 The *ABT* Family

In the following, we explore ways to remove obsolete information from ABT_{kernel} in finite time, producing several correct and complete algorithms. This allows us to rediscover already existent algorithms, like *ABT* [Yokoo et al., 1998] or *DIBT* [Hamadi et al., 1998], derived from ABT_{kernel} in a clean and elegant way. A first way to remove obsolete information is to add new links thus allowing a nogood owner to determine whether a given nogood is obsolete or not. An added link from agent i to agent j can be seen as the universal constraint between x_i and x_j, permitting all value tuples. x_i should be included in $\Gamma^-(x_j)$ and x_j in $\Gamma^+(x_i)$, which implies that x_j will be informed of the x_i's value changes. These added links were proposed in the original *ABT* algorithm [Yokoo et al., 1998]. A second way to remove obsolete information is to detect when a nogood could become obsolete. In that case, the hypothetically obsolete nogood and the values of unrelated agents are forgotten. These two alternatives lead to the following four algorithms,

- *Adding links as preprocessing: ABT_{all}.* This algorithm adds all the potentially useful new links during a preprocessing phase. New links are permanent.
- *Adding links during search: ABT.* This algorithm adds new links between agents during the search. A link is requested by $self$ when it receives a

ngd message containing unrelated agents above $self$ in the ordering. New links are permanent.

- *Adding temporary links:* ABT_{temp}. This algorithm adds new links between agents during the search, as in *ABT*. The difference is that new links are temporary. This idea has been informally proposed in [Silaghi et al., 2001c]. A new link remains until a fixed number of messages have been exchanged through it. After that, it is removed.

- *No links:* ABT_{not}. No new links are added between agents. To achieve completeness, this algorithm has to remove obsolete information in finite time. To do so, when an agent backtracks it forgets all nogoods that hypothetically could become obsolete.

In the following we present each of these algorithms in additional detail.

5.3.1 ABT_{all}: Adding links as preprocessing

In a preprocessing phase, ABT_{all} adds a permanent link between every pair of unrelated agents i and j such that x_j may receive a nogood mentioning x_i during the execution of ABT_{kernel}. This is done adding exactly the same links as in the computation of the induced constraint graph from the initial ordered constraint graph [Dechter and Pearl, 1988]. These new links are computed as follows. Agents (graph nodes) are processed from last to first, according to the total ordering of agents. When an agent is processed, all its parents (related agents before it in the ordering) are connected by new links if they were not connected before. These new links are directed, following the total ordering of agents. The structure of the induced graph is recorded in the sets Γ^- and Γ^+ of each agent. During the search phase, ABT_{all} behaves exactly like ABT_{kernel}, which is now a complete algorithm because each agent is directly connected to every other agent that could appear in a nogood contained in a **ngd** message. Obsolete nogoods will be removed in finite time, so ABT_{all} is a correct and complete algorithm that terminates with a correct answer. Interestingly, it is possible to modify ABT_{all} in such way that agents do not store nogoods anymore, by fixing the agent to backtrack to the closest agent in $\Gamma^-(self)$. A somewhat erroneous form of this algorithm was published in [Hamadi et al., 1998] as the *DIBT* algorithm.

5.3.2 *ABT*: Adding links during search

Instead of linking all possible sources of conflict beforehand, we can wait until the conflict actually happens, and add a link at that point. The original *ABT* takes this approach. *ABT* uses a fourth type of message, **adl**, to request the addition of a new link. Each time an agent j receives information about a higher priority agent i previously unheard of, an **adl** message is sent. As a result, x_i extends its Γ^+ to include x_j, and sends its current value on the newly created link. This way, each agent storing a nogood is guaranteed to be informed whenever one of

the variables in the nogood changes its value. The ABT algorithm appears in Figure 5.3, only those parts which differ from ABT_{kernel}. The main procedure `ABT` includes the reception of the **adl** message (line 9.1), which is processed by `SetLink`. When a link request arrives, the sender is included in $\Gamma^+(self)$ (line 1) and $self$ sends its value through an **ok?** message (line 2). When a **ngd** message is received, `ResolveConflict` considers if a request for a new link must be sent (`CheckAddLink` call, line 2.1). Also, the condition for resending $self$ value to senders of obsolete **ngd** messages is simplified (line 5.1). `CheckAddLink` checks if unrelated agents appear in the received nogood (lines 1-2). In such case, it sends a request of new link for each unrelated agent, adding it to $\Gamma^-(self)$ (lines 3-4). Finally, it updates its agent view taking as the value of the unrelated agent the value coming in the nogood (line 5). This value will be confirmed or discarded later, when the link request will cause the just related agent to send its value to $self$.

procedure `ABT()`
1 $myValue \leftarrow$ empty; $end \leftarrow$ false;
2 `CheckAgentView`();
3 **while** ($\neg end$) **do**
4 $msg \leftarrow$ `getMsg`();
5 **switch**($msg.type$)
6 **ok?** : `ProcessInfo`(msg);
7 **ngd** : `ResolveConflict`(msg);
8 **stp** : $end \leftarrow$ true;
9.1 **adl** : `SetLink`(msg);

procedure `ResolveConflict`(msg)
1 **if** `Coherent`($msg.Nogood, \Gamma^-(self) \cup \{self\}$) **then**
2.1 `CheckAddLink`(msg);
3 `add`($msg.Nogood, myNogoodStore$); $myValue \leftarrow$ empty;
4 `CheckAgentView`();
5.1 **else if** `Coherent`($msg.Nogood, self$) **then** `sendMsg`:**ok?**($msg.sender, myValue$);

procedure `SetLink`(msg)
1 `add`($msg.sender, \Gamma^+(self)$);
2 `sendMsg`:**ok?**($msg.sender, myValue$);

procedure `CheckAddLink`(msg)
1 **for each** ($var \in$ `lhs`($msg.Nogood$))
2 **if** ($var \notin \Gamma^-(self)$) **then**
3 `sendMsg`:**adl**($var, self$);
4 `add`($var, \Gamma^-(self)$);
5 `Update`($myAgentView, var \leftarrow varValue$);

Figure 5.3: The ABT algorithm with permanent links. Only the new or modified parts with respect to ABT_{kernel} in Figure 5.2 are shown.

5.3.3 ABT_{temp}: Adding temporary links

Given that links added in ABT serve the sole purpose of informing $self$ when some nogood becomes obsolete, we may add them during search on a temporary basis. In fact, as soon as $self$ knows the new value for the linked agent, obsolete nogoods are discarded and no further information from that agent is needed at this time, so this additional link could be dropped. It may happen that future **ngd** messages will also mention this agent, so the link will have to be established again. If this happens often, it may be more efficient to keep the link active for a number of **ok?** messages, carrying the value changes of the linked agent to $self$. This is the approach taken by ABT_{temp}. When a new link is set from agent i to j, it is maintained for a fixed number k of **ok?** messages going from x_i to x_j. After this number of messages has been sent, the link is removed and agents i and j become disconnected. The number k of messages for a link is known *a priori* by both agents, so two simple counters —one in each agent— allow for an effective implementation of this technique. When reporting results the number k is essential, and then this algorithm is mentioned as $ABT_{temp}(k)$.

5.3.4 ABT_{not}: No links any more

Instead of trying hard to be informed when an unconnected agent changes its value, $self$ can study its own course of action and update its knowledge accordingly. More precisely, when all values of $self$ have been removed, a new nogood is generated and sent to the nearest culprit. $self$ knows that this nogood will possibly reach every variable it contains, forcing them all, in the worst case, to change their value. For those variables in $\Gamma^-(self)$, there is no need to worry, because they are bound to inform $self$. For the others, the very action of backtracking can lead to the obsolescence of any nogood inside which they appear. Hence, $self$ will forget those insecure variables and nogoods upon backtracking. There are two cases which deserve some attention. First, it may happen that a forgotten nogood does not become obsolete after all. If $self$ takes the value that this nogood was removing, then $self$ will necessarily receive again this nogood, rediscovered by a lower priority agent. Second, it may happen that a nogood becomes obsolete because an unrelated, higher priority agent has changed its value and $self$ has not been notified. If the value suppressed by the obsolete nogood is not mandatory to find a solution, this mistake does not compromise finding a solution. On the contrary, if that value is mandatory, $self$ will be forced to try every other value in its domain before backtracking. A new nogood resolving all nogoods removing $self$ values will be produced. This nogood will include the agent that had changed its value, so when sending the **ngd** message, its value will be forgotten and search will be resumed. The ABT_{not} algorithm takes this approach. This algorithm was described in [Bessière et al., 2001], under the name *DisDB*. We call it here ABT_{not} to follow our scheme. ABT_{not} only differs from ABT_{kernel} in the forgetting policy of nogoods that could become obsolete, and this concerns the procedure `Backtrack` that appears in Figure 5.4. This procedure computes the new nogood as the resolvent of the nogoods that justify

```
procedure Backtrack()
1 newNogood ← solve(myNogoodStore);
2 if (newNogood = empty) then
3   end ← true; sendMsg:stp(system);
4 else
5   sendMsg:ngd(newNogood);
6   Update(myAgentView,rhs(newNogood) ← unknown);
6.1 for each var ∈ lhs(newNogood) \ Γ⁻(self) do
        Update(myAgentView, var ← unknown);
7   CheckAgentView();
```

Figure 5.4: The ABT_{not} algorithm with no links. Only the new or modified parts with respect to ABT_{kernel} in Figure 5.2 are shown.

that every value of $self$ is forbidden. If the new nogood is not empty, it is sent in a **ngd** message. Then, $self$ forgets the values of agents not in $\Gamma^-(self)$, and the nogoods including those agents (line 6.1). Finally, a new value consistent with the agent view is searched.

5.3.5 Discussion

Consider two agents i and j (i before j in the ordering) no originally constrained but connected in the induced constraint graph. Previous algorithms differ in the way information flows between these two agents. If i takes a new assignment, we say that j is informed about this new assignment when j knows it. The cost of informing j is the minimum number of messages required since i takes the new assignment until j is aware of it. Algorithm A is *better informed* than algorithm B if, for the same problem and the same agent ordering, the cost of informing j of i changes using A is less than or equal to the cost of informing j using B. From this concept, we observe a monotonic decrement in the quality of information handled by the ABT family algorithms, from ABT_{all} to ABT_{not}. ABT_{all} is better informed than ABT because both behave in the same way except when ABT detects a conflict between i and j for first time. In this case, ABT requires more messages than ABT_{all}. ABT is better informed than ABT_{temp} since the latter requires some extra messages to set up again temporary links. And ABT_{temp} is better informed than ABT_{not} because the former could inform j in one or two messages, while the latter always requires at least two messages.

5.4 Implementation Details

In this section we propose some new ideas that could improve the performance (regarding communication and computation cost) of the ABT-like algorithms. In order to decrease the number of the exchanged messages, we implemented the ABT family algorithms considering the following two improvements,

1. *Value in **adl**.* When a new link with agent k is requested by $self$, instead of sending the **adl** message and wait for answer, ABT and ABT_{temp} include in the **adl** message the value of x_k recorded in the received nogood. After reception of the **adl** message, agent k informs $self$ of its current value only if it is different from the value contained in the **adl** message. In this way, some **ok?** messages can be saved.

2. *Avoid resending same values.* ABT family algorithms keep track of the last value taken by $self$. When selecting a new value, if it happens that the new value is the same as the last value, $self$ does not resend it to $\Gamma^+(self)$, because this information is already known. (See line 3 of `CheckAgentView` in Fig. 5.2.) Again, this may save some **ok?** messages.

3. *Sequence numbers ($ABT_{not}(seq)$).* It is possible to enhance slightly the quality of the information stored by ABT_{not} in the agent view, as follows. Each agent keeps a sequence number, which is incremented each time its value changes. Each time it sends its value, the sequence number is attached. The agent view stores the values and sequence numbers of previous agents in the ordering. When $self$ receives a message, it keeps the newest value for each variable in its agent view. In particular, a **ngd** message is discarded as obsolete if it contains older values than those recorded in $self$'s agent view. When $self$ sends a **ngd** message, the computed nogood contains the values and sequence numbers of involved variables, forgetting the values of unconnected variables but keeping their sequence numbers.

The network load and search effort could be reduced if agents store the "best" nogood as a justification of a forbidden value. In the next subsection this idea is discussed in details.

5.4.1 Selecting the Best Nogood

Regarding polynomial space asynchronous algorithms, agents can store a constant number of nogood for removed value. However, if several nogoods are available for each value, it may be advisable to choose the most appropriate resolvent in order to speed up search. Unfortunately, in the most general case, selecting the most suitable nogoods with respect to one particular criterion (or set thereof) means generating all possible candidates in order to extract the best one, which could be prohibitively expensive. Heuristics are usually considered to tackle such issues: a polynomial-time process, although unable to find the best candidate, should help to select a worthy candidate in order to make search more accurate. In this case, when comparing two nogoods we have devised the following heuristic: select the nogood with *the highest possible lowest variable involved*. The rationale for this heuristic is to ensure that each time an agent discovers that it does not have a consistent value in its domain, the **ngd** message is sent as high as possible in the agent ordering, thus saving unnecessary search effort. A similar idea was proposed in [K. and Yokoo, 2000].

Our proposed heuristic selects *the best nogood* after a new nogood is received without generating all possible nogoods. The new nogood is compared with the stored nogood choosing the one which has the higher possible lowest variable involved. When any agent in ABT accepts a **ngd** message, the incoming nogood is coherent with its whole view, including its own assignment. Once this nogood is stored, the local value it refers to is eliminated, which makes the nogood coherent with the whole agent view except for the local assignment. It is *all-but-self* relevant. If *self* receives a **ngd** message with a nogood coherent with its agent view but not with its own assignment, this nogood is *all-but-self* relevant. In this case, this nogood deserves to be considered because it brings valuable information: it gives a valid reason to discard a value, even if that value may already has been discarded. Thus, the incoming nogood has to be compared against the current nogood for its target value, and replace it if it is better from the heuristic point of view.

5.5 Experimental Results

We have tested ABT algorithms, with or without the heuristic of selecting the best nogood, on distributed random *DisCSP*. In our experiments, we have generated instances of 16 agents and 8 values per agent, considering two connectivity classes, sparse ($p_1 = 0.2$) and medium ($p_1 = 0.5$). Experiments are at the complexity peak. Specifically, we tested the random classes $\langle 16, 8, 0.2, 0.7\rangle$ (20 solvable instances out of 50) and $\langle 16, 8, 0.5, 0.42\rangle$ (27 solvable instances out of 50). In Tables 5.1 and 5.2, we report the number of non-concurrent constraint checks ($nccc$) and the total number of messages exchanged (msg), averaged over 100 instances.

Table 5.1 contains the results for the plain ABT algorithms. The parameter k for ABT_{temp} was adjusted manually after some trials. Only the results for the best value of k are given. Considering the three algorithms adding links, ABT_{all}, ABT, and ABT_{temp}, we observe that the better informed the algorithm is, the less non-concurrent constraint checks it required to solve the problem. This is at the cost of exchanging more messages. ABT_{temp} is the algorithm exchanging less messages, followed by ABT and ABT_{all}. ABT_{not} requires the highest number of non-concurrent constraint checks. Because it is the worst informed algorithm, it is more likely to make wrong decisions, requiring more effort than previous algorithms to solve the same problem. This also implies a higher number of messages exchanged. $ABT_{not}(seq)$ dominates ABT_{not} because sequence numbers avoid some of the wrong decisions taken by ABT_{not}.

Table 5.2 shows the effect of using the nogood selection heuristic. We observe that the number of non-concurrent constraint checks and the number of exchanged messages decreases consistently for all the algorithms, showing the benefits of the heuristic. The relative performance of the algorithms remains unchanged with respect to the plain versions.

Although ABT_{not} does not add links between any pair of agents not sharing constraints, it may happen that the agent with lower priority backtracks to the

p_1=0.20	nccc	msg	p_1=0.50	nccc	msg
ABT_{all}	4,113	5,060	ABT_{all}	45,917	42,864
ABT	4,077	4,831	ABT	45,571	42,234
$ABT_{temp}(10)$	4,014	4,709	$ABT_{temp}(5)$	44,782	39,688
ABT_{not}	14,908	19,168	ABT_{not}	53,145	55,869
$ABT_{not}(seq)$	9,491	12,243	$ABT_{not}(seq)$	47,402	49,857

Table 5.1: Plain ABTs for solving random *CSP*.

p_1=0.20	nccc	msg	p_1=0.50	nccc	msg
ABT_{all}	3,701	4,561	ABT_{all}	42,283	39,422
ABT	3,795	4,482	ABT	42,333	39,223
$ABT_{temp}(10)$	3,731	4,329	$ABT_{temp}(5)$	41,056	36,171
ABT_{not}	10,319	13,028	ABT_{not}	45,320	50,284
$ABT_{not}(seq)$	8,127	9,431	$ABT_{not}(seq)$	43,981	48,907

Table 5.2: ABTs with nogood selection heuristic for solving random *CSP*.

agent with higher priority. Thus, the lower priority agent may know the valuation of the higher priority one, received via a **ngd** message from an agent connected to the higher priority agent. In Table 5.3, we report the total number of different values that are revealed via added links in ABT_{all}, ABT and ABT_{temp} and via **ngd** messages in ABT_{not}. For both tested classes, we observe that the more informed an algorithm is, the larger number of values it exchanges among unrelated agents. ABT_{all} and ABT_{not}, both with and without nogood selection heuristic, are the algorithms that show the worst and best results, respectively. Again, the use of sequence numbers in ABT_{not} slightly improves the original ABT_{not} algorithm.

p_1=0.20	added links	plain	nogood selection heuristic	p_1=0.50	added links	plain	nogood selection heuristic
ABT_{all}	26	193	193	ABT_{all}	50	301	301
ABT	26	182	177	ABT	40	286	284
$ABT_{temp}(10)$	26	171	168	$ABT_{temp}(5)$	40	275	273
ABT_{not}	-	126	124	ABT_{not}	-	255	253
$ABT_{not}(seq)$	-	120	119	$ABT_{not}(seq)$	-	250	249

Table 5.3: Number of values revealed through added new links in ABT_{all}, ABT, ABT_{temp} and through **ngd** messages in ABT_{not}.

From the above results, we observe the following facts. Regarding computation effort, consistently for all problems, the more informed an algorithm is, the smaller *nccc* it requires. Regarding communication cost, the dynamic links of ABT improve over the static approach of ABT_{all}. Temporary links of ABT_{temp} dominate the permanent link approach of ABT. ABT_{not}, the algorithm not adding links, has the worst results in both of *nccc* and *msg*, showing a slight improvement when sequence numbers are used ($ABT_{not}(seq)$). Regarding the information revealed among unrelated agents, ABT_{not} improves the rest of ABT family algorithms. This leads us to conclude that ABT_{not} has to be selected only

if some privacy policy justifies its use. Regarding the nogood selection heuristic, we observe clear benefits.

5.6 Summary

In this Chapter, we proposed ABT_{kernel}, a simple basic procedure for asynchronous backtracking search. We proved that ABT_{kernel} is correct but does not guarantee termination. We have presented some extensions of ABT_{kernel} that handle nogoods and links in a way such that termination is ensured. They differ only in their extensions with respect to the basic kernel. These extended algorithms are: ABT, ABT_{all}, ABT_{temp} and ABT_{not}, the first asynchronous algorithm that does not add links between agents not sharing constraints. In addition to presenting the ABT family, throughout this chapter we have discussed some improvements such as the heuristic of selecting the best nogood. Finally, we compared experimentally the derived algorithms from the ABT_{kernel} on distributed random problems. Our experimental results that the earlier links are added between agents not sharing constraints, the smaller number of messages the algorithm needs. On the other hand, the longer the duration of added links is, the greater number of messages the algorithm needs. Although ABT_{not} is the least economic algorithm, it exchanges much less information between agents not sharing constraints.

Chapter 6

Synchronous versus Asynchronous Backtracking

Distributed algorithms are divided in two main classes: synchronous and asynchronous. There was some debate around the efficiency of these two types of algorithms for *DisCSP*. The general opinion was that asynchronous algorithms were more efficient than the synchronous ones, because of their higher concurrency.[1] In the last decade, attention was mainly devoted to the study and development of asynchronous procedures, which represented a new approach with respect to synchronous ones, directly derived from centralized algorithms. In this chapter, we continue this line of research, studying the effect of fully asynchronous search in the context of *ABT*.

ABT agents assign values to their variables and exchange messages asynchronously and concurrently. When an agent sends a backtracking message, it continues working without waiting for an answer. This strategy may result costly in some cases because performing two tasks concurrently could be inefficient if these tasks keep a dependency relation between them. In this chapter, we identify a case in which *ABT*'s efficiency can be improved if, after backtracking, an agent waits for receiving a message showing the effect of backtracking on higher priority agents. We implement this idea on ABT_{hyb}, a new *ABT*-like algorithm, that combines asynchronous and synchronous elements to avoid redundant messages.

This chapter is organized as follows. We identify a source of inefficiency in *ABT* in Section 6.1. To overcome this, we present ABT_{hyb}, the new hybrid algorithm, in Section 6.2. We provide some theoretical results of ABT_{hyb} in Section 6.3. We also propose a formal protocol to allow agents, in asynchronous and hybrid algorithms, to process messages by packets instead of one by one in Section 6.4. Empirically, we compare synchronous, asynchronous and hybrid

[1] However, a careful reading of [Yokoo et al., 1998] shows that "synchronous backtracking might be as efficient as asynchronous backtracking due to the communication overhead" (footnote 15).

backtracking algorithms in Section 6.5. Finally, we resume the main proposal we present in this chapter in Section 6.7.

6.1 Asynchrony in *ABT*

During the resolution of *DisCSP*, the largest number of messages exchanged by *ABT* agents are **ok?** and **ngd** messages. **ok?** messages are always accepted. However, some **ngd** messages may be discarded as obsolete when they arrive to the receiver. *ABT* could save some work if these messages were not sent. Although the sender agent cannot detect which messages will become obsolete when reaching the receiver, it is possible to avoid sending those which are redundant.

When agent j sends a **ngd** message, it performs a new assignment and informs about it to lower priority agents, without waiting the reception of any message showing the effect of the just sent **ngd** on higher agents. This can be a source of inefficiency in the following situation. If k sends a **ngd** message to j causing j to have no consistent value, then j sends a **ngd** message to some previous agent i. If j takes the same value as before and sends an **ok?** message to k before i changes its value, k will find again the same inconsistency so it will send the same nogood to j in a **ngd** message. Agent j will discard this message as obsolete, sending again its value in an **ok?** message. The process is repeated generating useless messages, until some higher variable changes its value and the corresponding **ok?** message arrives to j and k. In the next section we propose a novel algorithm to avoid sending these redundant messages.

6.2 The *ABT* Hybrid Algorithm

Based on the intuition described above, we present ABT_{hyb}, a hybrid algorithm that combines asynchronous and synchronous elements. ABT_{hyb} behaves like *ABT* when no backtracking is performed: agents take their values asynchronously and inform lower priority agents. However, when an agent has to backtrack, it does it synchronously as follows. If k has no value consistent with its agent view, it sends a **ngd** message to j and enters in a *waiting* state. In this state, k has no assigned value, and it does not send out any message. Any received **ok?** message is accepted, updating k's agent view accordingly. Any received **ngd** message is treated as obsolete, since k has no value assigned. Agent k leaves the waiting state when receiving one the following messages:

1. an **ok?** message that breaks the nogood sent by k or,
2. an **ok?** message from j, the receiver of the last **ngd** message or,
3. a **stp** message informing that the problem has not solution.

The justification for leaving the waiting state follows. Case (1) is the confirmation that the **ngd** message has generated a change in a higher priority agent that

```
procedure ABT-hyb()
  myValue ← empty; end ← false; wait ← false;
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?    : ProcessInfo(msg);
      ngd    : if ¬wait then ResolveConflict(msg);
      adl    : SetLink(msg);
      stp    : wait ← false; end ← true;
 procedure ProcessInfo(msg)
  Update(myAgentView, msg.Assig);
  if wait then
    if (msg.sender ∈ rhs(lastNogood)) ∨ (msg.sender ∈ lhs(lastNogood)∧
          msg.Assig ≠ lastNogood[msg.sender]) then wait ← false;
  if ¬wait then CheckAgentView();
 procedure SetLink(msg)
  add(msg.sender, Γ+(self));
  if ¬wait then sendMsg:ok?(msg.sender, myValue);
 procedure Backtrack()
  newNogood ← solve(myNogoodStore);
  if (newNogood = empty) then
    end ← true; sendMsg:stp(system);
  else
    sendMsg:ngd(newNogood);
    lastNogood ← newNogood; wait ← true;
```

Figure 6.1: The ABT_{hyb} algorithm for *DisCSP*. Only the new or modified parts with respect to ABT in Figure 5.3 are shown.

breaks the nogood. So k has to leave the waiting state, returning to ordinary ABT operation. Case (2) considers the situation in which k has a more updated information than j. Then, until j does not receive the updated information, it will reject the **ngd** message as obsolete and resend to k its value in an **ok?** message. After receiving it, if k remains in the waiting state, the communication with j might be broken, since j may say nothing when receiving the updated information, k will have no notice of this updated information and the algorithm would be incomplete. Therefore, k has to leave the waiting state, just to rediscover the same nogood, send it to j and enter in the waiting state again. This loop breaks when the updated information reaches j: it will no longer reject the **ngd** message because it is not obsolete according to its updated agent view. Case (3) is the reception of a **stp** message (the empty nogood has been generated somewhere), so every agent has to finish its execution.

At this point, ABT_{hyb} switches to ABT. ABT_{hyb} detects that a *DisCSP* is unsolvable if during the resolution an empty nogood is derived. Otherwise, ABT_{hyb} claims that it has found a solution when no messages are traveling through the network (i.e.quiescense is reached in the network).

The ABT_{hyb} algorithm appears in Figure 6.1. Its difference with the code of ABT, given in Figure 5.3 (Chapter 5), is around variable *wait*, that appears in `ABT-hyb`, `ProcessInfo`, `Backtrack` and `SetLink`.

6.3 ABT_{hyb}: Theoretical Results

6.3.1 Correctness, Completeness and Termination

No matter synchronous points introduced, ABT_{hyb} inherits the good theoretical properties of ABT, namely correctness, completeness and termination. To proof these properties, we start with some lemmas.

Lemma 6.3.1. *In ABT_{hyb}, no agent will continue in a waiting state forever.*

Proof. In ABT_{hyb}, an agent enters the waiting state after sending a **ngd** message to a higher priority agent. The first agent (x_1) in the ordering will not enter in the waiting state because no **ngd** message departs from it. Suppose that no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, and suppose that x_k enters the waiting state after sending a **ngd** message to x_j ($1 \leq j \leq k-1$). We will show that x_k will not be forever in the waiting state.

When x_j receives the **ngd** message, there are two possible states:

1. x_j is waiting. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point. If x_j has a value consistent with its new agent view, it will send it to x_k in an **ok?** message. If x_j has no value consistent with its new agent view, it will backtrack and enter again in a waiting state. This can be done a finite number of times (because there is a finite number of values per variable) before finding a consistent value or discovering that the problem has no solution generating an **stp** message. In both cases, x_k will leave the waiting state.

2. x_j is not waiting. The **ngd** message could be:

   (a) Obsolete in the value of x_j. In this case, there is an **ok?** message traveling from x_j to x_k that has not arrived to x_k. After receiving such a message, x_k will leave the waiting state.

   (b) Obsolete not in the value of x_j. In this case, x_j resends to x_k its value by an **ok?** message. After receiving such a message, x_k will leave the waiting state.

   (c) Not obsolete. The value of x_j is forbidden by the nogood in the **ngd** message, and a new value is tried. If x_j finds another value consistent with its agent view, it takes it and sends an **ok?** message to x_k, which will leave the waiting state. Otherwise, x_j has to backtrack to a previous agent in the ordering, and enters the waiting state. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point, and as explained in the point 1 above, it will cause that x_k will leave the waiting state as well.

Therefore, we conclude that x_k will not stay forever in the waiting state. □

Lemma 6.3.2. *In ABT_{hyb}, if an agent is in a waiting state, the network is not quiescent.*

Proof. An agent is in a waiting state after sending a **ngd** message. Because Lemma 6.3.1, this agent will leave the waiting state in finite time. This is done after receiving an **ok?** or **stp** message. Therefore, if there is an agent in a waiting state, the network cannot be quiescent at least until one of those messages has been produced. □

Lemma 6.3.3. *A nogood, discarded as obsolete because the receiver is in a waiting state, will be resent to the receiver until the sender realizes that it has been solved, or the empty nogood has been derived.*

Proof. If an agent k sends a nogood to an agent j that is in a waiting state, this nogood is discarded and agent k enters the waiting state. From Lemma 6.3.1, no agent can stay forever in a waiting state, so agent k will leave that state in finite time. This is done after receiving either,

- An **ok?** message from j. If this message does not solve the nogood, it will be generated and resend to j. If it solves it, this nogood is not generated, exactly in the same way as ABT does.
- An **ok?** message allowing a consistent value for k. In this case, the nogood is solved, so it is not resent again.
- A **stp** message. The process terminates without solution.

Therefore, we conclude that the nogood is sent again until it is solved (either by an **ok?** message from j or from another agent) or the empty nogood is generated. □

Proposition 6.3.4. *ABT_{hyb} is correct.*

Proof. From Lemma 6.3.2, ABT_{hyb} reaches quiescence when no agent is in a waiting state. From this fact, ABT_{hyb} correctness derives directly from ABT correctness: when the network is quiescent all agents satisfy their constraints, so the current assignments of agents form a solution. If this would not be the case, at least one agent would detect a violated constraint and it would send a message, breaking the quiescence assumption. □

Proposition 6.3.5. *ABT_{hyb} is complete and terminates.*

Proof. From Lemma 6.3.3, the synchronicity of backtracking in ABT_{hyb} does not cause to ignore any nogood. Then, ABT_{hyb} explores the search space as good as ABT does. From this fact, ABT_{hyb} completeness comes directly from ABT completeness. New nogoods are generated by logical inference from the initial constraints, so the empty nogood cannot be derived if there is a solution. Total agent ordering causes that backtracking discards one value in the highest

variable reached by the **ngd** message. Since the number of values is finite, the process will find a solution if it exists, or it will derive the empty nogood otherwise.

To see that ABT_{hyb} terminates, we have to prove that no agent falls into an infinite loop. This comes from the fact that agents cannot stay forever in the waiting state (Lemma 6.3.1), and that ABT agents cannot be in an endless loop. □

6.3.2 Comparison with *ABT*

In practice (see Section 6.5), we observed that ABT_{hyb} improves ABT performance. However, there are some cases where the number of messages required by ABT is shorter than the number of messages required by ABT_{hyb}. As an example, let us consider the following instance,

- 8 variables ($\{x_1, \ldots, x_8\}$) with the domains of values: $\{a, b, c\}$;
- 8 agents ($\{A_1, \ldots, A_8\}$), each one holding one variable (x_i belongs to A_i, $1 \leq i \leq 8$);
- 28 constraints, one per pair of variables.

For ABT and ABT_{hyb}, we consider the same network conditions (i.e. messages are received by agents in the same order in both algorithms), the same priority ordering of agents (lexicographical) and the same strategy of selection of values. The algorithms start when each agent assigns a value to its variable and informs the assignment to the agents with lower priority via **ok?** messages. In our example, when A_7 receives the assignments of the previous agents and a **ngd** message from A_8, it does not find a consistent value in its domain. The first value of x_7's domain is forbidden by a conflict found and sent by A_8. The second and the third values are forbidden by the constraint c_{27}, which restricts the values of x_2 and x_7. Thus, the nogood store of A_7 has the form:

$$x_2 = a \wedge x_3 = b \wedge x_4 = c \wedge x_5 = b \wedge x_6 = a \Rightarrow x_7 \neq a$$

$$x_2 = a \Rightarrow x_7 \neq \{b, c\}$$

In this situation both algorithms take the same action: Agent A_7 backtracks to A_6, which is the lowest priority agent involved in the new nogood obtained by the resolution of the nogood store of A_7. After sending the **ngd** message, in ABT_{hyb} agent A_7 enters a waiting state and x_7 will remain unassigned until leaving that state; in ABT it discards x_6's value (the variable belonging to recipient of the **ngd** message sent by A_7) and every nogood in A_7's nogood store which mention x_6. As result, value a of x_7 becomes permitted, and A_7 assigns a to x_7.

Afterwards the conflict was discovered by A_6, in ABT_{hyb} the number of messages sent is 1 (1 **ngd** message from A_7 to A_6), while in ABT is 2 messages (1 **ngd** message from A_7 to A_6 and 1 **ok?** from A_6 to A_7) in ABT. Let us assume

that the conflict sent by A_7 to A_6 (from now on, referred as conflict α) will be solved when A_2 changes the value of x_2. It may happen that before α is solved, A_7 receives an assignment from A_1, which forbids A_7 to assign value a to x_7. The new conflict (referred as β) will be considered by agent A_7 in ABT_{hyb} when α has been solved. In contrast, in ABT, both conflicts are going to be tried in parallel. In ABT, conflict β causes A_7 to assign value b to x_7. Since this value is new, A_7 has to send a **ok?** message informing A_8 of the new assignment.

In ABT, suppose that β is solved by A_2 before α arrives to A_6. If β has been solved, A_2 has assigned a new value to x_2, which in this example, it is also consistent with all the current values of lower priority agents. Therefore, A_2 has sent several messages (6 **ok?** messages) to inform lower priority agents ($\{A_3, \ldots, A_8\}$) about x_2's value. These messages do not cause the recipients to send out any more message. In A_6, the new assignment of x_2 will arrive first that the **ngd** message containing α. Therefore, A_6 detects that the nogood is obsolete because x_2's value. This causes A_6 to send again its assignment to A_7 (1 **ok?** message). In total, the number of messages sent by ABT agents is 17: 1 **ngd** message from A_7 to A_6; 7 **ok?** messages from A_1 to lower priority agents; 1 **ngd** message from A_7 to A_2; 1 **ok?** message from A_7 to A_8; 6 **ok?** messages from A_2 to lower priority agents; 1 **ok?** message from A_6 to A_7.

In ABT_{hyb}, an agent resolves one conflict after the other. A_7 encounters first α and tries to solve it. The resolution of α implies that all the agents between A_7 and A_2 will enter in a waiting state before x_2 takes a new value. This implies several backtracking messages (5 **ngd** messages). Similar to ABT, A_2 has to inform its lower priority agents of the new assignment for x_2 (6 **ok?** messages). During this period of time, 7 **ok?** messages will be sent from A_1 to lower priority agents (the assignment that proves conflict β). Since we have assumed that both α and β will be solved when A_2 assigns a new value to x_2, β will be solved as soon as α is solved. In total, the number of messages sent by ABT_{hyb} agents is 18: 5 **ngd** messages from A_7 to A_6, A_6 to A_5, A_5 to A_4, A_4 to A_3 and A_3 to A_2; 6 **ok?** messages from A_2 to lower priority agents; 7 **ok?** messages from A_1 to lower priority agents. In this example, ABT_{hyb} needs one message more than ABT to solve a pair of conflicts. This occurs since an ABT agent tries to solve several conflicts concurrently while an ABT_{hyb} agent considers one by one.

Next, we prove that if an ABT agent does not find another conflict when a previous one has not been solved yet, then the number of messages needed by ABT_{hyb} to solve this conflict is less or equal as the number of messages required by ABT.

Proposition 6.3.6. *The number of messages required by ABT_{hyb} to solve a conflict is the same to or less than the number of messages that ABT needs, if during the resolution of the first conflict no other conflict will be found among conflicting variables.*

Proof. We prove this property by counting the number of **ok?** and **ngd** messages sent by conflicting agents. Let be agents A_j and A_i ($A_j < A_i$) the agent that finds the conflict and the agent that resolves it after changing the value of its local variable, respectively. Let S the set of agents $\mathrm{S} = \{A_{i+1} \ldots A_j\}$.

Regarding **ngd** messages, before A_i receives the **ngd** message that forces it to change the value of its variable, agents in S have sent several backtracking messages. In ABT, after sending each one of these backtracking messages, the sender has to assign a new value to its variable without knowing the new value that A_i's variable will take. Since we have assumed that no any other conflict between the agents in S will be discovered before the first conflict is solved, these assignments will not produce any **ngd** message in ABT. Therefore, the number of **ngd** messages exchanged by agents in S is the same number in both algorithms.

Regarding **ok?** messages, each ABT_{hyb} agent in S will leave the waiting state after receiving one of the following messages: (1) an **ok?** announcing the new assignment of A_i's variable, (2) an **ok?** announcing the new assignment of any higher priority agents in S that has abandoned the waiting state or (3) a **stp** message. Next, we count the number of messages sent by A_k, an agent in S, considering the above three cases.

1. In ABT_{hyb}, A_k takes a value consistent with the assignment of A_i and sends 1 **ok?** message to each lower priority agent. In ABT, after backtracking, A_k has to assign a value to its variable without knowing the new value of A_i's variable. This value may be consistent or inconsistent with the value taken by A_i. If it is consistent, A_k will send 1 **ok?** message for each lower priority agent. Therefore, the number of messages sent by A_k in both algorithms is the same. If the value of A_k is inconsistent with the new assignment of A_j's, A_k will send 2 **ok?** messages for each lower priority agent: one message because the inconsistent value and the other after assigning a value consistent with A_i's assignment. Therefore, the number of messages sent by A_k in ABT_{hyb} is less than or equal to the number of messages sent by A_k in ABT.

2. In ABT_{hyb}, if A_k takes a new value before receiving the assignment of A_i, this means that the assignment may be consistent or inconsistent. If A_k's assignment is consistent with the new value of A_i's variable, then A_k will send one **ok?** message for each lower priority agents. Otherwise, it will send two **ok?** messages for each lower priority agent. This is the same situation that happens in ABT discussed in the previous point. Therefore, the number of messages sent by A_k is the same in both algorithms.

3. In both algorithms when agent A_k receives a **stp** message, this means that the search ends because the problem is unsolvable, therefore A_k does not send any more message. Thus, the number of message sent by A_k in both algorithms is the same.

For the three cases, no agents in S never sends more messages in ABT_{hyb} than in ABT. Thus, the number of messages sent by ABT_{hyb} agents appearing in the conflict is less than or equal to the number of messages sent by the same agents in ABT. □

6.4 Processing Messages by Packets

In *ABT*, agents can process messages one by one, reacting as soon as a message is received. However, this strategy of *single-message process* may cause to perform some useless work. For instance, consider the reception of an **ok?** message reporting a change of an agent value, immediately followed by another **ok?** from the same agent. Processing the first message causes some work that becomes useless as soon as the second message arrives. More complex examples (involving **ok?** and **ngd** messages) can be devised, causing to waste substantial effort. To prevent this kind of useless work, we consider an alternative strategy. Instead of reacting after each received message, the algorithm reads all messages that are in the input buffer and stores them in internal data structures. Then, the algorithm processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message.

We call this strategy *processing messages by packets*, where a packet is the set of messages that are read from the input buffer until it becomes empty. Somehow, this idea was mentioned in [Yokoo et al., 1998] and [Zivan and Meisels, 2003]. In the latter, a comparison between processing messages one by one and processing messages by packets is presented. However, in none of them a formal protocol for processing messages by packets is completely developed. Instead of reading and processing only one message, when an agent processes messages by packets, it has to read all its messages from the input buffer, and processing them as a whole. It means that an agent looks for any consistent value after its agent view and its nogood store are updated with these incoming messages. Based on that, we propose a protocol for processing messages by packets in *ABT*.[2] This protocol requires three lists to store the incoming messages: *ok?-list*, *ngd-list* and the *adl-list*. In each list we store the messages of the corresponding type, following the reception order. Each packet of messages is processed as follows:

1. *ok?-list*. First, the *ok?-list* is processed. For each sender agent, all **ok?** messages but the last are ignored. The remaining **ok?** messages update $self$ agent view, removing nogoods if needed.

2. *ngd-list*. Second, the *ngd-list* is processed. Obsolete **ngd** messages are ignored. $self$ stores nogoods of no obsolete messages, and it sends **adl** messages to unrelated agents appearing in those nogoods. For those messages containing the correct current value of $self$, the sender is recorded in *RemainderSet*.

3. *adl-list*. Third, the *adl-list* is processed updating $\Gamma^+(self)$ without sending the **ok?** message.

4. Consistent value. Fourth, $self$ tries to find a value consistency with the agent view. If such value does not exit, $self$ will backtrack to a lower

[2] Similarly, in the rest of algorithms that form the *ABT* family, agents may process messages by packets.

priority agent, sending a **ngd** message, and again, it will search a consistent value.

5. **ok?** sent. Fifth, **ok?** messages containing $self$ current value are sent to all agents in $\Gamma^+(self)$ and to all agents in *RemainderSet*. The three lists become empty.

The search ends when quiescence is reached (i.e. all agents are happy with their current assignment) or an empty nogood is derived.

6.5 Experimental Results

We have tested ABT and ABT_{hyb} algorithms on the same instances of the distributed n-queens problem and random binary problem that we studied in Chapter 4. This allows us to compare the experimental results of synchronous against asynchronous algorithms on those problems. In both algorithms, agents process messages by packets instead of processing one by one. In addition, when an agent detects that it has multiple justifications for a forbidden value it follows the strategy of selecting the *best nogood* discussed in Chapter 5. Both algorithms also include the implementation details explained in Section 5.4 of the same chapter.

Algorithmic performance is evaluated according to the computation effort, in terms of non-concurrent constraint checks ($nccc$), and the communication costs, in terms of the total number of messages exchanged among agents (msg).

6.5.1 Distributed n-queens Problem

We have evaluated ABT and ABT_{hyb} algorithms for 4 dimensions (n = 10,15,20,25) of the n-queens problem. Table 6.1 shows the results in terms of $nccc$ and msg, averaged over 100 executions with different random seeds (ties are broken randomly). Three value ordering heuristics have been tested *lex* (lexicographic), *rand* (random) and *min* (min-conflicts) [Minton et al., 1992] on both algorithms. Likewise for $SCBJ$ in Chapter 4, we have made an approximation of *min*, for avoiding extra messages. This is approximation is computed in a preprocessing step and consists of computing the heuristic assuming initial domains.

We observe that the random value ordering provides the best performance for every algorithm and every dimension tested. Considering the relative performance of these algorithms, ABT_{hyb} is always better than ABT, in both $nccc$ and msg.

It is relevant to scrutinize the improvement of ABT_{hyb} over ABT with respect to the type of messages. In Table 6.2, we provide the total number of messages per message type for $SCBJ$ (taken from the experimentation results given in chapter 4), ABT and ABT_{hyb} with random value ordering. In ABT_{hyb}, the number of obsolete **ngd** messages decreases in one order of magnitude with respect the same type of messages in ABT, causing ABT_{hyb} to improve over

lex	ABT		ABT_{hyb}	
n	$nccc$	msg	$nccc$	msg
10	2,223	740	1,699	502
15	56,412	13,978	32,373	6,881
20	11,084,012	2,198,304	6,086,376	995,902
25	3,868,136	693,832	1,660,448	271,092
rand	ABT		ABT_{hyb}	
n	$nccc$	msg	$nccc$	msg
10	1,742	332	916	238
15	7,697	1,185	4,007	786
20	20,661	4,772	15,720	2,748
25	31,849	6,553	27,055	3,863
min	ABT		ABT_{hyb}	
n	$nccc$	msg	$nccc$	msg
10	3,716	896	2,988	555
15	49,442	11,055	32,303	5,906
20	320,278	63,378	165,338	28,686
25	38,450,786	6,716,505	17,614,330	2,795,319

Table 6.1: Results for the distributed n-queens problem with *lex*, *rand* and *min* value ordering approaches.

ABT. However, this improvement goes beyond the savings in obsolete **ngd** messages, because **ok?** and **ngd** messages decrement to a larger extent. This is due to the following collective effect. When an ABT agent sends a **ngd** message, it tries to get a new consistent value without knowing the effect that backtracking causes in higher priority agents. If it finds such a consistent value, it informs to lower priority agents using **ok?** messages. If it happens that this value is not consistent with new values that backtracking causes in higher priority agents, these **ok?** messages would be useless, and new **ngd** messages would be generated. ABT_{hyb} tries to avoid this situation. When an ABT_{hyb} agent sends a **ngd** message, it waits until it receives notice of the effect of backtracking in higher priority agents. When it leaves the waiting state, it tries to get a new consistent value. At this point, it knows some effect of the backtracking on higher priority agents, so the new value will be consistent with it. In this way, the new value has more chance to be consistent with all higher priority agents, and the **ok?** messages carrying it will be more likely to make useful work.

Considering the performance of synchronous versus asynchronous backtracking algorithms (Table 4.1 in Section 4.4 vs. Table 6.1 here), we compare $SCBJ$ against ABT_{hyb} with random value ordering. In terms of computation effort $SCBJ$ performs better than ABT_{hyb} for $n = 25$ and worse for $n = 20$, with very similar results for $n = 10, 15$. In terms of communication cost, $SCBJ$ uses

rand	$SCBJ$		ABT			ABT_{hyb}		
n	**ok?**	**ngd**	**ok?**	**ngd**	$obso$	**ok?**	**ngd**	$obso$
10	55	36	251	81	24	195	43	2
15	146	101	901	284	91	649	137	10
20	539	382	3,612	1,160	408	2,293	455	38
25	452	294	5,027	1,526	520	3,240	623	50

Table 6.2: Number of messages exchanged by $SCBJ$, ABT and ABT_{hyb} per message type, for the distributed n-queens problem with random value ordering.

less messages than ABT_{hyb} for the four dimensions tested. This comparison should be qualified comparing results in Table 6.2 and noting that the length of **ok?** messages differ from synchronous to asynchronous backtracking algorithms. In *SCBJ*, an **ok?** message contains the partial solution which could be of size n, while in ABT_{hyb} an **ok?** message contains a single assignment of size 1. Assuming that the communication cost depends more crucially on the number of messages than on their length, we conclude that *SCBJ* is more efficient in communication terms than ABT_{hyb}. Considering both aspects, computation effort and communication cost, *SCBJ* seems to be the algorithm of choice for the n-queens problem. However, synchronous algorithms are less steadfast than asynchronous and hybrid ones when the network crashes.

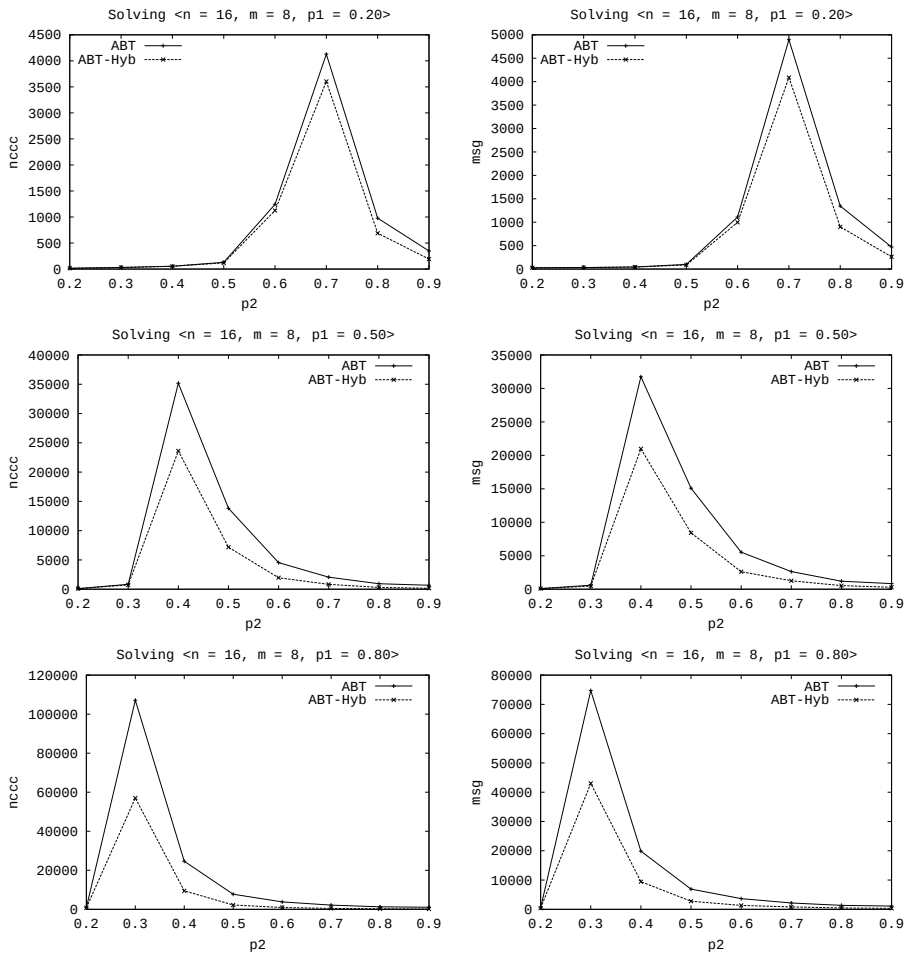


Figure 6.2: Number of non-concurrent constraint checks and messages for ABT and ABT_{hyb} on binary random problems.

6.5.2 Random Binary *DisCSP*

Similar to Section 4.4 in Chapter 4, we have tested random instances of 16 agents and 8 values per agent, considering three connectivity classes, sparse (p_1=0.2), medium (p_1=0.5) and dense (p_1=0.8). Figure 6.2 gives results averaged over 100 executions for ABT and ABT_{hyb} with random value ordering. We observe again that ABT_{hyb} is always better than ABT for the three problem classes, in both computation effort and communication cost. We believe that this is due to the effect already described for the distributed n-queens problem. This is confirmed after analyzing the number of messages per message type of Table 6.3.

rand	$SCBJ$		$SCBJ_{amd1}$		ABT				ABT_{hyb}			
p_2	**ok?**	**ngd**	**ok?**	**ngd**	**ok?**	**ngd**	*obso*	**adl**	**ok?**	**ngd**	*obso*	**adl**
0.20	2,647	1,254	100	63	3,587	1,310	320	26	3,141	949	53	24
0.50	6,913	3,556	477	321	24,725	7,025	2,336	40	17,650	3,335	321	37
0.80	9,761	5,265	1,052	758	58,283	16,432	6,497	19	37,046	5,956	755	18

Table 6.3: Number of messages exchanged by $SCBJ$, $SCBJ_{amd1}$, ABT and ABT_{hyb} per message type, for random binary problems with random value ordering.

Contrasting these results with those given in Chapter 4, Section 4.4, Figure 4.1, we observe the following. In terms of computation effort (constraint checks), $SCBJ$ is always worse than ABT_{hyb}, and $SCBJ$ is often the worst algorithm (except in the $\langle 16, 8, 0.8\rangle$ class, where it is the second worst). This behavior changes dramatically when adding the minimum domain heuristic approximations: $SCBJ_{amd1}$ and $SCBJ_{amd2}$ are the best and second best algorithms in the three classes tested, and they are always better than ABT_{hyb}.

Regarding communication costs, synchronous backtracking algorithms are always better than asynchronous ones: consistently in the three classes tested, $SCBJ_{amd1}$, $SCBJ_{amd2}$ and $SCBJ$ are the three best algorithms (in this order). Again, the addition of minimum domain approximations is very beneficial. As mentioned above for the n-queens problem, **ok?** messages are of different sizes in synchronous and asynchronous backtracking algorithms. Under the same assumptions (communication costs depends more on the number of messages exchanged than on their length), we conclude that for solving random binary problems, $SCBJ_{amd1}$ is the algorithm of choice. However, the selection of a synchronous algorithm implies that it could fail, if the network crashed.

We have also tested the three problem classes using the min-conflict value ordering. Results appear in Table 6.4 for the peak of maximum difficulty. We observe a minor but consistent improvement of all the algorithms with respect

min	$SCBJ$		$SCBJ_{amd1}$		$SCBJ_{amd2}$		ABT		ABT_{hyb}	
p_1	$nccc$	msg	$nccc$	msg	$nccc$	msg	$nccc$	msg	$nccc$	msg
0.20	7,100	3,277	907	153	1,811	687	3,771	4,006	3,448	3,535
0.50	44,024	9,367	5,637	783	11,677	2,669	30,719	26,840	22,227	19,141
0.80	102,153	15,111	16,206	1,843	40,449	7,142	101,492	70,033	58,428	43,459

Table 6.4: Results near of the pick of difficulty on binary random classes $\langle n = 16, m = 8\rangle$ with *min-conflict* value ordering.

to the random value ordering. In this case, the relative ranking of algorithms obtained with random value ordering remains, $SCBJ_{amd1}$ being the algorithm with the best performance.

6.6 Related Work

Alternatively to add synchronization points, we can avoid resending redundant **ngd** messages assuming exponential-space algorithms. Let assume that $self$ stores every nogood sent, while it is not obsolete. When $self$ finds an empty compatible domain, if the new generated nogood is equal to one of the stored nogoods, it is not sent. This allows $self$ not sending identical nogoods until some higher agent changes its value and the corresponding **ok?** arrives to $self$. But it requires exponential space, since the number of nogoods generated could be exponential in the number of agents with higher priority than $self$. A similar idea is also found in [Yokoo, 1995] for the asynchronous weak-commitment algorithm (AWC).

Very recently, Annon Zivan and Roie Meisels have studied concurrency of the agents' work in distributed algorithms from a point of view different from ABT. In the Concurrent Backtracking Search ($ConBT$) several search processes scan asynchronously disjoined parts of the search space. The search that each process performs is completely synchronous and when an agent cannot find a consistent value, it backtracks following a chronological ordering [Zivan and Meisels, 2004a]. An improved version of this algorithm considers dynamic backtracking techniques [Zivan and Meisels, 2004b].

Regarding variable ordering in asynchronous backtracking, [Zivan and Meisels, 2005b] presented a generic method for allowing agents to choose orders dynamically and asynchronously. That work evaluated the combination of ABT-DO with the heuristics for variable reordering that we presented in Chapter 4 for synchronous algorithms. However, an heuristic inspired by the idea used for dynamic backtracking in $CSPs$ [Ginsberg, 1993] was more effective.

6.7 Summary

We have proposed ABT_{hyb}, a new hybrid algorithm for distributed CSP that combines synchronous and asynchronous elements. This algorithm avoids agents to send some redundant messages after backtracking. We have demonstrate theoretical properties of the new algorithm. Empirically, we have compared ABT_{hyb} and ABT on two benchmarks. For all considered problems, ABT_{hyb} outperforms ABT in terms of the computation effort and communication cost. This improvement in performance is achieved after adding synchronization points when backtracking. These points make ABT_{hyb} less robust than ABT to network failures. We also compared synchronous against asynchronous algorithms. Experimental results shows that synchronous approaches improve over ABT and

ABT_{hyb}. This does not mean that synchronous backtracking algorithms should always be preferred to asynchronous and hybrid ones, since they offer different functionalities (synchronous algorithms are less robust to network failures, privacy issues are not considered, etc.). But for applications where efficiency is the main concern, synchronous algorithms seems to be quite good candidates to solve *DisCSP*.

Chapter 7

Non-binary *DisCSP*

It is widely acknowledged that many real world problems can be modeled naturally with non-binary constraints. In *CSP*, this question has been addressed in various papers, producing a corpus of knowledge that currently allows for an effective resolution of non-binary constraint problems [Bacchus and van Beek, 1998]. In *DisCSP*, however, most solving algorithms have been designed for binary constraints.

In this chapter, we present some approaches to deal with non-binary constraints in *DisCSP*. We start mentioning the works that have been published to handle non-binary constraints in *CSP* (Section 7.1). Then, we consider asynchronous backtracking (Section 7.2) and second we study synchronous backtracking (Section 7.3). Considering asynchronous backtracking, the extension to the non-binary case is straightforward. The existence of non-binary constraints will cause to add new links among agents, links that are used for transmitting information about new assignments. However, agents receiving these new links do not check any constraint. We suggest to add constraint projections, so these agents could check them, speeding-up the constraint checking process. We evaluate this idea comparing three algorithms: ABT, ABT with projections and ABT_{not}, which does not add any links during the search. Considering synchronous backtracking, we present two extensions of *SCBJ* for non-binary constraints, *SCBJ* and *SCBJ* with constraints projections. We evaluate the proposed algorithms on distributed ternary random problems (Section 7.4). A summary of this chapter appears in Section 7.5.

7.1 Related Work

In Chapter 3, we describe two distributed models for representing *DisCSP*: the variable-based model [Yokoo et al., 1992]; and the constraint-based model [Silaghi et al., 2000]. AAS [Silaghi and Faltings, 2005] is an asynchronous backtracking algorithm which assumes that the problem to be solved is expressed following the constraint-based model. Since this model considers that each problem

constraint belongs to one agent, a non-binary constraint can be locally managed by using the same techniques developed for *CSP*. Thus, *AAS* may consider constraints of any arity. The use of this algorithm for solving a *DisCSP*, which is originally expressed according to the variable-based model, requires that the problem be transformed to the constraint-based model. This transformation may be inadequate in many naturally distributed problems in which the initial problem structure must remain unchanged.

There are two options for solving a non-binary *DisCSP* expressed in the variable-based model: (i) translating it into a binary *DisCSP* and applying binary algorithms, or (ii) extending the binary algorithms to the non-binary case.

Regarding the first option, it is well known that a non-binary *CSP* can be translated into an equivalent binary *CSP*. Two general methods are known: the dual problem method [Dechter and Pearl, 1989]; and the hidden variable method [Dechter, 1990]. Both require the addition of new variables with exponentially large domains, which is usually seen as a serious drawback in the centralized case. However, once again losing the problem structure seems to be a real issue in *DisCSP*, when original variables are owned by agents. These translations generate new variables, which should be allocated to some "virtual" agents, while some original variables disappear from the solving process. This exchange of information may be undesirable for security or privacy reasons.

In this thesis we develop the second option, that is, we analyze how we can extend some existent algorithms for binary *DisCSP* to handle non-binary constraints.

7.2 Asynchronous Backtracking

In this section, we focus on the resolution of non-binary *DisCSP* in the context of asynchronous backtracking algorithms. First, we consider the straightforward extension of *ABT* to the non-binary case. Second, we analyze the inclusion of redundant constraint projections into the problem.

7.2.1 Non-binary *ABT*

The *ABT* algorithm, described in Chapter 5, is a reference systematic algorithm for *DisCSP* solving. Originally, *ABT* was designed to handle binary constraints [Yokoo et al., 1992]. A binary constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link departs, and the constraint-evaluating agent, to which the link arrives. To make the constraint graph cycle-free there is a total order among agents, which is followed by the directed links.

We analyze how the inclusion of non-binary constraints in the original *ABT* algorithm affects nogoods.[1] and the relationship between each constraint's

[1] In *ABT*, nogoods are generated by logical inference of problem constraints. *ABT* agents consider generated nogoods as new problem constraints.

constraint-evaluating and value-sending agents. Since ABT is able to manage nogoods of any size, the inclusion of non-binary constraints does not cause significant issues in its behavior. The only novelty is the representation of a non-binary constraint in terms of links and constraint checking [Brito and Meseguer, 2006a].

Let us consider a non-binary constraint c_i, such that $var(c_i) = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$. Assuming that agents are totally ordered from $i_1, i_2, \ldots, i_k$, the inclusion of c_i requires the addition of new links from $i_1, i_2, \ldots, i_{k-1}$ to i_k. The lowest priority agent i_k will receive the variable values $x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}}$ though these links and it will check c_i.

As in the binary case, the agent with lowest priority among those involved in the constraint is in charge of checking the constraint. The existence of links from the other agents to the constraint-evaluating agent is required, in order to inform the constraint-evaluation agent about assignments of the value-sending agents.

If i_k, the constraint-evaluating agent of c_i, upon receiving the assignments of $x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}}$, does not find any consistent value, it will send a backtrack message to i_{k-1}. This agent will receive a nogood including variables $x_{i_1}, x_{i_2}, \ldots, x_{i_{k-2}}$. But i_{k-1} has no direct link with the agents owning those variables, so it will ask them to set up a link, to confirm the values of these variables. So new links, from $i_1, i_2, \ldots, i_{k-2}$ to i_{k-1} will be added. If it happens that i_{k-1} sends another backtrack message to i_{k-2}, for the same reason it will request new links from $i_1, i_2, \ldots, i_{k-3}$ to i_{k-2}. So it is very likely that finally a clique of links will appear among the agents involved in the constraint, connecting any agent with all other lower priority agents.

7.2.2 Non-binary ABT_{not}

The ABT_{not} algorithm, described in Chapter 5, is an ABT-based algorithm that does not add any new links during the solving process. This is due to the fact that it is able to forget all those variable assignments that may become obsolete upon backtracking. It is "less informed" than pure ABT, since an agent cannot ask another agent to set up a new link. Changes in variables not directly connected are detected by backtracking messages. In Chapter 5, we prove that ABT_{not} is correct and complete.

Non-binary constraints can be added to ABT_{not} exactly in the same way they are added to ABT [Brito and Meseguer, 2006a]. Since ABT_{not} does not generate new links, it will solve the problem using the original connection topology among agents. Typically, this requires an extra effort (in computation and in communication) with respect to ABT performance.

7.2.3 Adding Constraints Projections

In the direct ABT extension, a constraint is checked by a single agent (the constraint-evaluating agent). New links are used for two proposes: to transmit the assignments among the agents owning variables involved in the non-binary constraint and to check backtracking messages for obsolescence. However, the

receiver of those new links added by ABT does not check any constraint. In [Brito and Meseguer, 2006a], we propose to add constraint projections on those new links that would be added by the ABT algorithm, in such a way that the receiving agent will be the agent that evaluates the constraint projection. Formally, a constraint projection is defined as follows:

Definition 7.2.1. The projection of a constraint C on a subset of variables S ($S \subseteq \mathcal{X}$, where $\mathcal{X}$ is the variables of the *DisCSP* to be solved) is a new constraint $C[S]$, whose permitted tuples are formed from C tuples removing the variable values not in S.

Projections are redundant constraints, in the sense that they do not convey more information than that contained in the non-binary constraint. But their inclusion distributes consistency checking among all the variables (except the one in the highest priority agent) since all these variables have to check one or several constraint projections. Constraint projections will never detect to be inconsistent a partial assignment that would be detected to be consistent for the whole constraint. On the contrary, the partial assignments found inconsistent by projections, will be found inconsistent by the whole constraint. The point here is that in checking constraint projections, inconsistencies may be detected sooner, at earlier tree levels or involving less variables than checking the whole constraint.

We call ABT_{proj} the version of non-binary ABT which includes constraint projections. In ABT_{proj}, projections are computed on ordered subsets of variables according to the total agent ordering as showed in the following example.

Example 7.2.1. *Let us consider constraint c_i, such that $var(c_i) = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$, and agents are ordered from i_1 to i_k. Then, we can add all unary projections, $c_i[x_{i_j}], (1 \leq j \leq k)$, binary projections of the form $c_i[x_{i_j}, x_{i'_j}], (1 \leq j < j' \leq k)$, ternary projections of the form $c_i[x_{i_j}, x_{i'_j}, x_{i''_j}], (1 \leq j < j' < j'' \leq k)$ and so on, until constraint projections of arity $k-1$. The constraint-evaluating agent for each of these redundant constraints is the lowest priority agent among the agents owning the projection variables.*

The idea of adding constraint projections has been previously studied in the context of *CSP* [Larrosa and Meseguer, 1998, Bessiere et al., 1999]. Somehow, ABT_{proj} is motivated by the significant improvement that several versions of *FC* with projections show over the classical *FC*.

7.2.4 Example

Let us consider a simple example to compare asynchronous backtracking versions for non-binary *DisCSPs*. There is a single non-binary constraint: $c = all - different(x_1, x_2, x_3, x_4)$, and there are four agents $1, \ldots, 4$ totally ordered by decreasing priority. Agent i owns variable x_i. Domains are $D(x_1) = \{a, d\}, D(x_2) = \{b\}, D(x_3) = \{c\}, D(x_4) = \{a\}$. The initial topology of links in the distributed constraint graph appears in Figure 7.1 (a).

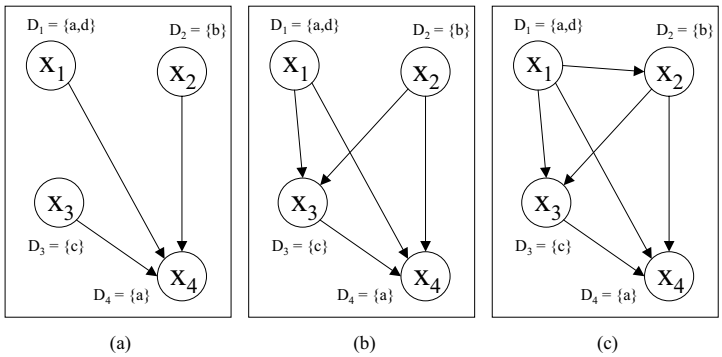


Figure 7.1: A simple problem solved by *ABT*. (a) Initial links (b) Links after x_3 received a backtracking message from x_4. (c) Links after x_2 received a backtracking message from x_3.

The direct *ABT* extension will start with the following links: $x_1 \rightarrow x_4$, $x_2 \rightarrow x_4$ and $x_3 \rightarrow x_4$. Assigning the first domain values, agent 4 will get the tuple $x_1 = a, x_2 = b, x_3 = c, x_4 = a$ which is a nogood for the constraint. Since x_4 has no other value, it will send a backtracking message to agent 3 with the nogood $x_1 = a, x_2 = b \Rightarrow x_3 = c$. Since x_3 has no direct connection with the nogood variables, two new links will be added: $x_1 \rightarrow x_3$ and $x_2 \rightarrow x_3$. Again, x_3 has no other value, so agent 3 will send a backtracking message to agent 2 with the nogood $x_1 = a \Rightarrow x_2 = b$. Since x_2 has no direct connection with x_1, this new link is added: $x_1 \rightarrow x_2$. Again, agent 2 has no other value, so it sends a backtracking message to agent 1, which now changes its value to d, informing all its neighbors. At this point, a solution is found and the network reaches quiescence. The evolution of links during *ABT* execution can be seen in Figure 7.1 (b) and (c).

The direct ABT_{not} extension will have the same links as *ABT* in its starting phase. Assigning the first domain values, agent 4 will get the tuple $x_1 = a, x_2 = b, x_3 = c, x_4 = a$ which is a nogood for the constraint. Since x_4 has no other value, agent 4 will send a backtracking message to agent 3 with the nogood $x_1 = a, x_2 = b \Rightarrow x_3 = c$. Agent 3 will update its agent view with this nogood. Since x_3 has no other value, agent 3 will send a backtracking message to agent 2 with the nogood $x_1 = a \Rightarrow x_2 = b$, and it will forget about the values of x_1 and x_2, not directly connected to it. Agent 2 will update its agent view with this nogood. Since x_2 has no other value, agent 2 will send a backtracking message to agent 1 with the nogood $x_1 = a$, and it will forget about the values of x_1, not directly connected to it. After receiving this message, x_1 changes its value to d, informing all its neighbors. Again, a solution is found and the network reaches quiescence.

The projections addition approach will consider the addition of the following redundant constraints: $c[x_1]$, $c[x_2]$, $c[x_3]$, $c[x_4]$, $c[x_1, x_2]$, $c[x_1, x_3]$, $c[x_1, x_4]$,

$c[x_2, x_3]$, $c[x_2, x_4]$, $c[x_3, x_4]$, $c[x_1, x_2, x_3]$, $c[x_2, x_3, x_4]$. Unary constraints can be processed, eliminating from the domains those values not permitted by them. In this way, value $a \in D(x_1)$ is eliminated. From each agent there are links towards any other agent of lower priority. With the first assignment, a solution is found.

7.3 Synchronous Backtracking

In this section, we examine the resolution of non-binary *DisCSP* in the context of *SCBJ*, a synchronous backtracking algorithm. *SCBJ* has been previously described and empirically evaluated in Chapter 4.

7.3.1 Non-binary *SCBJ*

In synchronous backtracking algorithms, only one agent is active at any time while the rest of the agents are waiting. The activation of agents is provoked by the reception of any message. As seen in Chapter 4, *SCBJ* is a synchronous backtracking algorithm that requires a static instantiation ordering of agents. In accordance with this priority order, agents try to extend a partial solution into a total one by adding consistent assignments for unassigned variables. When an agent does not have a value for one of its variables, which is consistent with the assignments of preceding variables in the partial solution, the agent backtracks to the closest preceding agent which does not permit a valid value.

The extension of *SCBJ* for non-binary constraints is straightforward. Like in *ABT*, we analyze how the inclusion of non-binary constraints affects nogoods and the relationship between each constraint's constraint-evaluating and value-sending agents. The original *SCBJ* can handle nogoods of any size. Thus, the inclusion of non-binary constraints does not affect nogoods.

In *SCBJ*, the lowest priority agent of each constraint is the agent in charge of checking the consistency of the constraint (i.e. the constraint-evaluating agent). Since, each *SCBJ* agent receives the assignments of all preceding variables, the constraint-evaluating agent of each constraint will receive from the value-sending agents all the information it needs to check the consistency of the constraint. Therefore, the inclusion of non-binary constraints does not alter the behavior of *SCBJ*.

7.3.2 Non-binary *SCBJ* with Projections

As seen in Section 7.2.3 for *ABT*, the addition of constraint projections to a distributed algorithm allows it to have multiple constraint-evaluating agents for each non-binary constraints. This may help agents to detect early inconsistency. In this subsection, we present a version of *SCBJ* for non-binary *DisCSPs* which also adds constraint projections. We call this algorithm $SCBJ_{proj}$. Projections are computed on ordered subsets of variables following the total agent ordering. The process to generate constraint projections is the same as explained in Example 7.2.1 for ABT_{proj}. This implies that, for a given *DisCSP*, ABT_{proj} and

$SCBJ_{proj}$ adds the same constraint projections for each non-binary constraint.

There is a significant difference between ABT_{proj} and $SCBJ_{proj}$. In ABT_{proj}, a non-binary constraint and its corresponding projections may be evaluated simultaneously due to the parallel nature of the algorithm. This may produce that a partial solution can be simultaneously found to be inconsistent because a constraint projection and the corresponding non-binary constraint. Regarding communication cost, this situation will produce several useless messages. Regarding computation cost, since these constraint checks are done in parallel, this situation does not affect substantially to the performance of ABT_{proj}.

In contrast, the above situation will never occur in *SCBJ*. That is, a partial solution will never be simultaneously found to be inconsistent because a projection and the non-binary constraint. This can be explained because, in *SCBJ*, constraint projections corresponding to a non-binary constraints are evaluated sequentially and always before the non-binary constraint. Nevertheless, due to the sequential order in which redundant constraint projections and the original non-binary constraint are evaluated, it may increase computation effort with respect to *SCBJ*.

7.4 Experimental Results

We have evaluated asynchronous (*ABT*, ABT_{not} and ABT_{proj}) and synchronous backtracking approaches ($SCBJ_{not}$ and $SCBJ_{proj}$) on random ternary problems. In accordance to [Larrosa and Meseguer, 1998], we use an extended version of the four parameter binary model [Smith, 1994] to ternary problems. A ternary random problem is defined by four parameters $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m is the cardinality of their domains, p_1 is the problem connectivity, the ratio between existent constraints and the maximum number of possible constraints (the problem has exactly $p_1 n(n-1)(n-2)/6$ constraints), and p_2 is the constraint tightness, the proportion of forbidden value triplets between three constrained variables (the number of forbidden value triplets is exactly $p_2 m^3$). The constrained variables and their nogoods are randomly selected following a uniform distribution.

In our experiments, each agent owns a variable. For all algorithms, each ternary constraint is evaluated by the lower priority agent involved in it. Agents store ternary constraints as lists of permitted combinations of variable values. These lists are called ternary constraint tables. ABT_{proj} and $SCBJ_{proj}$ add at most three unary projections and one binary projection for each ternary constraint. In both algorithms, agents compute unary and binary projections from ternary constraint tables in a preprocessing step. We assume that each variable involved in a ternary constraint completely knows the constraint. Therefore, agents do not need to communicate in the preprocessing step.

Given a ternary constraint $T[x_i, x_j, x_k]$ between variables x_i, x_j and x_k, the binary constraint projection between x_i and x_j is computed by x_j as follows. First, x_j removes from T the column corresponding to x_k. Second, x_j removes duplicated rows from the resulting table. The output table $B[x_i, x_j]$ contains

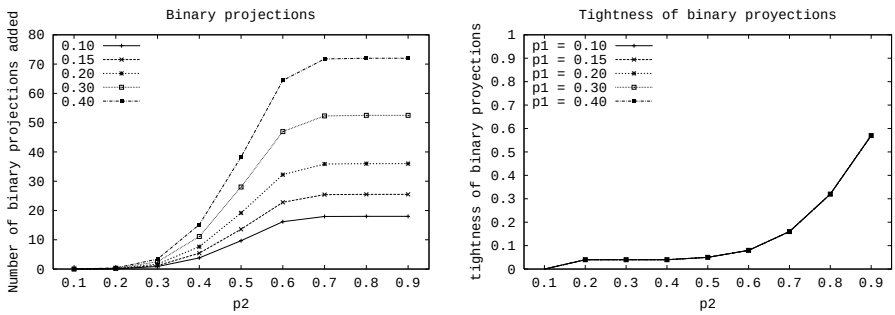


Figure 7.2: Number of binary projections added and their tightness.

the list of tuples allowed for the binary projection. Similarly, agents x_i, x_j, x_k compute their unary projections by removing from their copies of T columns related to other two variables, respectively. Thus, agents do not need to perform any constraint check in the preprocessing step. In the experiments, ABT_{proj} and $SCBJ_{proj}$ add only unary/binary projections with at least one forbidden value/tuple.

We consider five different classes of random ternary problems with ten variables ($n = 10$) and five values per domain ($m = 5$). p_1 takes the following values: 0.1, 0.15, 0.2, 0.3 and 0.4. Higher values of p_1 generate instances with many ternary constraints. These instances are not particularly representative of real problems. For these instances, there is a high probability that an agent is linked to all higher priority agents. Therefore, in these instances ABT and ABT_{not} have a similar performance (as seen in the plots). p_2 varies between 0.1 and 0.9, in increments of 0.1.

In Figure 7.2, we give the number of binary projections added by ABT_{proj} and $SCBJ_{proj}$ and their average tightness for each value of p_1 and p_2. Both parameters clearly depends on the value of p_2. Larger value of p_2 means a higher number of binary projections are considered by (figure on left) and these redundant constraints have higher tightness (figure on right).

For both algorithms, agents implement the heuristic of selecting the best nogood seen in Chapter 5 and process messages by packets as discussed in Chapter 6. We compare performance algorithms according to computation effort, in terms of the number of non-concurrent constraint checks (*nccc*), and communication cost, in terms of total number of exchanged messages (*msg*). In Figure 7.3 and Figure 7.4, we report *nccc* and *msg* needed by algorithms for solving the five problem classes studied. Results are averaged on 100 instances for each value of p_1 and p_2.

Considering asynchronous backtracking approaches, we observe that ABT_{not} always requires more *nccc* than ABT for each tested class. This can be explained by the fact that agents of ABT_{not} are worse informed that agents of ABT, since ABT_{not} does not add new links during the search. Differences evolve with constraint density: in classes with very low p_1, ABT is much better than

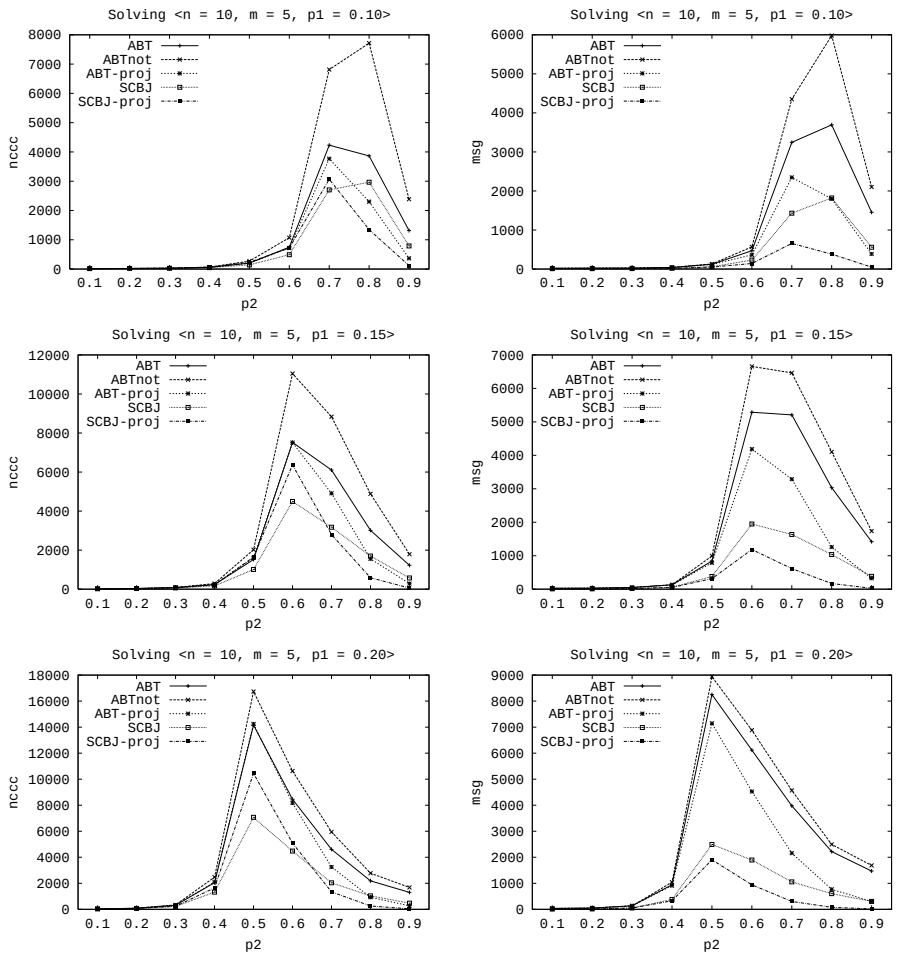


Figure 7.3: Computation and communication cost of ABT, ABT_{not}, ABT_{proj}, $SCBJ$ and $SCBJ_{proj}$ for solving ternary random instances with low constraint density (n = 10, m = 5, $p_1 = 0.1, 0.15, 0.2$).

ABT_{not}, but as p_1 increases their difference decreases, until $p_1 = 0.4$ and on, where both algorithms show a similar behavior. From this connectivity on, the probability of any two agents being linked is very high, so ABT and ABT_{not} behave in the same way most of the time.

ABT_{proj} requires a number of *nccc* equal to or less than ABT for each tested class. This depends on constraint tightness. For $p_2 < 0.5$, both require a similar number of *nccc*. In this region, the tightness of the added projections is quite low (see Figure 7.2 right), so they have practically no effect. For $p_2 \geq 0.5$, the tightness of the added projections rises steadily (see Figure 7.2 right), producing the expected benefits in distributed search. As connectivity increases, differences

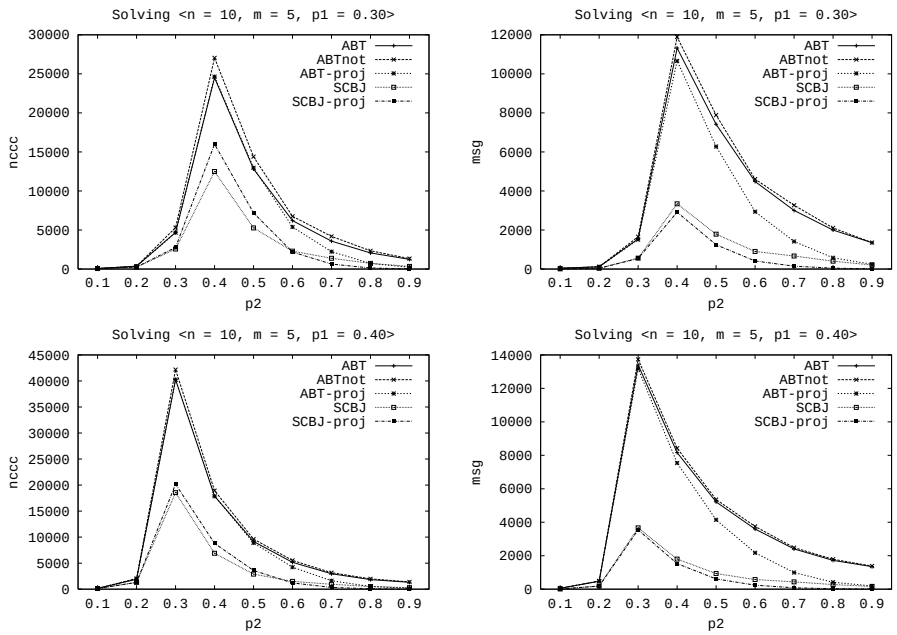


Figure 7.4: Computation and communication cost of ABT, ABT_{not}, ABT_{proj}, $SCBJ$ and $SCBJ_{proj}$ for solving ternary random instances with higher constraint density (n = 10, m = 5, $p_1 = 0.3, 0.4$).

between ABT and ABT_{proj} decrease slightly (observe the different scales in plots in Figures 7.3 and 7.4). As the probability of any two agents being linked increases, information about assignments is more available on the network, so the probability of backtracking for a ternary constraint also increases. In this way, the relative efficiency of ABT_{proj} is diminished.

In terms of communication effort, the relative order of three asynchronous backtracking remain unchanged. ABT_{not} always requires more messages than ABT. The difference is higher for low connectivity classes, and it decreases steadily until achieving a similar performance for $p_1 \geq 0.4$. This is due to the high probability of two agents being linked for this constraint density, as discussed above. ABT_{proj} requires a number of messages equal to or less than ABT for each tested class. Again, this depends on constraint tightness. For $p_2 < 0.5$, both require a similar number of messages, but for $p_2 \geq 0.5$, ABT_{proj} requires less messages than ABT. This is due to the tightness of the added projections in Figure 7.2 as discussed above.

Considering synchronous backtracking approaches, we observe that, in terms of number of non-concurrent constraint checks, $SCBJ_{proj}$ is worse than $SCBJ$ at the complexity peak for all tested classes. This is because the effect of adding constraint projections in the number of non-concurrent constraint checks is two-fold. On one hand, an agent in charge of checking a constraint projection will

increase its constraint check counter each time it makes a constraint check. Since there are more agents increasing their counters, the global *nccc* may increase. On the other hand, since backtracking may appear at earlier levels of the tree, some *nccc* may be saved and the global *nccc* may decrease. The added effect of these two tensions is a *nccc* saving in ABT_{proj} and a *nccc* increasing in $SCBJ_{proj}$. To understand this one must keep in mind the way in which constraint projections and non-binary constraints are checked by each algorithm. In $SCBJ_{proj}$, every non-binary constraint is always checked after the corresponding constraint projection has been checked. Conversely, constraint projections and non-binary constraints may be checked concurrently in ABT_{proj}.

Regarding communication cost, the number of messages sent by $SCBJ_{proj}$ agents is approximately larger than or equal to the number of messages sent by *SCBJ*. The benefic of adding constraint projections for problems at the complexity peak decreases as p_1 increases. This is because as soon as p_1 increases the complexity peak shifts to the left, where non-binary constraints become looser. This causes the added constraint projections to be looser as well and, therefore, to have a low pruning capability as can be seen in Figure 7.2. For each tested classes, $SCBJ_{proj}$ outperforms *SCBJ* on instances on the left of the complexity peak. In this region, the improvement of ABT_{proj} over *ABT* is larger than the improvement of $SCBJ_{proj}$ over *SCBJ*. However, ABT_{proj} is never better than $SCBJ_{proj}$.

Considering asynchronous and synchronous backtracking approaches not adding constraint projections, we note that *SCBJ* is always better than *ABT* and ABT_{not} with respect to *nccc* and *msg* for all tested problem classes. These results show what we have discussed in Chapter 6 for binary *DisCSPs*. Again, the use of synchronous backtracking approaches seems to be more effective on problems where efficiency is the main concern.

7.5 Summary

We have presented asynchronous and synchronous backtracking approaches to deal with non-binary *DisCSPs*. Considering asynchronous backtracking, we presented versions of *ABT* and ABT_{not} to handle non-binary constraints. In addition, we proposed to add constraints projections over those new links that are dynamically added by *ABT* but are no used to check any constraint. These projections are redundant constraints of smaller arity, which can be exploited by *ABT* to detect inconsistencies at earlier levels of the search tree. Considering synchronous backtracking, we presented a version of *SCBJ* to handle non-binary *DisCSPs*. Similar to *ABT*, we propose to add constraint projections to *SCBJ* in order to improve the algorithmic performance. The experimental results show that synchronous algorithms outperforms asynchronous ones. In both search types, the addition of constraint projections is much effective when their pruning power is high.

Part III

Privacy

Chapter 8

Privacy in *DisCSP*

Privacy is one of the main motivations to solve *DisCSPs* in a distributed form. Many problems appear to be naturally distributed, each part belonging to a different agent. In that setting, agents may desire to keep their information as private as possible. Take for example a *DisCSP* in which agents want to keep their variable domains private or to hide assignments and constraints from other agents considered as potential competitors.

Generally speaking, most distributed algorithms leak some kind of information in the solving process, which can be exploited by some agents to deduce the reserved information of other agents. Although the original *ABT* was not concerned with privacy issues (agents exchanged their values freely), privacy has been a key aspect in the development of new *DisCSP* solving algorithms. So far, there are two main approaches to enforcing privacy. One considers the use of encryption techniques to conceal values and constraints [Silaghi and Mitra, 2004, Yokoo et al., 2005, Nissim and Zivan, 2005]. Alternatively, the other aims at enforcing privacy by different strategies but excluding cryptography [Silaghi, 2002, Brito and Meseguer, 2003, Brito and Meseguer, 2005b, Zivan and Meisels, 2005a].

In this chapter, we investigate how privacy can be enhanced in *ABT* without using encryption methods.[1] We analyze privacy from three perspectives related to different elements of *DisCSP*: domain privacy (Section 8.1), assignment privacy (Section 8.2) and constraint privacy (Section 8.3). We discuss the evolution of distributed algorithms for trying to maintain constraint and/or assignment privacy during resolution. This includes two families of algorithms: $DisFC_2/DisFC_1$ and ABT_2/ABT_1.

These algorithms do not keep agents' information completely secure as agents may leak some data during the search. We propose a way to count the amount of information related to problem constraints that agents reveal to other agents. We assess the algorithms on random *DisCSP* instances (Section 8.4). Experimental results show that *ABT* is worse than the algorithms we present here in terms of

[1] All the strategies that we propose here are also applicable to the rest of algorithms that form the *ABT* family seen in Chapter 5. Their extensions are straightforward.

privacy issues but it is more economic than them in terms of computation effort and communication cost.

8.1 Domain Privacy

Domain privacy is concerned with the idea that agents may want to hide the domains of their variables. In the variable-based model for *DisCSP*, every variable belongs to one agent. This implies that the agent owns a variable and is the only one that knows its domain. Hence, this model guarantees the desired domain privacy. All the distributed algorithms seen in previous chapters, *ABT* included, assume this model.

Regarding solving methods, except for the value dynamic ordering heuristic seen in Chapters 4 and 6, none of the algorithms previously described requires that agents know the whole domains of other agents[2]. Let us analyze how *ABT* execution affects domain privacy. During problem resolution, agents exchange assignments and nogoods to find a solution. Thus, an agent may detect some values from the domains of the variables held by others agents through received messages. If an agent stores all the values it receives from any other agent then, at the end of the algorithm, it will have enough values to partially reconstruct the other agents' domains. The more values an agent receives from the variable domain of another agent, the closer it will be to know the whole domain of this variable. Therefore, the lower number of values an agent tries when assigning consistent values to its variable, the higher domain privacy it achieves.

In Section 5.4, Chapter 5, we presented several heuristics to reduce the number of exchanged messages in asynchronous backtracking algorithms. One of them is that, after backtracking, an agent tries first the value it had assigned to the variable before it found an empty consistent domain. This strategy can be generalized to enforce domain privacy. That is, each time an agent searches a consistent value, it should try first the values that it has considered before. This strategy can be easily implemented by agents either in synchronous or asynchronous algorithms.

For solvable *DisCSPs*, it is impossible for an agent to detect if another agent has shared all its values. Only when an agent j backtracks, the agent i which receives the **ngd** message can infer that j has checked the consistency of all its values. But this does not mean that all those values have been assigned to j's variable and, consequently, sent out to lower priority agents via **ok?** messages. Take for example a problem where a value of j's variable is always ruled out by constraints provided by other higher priority agents, so, j will never assign this value to its variable. In addition, notice that the agent i that receives a **ngd** message from j will never receive an **ok?** message from j, because i appears first in the priority ordering. In consequence, when a solution is found, no agent

[2]This is only true in the case that problem constraints are expressed implicitly. Since all the algorithms described so far assume that the lowest priority agent involved in each constraint has to know the complete constraint, if this constraint is given explicitly, the lowest priority agent of each constraint must know the variable domain of the higher priority one.

knows if it has received the complete variable domains that belong to other agents.

For unsolvable *DisCSPs*, however, if the identity of the agent i that detects inconsistency is revealed, all lower priority agents that shares a constraint with i will receive all the values of i's domain.

Proposition 8.1.1. *If an ABT agent detects inconsistency, every lower priority agent directly connected with it has received all the values of its domain.*

Proof. Let i an agent detecting inconsistency. If i finds the empty nogood, it means that there is a nogood for every value of i's domain. These nogoods have an empty left-hand side (otherwise, i could not deduce the empty nogood). So they have been produced as a result of **ngd** messages coming from lower priority agents. Therefore, every possible value of i has been taken, so i has sent every value to its lower priority agents through **ok?** messages. □

In order to hide the identity of the agent i that detects inconsistency, one could add an extra agent for announcing to the agents that the search has finished or allow the agent i to send **stp** messages to the rest of agents in such way that the identity of i in those messages is concealed.

8.2 Assignment Privacy

Assignment privacy is concerned with the idea that agents may want to conceal their assignments. An *ABT* agent sends its current assignment in two messages (between i and j, $i < j$):

1. **ok?**: when agent i informs low priority agents of its value. This message contains i value. It is used by j to find a compatible value with i, so j has to know the constraint C_{ij}.

2. **ngd**: when agent j sends a backtrack message to i. This message contains the values of the agent view of j. It is used by i to check if the nogood message is obsolete, testing whether the assignments of common variables with higher priority than i in the agent views of i and j are the same or not.

In order to preserve assignments private, agents must avoid sending their assignments to other agents. In the following subsection we present an *ABT*-like algorithm that resolves the above two points.

8.2.1 Distributed Forward Checking

The *Distributed Forward Checking* (*DisFC*) algorithm [Brito and Meseguer, 2003] avoids that agents send their assignments in **ok?** and **ngd** messages. About **ok?** messages, instead of sending i current value, the **ok?** message contains the subset of D_j values that are compatible with i current value. From this subset, j may be consistently assigned without

```
procedure DisFC()
  compute Γ⁻, Γ⁺;
  myValue ← empty; mySeq ← 0; end ← false;
  DisFC();
procedure DisFC()
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?    : ProcessInfo(msg);
      ngd    : ResolveConflict(msg);
      adl    : SetLink(msg);
      stp    : end ← true;
procedure ProcessInfo(msg)
  UpdateAgentView(msg.Sender = msg.Seq);
  UpdateDomain(msg);
  CheckAgentView();
procedure ResolveConflict(msg)
  if coherent(msg.Nogood, Γ⁻(self) ∪ {self}) then
    add(msg.Nogood, myNogoodStore); myValue ← empty;
    CheckAgentView();
  else if coherent(msg.Nogood, self) then
    SendMsg:ok?(msg.Sender, mySeq, compatible(D(msg.Sender), myValue));
procedure CheckAgentView()
  if (myValue = empty ∨ myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      for each child ∈ Γ⁺(self)
        do sendMsg:ok?(child, mySeq, compatible(D(child), myValue));
    else Backtrack();
procedure UpdateDomain(msg)
  for each v ∈ D(self) ∧ v ∉ msg.Domain do
    add(msg.Sender = msg.Seq ⇒ self ≠ v, myNogoodStore);
```

Figure 8.1: The *DisFC* algorithm. Missing procedures appear in Figure 5.2, Chapter 5.

further checking. This is the idea of Forward Checking in the centralized case [Haralick and Elliot, 1980]. About **ngd** messages, we propose the use of identifiers. Each variable keeps a sequence number that starts from 1 (or some random value), and increases monotonically each time the variable changes its assignment, acting as a unique identifier for each assignment. Messages include the sequence number of the assignment of the sending agent. The agent view of the receiver is composed of the sequence numbers it received in **ok?** messages from higher priority agents. Nogoods are formed by variables and

their sequence numbers.

DisFC is an *ABT*-like algorithm that combines the above two strategies: sending filtered domains to other agents and replacing the own value by its sequence number. *DisFC* allows an agent to exchange enough information with other agents to reach a global consistent solution (or proving that no solution exists) without revealing its own assignment at any time. The code of *DisFC* appears in Figure 8.1. Only the new or modified parts with respect to *ABT* are shown. Missing procedures can be found in Figure 5.2, Chapter 5.

8.2.2 *DisFC*: Theoretical Results

Here, we formally prove that *DisFC* inherits the good properties of *ABT*: completeness, correctness and termination. Regarding nogoods, the only difference between *ABT* and *DisFC* is that actual values are replaced by sequence numbers. This is fine, as the only role of values/sequence numbers is to detect that an assignment is obsolete. However, it might occur that two different sequence numbers for one variable would represent the same value. If this happens after receiving a backtrack message, when comparing the agent view of the message with the agent view of the receiver, the message will be discarded as obsolete. But this will cause no problem. Since each time the sequence number changes, an **ok?** message is sent, if either the sender or the receiver of the backtrack message are not updated, it means that the message with the most updated sequence number has not arrived yet, but it is on its way. After its arrival, the backtrack message will be accepted. After this clarification, we prove that the good theoretical properties of *ABT* also held for *DisFC*.

The search space is defined by the variables and domains of the problem instance. The way this space is traversed depends on (i) the total order among agents and (ii) the set of nogoods generated during asynchronous search. Assuming that all algorithms follow the same agent ordering, the proof will be based on the fact that all algorithms can generate the same nogoods.

Lemma 8.2.1. *A nogood can be generated by DisFC iff it can be generated by ABT.*

Proof. Let us differentiate between explicit and implicit nogoods. In *ABT* and *DisFC*, an explicit nogood is generated as a consequence of an **ok?** message. An implicit nogood is generated by resolution of the set of nogoods that forbid every value of a variable.

Explicit nogoods. Let us consider two constrained agents i, j, $i < j$. The only difference with *ABT* is that **ok?** messages may generate more than one nogood. In fact, an **ok?** message from i to j generates as many nogoods as values considered inconsistent in D_j. However, these nogoods would have been generated by *ABT* if these values would have been successively assigned to j.

Implicit nogoods. (Proof by induction on the number of implicit nogoods in a sequence of backtracking steps). The first implicit nogood in the sequence that appears in *DisFC* is generated by resolving explicit nogoods. Since all explicit nogoods of *DisFC* can be generated by *ABT*, and the nogood resolution

mechanism is the same, this first implicit nogood can also be generated by *ABT*. Let us assume that this is true up to the n-th implicit nogood generated by *DisFC*, and let us prove that this is the case for the $n+1$-th implicit nogood. Let us consider the $n+1$-th implicit nogood generated. It has been computed by resolving previous nogoods, either explicit or implicit. We have already proved that explicit nogoods can also be generated by *ABT*. All previous (n) implicit nogoods can be generated by *ABT* by the induction step. Therefore, since the nogoods involved in the resolution can all be generated by *ABT* and the resolution process is the same, the resolvent nogood could also be generated by *ABT*. A similar argument holds in the other direction of the lemma, starting from *ABT* implicit nogoods and implying nogoods of *DisFC*. □

Proposition 8.2.2. *DisFC is correct, complete and terminates.*

Proof. *Correctness.* If *DisFC* reports a solution, it is because the network has reached quiescence. In that case, every constraint is satisfied (otherwise, quiescence cannot be reached). Therefore, the reported solution is a true solution.

Completeness. DisFC performs the same kind of search as *ABT*: total ordering of agents, asynchronous instantiation, resolving nogoods, adding links, etc. The only differences of *DisFC* with respect to *ABT* are that agents exchange filtered domains instead of current assignments and replace the own value by its sequence number. But we know, by Lemma 8.2.1, that these changes do not cause any modification in the set of nogoods generated by *DisFC* with respect to *ABT*. Consequently the *DisFC* algorithm will discard the same parts of the search space as *ABT*, but not other parts. Since *ABT* is complete, *DisFC* is also complete.

Termination. An argument similar to the one used in completeness applies here. Nogoods rule out parts of the search space. Because the total ordering of agents, discarded parts accumulate, so *ABT* terminates in a finite search space (see Proposition 5.2.8, Chapter 5). Since *DisFC* generates the same nogoods as *ABT*, and they perform the same kind of search, *DisFC* also terminates. □

8.3 Constraint Privacy

ABT assumes that an inter-agent constraint C_{ij} is *totally known* by the agents owning their related variables, that is, C_{ij} is totally known by agent i and agent j (see Section 2.2 of [Yokoo et al., 1998]). We say that they follow the *Totally Known Constraints*(*TKC*) model. In fact, it is enough for *ABT* that the lower priority agent in each constraint knows the set of permitted tuples. However, versions of *ABT*, in which the order of agents can dynamically change during the execution of the algorithm, require that a constraint be totally known by both agents involved in it [Zivan and Meisels, 2005b].

8.3.1 Partially Known Constraint Model

To enforce constraint privacy, we presented the *Partially Known Constraints* (*PKC*) model of *DisCSP* [Brito and Meseguer, 2003]. In *PKC*, a constraint C_{ij} is partially known by its related agents. Agent i knows the constraint $C_{i(j)}$ where:

- $vars(C_{i(j)}) = \{x_i, x_j\}$;
- $C_{i(j)}$ is specified by three disjoint sets of value tuples for x_i and x_j:
  - $prm(C_{i(j)})$, the set of tuples that i knows to be permitted;
  - $fbd(C_{i(j)})$, the set of tuples that i knows to be forbidden;
  - $unk(C_{i(j)})$, the set of tuples whose consistency is not known by i;
- every possible tuple is included in one of the above sets, that is, $prm(C_{i(j)}) \cup fbd(C_{i(j)}) \cup unk(C_{i(j)}) = D_i \times D_j$.

Similarly, agent j knows $C_{(i)j}$, where $vars(C_{(i)j}) = \{x_i, x_j\}$. $C_{(i)j}$ is specified by the disjoint sets $prm(C_{(i)j})$, $fbd(C_{(i)j})$ and $unk(C_{(i)j})$. For the model to be truly partial, it is required that, there is at least one pair of constrained agents i and j that do not have the same information about the shared constraint (i.e. they differ in at least one of the three sets of tuples). The relation between a totally known constraint C_{ij} and its corresponding partially known constraints $C_{i(j)}$ and $C_{(i)j}$ is

$$C_{ij} = C_{i(j)} \otimes C_{(i)j}$$

where $\otimes$ depends on the semantic of the constraint. The above definitions satisfy the following conditions:

- If the combination of values k and l, for x_i and x_j is forbidden in at least one partial constraint, then it is forbidden in the corresponding total constraint: if $(k,l) \in fbd(C_{i(j)})$ or $(k,l) \in fbd(C_{i(j)})$ then $(k,l) \in fbd(C_{ij})$.
- If the combination of values k and l, for x_i and x_j is permitted in both partial constraints, then it is also permitted in the corresponding total constraint: if $(k,l) \in prm(C_{i(j)})$ and $(k,l) \in prm(C_{i(j)})$ then $(k,l) \in prm(C_{ij})$.

In this chapter, we only consider constraints for which $unk(C_{(i)j}) = unk(C_{i(j)}) = \emptyset$. In this case, a partially known constraint $C_{i(j)}$ is completely specified by its permitted tuples (tuples not in $prm(C_{i(j)})$ are in $fbd(C_{i(j)})$). Furthermore, the intersection of the constraints known by agent i and agent j is the actual constraint existing between i and j, that is,

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

Next, we give an example of the application of the *PKC* model to the *n-pieces m-chessboard* problem [Brito and Meseguer, 2003]. This problem consists of n chess pieces and an $m \times m$ chessboard. The goal is to put all pieces on the chessboard in such a way that no piece attacks any other.

Example 8.3.1. *Let us consider the n-pieces m-chessboard problem. As distributed CSP, the problem can be formulated as follows,*

- *Variables: one variable per piece.*
- *Domains: all variables share the domain* $\{1, \ldots, m^2\}$ *of chessboard positions.*
- *Constraints: one constraint between every pair of pieces, following chess rules.*
- *Agents: one agent per variable.*

For instance, we can take $n = 5$ *with the set of pieces* $\{$*queen, castle, bishop, bishop, knight*$\}$*, on a* 4×4 *chessboard, with the variables,*

$$x_1 = queen,\ x_2 = castle,\ x_3 = bishop,\ x_4 = bishop,\ x_5 = knight.$$

If agent 1 knows that agent 5 holds a knight, and agent 5 knows that agent 1 holds a queen, the result is a complete known constraint C_{15} *including the following tuples,*

$$C_{15} = \{(1,8), (1,12), (1,14), (1,15), \ldots\}$$

With the PKC model, agent 1 does not know which piece agent 5 holds. It only knows how a queen attacks, from which it can develop the constraint,

$$C_{1(5)} = \{(1,7), (1,8), (1,10), (1,12), \ldots\}$$

Analogously, agent 5 does not know which piece agent 1 holds. Its only information is how a knight attacks, from which it can develop the constraint,

$$C_{(1)5} = \{(1,2), (1,3), (1,4), (1,5), (1,6), (1,8), \ldots\}$$

The whole constraint C_{15} *appears as the intersection of these two constraints,*

$$C_{15} = C_{1(5)} \cap C_{(1)5} = \{(1,8), \ldots\}$$

In fact, $C_{1(5)}$ *does not depend on agent 5. It codifies the way a queen attacks, which is independent of any other piece. In this problem, the PKC model allows each agent to represent itself, independently of other agents.*

8.3.2 Two-Phase Strategy for *PKC*

In [Brito and Meseguer, 2003], we presented the first strategy that incorporates the *PKC* model of constraints: *the two-phase strategy*. It consists of a cycle of two phases. In the first phase (**phase I**), the original problem is relaxed considering only one partial constraint for each pair of constrained agents. If no solution is found in **phase I**, the procedure ends returning failure, since no solution exists for the whole problem. If a solution is found, it is passed to the second phase (**phase II**) where it is checked against the partial constraints

disregarded in **phase I**. If it is also a solution of **phase II**, then it is a solution for the whole problem. Otherwise, one or several nogoods are generated and search is resumed in **phase I**. In this way, nogoods found in **phase II** are used in **phase I** to escape from incompatible assignments. The two-phase strategy is a generic method which implementation details depend on the algorithm used to find a solution in **phase I**. In the following, we present $DisFC_2$ and ABT_2 algorithms, which have been obtained by combining the two-phase strategy with *DisFC* and *ABT*, respectively. The order in which algorithms are described here correspond to the order in which they were published.

$DisFC_2$

Originally, we implemented the two-phase strategy using *DisFC* [Brito and Meseguer, 2003]. This combination was conceived for enforcing assignment and constraint privacy. We call this algorithm $DisFC_2$. It works as follows:

- **phase I**. Constraints are directed, forming a directed acyclic graph (in short, *DAG*), and a compatible total order of agents is selected. The standard *DisFC* algorithm finds a solution with respect to constraints $C_{i(j)}$, where agent i has higher priority than j (the constraint $C_{(i)j}$ is checked by the lower priority agent, j). A solution is identified by detecting quiescence in the network. If no solution is found, the process stops, reporting failure.
- **phase II**. Constraints and the order of agents are reversed. Now $C_{(i)j}$ are considered, where j has higher priority than i (e.g. in the *reversed* order). j informs i of its value. If the value of i is consistent, i does nothing. Otherwise, i sends a **ngd** message to j, which receives that message and does nothing. Quiescence is detected.

Figure 8.2 presents the code for the $DisFC_2$ algorithm. $DisFC_2$ agents have the same data structures as *ABT* agents. In the main procedure (procedure `DisFC-I()`), the agents perform standard *DisFC* to find a solution compatible with all constraints held by lower priority agents. If such a solution is obtained, the agents reverse the total order by exchanging Γ^- and Γ^+ (as seen Chapter 5, the sets Γ^- and Γ^+ of an agent i are the higher and lower priority agents which are constrained with i). Then **phase II** is performed (procedure `DisFC-II()`). If it is successful (no nogood generated during **phase II**), the algorithm ends, otherwise Γ^- and Γ^+ are exchanged again and **phase I** is resumed. Agents exchange *ABT* message types, plus the messages **qes**, **qnn** meaning quiescence in the network after **ngd** and after no **ngd** messages, respectively. $DisFC_2$ inherits the good properties of *DisFC*: completeness, correctness and termination.

ABT_2

[Zivan and Meisels, 2005a] propose the ABT_2 algorithm. This algorithm results of combining the two-phase strategy for *PKC* and the well-known *ABT* algorithm. In this case, it is assumed that agents do not consider relevant to hide

```
procedure DisFC2()
  compute Γ⁻, Γ⁺;
  myValue ← empty;
  end ← false; nogoods ← false;
  repeat
    DisFC-I();
    if (¬end)
      exchange Γ⁻, Γ⁺;
      DisFC-II();
      exchange Γ⁻, Γ⁺;
  until end or ¬nogoods
procedure DisFC-I()
  quiescence ← false;
  CheckAgentView();
  while (¬end ∧ ¬quiescence) do
    msg ← getMsg();
    switch(msg.type)
      ok?     : ProcessInfo(msg);
      ngd     : ResolveConflict(msg);
      adl     : SetLink(msg);
      stp     : end ← true;
      qes     : quiescence ← true;
procedure ProcessInfo(msg)
  UpdateAgentView(msg.Sender = msg.Seq);
  UpdateDomain(msg);
  CheckAgentView();
procedure ResolveConflict(msg)
  if coherent(msg.Nogood, Γ⁻(self) ∪ {self}) then
    CheckAddLink(msg);
    add(msg.Nogood, myNogoodStore); myValue ← empty;
    CheckAgentView();
  else if coherent(msg.Nogood, self) then
    SendMsg:ok?(msg.Sender, mySeq, compatible(D(msg.Sender), myValue));
procedure CheckAgentView()
  if (myValue = empty ∨ myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      for each child ∈ Γ⁺(self) do
        sendMsg:ok?(child, mySeq, compatible(D(child), myValue));
    else Backtrack();
procedure UpdateDomain(msg)
  for each v ∈ D(self) ∧ v ∉ msg.Domain do
    add(msg.Sender = msg.Seq ⇒ self ≠ v, myNogoodStore);
```

```
procedure DisFC-II()
  quiescence ← false;
  for each child ∈ Γ0+(self) do sendMsg:ok?(child, mySeq, compatible(D(child), myValue));
  while (¬quiescence) do
    msg ← getMsg();
    switch(msg.type)
      ok?   : if myValue ∉ msg.Domain then
                 sendMsg:ngd(self = mySeq ⇒ msg.Sender ≠ msg.Seq);
      ngd   : add(msg.Nogood, myNogoodStore); myValue ← empty;
      qes   : quiescence ← true; nogoods ← true; /*quiescence with ngd messages*/
      qnn   : quiescence ← true; nogoods ← false; /*quiescence without ngd messages*/
```

Figure 8.2: The $DisFC_2$ algorithm for the PKC model (continued from previous page). Missing procedures appear in Figure 5.2, Chapter 5.

their assignments from other agents. Essentially, ABT_2 works like $DisFC_2$ with the difference in which constraints are checked in each phase.

Let us consider ABT_2 and two constrained agents i, j, $i < j$. In **phase I**, the partial constraint $C_{(i)j}$ is tested by j, while in **phase II** $C_{i(j)}$ is tested by i. In $DisFC_2$ it happens exactly in the opposite order: in **phase I** $C_{i(j)}$ is tested by i and in **phase II** $C_{(i)j)}$ is tested by j. This is due to the type of information sent (values in ABT_2, consistent sub-domains in $DisFC_2$) and the partial constraint owned by each agent, but this is not a fundamental difference. ABT_2 is complete, correct and terminates [Zivan and Meisels, 2005a]. The ABT_2 algorithm appears in Figure 8.3. It uses the same types of messages as $DisFC_2$.

8.3.3 Single Phase Strategy for *PKC*

[Zivan and Meisels, 2005a] suggest that, instead of checking some constraints in **phase I** and the rest in **phase II**, all constraints can be simultaneously tested in a single phase. To achieve this, they propose *the single phase strategy*, in which each agent has to check all its partially known constraints with both higher and lower priority agents. In the single phase strategy, an agent has to inform all its neighboring agents when it takes a new value, and nogood messages can go in both directions (from lower priority to higher priority agents as in ABT but, also from higher to lower). In the following, we present $DisFC_1$ and ABT_1 algorithms, which have been obtained by combining the single phase strategy with $DisFC$ and ABT, respectively. Similarly to the two-phase strategy, the description of the algorithms follows the order in which they were published.

ABT_1

The single phase Asynchronous Backtracking for PKC (ABT_1) results from combining ABT with the single phase strategy [Zivan and Meisels, 2005a]. ABT_1 enforces only constraint privacy. Unlike ABT_2, each ABT_1 agent checks simul-

```
procedure ABT_2()
  compute Γ⁻, Γ⁺; myValue ← empty; end ← false; nogoods ← false;
  repeat
    ABT-I();
    if (¬end)
      exchange Γ⁻, Γ⁺;
      ABT-II();
      exchange Γ⁻, Γ⁺;
    until end or ¬nogoods
procedure ABT-I()
  quiescence ← false;
  CheckAgentView();
  while (¬end ∧ ¬quiescence) do
    msg ← getMsg();
    switch(msg.type)
      ok?     : ProcessInfo(msg);
      ngd     : ResolveConflict(msg);
      adl     : SetLink(msg);
      stp     : end ← true;
      qes     : quiescence ← true;
procedure ABT-II()
  quiescence ← false;
  for each child ∈ Γ⁺(self) do sendMsg:ok?(child, myValue));
  while (¬quiescence) do
    msg ← getMsg();
    switch(msg.type)
      ok?   : if ¬ consistent(myValue, msg.Value) then
                sendMsg:ngd(self = myValue ⇒ msg.Sender ≠ msg.Value);
      ngd   : add(lhs(msg.Nogood, myNogoodStore)); myValue ← empty;
      qes   : quiescence ← true; nogoods ← true; /*quiescence with ngd messages*/
      qnn   : quiescence ← true; nogoods ← false; /*quiescence without ngd messages*/
```

Figure 8.3: The ABT_2 algorithm for the *PKC* model. Missing procedures appear in Figure 5.2, Chapter 5.

taneously its constraints with all constraining agents (i.e. higher priority and lower priority constraining agents.). This means that, after an assignment, the agent has to send its current value to all constraining agents via **ok?** messages. Despite an agent may know the assignments of all its neighboring agents, it searches a value which is only consistent with the assignments of higher priority agents. That is, values from the domain of agents are eliminated *only if they violate constraints with higher priority agents* [Zivan and Meisels, 2005a]. After an agent finds a value which is consistent with all assignments of higher priority agents, it checks the assignment against the assignment of each lower priority agent. If a conflict is detected, the agent keeps its assignments but sends a **ngd** message to the lower priority agent. This message includes the assignments of

both agents. ABT_1 processes all **ngd** messages in the same way as ABT, no matter they come from higher or lower priority agents. [Zivan and Meisels, 2005a] prove that ABT_1 inherits the good properties of ABT: completeness, correctness and termination. The code of ABT_1 appears in Figure 8.4. Only the new or modified parts with respect to ABT (Figure 5.2, Chapter 5) are shown. ABT_1 agents have the same data structures as ABT agents.

```
procedure ABT1()
  myValue ← empty; end ← false; compute Γ+, Γ−;
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?    : ProcessInfo(msg);
      ngd    : ResolveConflict(msg);
      adl    : SetLink(msg);
      stp    : end ← true;
procedure ProcessInfo(msg)
  UpdateAgentView(msg.Assig);
    if ¬consistent(myValue, msg.Assig) then
      if (msg.Sender ∈ Γ+) then
        SendMsg:ngd(msg.Sender, self = myValue ⇒ msg.Sender ≠ msg.Assig);
      else CheckAgentView();
procedure CheckAgentView(msg)
  if ¬consistent(myValue, myAgentView[Γ−]) then
    myValue ← ChooseValue();
    if (myValue) then
      for each child ∈ Γ+(self) ∪ Γ−(self) do sendMsg:ok?(child, myValue);
      for each child ∈ Γ+(self) such that ¬consistent(myValue, child.Assig) do
              sendMsg:ngd(child, self = myValue ⇒ ¬child.Assig);
    else Backtrack();
```

Figure 8.4: The ABT_1 algorithm for the PKC model. Missing procedures appear in Figure 5.2, Chapter 5.

DisFC$_1$

The $DisFC_1$ algorithm results of applying the single phase strategy proposed by [Zivan and Meisels, 2005a] to $DisFC$. $DisFC_1$ enforces assignment privacy while keeping constraint privacy in the PKC model. The algorithm works like ABT_1 with the following differences: (i) instead of sending its current value, agent i sends the subset of D_j consistent with it, and (ii) the assigned value is replaced by a sequence number. The code of the $DisFC_1$ algorithm appears in Figure 8.5. In addition to the data structures of $DisFC_2$, each agent keeps in $myFilteredDomains$ the received filtered domains from lower priority constraining agents.

```
procedure DisFC1()
  myValue ← empty; end ← false; compute Γ+, Γ−;
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?   : ProcessInfo(msg);
      ngd   : ResolveConflict(msg);
      adl   : SetLink(msg);
      stp   : end ← true;
procedure ProcessInfo(msg)
  UpdateAgentView(msg.Sender = msg.Seq);
  if (msg.Sender ∈ Γ+(self)) then myFilteredDomain[msg.Sender] ← msg.Domain;
  if (msg.Sender ∈ Γ−(self)) then UpdateDomain(msg);
  if ¬(myValue ∈ msg.Domain) then
    if (msg.Sender ∈ Γ+(self)) then
        SendMsg:ngd(msg.Sender, self = mySeq ⇒ msg.sender ≠ msg.Seq);
    else CheckAgentView();
procedure CheckAgentView()
  if (myValue = empty ∨ myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      for each child ∈ Γ+(self) ∪ Γ−(self) do
            sendMsg:ok?(child, mySeq, compatible(D(child), myValue));
      for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
            sendMsg:ngd(child, self = mySeq ⇒ child ≠ child.Seq);
    else Backtrack();
```

Figure 8.5: The $DisFC_1$ algorithm for the *PKC* model. Missing procedures appear in Figure 8.2.

8.3.4 $DisFC_2/DisFC_1$: Theoretical Results

Here, we prove that $DisFC_2$ and $DisFC_1$ inherit the good properties of *DisFC*: completeness, correctness and termination. The proof that ABT_2/ABT_1 hold these properties appears in [Zivan and Meisels, 2005a]. We assume that all algorithms follow the same agent ordering. The proof will be based on the fact that all algorithms can generate the same nogoods.

Lemma 8.3.2. *A nogood can be generated by $DisFC_2/DisFC_1$ iff it can be generated by DisFC.*

Proof. We distinguish between two types of nogoods: explicit and implicit. In *DisFC* and $DisFC_2/DisFC_1$, an explicit nogood is generated as a consequence of an **ok?** message. An implicit nogood is generated by resolution of the set of nogoods that forbid every value of a variable.

Explicit nogoods. Let i and j be two agents, $i < j$ in the total order, and let $x_i = v \Rightarrow x_j \neq w$ be a nogood generated by $DisFC_2/DisFC_1$. If the pair

(v, w) is forbidden in $C_{i(j)}$, this nogood will be generated in j after receiving the **ok?** message containing the filtered domain of x_j associated to the assignment $x_i = v$, and it will be stored in j. Otherwise, if the pair (v, w) is permitted in $C_{i(j)}$ but forbidden in $C_{(i)j}$, it will be generated in i, and it will be sent from i to j and stored in j. In any case, if it is forbidden by, at least, one partial constraint it is enough for the pair (v, w) to be forbidden by the total constraint C_{ij}. Therefore, it will be generated by *DisFC*.

Let us assume that (v, w) is forbidden by C_{ij}, so *DisFC* will generate the nogood $x_i = v \Rightarrow x_j \neq w$ when sending the **ok?** message from i to j containing the filtered domain of x_j associated to the assignment $x_i = v$. By the definition of *PKC* model, we know that the pair (v, w) will be forbidden by, at least, one of the partial constraints. If it is forbidden by $C_{i(j)}$, the nogood will be generated in j after receiving the **ok?** message containing the filtered domain of x_j associated to the assignment $x_i = v$, and stored in j. If it is forbidden by $C_{(i)j}$, the nogood will be generated in i when i sends j an **ok?** message containing the filtered domain associated to $x_i = v$ and j sends i an **ok?** message containing the filtered domain associated to $x_j = w$ (this requires two phases in $DisFC_2$ but a single one in $DisFC_1$). This nogood will be sent to j, and stored there. So, in both cases the nogood is generated (and stored in j).

Implicit nogoods. Analogous to the proof given in Proposition 8.2.1 for implicit nogoods. □

Proposition 8.3.3. *$DisFC_2$/$DisFC_1$ are correct, complete and terminate.*

Proof. *Correctness.* If $DisFC_2/DisFC_1$ network has reached quiescence. In that case, every partial constraint is satisfied (otherwise, quiescence cannot be reached). If every partial constraint is satisfied, every total constraint is satisfied as well (by definition of partial constraints, see Subsection 8.3.1). Therefore, the reported solution is a true solution.

Completeness. $DisFC_2/DisFC_1$ perform the same kind of search as *DisFC*: total ordering of agents, asynchronous instantiation, resolving nogoods, adding links, etc. Their only difference is that (i) agents send filtered domains to higher and lower priority agents through **ok?** messages, and (ii) if an agent i receives an **ok?** message a one lower priority agent j and the filtered domain included in the message is inconsistent with the i's current assignment, the agent will sends a nogood message to j. These points are crucial in the generation of nogoods. However, by Lemma 8.3.2, we know that these changes do not cause any modification in the set of nogoods generated by these algorithms with respect to *DisFC*. Consequently the $DisFC_2/DisFC_1$ algorithms will discard the same parts of the search space as *DisFC*, but not other parts. Since *DisFC* is complete, $DisFC_2/DisFC_1$ are also complete.

Termination. An argument similar to the one used in completeness applies here. Nogoods rule out parts of the search space. Because the total ordering of agents, discarded parts accumulate, so *DisFC* terminates in a finite search space. Since $DisFC_2/DisFC_1$ generate the same nogoods as *DisFC*, and they perform the same kind of search, $DisFC_2/DisFC_1$ also terminate. □

8.3.5 An Example

Let us consider the problem of safely locating a queen Q and a knight k on a 4×4 chessboard (see Section 8.3.1 for the formal definition of the n-pieces m-chessboard problem). Each piece is handled by an independent agent, and none knows the identity of the other piece, so the *PKC* model applies here. There is a single constraint between Q and k. Initially we assume that Q has higher priority than k, so the constraint is directed $Q \rightarrow k$. We show the execution of $DisFC_2$, ABT_2, ABT_1 and $DisFC_1$ on this problem.

Considering $DisFC_2$, the algorithm execution is as follows (assuming that values are selected lexicographically). When **phase I** starts, both pieces take value 1. Q informs k of its filtered domain with respect to Q value, and k changes its value to 7. Quiescence is reached and **phase II** starts. The constraint direction is reversed, now $k \rightarrow Q$. k informs Q of its filtered domain with respect to k value. Q realizes that its current value is forbidden, so it sends a **ngd** message to k. Quiescence is reached and the constraint direction is reversed again, now $Q \rightarrow k$. **phase I** starts. k realizes that its value is eliminated by the nogood contained in the **ngd** message received in **phase II**. Therefore, k changes its value to 8. Quiescence is reached and **phase II** starts. The constraint direction is reversed, now $k \rightarrow Q$. k informs Q of its filtered domain with respect to k value. Q finds that its current value is permitted, so it does nothing. Quiescence is reached causing termination because no **ngd** message has been generated in **phase II**.

ABT_2 works as follows. **phase I** starts locating each piece in the first position of the chessboard. Afterwards, Q informs k of its current assignment. This causes k to take value 2, which is consistent according to its partial constraint with the value of Q. Then, quiescence is reached and **phase II** starts. The constraint direction is reversed, now $k \rightarrow Q$. k informs Q of its value. Q notes that its current value is not consistent with the value of k, so it sends a **ngd** message to k. Quiescence is reached and the constraint direction is reversed again, now $Q \rightarrow k$. **phase I** starts. k realizes that its value is eliminated by the *Nogood* contained in the **ngd** message received in **phase II**, so k changes its value to 3. Quiescence is reached and **phase II** starts. The constraint direction is reversed, now $k \rightarrow Q$. k informs Q of its value. Q finds that its current value is not consistent with the value 3 of k and then sends a Nogood message to k. Quiescence is reached and the search is resumed in **phase I**. This process continues until k takes value 8. Then, k informs Q of the new assignment. Q checks that its assignment is consistent with k value. Quiescence is reached causing termination because no **ngd** message has been generated in **phase II**.

ABT_1 works as follows. First, each piece takes value 1. Agents exchange two **ok?** messages. Q informs k that it has taken value 1, and k informs Q that it has taken value 1. When Q receives the **ok?** message, it sends a **ngd** message to k informing that its current value it is not consistent with the value of Q (both pieces are in the same chessboard cell). When k receives the **ok?** and **ngd** messages it takes value 2 and informs this change to Q via an **ok?** message. When Q receives this message it notes that its current value is not consistent

with k value, so Q sends an **ngd** message to k informing it to change its value. Then, k takes the value 3 and informs Q. This process continues until k takes value 8. Then, it informs Q of the new assignment. Q finds that its current value is consistent with the value of k, so it does nothing. Quiescence is reached causing termination.

When $DisFC_1$ starts, both pieces take value 1. Agents exchange two **ok?** messages. Q informs k its filtered domain with respect to Q value, and k informs Q its filtered domains with respect to k value. When Q receives the message it sends an **ngd** message to k informing that its current value it is not consistent with Q value. When k receives the **ok?** and **ngd** messages it takes value 7 and informs this change to Q via an **ok?** message. When Q receives this message it notes that its current value is not permitted, so Q sends an **ngd** message to k informing that it has to change its value. Then, k takes value 8 and informs Q. Q finds that its current value is permitted, so it does nothing. Quiescence is reached causing termination.

8.3.6 Evaluating Privacy Loss of Constraints

The agents in the four proposed algorithms have to reveal some information about their constraints during the search process. In order to compare the algorithms with respect to privacy loss of constraints that they provide, we must be able to count the amount of information that can be inferred by agents in each algorithm. We look at each binary constraint as a matrix in which every entry represents the compatibility of two values, one for each agent. In the *PKC* model, for two mutually constrained agents, each owns a partial constraint. In matrix terms, each agent has a matrix, and these matrixes could be different.

To measure the privacy loss of constraint $C_{i(j)}$ held by i, we propose to count the number of constraint matrixes that are consistent with information about $C_{i(j)}$ that i reveals to j. Of these matrixes, one is the actual $C_{i(j)}$. In ABT algorithms, this number is computed by simply counting the number of entries (e) of $C_{i(j)}$ that i has shared with j. Thus, we can define the privacy loss of $C_{i(j)}$ by the following equation:

$$2^{(|D_i||D_j|-e)}$$

where $|D_i|$ and $|D_j|$ are the sizes of domains of i and j, and e is the number of entries that i has shared with j. The larger this number is, the less information j can infer about $C_{i(j)}$. The critical privacy value is 1, that occurs when j deduces the values of all entries in $C_{i(j)}$.

In ABT, if i has higher priority than j, e is equal to $|D_i||D_j|$, otherwise $e = 0$. This is due to ABT assumes that the lower priority agent in each constraint has to know the whole constraint and higher priority agents never receive the assignments of lower priority ones.

In ABT_1, e is equal to the number of different *negative entries* of $C_{i(j)}$ that i reveals to j. An entry is negative if the combination of values that the entry represents is incompatible. Each different explicit nogood holds a negative entry.

As we have defined before, a nogood is said explicit if it is a direct consequence of a conflict defined by a problem constraint. In ABT_1, when an agent i receives an **ok?** message from a lower priority agent j, i checks if the j's value containing in that message and its own current value are valid with respect to $C_{i(j)}$. If i detects that such values are inconsistent, i sends an explicit **ngd** message to j. Explicit nogoods are differentiable from the other nogood, because they always are sent from higher to lower priority agents. Hence, if i has higher priority than j, e is equal to number of different negative entries revealed by i to j, otherwise $e = |D_i||D_j|$.

In ABT_2, e is equal to the number of different negative entries, like in ABT_1, plus the number of different *positive entries* that j can infer from $C_{i(j)}$. An entry is positive if the combination of values that the entry represents is compatible. Agents reveal negative and positive entries in **phase II**. At this phase, an agent j sends its value to every constraining agent i with higher priority. Then, agent i detects that j's value is either inconsistent or not. If it is inconsistent, i will send to j an explicit nogood, otherwise i will do nothing. If j receives such a **ngd** message, j deduces that the entry in $C_{i(j)}$ that correspond to the values of i and j is negative. Otherwise, j deduces that this entry is positive. Hence, if i has higher priority than j, e is equal to number of different explicit nogoods and explicit goods that j can infer from $C_i(j)$, otherwise $e = |D_i||D_j|$.

Regarding $DisFC_2/DisFC_1$, it is more costly to compute the number of constraint matrixes which are consistent with the information exchanged between every pair of constraining agents. In both algorithms, if agent i is constrained with j and i has higher priority, instead of sending the actual value of i to j like in ABT algorithms, it sends the subset of D_j that is compatible with the actual value of i. After reception, j does not know the actual value of i, but it knows a complete row of $C_{i(j)}$ without knowing its position in the matrix. As search progresses, j may store new rows of $C_{i(j)}$. At the end, j has a subset of rows without knowing their position. In addition, some search episodes (information exchanged by agents in **phase II** in $DisFC_2$, nogood messages from high to low priority agents in $DisFC_1$) may reduce the number of acceptable positions for a particular row [Meisels and Zivan, 2006]. With all this, we construct a *CSP* instance where the variables are the rows, their domains are the acceptable positions, under the constraints that two different rows cannot go to the same position and every row must get a position. Computing all solutions of this instance we obtain all matrixes which are compatible with the information obtained from the search using the following equation:

$$\sum_{i=0\ldots s} 2^{(|D_i||D_j|-|D_i|rows_i)} - rep$$

where s is the number of solution of the *CSP* instances, $rows_i$ is the number of rows that solution i locates in $C_{i(j)}$ and rep is the number of matrixes that are counted more than once. Of all the matrixes included in the above equation, one is $C_{i(j)}$.

Let us consider an example that illustrates how we generate the *CSP* in-

stance. Figure 8.6 shows an example of a part of $DisFC_1$ execution. The domain of both variables is: $\{a, b, c, d, e\}$. Matrixes are expressed assuming that values of each variable are lexicographically ordered. An entry is positive $(+)$ if the combination of values that this entry represents is consistent for the constraint. Conversely, an entry is negative $(-)$ if the combination of values that the entry represents is invalid.

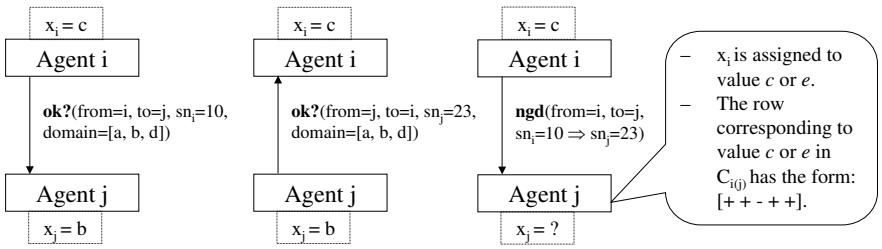


Figure 8.6: Information deduced by a $DisFC_1$ agent after receiving a **ngd** message from a higher priority agent.

This example starts when agent i sends an **ok?** message to agent j saying that i's current value is compatible only with the values $\{a, b, d\}$ of j's domain (Figure 8.6.a). From this message, j can deduce that there exits a row in $C_{i(j)}$ with the form: $[+ + - + -]$. However, with this information j cannot infer the position of this row in $C_{i(j)}$. After receiving the **ok?** message, j takes a new value and sends an **ok?** message to i saying that the only valid values for i are in the domain $\{a, b, d\}$ (Figure 8.6.b). Similar to j, i may deduce that $C_{(i)j}$ has a row with the form $[+ + - + -]$. When i receives that message from j, it discovers that the received **ok?** message is incompatible with its own value and sends a **ngd** to j (Figure 8.6.c). From this message, j can deduce that i's current value does not belong the compatible domain it has sent it previously, therefore, the current value of i is either c or e. Thus, j may deduce that the row corresponding to c or e in $C_{i(j)}$ has the form $[+ + - + -]$ (i.e. the value c or e for agent i are compatible only with a, b and d for agent j). Suppose that the example continues; j changes its value and, following the same reasoning we have described before, j discovers that $C_{i(j)}$ has a row with the form $[+ + - + -]$ that corresponds to value c or d. Then, the algorithm ends.

In this example, we construct a *CSP* instance that includes two variables x_1 and x_2; one for each time the row $[+ + - + -]$ has been discovered. The domains of these variables are: $\{c, e\}$ and $\{c, d\}$, respectively. There exists a constraint between variables in order to avoid that both rows be associated to value c. The *CSP* has 4 solutions: $s_1 = \{x_1 = x_2 = c\}$; $s_2 = \{x_1 = c, x_2 = d\}$; $s_3 = \{x_1 = e, x_2 = c\}$; $s_2 = \{x_1 = e, x_2 = d\}$. The privacy loss of $C_{(i)j}$

is: $2^{(25-5)} + 3 \times 2^{(25-10)} - rep$, where $rep = 2^{(25-5)} + 2 \times 2^{(25-10)}$, because the matrixes for s_2 and s_3 are included in the matrixes for s_1, and as well as matrixes for s_1 are included in the matrixes for s_4. Hence, according to the information that j has at the end of the search, the original $C_{i(j)}$ could be one of those $2^{15} = 32768$ matrixes.

In $DisFC_2$, the process to build the *CSP* instance is the almost same as for $DisFC_1$ with some minor differences. Similar to ABT_2, all inferences are done at **phase II**. When j receives a **ngd** message from a higher priority agents, the process is the same as for $DisFC_1$. In addition, if after j has sent an **ok?** message to a higher priority agent i, j does not receive a **ngd** message that agent, this mean that i's value appears in the domains sent by j to i in the **ok?** message. Thus, the rows previously sent from i to j corresponds to one of the value that appears in domain included in the **ok?** messages sent from j to i.

Contrary to *ABT* algorithms, breaking constraint privacy in $DisFC_2/DisFC_1$ requires that all solutions of a *CSP* instance have to be computed (an NP-hard task). In practice, solving this instance requires significant effort and in some cases subsumption testing is required.

Because of the required static ordering of agents, in all proposed algorithms lower priority agents tend to work more than higher priority ones. This causes lower priority agents to reveal more information than higher priority ones. Therefore, they tend to have a higher privacy loss. To overcome these differences, we propose to use three functions to aggregate the values of privacy loss of constraints of all agents. These functions are: minimum (min), median (med) and average (avg). The minimum function measures the amount of information that a better informed agent has about partial constraints of other agents. The median function measures the amount of information that the agent with the median value of privacy loss of constraints has. The average function measures the average value of the privacy loss of constraints for all agents.

8.4 Experimental Results

We have performed some experiments to assess the potential of proposed algorithms when solving *DisCSPs* under privacy requirements. In our experiments we have used distributed random problems to evaluate privacy loss of constraints during algorithm execution. We evaluate the performance of algorithms according to communication cost, computation effort and privacy loss of assignments and constraints. As discussed in Chapter 3, we measure the computation effort in terms of the number of non-concurrent constraint checks ($nccc$) and the communication cost in terms of the total number of exchanged messages (msg). Regarding privacy constraints, we evaluate privacy loss of constraints in terms of the number of constraint matrixes consistent with the information exchanged among agents (see Subsection 8.3.6).

We performed experiments on the following class of random problems: $\langle 15, 10, 0.4, p_2 \rangle$. Problems include 15 agents ($n = 15$) each holding one variable and 10 values for each variable ($k = 10$). The value of the constraint one

density is $p_1 = 0.4$. The tightness (p_2) varies between 0.1 and 0.9 to cover all ranges of problem difficulty. For each pair of density and tightness (p_1, p_2), we generated 100 different instances. We assumed that problems are initially expressed under the *PKC* model. The p_2 values correspond to the tightness of the total constraints. For each pair of constraining agents i and j, the set of forbidden pairs of values of the total constraint C_{ij} are randomly split between the two partial constraints $C_{i(j)}$ and $C_{(i)j}$. Since ABT requires that j knows the whole constraint C_{ij}, j must receive, via message(s), the partial constraint $C_{i(j)}$ from i before running ABT.

The results of ABT_2/ABT_1 and $DisFC_2/DisFC_1$ are reported and discussed in the following subsections. For all algorithms, messages are processed by packets, as described in Chapter 6.

8.4.1 *ABT*, ABT_2 and ABT_1

Figure 8.7 (left) shows the average number of non-concurrent constraint checks required by ABT, ABT_2 and ABT_1 to solve the considered random instances. The overhead of algorithms that preserve constraint privacy is clear. ABT_2 and ABT_1 run more than twice slower than standard ABT. For problems in the phase transition region, ABT_1 outperforms ABT_2 by 30%. On instances with high tightness, ABT_1 behaves like the standard algorithm (i.e. the difference between the algorithms is constant) while the performance of ABT_2 deteriorates. This phenomenon occurs because the problem solved by ABT_2 in **phase I** is actually less tight than the problem solved by ABT_1. Therefore when ABT_1 detects that the problem is too tight to be solved, ABT_2 works hard to solve a problem with lower tightness.

Figure 8.7 (right) presents the results for the total number of exchanged messages among agents during algorithm execution. Comparing ABT_2 and ABT_1 with the standard ABT, we observe the similar result previously seen in terms of *nccc*: algorithms that preserve constraint privacy are much more costly than ABT. In contrast to the results for *nccc*, for low tightness instances, the number of messages sent by ABT_2 is smaller than for ABT_1. This is because the agents in ABT_1 send **ok?** messages to all their neighbors while in ABT_2 in each phase the agents send **ok?** messages only in one direction. For tighter problems, ABT_1 sends less messages than ABT_2 when solving instances to the right of the complexity peak.

Table 8.1 reports the minimum (*min*), median (*med*) and average (*avg*) of the numbers of constraint matrixes that are consistent with information exchanged between every pair of constraining agents. Larger values mean higher privacy. The critical privacy occurs when the number of constraint matrixes is 1, which means that one agent may build the whole partial constraint matrix of one other agent.

According to *min*, ABT and ABT_1 are the algorithms with lower and higher constraint privacy, respectively. ABT reaches the value of critical privacy for every value of p_2. This is because ABT requires that the lower priority agent of each constraint knows the entire matrix in order to check the consistency of the

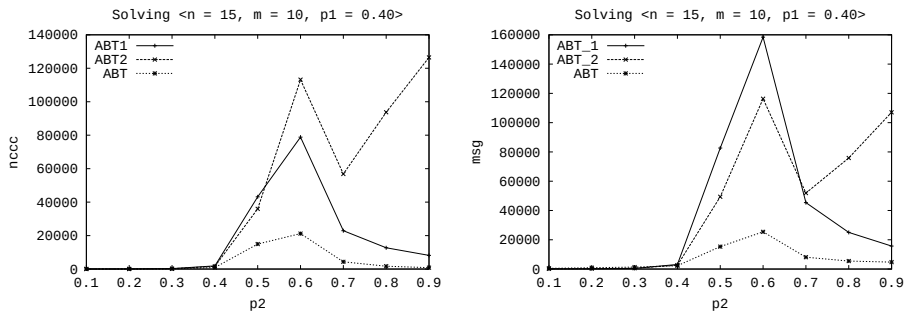


Figure 8.7: Non-concurrent constraint checks (left) and number of messages (right) for ABT, ABT_1, ABT_2 on binary random *DisCSP*.

constraint. In contrast, no agent in ABT_1 and ABT_2 receive enough information to infer completely the partial constraint of other agent. Except for $p_2 = 0.9$, ABT_2 agents can infer more information about the partial constraint of the other agents than ABT_1 agents. According to *med*, the number of consistent matrixes is the same for the three algorithms. This means that, for half of the agents both algorithms provide the same privacy. In terms of *avg*, ABT, ABT_2 and ABT_1 shows very similar results. For $p_2 \geq 0.8$, ABT_1 is just one order of magnitude worse than the other two algorithms. This is due to the first phase of $DisFC_2$, where only some constraints are considered, is enough to detect that instances are unsolvable.

These results show a trade-off between efficiency and constraint privacy on ABT algorithms. The more private an algorithm is, the more inefficient it is. ABT requires less computation effort and communication cost than ABT_1 and ABT_2. According to constraint privacy, we conclude that: (1) ABT is less private than ABT_2 and ABT_1 because ABT needs that all the information about constraints must be revealed in advance; (2) the better informed agent in ABT_2 have more information about other agents' partial constraints that the

	ABT			ABT_1			ABT_2		
p_2	min	med	avg	min	med	avg	min	med	avg
0.1	**1**	10^{30}	10^{30}	10^{29}	10^{30}	10^{30}	10^{29}	10^{30}	10^{30}
0.2	**1**	10^{30}	10^{30}	10^{29}	10^{30}	10^{30}	10^{28}	10^{30}	10^{30}
0.3	**1**	10^{30}	10^{30}	10^{29}	10^{30}	10^{30}	10^{27}	10^{30}	10^{30}
0.4	**1**	10^{30}	10^{30}	10^{27}	10^{30}	10^{30}	10^{26}	10^{30}	10^{30}
0.5	**1**	10^{30}	10^{30}	10^{23}	10^{30}	10^{30}	10^{18}	10^{30}	10^{30}
0.6	**1**	10^{30}	10^{30}	10^{20}	10^{30}	10^{30}	10^{10}	10^{30}	10^{30}
0.7	**1**	10^{30}	10^{30}	10^{19}	10^{30}	10^{30}	10^{15}	10^{30}	10^{30}
0.8	**1**	10^{30}	10^{30}	10^{19}	10^{30}	10^{29}	10^{19}	10^{30}	10^{30}
0.9	**1**	10^{30}	10^{30}	10^{18}	10^{30}	10^{29}	10^{25}	10^{30}	10^{30}

Table 8.1: Constraint privacy of ABT, ABT_1 and ABT_2 measured by the minimum (*min*), median (*med*) and average (*avg*) of the numbers of consistent constraint matrixes.

better informed agent in ABT_1; (3) globally, ABT_1 agents have slightly more information about partial constraint matrixes of other agents than ABT_2 agents.

8.4.2 *DisFC*$_2$ and *DisFC*$_1$

We have tested $DisFC_2$ and $DisFC_1$ on the same class of random problems used in the evaluation of ABT_2/ABT_1: $\langle n = 15, m = 10, p_1 = 0.4, p_2 \rangle$. Similarly, we evaluate the performance of algorithms by the number of non-concurrent constraint checks ($nccc$), the number of exchanged messages (msg) and the minimum (min), median (med) and average (avg) of the numbers of constraint matrixes that are consistent with the information that agents reveal during resolution.

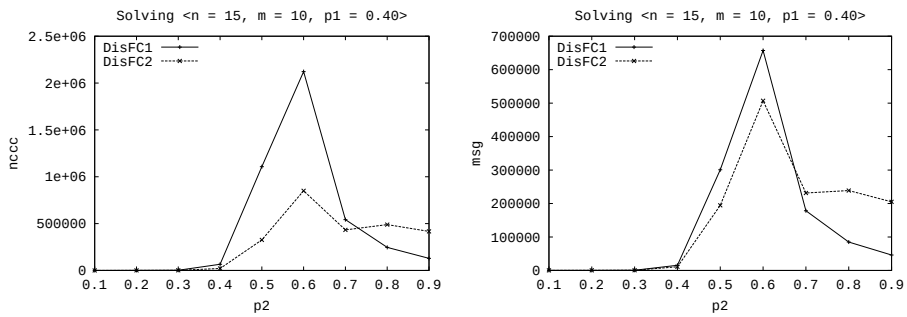


Figure 8.8: Non-concurrent constraint checks (left) and number of messages (right) for $DisFC_1$, $DisFC_2$ on binary random *DisCSP*.

Figure 8.8 (left) shows the number of non-concurrent constraint checks required by $DisFC_1$ and $DisFC_2$ to solve considered problems. We observe that $DisFC_1$ is more than twice slower than $DisFC_2$ for problems close to the complexity peak. However, $DisFC_2$ is worse than $DisFC_1$ for tighter problems ($p_2 \geq 0.7$). These results differ from those obtained for ABT versions where ABT_1 is faster than ABT_2 for instances close to the complexity peak. This is explained by the following fact. Before sending an **ok?** message, a $DisFC_1$ agent has to check consistency with each value in the domain of every agent constrained with it. Conversely, a $DisFC_2$ agent has to check consistency with every lower priority agent constrained with it, which generates a lower number of constraint checks. Comparing these algorithms with ABT_2/ABT_1, the former have to perform a much larger number of constraint checks than the latter (because they send filtered domains, not just single values), which justify these results.

Figure 8.8 (right) presents the results of communication cost in term of the number of messages exchanged among agents in $DisFC_1$ and $DisFC_2$. Mainly, the relative ordering of algorithms is the same as that shown in the ABT versions. $DisFC_1$ agents exchange more messages than $DisFC_2$ for problems with

constraint tightness lower than 0.6 ($p_2 \leq 0.6$). Although, *DisFC*$_2$ is more costly than *DisFC*$_1$ for the rest of the problems ($p_2 \geq 0.7$).

Comparing these results with the *ABT-2ph/ABT-1ph* in problems with low and high density, we see that *DisFC* algorithms are much slower. Similarly, agents in *DisFC* algorithms send more messages. This inefficiency of *DisFC* algorithms can be explained. In *ABT-2ph/ABT-1ph* as in standard *ABT*, assignments are sent to neighboring agents which concurrently check there consistency with the local assignments. In *DisFC*, in order to keep the assignments private, agents must perform the consistency checks of their proposed assignments sequentially, checking the entire domains of their neighboring agents. This increases the non-concurrent effort of the *DisFC* algorithms.

We also evaluate *DisFC*$_1$ and *DisFC*$_2$ according to privacy issues. Loss in assignment privacy depends on constraint tightness. In *DisFC*$_1$, when an agent j sends a filtered domain to a higher priority agent i, i may find inconsistency between i's assignment and the filtered domain sent by j. In this situation, i sends a **ngd** message to j. When j receives this **ngd** message, it can deduce that i's assignment is one of those values that are not included in the filtered domain sent to i. Because agents work asynchronously, j cannot infer anything about i's assignment if no **ngd** message is received from i. This is true because i may have changed its assignment before receiving the **ok?** message.

Following a similar reasoning, every *DisFC*$_2$ agent can infer the assignments of higher priority agents when receiving **ngd** messages from them. All these messages are exchanged in the second phase of *DisFC*$_2$. Since agents' assignments remain unchanged in the second phase, an agent j can deduce the assignments of a higher priority agent i even when it does not receive a **ngd** message from i. In this case, j knows that i's assignment is one of the values that is valid according to the filtered domain sent by j to i in the second phase of the algorithm. Note that all these inferences are produced in the second phase of the algorithm, which is only invoked at certain points of the execution. One can say that, *DisFC*$_1$ agents have a higher certainty of the assignments of higher priority agents during the whole execution of the algorithm, while *DisFC*$_2$ agents may suspect the current assignments of other agents only at some points of the execution. Similarly to the standard *ABT*, assignments of lower priority agents are keeping completely private for higher priority agents during the execution of both algorithms.

Table 8.2 shows the minimum (*min*), median (*med*) and average (*avg*) numbers of constraint matrixes which are consistent with information exchanged between constraining agents. Again, larger values mean higher privacy and value 1 represents the critical privacy loss. In terms of *min*, we observe that *DisFC*$_1$ improves over or ties with *DisFC*$_2$, for all values of p_2 up to $p_2 = 0.4$. This is due to the following fact. Synchronization points required by *DisFC*$_2$ to change from one phase to the other make every agent i to know exactly the set of compatible values of other agent j that shares a constraint with it. This reduces the number of possible acceptable positions for the row of partial constraint $C_{(i)j}$ that corresponds to the filtered domain sent by j to i before phase change. In

	$DisFC_1$			$DisFC_2$		
p_2	min	med	avg	min	med	avg
0.1	10^{23}	10^{28}	10^{29}	10^{26}	10^{28}	10^{28}
0.2	10^{23}	10^{27}	10^{29}	10^{25}	10^{27}	10^{28}
0.3	10^{16}	10^{24}	10^{29}	10^{21}	10^{26}	10^{27}
0.4	10^{7}	10^{14}	10^{28}	10^{3}	10^{24}	10^{25}
0.5	**1**	10^{9}	10^{25}	**1**	10^{6}	10^{20}
0.6	**1**	10^{6}	10^{9}	**1**	10^{6}	10^{10}
0.7	**1**	10^{6}	10^{12}	**1**	10^{6}	10^{10}
0.8	**1**	10^{6}	10^{10}	**1**	10^{6}	10^{12}
0.9	**1**	10^{6}	10^{10}	**1**	10^{6}	10^{17}

Table 8.2: Constraint privacy of $DisFC_1$ and $DisFC_2$ measured by the minimum (*min*), median (*med*) and average (*avg*) of the numbers of consistent constraint matrixes.

contrast, an $DisFC_1$ agent may deduce the position of a particular row only after receiving an explicit **ngd** message (one coming from a higher priority agents). For tighter instances ($p_2 \geq 0.5$), both algorithms reveal enough information allowing at least one agent to build completely the partial constraint matrix of other one.

In terms of *med*, we find that both algorithms reveal almost the same up to $p1 = 0.4$, where $DisFC_2$ outperforms $DisFC_1$. For tighter instances, we conclude that approximately in half of partial constraint matrixes all rows are revealed during search since 10^6 is close to $10! = 3.6 \times 10^6$ (the number of permutations of 10 rows). According to *min* and *med*, we observe that privacy loss is high as p_2 increases. In terms of *avg*, higher privacy loss occurs at the complexity peak ($p_2 = 0.6$). For $p_2 > 0.7$, it seems that $DisFC_2$ tends to increase. This occurs because the first phase of the algorithm does not find any solution. Therefore, the second phase is not performed and explicit **ngd** are not sent. In contrast, in $DisFC_1$, explicit **ngd** are sent throughout the algorithm.

Comparing these results with those obtained about privacy loss of constraints in *ABT* algorithms, we observe the following. *DisFC* algorithms reveal more information about partial constraints than agents in *ABT* algorithms. Actually, in some problems, all rows of the partial constraint matrixes are revealed. This happens because *DisFC* agents exchange filtered domains which correspond to rows in the partial constraint matrixes. Since no agent i reveals its assignments during the search, an agent j cannot easily infer which values of i's domain correspond to the filtered domains received from i. Although apparently, in *DisFC* algorithms, an agent may find out the partial constraints of another agent, as happens in *ABT*, it is important to keep in mind that in *DisFC* algorithms this process has a cost that is equal to search cost for finding all a *CSP*'s solutions.

8.5 Summary

In this chapter, we have treated privacy in the context of *ABT*. To this end, we have differentiated among privacy of domains, assignments and constraints.

The first considers that the whole domain of every variable must be concealed by holding agent during resolution. The second considers that actual assigned values are not made public in the solving process. The third is concerned with constraints that are initially private (the *PKC* model) between agents, and they remain as private as possible during the solving process.

Somehow, the variable-based model assumed by *ABT* guarantees a substantial domain privacy because only the holding agent of each variable knows its domain. Here, we have discussed some ideas to enhance further domain privacy in *ABT*. Referring to assignment and constraint privacy, we have presented two families of *ABT*-like algorithms, ABT_2/ABT_1 and $DisFC_2/DisFC_1$, to perform the actual solving while trying to keep the above mentioned privacy levels. They were initially conceived as two phase algorithms, although later both phases were joined into a single one. These algorithms are not perfect and leak some information in the solving process, but less than standard *ABT*.

The proposed algorithms have been implemented and evaluated on random *DisCSP* instances. Empirically we observe an expected fact: to achieve some privacy, algorithms degrade their performance (because they have to conceal some values, exchange more messages etc). In addition, the more privacy is required the less performance is obtained by the solving algorithm. This is the price one has to pay to achieve the required privacy.

Chapter 9

Enhancing Privacy with Lies

In the preceding chapter we introduced $DisFC_1$ and $DisFC_2$, two asynchronous backtracking algorithms that are concerned with privacy of assignments and constraints. In both algorithms, each time an agent changes its value, it sends, via **ok?** messages, the domain of the low priority agent that is compatible with its current value. Although, this strategy allows agents to not reveal their current assignments, it may cause a privacy loss on shared constraints that was initially overlooked when assuming the *PKC* model. This phenomenon is clearly observed in the experimental evaluation of the preceding chapter, in particular, in Table 8.2 which measures privacy loss of constraints. In those experiments, at least one $DisCSP_1/DisCSP_2$ agent can infer totally the partial constraint matrix of other agent (after computing all the solution of a *CSP*). To further enhance privacy of constraints, in this chapter we propose a novel algorithm that works like $DisFC_1$ but it may lie about the compatible domains of other agents. The new algorithm requires a single extra condition: if an agent sends a lie, it has to tell the truth in finite time afterwards.

The structure of this chapter is as follows. In Section 9.1, we present the strategy of false domains to reduce the information that $DisFC_1/DisFC_2$ agents can deduce from other's partial constraints. In Section 9.2, we present $DisFC_{lies}$, an algorithm that works as $DisFC_1$ and implements the strategy of false domains. We formally prove that the $DisFC_{lies}$ is correct, complete and terminates in Section 9.3. We discuss privacy improvements of $DisFC_{lies}$ with respect to $DisFC_1$ in Section 9.4. In Section 9.5, we evaluate $DisFC_{lies}$ on random *DisFC*. Finally, in Section 9.6, we summarize this chapter.

9.1 The Strategy of False Domains in $DisFC_1/DisFC_2$

To enhance constraint privacy in $DisFC_1/DisFC_2$ we propose that agents could lie. Instead of sending true rows of $C_{i(j)}$, the algorithm may send true and *false* rows. Each false row represents a lie. False rows will make much more difficult the hypothetical reconstruction of $C_{i(j)}$ by agent j, but it has to be done keeping the correctness and completeness of the algorithm.

This idea can be formalized as follows. If i has d values $D_i = \{v_1, v_2, \ldots, v_d\}$, it is assumed that i has an extended domain $D'_i = \{v_1, v_2, \ldots, v_d, v_{d+1}, \ldots, v_{d+k}\}$ of $d + k$ values. We call *true values* the first d values, while the rest are *false values*. When i assigns the true value $v_p, 1 \leq p \leq d$, it sends to agent j the subset of values that are compatible with v_p (that is, a true row of $C_{i(j)}$). When i assigns the false value $v_q, d < q \leq d+k$, it sends an invented subset of compatible values to j (that is a row which does not exist in $C_{i(j)}$). The only concern that an agent must have after assigning a false value is that it must tell the truth (assign a true value or perform backtracking if no more true values are available) in finite time. The point is that no solution could be based on a false value, so assignments including false values have to be removed in finite time (in fact, in a shorter time than required to detect quiescence).

9.2 The $DisFC_{lies}$ Algorithm

$DisFC_1$ offers a better platform for privacy than $DisFC_2$, because it has no synchronization points between phases. For this reason, we implement the strategy of false domains on top of $DisFC_1$ (although it can also be implemented on top of $DisFC_2$).

We call $DisFC_{lies}$ the new version of $DisFC_1$ where agents may exchange false pruned domains. $DisFC_{lies}$ appears in Figure 9.1. It includes most of the procedures, functions and data structures of $DisFC_1$, and uses the same types of messages. Each agent has a local clock to control when it has to tell the truth after a lie. In the structure $FalseDomains$, each agent puts away the false domains that it will send to its neighbors for each false value the agent's variable can take. $D_{true}(self)$ is the set of true values for $self$, while $D_{false}(self)$ is the set of its false values. $D(self)$ is the union of these two sets.

In the main procedure, $self$ first initializes its data structures and generates the false domain that it will send for each false value. Secondly, $self$ assigns a value to its variable by invoking the function $CheckAgentView$. This value may be false or not. Then, $self$ enters in a loop where incoming messages are received and processed. This loop ends, and therefore the algorithm, when $self$ receives either an **stop** or a **qcc** message from *system*. This is a special agent that handles these messages in the same way it did in $DisFC_1$. If $self$ ends the search because a **qcc** message, it means that a problem has at least one solution, otherwise, the problem is unsolvable. Quiescence state can be detected by specialized algorithms [Chandy and Lamport, 1985]. However, in

```
procedure DisFC_lies()
  myValue ← empty; end ← false; compute Γ+, Γ−; tsaytrue ← 0;
  for each value ∈ D_false(self) do
    for each neig ∈ Γ+(self) ∪ Γ−(self) do generate FalseDomain[value][neig];
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?    : ProcessInfo(msg);
      ngd    : ResolveConflict(msg);
      adl    : SetLink(msg);
      stp, qcc : end ← true;
    if (value ∈ D_false(self)) and (gettime() ≥ tsaytrue) then TakeATrueValue();

procedure CheckAgentView()
  if (myValue = empty or myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      if (myValue ∈ D_false(self)) then
        for each neig ∈ Γ+(self) ∪ Γ−(self) do
          sendMsg:ok?(neig, mySeq, FalseDomain[myValue][neig]);
        tsaytrue ← gettime() + t_lies;                    /* t_lies < t_quies */
      else
        for each neig ∈ Γ+(self) ∪ Γ−(self) do
          sendMsg:ok?(neig, mySeq, compatible(D(neig), myValue));
        for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
          sendMsg:ngd(child, self = mySeq ⇒ ¬child.Assig);
        tsaytrue ← 0;
    else Backtrack();

procedure ResolveConflict(msg)
  if coherent(msg.Nogood, Γ−(self) ∪ {self}) then
    CheckAddLink(msg);
    add(msg.Nogood, myNogoodStore); myValue ← empty;
    CheckAgentView();
  else if coherent(msg.Nogood, self) then
    if (myValue ∈ D_false(self)) then
      sendMsg:ok?(neig, mySeq, FalseDomain[myValue][neig]);
    else
      SendMsg:ok?(msg.Sender, mySeq, compatible(D(msg.Sender), myValue);

procedure TakeATrueValue()
  tsaytrue ← 0; myValue ← ChooseATrueValue();
  if (myValue) then
    mySeq ← mySeq + 1;
    for each neig ∈ Γ+(self) ∪ Γ−(self) do
        sendMsg:ok?(neig, mySeq, compatible(D(neig), myValue);
    for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
        sendMsg:ngd(child, self = mySeq ⇒ ¬child.Assig);
  else Backtrack();

function ChooseATrueValue()
  for each v ∈ D_true(self) not eliminated by myNogoodStore do
    if consistent(v, myAgentView[Γ−]) then return (v);
    else add(x_j = val_j ⇒ self ≠ v, myNogoodStore); /*v is inconsistent with x_j's value */
  return (empty);
```

Figure 9.1: The $DisFC_{lies}$ algorithm for asynchronous backtracking search. Missing procedures/functions appear in Figure 8.5, Chapter 8.

order to assure the completeness and correctness of the algorithm, the time t_{quies} required by *system* to assure quiescence in the network (i.e no message has traveled through the network within the last t_{quies} units of time) must be larger than t_{lies}, the maximum time agents may wait until rectifying their lies, thus $t_{lies} < t_{quies}$.

In the following, we prove that $DisFC_{lies}$ is correct, complete and terminates.

9.3 Theoretical Results

Lemma 9.3.1. *When $DisFC_{lies}$ finds a solution, the last filtered domain received by agent i from agent j corresponds to a (true) row in the partial constraint matrix $C_{i(j)}$.*

Proof. For $DisFC_{lies}$ the current variables' assignments are a solution if no constraint is violated and network has reached quiescence. Let us assume that $DisFC_{lies}$ reports a solution in which variable x_i takes a false value. So the last filtered domains sent by agent i are false too. However, $DisFC_{lies}$ requires that, after lying, an agent must rectify in finite time. That is, assigning a true value and sending to its neighbors the true filtered domains, or performing backtrack. So, at least one **ok?** message or a **ngd** message has traveled through the network after i lied, in contradiction with the initial assumption that the network had reached quiescence. Therefore, the solution condition cannot be reached unless true filtered domains are sent in the last messages from any agent. □

Proposition 9.3.2. *$DisFC_{lies}$ is correct.*

Proof. If a solution is claimed, we have to prove that current agents' assignments satisfy their partial constraints. Lemma 9.3.1 shows that if $DisFC_{lies}$ reports a solution the last variables's assignments correspond to true values. Therefore, one can prove that $DisFC_{lies}$ is correct by using the same arguments to prove that $DisFC_1$ is correct.

Let us assume quiescence in the network. If the current assignment is not a solution, there exists at least one partial constraint that is violated by agent j. In that case, agent j has sent a **ngd** message to agent i, the closest agent involved in the conflict. This **ngd** is either discarded as obsolete or accepted as valid by agent i. If the message is discarded, it means that some message has not yet reached its recipient, which breaks our assumption of quiescence in the network. If the message is valid, i has to find a new consistent values, which will produce several **ok?** messages or one new **ngd** message, which again breaks our assumption of quiescence in the network. □

Proposition 9.3.3. *$DisFC_{lies}$ is complete.*

Proof. Considering only nogoods based on true values, we can prove that $DisFC_{lies}$ is complete by using the same arguments to prove that $DisFC_1$ is complete. Since nogoods resulting from an **ok?** message are redundant with

respect to the partial constraint matrixes, and the additional nogoods are generated by logical inference, the empty nogood cannot be inferred if the problem is solvable.

Let us prove that $DisFC_{lies}$ cannot infer inconsistency based on false values if the problem is solvable. Suppose that agent j detects inconsistency because a lie introduced by agent i. We know that j detects inconsistency when it infers an empty nogood. Besides, we know that the left-hand side of the nogoods (justifications of forbidden values) stored by j is either empty or includes agents with higher priority than j. Since we assume that inconsistency discovered by j is based on the false value of i, i is before j in the agents' ordering and there is at least one nogood stored by j including i in its left-hand side. Therefore, when j finds no consistent value, it has to send a backtracking messages to i, which breaks our assumption that j derives an empty nogood. □

Lemma 9.3.4. *$DisFC_{lies}$ agents will not store indefinitely nogoods based on false values.*

Proof. Let us assume that a false nogood (i.e. a nogood including an agent with a false value) will be stored indefinitely by an agent. In that case, the lying agent cannot change its variable's assignment, otherwise the nogood will become obsolete and, therefore, deleted by the holder agent. But a lying agent *must* tell the truth in finite time. So, in finite time, the agent storing the false nogood will be informed of a new true value, the false nogood will become obsolete and, therefore, it will be deleted by the holder agent. This breaks our assumption that the false nogood lasts forever. □

Proposition 9.3.5. *$DisFC_{lies}$ terminates.*

Proof. By Lemma 9.3.4, nogoods based on false values are discarded in finite time. About nogoods based on true values, $DisFC_{lies}$ performs the same treatment as $DisFC_1$. Since $DisFC_1$ terminates in finite time, $DisFC_{lies}$ also terminates in finite time. □

Proposition 9.3.6. *If a $DisFC_{lies}$ agent detects inconsistency, every agent directly connected with it has received d true rows.*

Proof. Let i be that agent. If i finds the empty nogood, it means that there is a nogood for every true value of i. These nogoods have an empty left-hand side (otherwise, i could not deduce the empty nogood). So they have been produced as result of **ngd** messages coming from the lower priority agents. Therefore, every possible true value of i has been taken, so i has sent to its neighbors d true rows. □

9.4 Privacy Improvements of $DisFC_{lies}$

The inclusion of false values has two direct consequences. First, agent j may receive false rows of $C_{i(j)}$. Then j has more difficulties to reconstruct $C_{i(j)}$, since

it is uncertain whether some received rows truly belong to $C_{i(j)}$ or not. Second, this strategy decreases performance, because any computation that includes a false assignment will not produce any solution, so it is a wasted effort, only useful for privacy purposes.

For a solvable instance, Lemma 9.3.1 shows that the last assignments correspond to true values. So, agent j knows that the last message from i corresponds to a true assignment, and it contains a true row of $C_{i(j)}$. Agent j cannot discriminate whether previous assignments are true or false, so it cannot include the rows of these messages when trying to compute $C_{i(j)}$. So j knows a single row of $C_{i(j)}$ but it does not know its location. The number of different constraint matrixes compatible with this information is approximately $d \cdot 2^{(d^2-d)}$ (d, the number of possible locations for the true row, times $2^{(d^2-d)}$, the number of compatible matrixes when d elements are known). This is a big difference with the approach without lies, where all received rows truly belong to $C_{i(j)}$.

For an unsolvable instance, Proposition 9.3.6 shows that every agent j directly connected with the agent i that detects inconsistency would have received d true rows. In addition, since all possibilities have been tried, they have received $d+k$ rows (observe that j cannot receive more than $d+k$ rows). Assuming that j has received $d+k$ *different* rows, if j wants to compute $C_{i(j)}$, it has to select d rows, take them as true rows and solve the corresponding *CSP*. j has to repeat this process $\begin{pmatrix} d+k \\ d \end{pmatrix}$ times, that is, once for each different subset of d rows. This increases the number of *CSPs* to solve, in order to compute the matrixes compatible with the leaked information. However, j may have received less than $d+k$ *different* rows. In that case, j considers that some rows are repeated. If there is no way to identify repeated rows, in addition to the previously described combinations, we have to consider each possible row as possibly repeated, largely increasing the number of *CSP* instances to solve. As a consequence, the privacy level of the solving process is improved.

9.5 Experimental Results

In this Section, we compare the performance of $DisFC_1$ and $DisFC_{lies}$ solving instances of binary random classes. As seen in Chapter 3, a binary random class is defined by $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d the number of values per variable, p_1 the ratio of existing constraints and p_2 the ratio of forbidden value pairs. We solved instances in the class $\langle 15, 10, 0.4, p_2 \rangle$ with varying tightness (p_2) between 0.1 to 0.9 in increments of 0.1. To create these instances in *PKC*, we first generate random instances and then we split the forbidden tuples of each constraint between its two partial constraints.

We consider three versions of $DisFC_{lies}$ that differ from each other in the number of false values that their agents add to initial domains: 1, 3 and 5 false values. When an agent takes a value, it chooses between true and false values with probability 0.5. t_{lies} is randomly chosen between 1 and 99 internal units of time. Messages are processed by packets, as described in

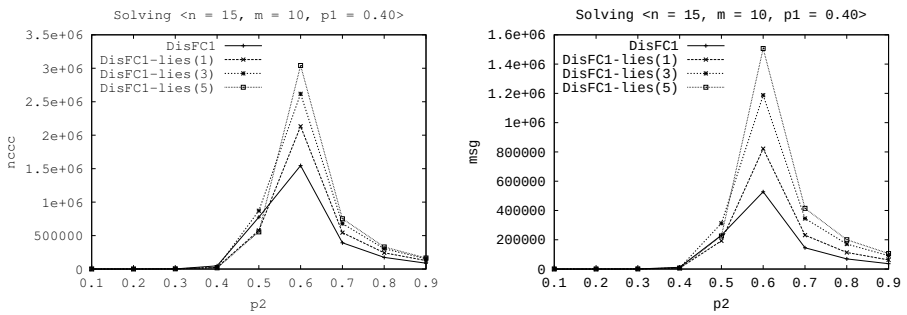


Figure 9.2: Computation and communication cost of *DisFC* and versions of $DisFC_{lies}$.

[Brito and Meseguer, 2004].

Algorithmic performance is evaluated by communication effort, computation cost and constraint privacy. Communication effort is measured by the total number of exchanged messages (msg). Computation cost is measured by the number of non-concurrent constraint checks ($nccc$) [Meisels et al., 2002]. Constraint privacy is measured by the number of constraint matrixes consistent with the information exchanged among agents (see Section 8.3.6). Generally, lower priority agents work more than higher priority ones, therefore they reveals more information than higher priority ones. Thus, we report the minimum (min), median (med) and average (avg) of the numbers of constraint matrixes that are consistent with information exchanged among agents.

Figure 9.2 shows the computation and communication costs. In both plots, results are averaged on 100 instances. In terms of computation cost, we observe that $DisFC_{lies}$ is more costly than *DisFC*, and the cost increases with the number of allowable lies. The difference between algorithm is greater at the complexity peak ($p_2 = 0.6$). Except for $p_2 = 0.5$, $DisFC_{lies}(5)$ always requires more $nccc$ than the others, while *DisFC* performs the lowest number of $nccc$. Similar results appear for communication costs.

Table 9.1 contains the values of parameters min, med and avg to measure constraint privacy. Larger values mean higher privacy. The critical privacy occurs when the number of constraint matrixes is 1 (at least one agent knows exactly the partial constraint matrix of one of its constraining agents). Regarding constraint privacy in $DisFC_1$, the values of min and med decrease when p_2 increases. Actually, in problems with constraint tightness greater than 0.4, at least one agent can infer exactly the partial constraint of one of its constraining agents (see column min). From med values in unsolvable instances ($p_2 \geq 0.6$), we conclude that approximately in half of partial constraint matrixes all rows are revealed during search since 10^6 is close to $10! = 3.6 \times 10^6$ (the number of permutations of 10 rows). In terms of avg, higher privacy loss occurs at the complexity peak ($p_2 = 0.6$).

Regarding constraint privacy in $DisFC_{lies}$, we notice the following. In solv-

	$DisFC_1$			$DisFC_{lies}(1)$			$DisFC_{lies}(3)$			$DisFC_{lies}(5)$		
p_2	min	med	avg	min	med	avg	min	med	avg	min	med	avg
0.1	10^{23}	10^{28}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.2	10^{23}	10^{27}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.3	10^{16}	10^{24}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.4	10^{7}	10^{14}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.5	**1**	10^{9}	10^{25}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.6	**1**	10^{6}	10^{9}	**3.3**	10^{30}	10^{29}	**20**	10^{30}	10^{29}	**221**	10^{30}	10^{29}
0.7	**1**	10^{6}	10^{12}	**2**	10^{30}	10^{29}	**10.7**	10^{30}	10^{29}	**163**	10^{30}	10^{29}
0.8	**1**	10^{6}	10^{10}	**2.3**	10^{30}	10^{29}	**50.3**	10^{30}	10^{29}	**270**	10^{30}	10^{29}
0.9	**1**	10^{6}	10^{10}	**3.3**	10^{30}	10^{29}	**25.3**	10^{30}	10^{29}	**426**	10^{30}	10^{29}

Table 9.1: Constraint privacy measured by the minimum (*min*), median (*med*) and average (*avg*) of the numbers of consistent constraint matrixes. Averaged on 10 instances.

able instances ($0.1 \leq p_2 \leq 0.5$), $DisFC_{lies}$ versions achieve the same level of privacy for *min*, *med* and *avg*, no matter the number of allowable lies. This occurs since each agent can only assure that the last filtered domain received from another agent truly corresponds to a row in the partial constraint matrix of that agent (see Lemma 9.3.1), which is independent to the number of false values that agents may have. In terms of *min* and *med*, $DisFC_{lies}$ versions are more private than $DisFC_1$. In unsolvable instances, $DisFC_{lies}$ versions have different level of privacy when considering *min*. $DisFC_{lies}(5)$ is one and two orders of magnitude more private than $DisFC_{lies}(3)$ and $DisFC_{lies}(1)$, respectively. $DisFC_{lies}(1)$ is the least private of these three algorithms although it is more private than *DisFC*. $DisFC_{lies}$ versions are equally private with respect to *med* and *avg*. For these parameters, $DisFC_{lies}$ versions are more private than $DisFC_1$.

9.6 Summary

We have shown in this chapter that lying is a suitable strategy to enhance privacy in *DisCSP* solving. We have presented $DisFC_{lies}$, a new version of the *DisFC* algorithm that may tell lies, sending false compatible domains to neighbor agents. The unique extra condition is that, after a lie, the lying agent has to tell the truth in finite time, lower than t_{quies}. We have proved that this algorithm is correct, complete and terminates. Second, we have shown analytical and experimentally that this idea effectively enhances constraint privacy in the *PKC* model, because it increases the number of partially known constraint matrixes that are compatible with the leaked information of the solving process. And third, although solving *DisCSP* lying is more costly than solving it without lies, experiments show that the extra cost required is not unreachable. It is clear that any strategy used to conceal information will have an extra cost, and this approach is not an exception. We believe that this approach could be useful for those applications with high privacy requirements.

Part IV

Applications

Chapter 10

Distributed Meeting Scheduling

Meetings are an important vehicle for human communication. The Meeting Scheduling problem (*MS*) consists of a set of people which use their personal calendars to determine *when* and *where* one or more meeting(s) could take place [Freuder et al., 2001].

The Meeting Scheduling problem is a naturally distributed problem because (1) each person knows only his/her own personal calendar before resolution and (2) people may desire to preserve the already planed meetings in their personal calendars during resolution. In the centralized approach, all people must give their private information to one person, who solves the problem and returns a solution. This approach results in a high privacy loss (each person must give his/her personal calendar to the solver). In a distributed approach, people work together, revealing some information of their personal calendars, in order to agree upon the time and the place that the meetings could be planned. In such context, it is natural to view *MS* as *DisCSPs* with privacy requirements.

In this chapter, we provide an empirical comparison of three distributed approaches (two synchronous and one asynchronous) for *MS* in terms of privacy loss. Among these approaches, two are *DisCSP* algorithms: *SCBJ* and *ABT*. We do not use the *PKC* model previously introduced because, in this problem all interagent constraints are equality constraints given in the implicit form (the concept of partially known constraint is not really applicable here).

This chapter is divided as follows. Section 10.1 gives a formal definition of the Meeting Scheduling problem and a *DisCSP* encoding for this problem. In Section 10.2, we discuss some issues related to privacy loss in the different distributed approaches. In Section 10.3, we compare empirically the considered algorithms in terms of computation effort, communication cost and privacy loss. Finally, we summarize this chapter in Section 10.5.

10.1 What is the Meeting Scheduling Problem?

The Meeting Scheduling [Freuder et al., 2001] problem (in short *MS*) is a decision-making process affecting several people, in which it is necessary to decide *when* and *where* several meetings could be scheduled.

Definition 10.1.1. Formally, an *MS* is defined by the following parameters:

- $P = \{p_1, p_2, ..., p_n\}$, the set of n people; each with his/her own calendar, which is divided into r slots, $S = \{s_1, s_2, ..., s_r\}$;
- $M = \{m_1, m_2, ..., m_k\}$, the set of k meetings;
- $At = \{at_1, at_2, ..., at_k\}$, the set of k collection of people that define which attendees must participate in each meeting, i.e. people in at_i must participate in the meeting m_i, $1 \leq i \leq k$ and $at_i \in P$;
- $c = \{pl_1, pl_2, ...pl_o\}$, the set of o places where meetings can be scheduled, places are separated by a given travel time.

Initially, people may have several slots reserved for already filled planning in their calendars. A solution to this problem answers the *where* and *when* of each meeting. This solution must satisfy the next rules:

- attendees of a meeting must agree *where* and *when* the meeting is to take place,
- no two meetings m_i and m_j can be held at same time if they have at least one attendee in common,
- each attendee p_i of a meeting m_j must have enough time to travel from the place where he/she is before the meeting starts to the place where the meeting m_j will be. Similarly, people need sufficient time to travel to the place where their next meetings will take place.

10.1.1 The Distributed Meeting Scheduling Problem

The Meeting Scheduling problem is a truly distributed benchmark, in which each attendee may desire to keep the already planned meetings in his/her calendar private. So this problem is very suitable to be treated by distributed techniques, trying to provide more autonomy to each person, and to keep preferences private. For this purpose, we define the Distributed Meeting Scheduling problem (in short *DisMS*).

Every *DisMS* instance can be encoded as a *DisCSP* as follows. There exists one agent per person. Every agent includes one variable for each meeting in which the corresponding person wishes to participate. The domains of the variables enumerate the possible alternatives of *where* and *when* meetings may occur. That is, each domain includes $k \times o$ values, where k means the number of places where meetings can be scheduling and o represents the number of slots in

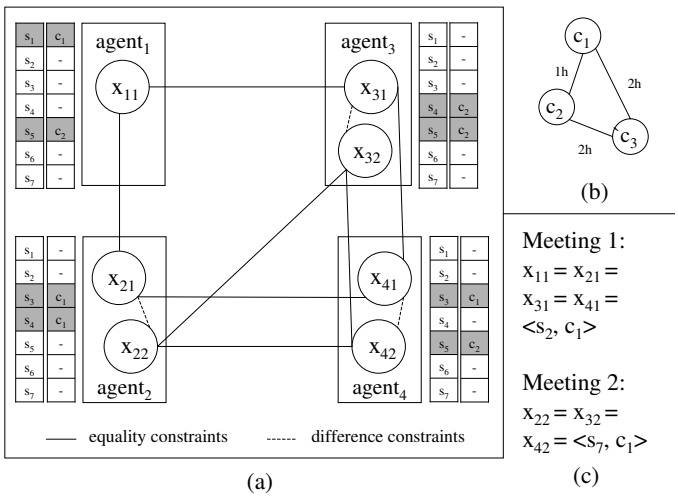


Figure 10.1: An instance of the *DisMS*. (a) The problem seen as a *DisCSP*. (b) Required times for traveling among cities. (c) A solution to the problem.

agents' calendars. There are two types of binary constraints between variables: equality and difference constraints. There exists a binary equality constraint between each pair of variables that belongs to different agents and corresponds to the same meeting. There exists a binary difference constraint between each pair of variable which belongs to the same agent.

Figure 10.1.(a) illustrates an instance of *DisMS* viewed as *DisCSP*. In that example, there are four people, $person_1$, $person_2$, $person_3$ and $person_4$, and two meetings, m_1 and m_2. Each agent has its own calendar divided into 7 slots. Regarding meeting m_1, people $person_1$, $person_2$, $person_3$ and $person_4$ are looking for a place and time when they can meet together. As for meeting m_2, people $person_2$, $person_3$ and $person_4$ have to agree on the place and time when they can meet. Meetings can take place in any of the following three places: c_1, c_2 and c_3. Two out of the seven slots in every calendar are already reserved for other personal meetings: $person_1$ has to be in place c_1 at time s_1 and in place c_2 at time s_5; $person_2$ has to be in place c_1 at time s_3 and time s_4; $person_3$ has to be in place c_2 at time s_4 and s_5; $person_4$ has to be in place c_1 at time s_3 and in place c_2 at time s_5.

In the *DisCSP* formulation for this *DisMS* instance, each person is represented by an agent, such that $agent_i$ corresponds to $person_i$. Variables x_{11} in $agent_1$, x_{21} in $agent_2$, x_{31} in $agent_3$ and x_{41} in $agent_4$ make up meeting m_1, while variables x_{22} in $agent_2$, x_{32} in $agent_3$, x_{42} in $agent_4$ make up meeting m_2. The *DisCSP* contains 10 constraints: one difference constraint between each pair of variables held by an agent and one equality constraint between variables that belong to the same meeting. The time required for travel among cities is given in Figure 10.1.(b). A solution to this example appears in Figure 10.1.(c): all the

people agree to meet at place c_1 at time s_2 and $person_2$, $person_3$ and $person_4$ agree to meet at place c_1 at time s_7.

10.2 Privacy on *DisMS* Algorithms

To solve a *DisMS* instance, agents must cooperate and communicate among them in order to determine *when* and *where* meetings will take place. During this process, agents reveal some information about their personal calendars. Privacy loss is concerned with the amount of information that agents reveal to other agents. In the *DisCSP* formulation for *DisSM*, variable domains represent the availability of people to hold a meeting at a given time and place, which actually is the information that agents desire to hide from other agents. In that sense, measuring the privacy loss of a *DisMS* modeled as *DisCSP* is actually the same as measuring the privacy loss of variable domains.

Later on this section we will analyze privacy loss on three distributed algorithms for *DisMS*. The first algorithm is based on a very simple communication protocol, in which agents make proposals about *when* and *where* meeting can occur following a *Round Robin* order [Freuder et al., 2001]. We refer this algorithm as *RR*, which is presented next. The other two algorithms are *SCBJ* and *ABT*, which have seen previously in Chapter 4 and 5, respectively.

From *DisMS* perspective, agents in these algorithms make proposals to other agents about *when* and *where* meetings could take place.[1] A proposal can be accepted or rejected by recipient agents. Depending on the answers of recipient agents, the proposing agent can infer some information about the other agents. Similarly, when an agent receives an assignment proposal, some information is leaked about the proposing agent. In following we describe which knowledge can be inferred by agents in each case [Franzin et al., 2004]:

1. When a proposal is rejected, the proposing agent can infer that it may be because the rejecting agent either has a meeting in that slot already or has a meeting that could not be reached if the proposed meeting was accepted.

2. When a proposal is accepted, the proposing agent can infer that the accepting agent does not have a meeting in that slot, that possible meetings that are incompatible with the proposal do not occur in the possible another agent's calendar.

3. When an agent receives a proposal from another agent, the recipient agent can infer that the proposing agent has not a meeting in that slot, nor in any slot that would be incompatible because of the distance constraints.

The aforementioned points constitute what we call *the process of knowledge inference*. In this work, we actually consider only part of the information that

[1] Notice that when an agent in $SCBJ/ABT$ assigns a value to its variable and inform of this to higher priority agents, actually it is proposing *when* and *where* a particular meetings could occur.

agents can infer by using the first point. The inferred knowledge in this case is very vague because the agent that receives a rejection cannot deduce anything for certain regarding the personal calendar of the rejecting agent. From the other two cases (points 2 and 3), we identify three kinds of information that can be deduced from agents:

Positive Information This is the information that denotes that can have a meeting in certain locations at certain times.

Negative Information This is the information that denotes that an agent cannot have a meeting in certain locations at certain times.

Open Slots This is the information related to slots in which an agent surely does not have any meeting already in any of the places.

The concepts of **Positive Information** and **Negative Information** are similar to the definitions of "present-meeting information" and "future-meeting information" given in [Franzin et al., 2004]. Regarding **Open Slots**, this information can be deduced by an agent if its proposal is accepted by another agent. In this case, the accepting agent does not any meeting already in a time-and-city that is incompatible with the proposal because the distance constraints.

In the following subsections we analyze the details of the process of knowledge inference within each considered algorithm presuming that the number of meetings to be scheduled is simply one ($k = 1$).

10.2.1 The *RR* Algorithm

In *RR*, one agent at a time proposes to the others agents the time and the location that meeting may occur. The ordering in which proposals are made follows the Round Robin strategy. When an agent receives a proposal, it responds only to the proposing agent if this proposal is possible according to its calendar.

RR was presented and used in [Freuder et al., 2001] to solve *DisMS*. We have rewritten this algorithm to present it in a similar way to the previously mentioned and discussed algorithms in this thesis. The new code appears in Figure 10.2. Agents exchange six types of messages: **pro**, **ok?**, **gd**, **ngd**, **sol**, **stp**. **pro** messages are used by agents to select the proposing agent. When an agent receives a **pro** message, this causes the agent to become the proposing agent. After the proposing agent chooses the time/place that the meeting can be scheduled, it informs about its decision to rest of agents via **ok?** messages. When an agent receives an **ok?** message, it checks if the received proposal is valid with respect to the previously scheduled appointments in its calendar. If this proposal is consistent, the agent sends a **gd** message to the proposing agent announcing it accepts the proposal. Otherwise, the agent sends a **ngd** message to the proposing agent saying that it rejects the proposal. Messages **sol** and **stp** are responsible for announcing to agents that a solution has been found or the problem is unsolvable, respectively.

```
procedure RR()
  end ← false; allgood ← true;
  received ← 0;
  if self is the first agent in the agent ordering then Propose();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      pro    : Propose();
      ok?    : Process-ok?(msg);
      gd, ngd:GetAnswers(msg);
      sol, stp: end ← true;

procedure Propose()
  allgood ← true; received ← 0;
  if all possible proposals have been rejected do
    for each other agent i do sendMsg:stp(i, self);
  else
    repeat
        generate a proposal p at random;
        check p to see if :
            the time-slot referred in p is empty in self's calendar;
            p has not conflict in the current agents's calendar;
            p has not been already rejected;
    until the above conditions are satisfied
    for each other agent i do sendMsg:ok?(i, self, p);
    end if

procedure GetAnswers(msg)
  received ← received + 1;
  switch(msg.type)
    gd     : SaveAndInferFromGood(msg);
    ngd    : allgood ← false;
             SaveAndInferFromNoGood(msg);
  if (received = (#agents − 1)) then
    if (allgood = true) do for each other agent i do sendMsg:sol(i, self);
    else
      /*the next agent in the ordering is selected as the proposing agent*/
      next ← (self.id + 1)%#agents;
      sendMsg:pro(i, self);
    end if
  end if

procedure Process-ok?(msg)
  check msg.p to see if :
      the time-slot referred in msg.p is empty in self's calendar;
      msg.p has not conflict in the current agents's calendar;
      msg.p has not been already rejected;
  if the above conditions are satisfied do sendMsg:gd(msg.from, self);
  else sendMsg:ngd(msg.from, self);
  end if
```

Figure 10.2: The code of *RR*.

Based on the previously discussed message system, it logically follows that the process of knowledge inference is clear-cut. The message system is simple: proposals are sent via **ok?** messages; , which are accepted via **gd** messages or rejected via **ngd** messages.

10.2.2 *SCBJ*

Agents in *SCBJ* assign variables sequentially. They exchange assignments and nogoods through **ok?** and **ngd** messages, respectively. From the point of view of *DisMS*, agents propose or reject the proposals made by other agents. **ok?** messages are used for the agents to send proposals regarding the time and the place that are acceptable for a meeting. Contrary to what happens in *RR*, **ngd** messages only mean that someone has rejected the proposal, but the agent who has done such is not easily discovered. It is important to note that *SCBJ* loses some possible privacy in the sense that as the agents send **ok?** messages down the line, each subsequent agent knows that all the previous agents have accepted this proposal.

For the purpose of clarification, take for example a problem consisting of five agents each one representing a person with its own personal calendar. Suppose that the first agent sends a proposal to the second agent about meeting Monday at 9:00 am in Barcelona. The second agent accepts the proposal and sends it to the third agent. Then, the third agent finds this proposal to unacceptable and therefore sends a **ngd** message to the second agent, effectively eliminating the possibility of meeting Monday at 9:00 in Barcelona. In this case, it is impossible for agent 1 or 2 to to know where the rejection originated, because any of the agents situated ahead of them, could be responsible, an the **ngd** message came via the third agent. Furthermore, it is impossible for both the first and second agents to discover the agent that rejected the proposal, as it could have been any of the agents situated ahead of them, as was simply relayed back to them via the third agent. However, in such systems, there is one specific case in which it is possible to determine which agent has rejected a proposal. In this example, such a case would occur if all of the agents 1-4 have already received the proposal and then the fifth agent rejects. When this happens, the fourth agent knows that it was the fifth agent that rejected the proposal as the latter is the only agent capable of sending such a message, assuming that the fourth agent knows that the system contains only five agents in total.

10.2.3 *ABT*

Agents in *ABT* assign the variables asynchronously and concurrently. Mainly, agents exchange assignments and nogoods through **ok?** and **ngd** messages respectively. Similar to *SCBJ*, **ok?** messages represent proposals, while **ngd** messages represent rejections. The pervious discussion regarding **ngd** message in *SCBJ* is still valid for this algorithm. However, there is a difference with respect to **ok?** messages: since a sending agent may send an **ok?** to all of the

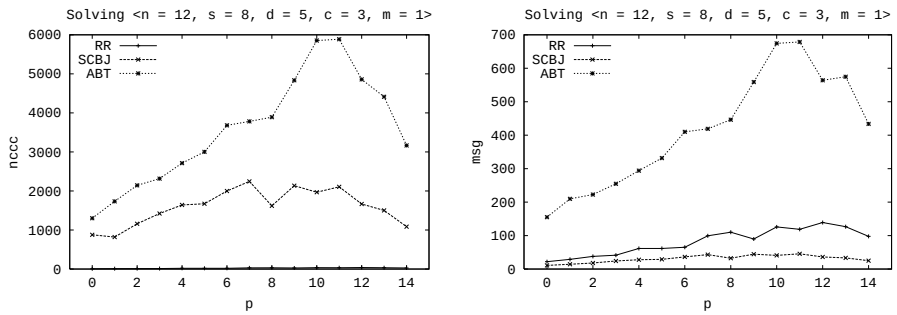


Figure 10.3: Constraint checks and number of messages for *RR*, *SCBJ* and *ABT* on Distributed Meeting Scheduling instances.

lower priority agents, the information contained in this message is only valid for the sending agent and is revealed only to the receiving agents.

10.3 Experimental Results

In this section, we evaluate two synchronous algorithms (*RR* and *SCBJ*) as well as one asynchronous algorithm (*ABT*) for solving *DisMS* instances. In order to compare the algorithms, we make use of three measures: computation effort, communication cost, and privacy loss. We measure computation effort using the number of non-concurrent constraint checks (*nccc*), communication cost in terms of the number of messages exchanged (*msg*) and privacy loss using the three types of information that agents may deduce regarding other agents' calendars: **Positive Information**, **Negative Information**, **Free Slots**.

Lower priority agents in *SCBJ* and *ABT* tend to work more than higher priority ones, which causes them to reveal more information than higher priority agents. In order to analyze the difference in the amount of privacy loss, we give the minimum, maximum and average amount data for each type of information that agents can find out from other agents' plans.

In our experiments, we deal with *DisMS* instances in which there has to be only one meeting scheduled and which admit at least one solution. Each problem is composed of 12 people, 5 days, with 8 time slots per day and 3 meeting places. This gives $5 \cdot 8 \cdot 3 = 120$ possible values in each agent's domain. Meetings and time slots are both one hour long. The time required for travel among the three cities is 1 hour, 1 hour and 2 hours. *DisMS* instances are generated by randomly establishing p predefined meetings in each agent's calendar. The value of p varies from 0 to 14.

We consider that an agent in *RR* performs a constraint check each time it checks if meeting can occur at a certain time/place. In all algorithms, each time an agent has to propose, it chooses a time/place at random. Agents in *ABT* process messages by packets instead of processing one by one (Section 6.4) and

implement the strategy of selecting the best nogood (Section 5.4.1).

Figure 10.3 gives the results in terms of *nccc* (on the left) and *msg* (on the right) for each algorithm averaged over 100 instances. For every value of p, we observe that *RR* requires less *nccc* than *SCBJ* and *ABT* has the worst results. This can be explained by looking at how agents reject invalid proposals in each algorithm. In *RR*, the proposing agent broadcasts its proposal to all the other agents. Then, the receiving agents check if this proposal is valid or not. This process can be performed concurrently by all receiving agents, and therefore, its computation effort is just one non-concurrent constraint check. (Actually, the *nccc* value for *RR* is equal to number of proposals made before finding a solution.) In *SCBJ*, the active agent sends the proposal (received from prior agents) to the next agent when this is valid for him/her. It could be happen that a proposal make by the proposing agent in *RR* and by the first agent in the ordering in *SCBJ* and *ABT* decide to meet at certain time and certain place which is inconsistent for an agent lower in the ordering for *SCBJ* and *ABT*. In *RR*, this inconsistency will be found as soon as this agent responds to the proposing agent. In *SCBJ*, otherwise, this will be found when this agent receives the proposal, after that all the prior agents have accepted it and have performed several non-concurrent constraint checks. Regarding *ABT*, this results can be explained because (1) agents choose their proposals randomly and (2) these proposals are made possibly without knowing the proposals of higher priority agents. The combination of these two facts make *ABT* agents more likely to fail when trying to reach an agreement regarding *where* and *when* the meeting could take place. Considering *msg*, the relative ordering among agents changes only in the sense that *RR* is worse than *SCBJ*. This difference between both algorithms occurs probably because *SCBJ* omits the accept messages used by *RR*.

Figure 10.4 and Figure 10.5 report the privacy loss with respect to each information type. Regarding **Positive Information** (plots on the left in Figure 10.4), we observe that according to minimum values of **Positive Information**, *ABT* and *SCBJ* have similar behavior, while *RR* is a little worse, especially for more difficult problems ($p > 6$). This plot indicates that the less informed agent in terms for each algorithm infers only the **Positive Information** derived from the problem solution. That is, when a solution is reached, this agent can deduce that all the other agents can meet at the time and the location given by the found solution.

In terms of maximum values it is apparent that the difference between algorithms is greater. *ABT* is always less private than *SCBJ* and *RR* is the algorithm with the best results. From these values we may conclude that the better informed agent in *RR* has less **Positive Information** than the better informed agent in the other two algorithms. In terms of average values of **Positive Information**, the plot shows that *ABT* agents discover on average approximately two time slots in which each agent is available while for agents in the other two algorithms this value is approximately one. *SCBJ* shows better results than *RR* on instances with larger numbers of already planned appointments.

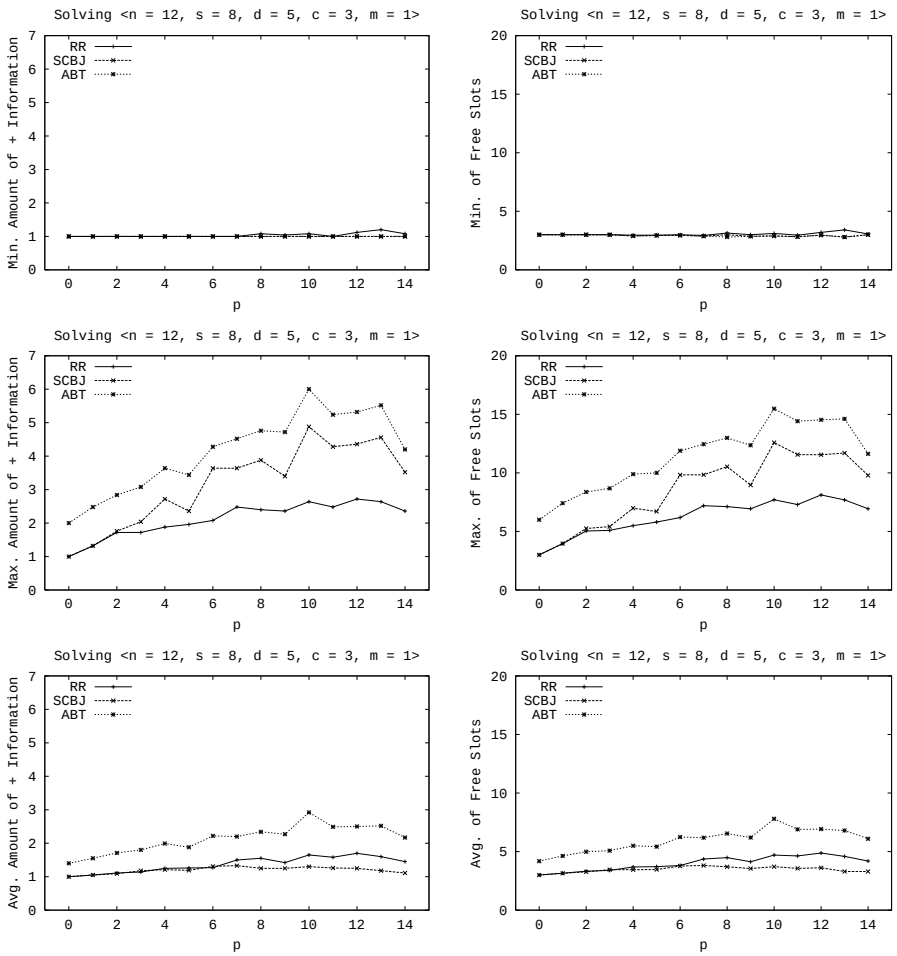


Figure 10.4: Privacy loss for *RR*, *SCBJ* and *ABT* on Distributed Meeting Scheduling instances.

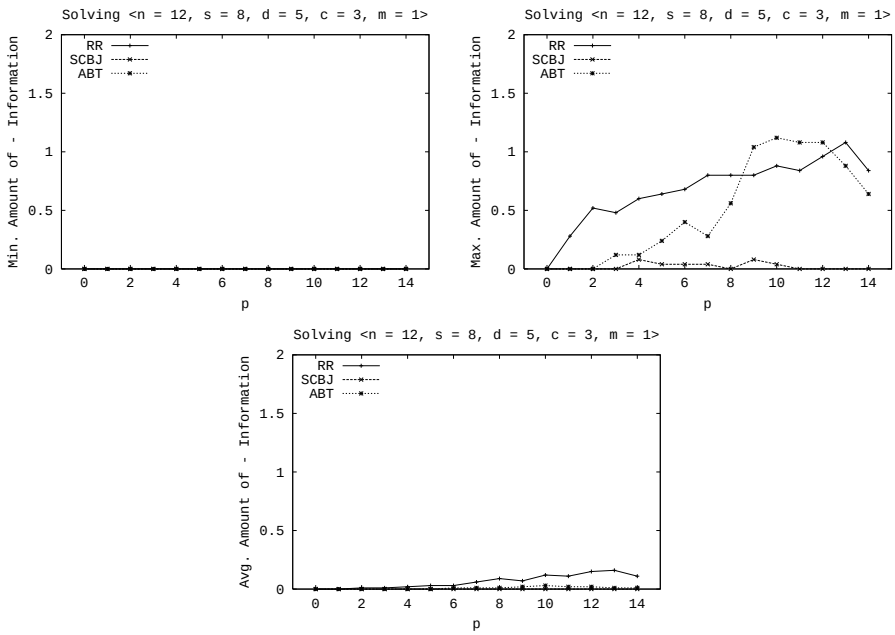


Figure 10.5: Privacy loss for *RR*, *SCBJ* and *ABT* on Distributed Meeting Scheduling instances.

Considering the number of **Free Slots** that agents leak from other agents, in terms of minimum values, we observe that the three algorithms have results similar to **Positive Information** ones. The less informed agent for each algorithms identifies almost 10 free slots from the other agents' calendars. In terms of maximum values, the better informed agent in ABT infers almost twice more **Free Slots** than the better informed agent in RR, while the better informed agent in $SCBJ$ discovers more than this agent in RR. In terms of average values, ABT agents also find more **Free Slots** than the other two algorithms. $SCBJ$ lightly outperforms RR on instances with $p > 6$.

The amount of **Negative Information** deduced in each algorithm is practically null (Figure 10.5). Only for instances with higher number of already planned appointments, the better informed agent in ABT/RR can identify at most one rejection from the other agents.

From the above results, we observe the following. Regarding computation effort and communication cost, ABT, the asynchronous algorithm, is less economic than the other two algorithms. This is because ABT agents work concurrently and select their proposals randomly, which makes more difficult ABT agents to reach an agreement regarding *when* and *where* they can meet together. Consistently for all tested instances, $SCBJ$ requires less messages than RR, however, it performs more constraint checks. Regarding privacy, for the three algorithm the greater amount of information revealed identifies time slots in which agents surely does not have any meeting in any of the places. In terms of this parameter, ABT is always worse than the synchronous algorithms. On average, $SCBJ$ agents reveal less information than RR. However, the better informed agent in $SCBJ$ deduce more information than in RR.

10.4 Related Work

The Meeting Scheduling problem has been widely studied in several works for a long time. Nevertheless, the paper that quantified first the privacy loss of information from people's calendars was published in this decade [Freuder et al., 2001]. This work, however, does not use for its resolution neither of the *DisCSP* algorithms presented in this thesis. In its experimental evaluation, authors included a version of RR whose agents can give one or all the justifications for a rejection. Very recently, [Franzin et al., 2004] propose a version of MS in which each person has a preference value associated to each place in each time slot. This work demonstrates some relations among solution quality according to people's preference, efficiency and privacy loss. Conversely to above approaches, [Silaghi, 2004b] proposes to use costly encryption tools for keeping people's calendars completely private.

10.5 Summary

We have studied in this chapter the Meeting Scheduling problem, a naturally distributed benchmark that requires some privacy. We have defined a distributed version for this problem that can be modeled as *DisCSP*. In the context of the distributed version, we have discussed some issues regarding privacy loss of domains in *DisCSP*, identifying three types of information that agents may reveal to other agents at resolution time. Empirically, we have compared two synchronous algorithms against one asynchronous one to solve instances of this problem. Our experimental results show that the two synchronous approaches outperform the asynchronous one regarding computation effort, communication cost as well privacy loss. These results do not imply that synchronous algorithms should be considered the chosen algorithms for solving this problem. As discussed in previous chapters, synchronous and asynchronous algorithms have different functionalities. Regarding privacy, neither of the distributed algorithms that we have considered in this chapter is worse than the centralized approach, which needs to gather the whole problem into a single agent to solve it.

Chapter 11

Distributed Stable Matching Problems

The Stable Marriage and the Stable Roommates problems are well-known instances of Stable Matching Problems [Gusfield and Irving, 1989]. The term *matching* implies that the participants (elements of some underlying set or sets) are to be matched or assigned to each other in some way to meet *stability*. The stability criterion depends on fixed preferences expressed by the participants.

Stable Matching problems are combinatorial problems with real applications in computer science, economics, game theory and operations research [Gusfield and Irving, 1989]. Dating from 1962, the most well-known application for the Stable Marriage problem is the assignment of medical residents to hospitals [Gale and Shapley, 1962]. Very recently, it has been considered for the Stable Roommates problem an application about pairwise kidney exchange between patient-donor pairs [Roth et al., 2005].

The Stable Marriage and the Stable Roommates problems can be solved by centralized algorithms (i.e. algorithms which assume that all the information required to solve a given problem is centralized into a single processor/computer). However, this requires to make public people's preferences, which people would like to keep private. With this aim, in this chapter, we define distributed versions of the Stable Marriage and the Stable Roommates problems, and provide a constraint-based approach that solves these distributed problems keeping people's preferences privacy. We also consider other versions of these problems in which participants may declare some others participants to be unacceptable (i.e. preference lists may be incomplete) and/or may be indifferent between a subset of possible partners (i.e preference lists may contain ties).

In order to preserve preferences private, the original the Stable Marriage and the Stable Roommates problems have also been solved with a secure protocol which uses cryptographic tools [Silaghi, 2004a, Atkinson et al., 2006]. The approaches that this thesis proposes do not use these techniques. Typically, the overhead in communication and computation in secured protocols is very large.

11.1 What is the Stable Marriage Problem?

The Stable Marriage problem (*SM*) consists of two finite equal-sized sets of players, commonly called men and women. Each man m_i ($1 \leq i \leq n$, n is the number of men) ranks women in strict order forming his preference list. Similarly, each woman w_j ($1 \leq j \leq n$) ranks men in strict order forming her preference list. A matching M is just a complete one-to-one mapping between the two sexes. The goal is to find a *stable matching* M.

Definition 11.1.1. A matching M is *stable* if there are not a man m and a woman w such that m prefers w to his partner in M and w prefers m to her partner in M.

If this pair (m, w) exists, we say that M is unstable and the pair (m,w) is a *blocking pair* for M.

Figure 11.1 shows an instance of *SM* with 3 men (m_1, m_2, m_3) and 3 women (w_1, w_2, w_3). Each person has its own preference list. The preference lists are given in decreasing order, that is, the most-preferred partner for each person is the person who appears first in his/her list. For this instance, the matching M = $\{(m_1,w_1),(m_2, w_2), (m_3, w_3)\}$ is unstable because the pair (m_1, w_2) blocks M. This instance only has one stable matching: $M1$ = $\{(m_1,w_2), (m_2,w_1), (m_3,w_3)\}$.

Gale and Shapley proved that at least one stable matching exists for every *SM* instance. They obtained a $O(n^2)$ solving algorithm, called the Gale-Shapley algorithm [Gale and Shapley, 1962]. The algorithm consists of a sequence of proposals from persons of one sex to the persons of the other sex. However, *GS* needs that:

Observation 11.1.1. *Each time a person has to propose marriage, he/she must propose marriage to the most preferred person in his/her current preference list.*

The Extended Gale-Shapley algorithm (*EGS*) is a version of the original Gale-Shapley algorithm, that avoids some extra steps by deleting from the preference lists certain pairs that cannot belong to a stable matching [Gusfield and Irving, 1989]. A man-oriented version of *EGS* appears in Figure 11.2. In this version, men propose marriage to women.

During *EGS* execution, some people are deleted from preference lists. The reduced preference lists that result of applying man-oriented Gale-Shapley algorithm are called *man-oriented Gale-Shapley lists* or *MGS-lists*. On termination,

m_1 :	w_2	w_3	w_1		w_1 :	m_1	m_2	m_3
m_2 :	w_1	w_2	w_3		w_2 :	m_1	m_3	m_2
m_3 :	w_2	w_1	w_3		w_3 :	m_2	m_1	m_3

Figure 11.1: A *SM* instance with three men and three women.

```
assign each person to be free;
while some man m is free and m has a nonempty list loop
    w := first woman on m's list; {m proposes to w}
    if m is not on w's preference list then
        delete w from m's preference list;
        goto line 3
    end if
    if some man p is engaged to w then
        assign p to be free;
    end if
    assign m and w to be engaged to each other;
    for each each successor p of m on w's list loop
        delete p from w's list;
        delete w from p's list;
    end loop;
end loop;
```

Figure 11.2: The man-oriented Extended Gale-Shapley algorithm for *SM*.

each man is engaged to the first woman in his (reduced) list, and each woman to the last man in hers. These engaged pairs constitute a stable matching, and it is called *man-optimal* (or *woman-pessimal*) stable matching since there is not other stable matching where a man can achieve a better partner (according to his ranking). Similarly, exchanging the role of men and women in *EGS* (which means that women propose), we obtain the *woman-oriented Gale-Shapley lists* or *WGS-lists*. On termination, each woman is engaged to the first man in her (reduced) list, and each man to the last woman in his. These engaged pairs constitute a stable matching, and it is called *woman-optimal* (or *man-pessimal*) stable matching.

The intersection of *MGS-lists* and *WGS-lists* is known as the Gale-Shapley lists (*GS-lists*). These lists have important properties (see Theorem 1.2.5 in [Gusfield and Irving, 1989]):

- all the stable matchings are contained in the *GS-lists*,
- in the *man-optimal* (*woman-optimal*), each man is partnered by the first (last) woman on his *GS-list*, and each woman by the last (first) man on hers.

Figure 11.3 shows the *GS-lists* for the example given in Figure 11.1. The reduced lists of all persons have only one possible partner which means that only one solution exits. In that case, the *man-optimal* matching and *woman-optimal* matching are the same.

$m_1:$	w_2	$w_1:$	m_2
$m_2:$	w_1	$w_2:$	m_1
$m_3:$	w_3	$w_3:$	m_3

Figure 11.3: GS-Lists for *SM* of Figure 11.1.

11.2 A Constraint Formulation

According to [Gent et al., 2001], every instance of the Stable Marriage can be modeled and solved using a binary *CSP* encoding. This constraint encoding is described next. Each person in the Stable Marriage instance is represented by a variable in the *CSP*: variables x_1, x_2, ..., x_n represent the men (m_1, m_2, ..., m_n) and variables y_1, y_2, ..., y_n represent the women (w_1, w_2 ..., w_n). $PL(q)$ is the set of people that belong to q's preference list. Domains are $D(x_i) = \{k : w_k \in PL(m_i)\}, D(y_j) = \{l : m_l \in PL(w_j)\}, 1 \leq i, j \leq n$. When x_i takes value j, it means that man m_i marries woman w_j. Constraints are defined between men and women. Given any pair i, j ($1 \leq i, j \leq n$), the constraint C_{ij} is a $|D(x_i)| \times |D(y_j)|$ conflict matrix that represents all possible partial matchings involving x_i and y_j. For any pair k, l ($k \in D(x_i)$, $l \in D(y_j)$), the element $C_{ij}[k, l]$ represents the partial matching $(m_i, w_k)(m_l, w_j)$. This element could be one of the following values:

- $C_{ij}[k, l] =$ **A**llowed, when $k = j$ and $l = i$. This represents the partial matching (m_i, w_j). At most one element is A.
- $C_{ij}[k, l] =$ **I**llegal, when either $k = j$ and $l \neq i$ or $k \neq j$ and $l = i$. This assures matching monogamy.
- $C_{ij}[k, l] =$ **B**locked by the pair (m_i, w_j), when m_i prefers w_j to w_k and w_j prefers m_i to m_l.
- $C_{ij}[k, l] =$ **S**upport, all other entries that are not A, I or B.

Constraint matrixes in terms of **A**, **I**, **B**, **S** are transformed in terms of 1/0 (permitted/forbidden) pairs, using the natural conversion **A**, **S** $\rightarrow$ 1, **I**, **B** $\rightarrow$ 0. Figure 11.4 shows the constraint matrix for man m_3 and woman w_1 of the example given in Figure 11.1. In the constraint matrix, the domains of x_3 and y_1 are listed in decreasing ordering of the preferences. From that example, we can see that assignment $x_3 = w_1$ does not block any of other pairs which involve variable x_3 or variable y_1.

Let be J the CSP that result of applying the above constraint encoding to an *SM* problem I. [Gent et al., 2001] proves that J produces as solutions all the stable matchings that I admits and that the number of solutions of J and the number of stable matchings I is the same. Special emphasis is put on the fact that achieving arc consistency on J produces a reduced domains which are exactly the GS lists obtained by the EGS algorithm.

11.3 Generalizations of the Stable Marriage

The Stable Marriage requires that each person's preference list has to be complete (i.e. all member of the opposite sex must be included) and totally ordered. A natural generalization of *SM* occurs when persons may declare one or more members of the opposite sex to be unacceptable, so they do not appear in the corresponding preference lists. This relaxed version is called the Stable Marriage Problem with Incomplete Lists (*SMI*) [Gale and Sotomayor, 1985]. Although people prefer be married to be unmatched, it is possible to find stable matching in which some persons are not matched. Thus, the goal in *SMI* is to find a stable matching which could be incomplete. Similar to *SM*, the meaning of the therm stability is defined by the Definition 11.1.1. Every *SMI* instance admits at least one stable matching [Gale and Sotomayor, 1985]. However, all stable matchings for a given *SMI* instance will have the same lengths (the number of coupled men in a matching) since every one involves the same men and women.

An alternative natural generalization of *SM* arises when persons need not to rank all the members of the opposite sex in a strict order, so ties entries in the preference lists are possible. That is, a person might be indifferent between a subset of his/her possible partners. This relaxed version is called the Stable Marriage Problem with Ties (*SMT*). For this problem the goal is also to find a stable matching. Three possible notions of stability have been formulated ([Gusfield and Irving, 1989]):

1. *Weak stability.* A matching M is *weakly stable* if it does not admit a *weak blocking pair* (m, w) such that m and w are not partners in M and each of whom strictly prefers the other to his/her partner in M. Note that this formulation is exactly the same as the definition of the stability given in Definition 11.1.1 for *SM* and *SMI*.

2. *Strong stability.* A matching M is *strongly stable* if it does not admit a *strong blocking pair* (m, w) such that m and w are not partners in M and one strictly prefers the other to his/her partner in M and the other is at least indifferent between them.

3. Super stability. A matching M is *super stable* if it does not admit a *super blocking pair* (m, w) such that m and w are not partners in M and each of whom either strictly prefers the other to his/her partner in M or it is indifferent between them.

	m_1	m_2	m_3
w_2	S	S	I
w_1	I	I	A
w_3	S	S	I

	m_1	m_2	m_3
w_2	1	1	0
w_1	0	0	1
w_3	1	1	0

Figure 11.4: C_{31} for example of Figure 11.1. Left: in terms of A,I,B,S. Right: in terms of 0/1.

SM version	Size	All solutions		Algorithm	Complexity
		Length	Partners		
SM	n	same	same	*EGS* [Gusfield and Irving, 1989]	polynomial
SMI	$\leq n$	same	same	*EGS* [Gusfield and Irving, 1989]	polynomial
SMT-weak	n	same	same	break ties + *EGS* [Irving, 1994]	polynomial
SMT-strong	n	same	same	*STRONG* [Irving, 1994]	polynomial
SMT-super	n	same	same	*SUPER* [Irving, 1994]	polynomial
SMTI-weak	$\leq n$	diff	diff	break ties + *EGS* [Manlove, 1999]	polynomial
SMTI-strong	$\leq n$	same	same	*STRONG2* [Manlove, 1999]	polynomial
SMTI-super	$\leq n$	same	same	*SUPER2* [Manlove, 1999]	polynomial
SMTI-weak-max	$\leq n$	same	diff	break ties in all possible ways + *EGS* [Manlove, 1999]	NP-hard

Table 11.1: Solvability conditions, solving algorithm (centralized case) and complexity for the different *SM* problems. Any instance of *SM*, *SMI*, *SMT-weak* and *SMTI-weak* always has a solution, while this is not guaranteed for other problem instances. For *SM* and the three *SMT* versions, a solution has size n, while for the other versions the solution size is $\leq n$. Given any instance, except of *SMTI-weak*, all its solutions have the same length. Given any instance, except of *SMTI-weak* and *SMTI-weak-max*, all its solutions involve the same partners. All these problems are solved in polynomial time, except *SMTI-weak-max* that is NP-hard.

The version of *SM* which considers the above two extensions (preference lists may include ties and be incomplete) is named the Stable Marriage with Ties and Incomplete Lists (*SMTI*). The three stability types, defined previously, weak, strong and super, have been also studied for *SMTI*. Each instance of *SMTI* admits at least one *weakly stable* matching. In this situation, different solutions may exists, with different lengths. It is of interest to find the weakly stable matching with maximum length [Manlove, 1999]. Regarding strong and super stability, it may happen that an instance admits no, one or several solutions. In latter case, all the *strongly stable/super-stable* matchings have the same lengths because they involve the same men and women [Manlove et al., 2002].

The solvability conditions, complexity and solving algorithms of each *SMI*, *SMT* and *SMTI* in the centralized case are detailed in Table 11.1. Regarding solving algorithms, *SMI* is solved by the *EGS* algorithm. *SMT-weak* is solved by a direct extension of *EGS* since the definition of weak stability is the same that the definition of stability for *SM*. *SMT-strong* is solved by the *STRONG* algorithm [Irving, 1994]. *SMT-super* is solved by the *SUPER* algorithm [Irving, 1994]. *SMTI-weak* is solved by a direct extension of *EGS*. *SMTI-strong* is solved by the *STRONG2* algorithm [Manlove, 1999]. *SMTI-super* is solved by the *SUPER2* algorithm [Manlove, 1999]. *STRONG*, *STRONG2*, *SUPER* and *SUPER2* are basically extensions of *EGS* and have polynomial time complexity with respect to the number of men. However, in *SMTI-weak*, different solutions may exist with different lengths, so it is of interest to find the matching of maximum cardinality. This is *SMTI-weak-max*, an optimization problem that is NP-hard.[1]

[1]The decision problem "given an instance, there exists a matching of size $\geq K$?", is NP-

In addition to the aforementioned algorithms, the constraint formulation given in [Gent et al., 2001] can be used to solve the new versions of *SM*. For instances with incomplete lists, the man and woman variables remain the same but their domains are enlarged with the dummy value $n+1$, that is always the least preferred. Whether a person p is not an accepted partner for a person q, of opposite sex, all entries in column or row assigning p to q on C_{pq} are **I**. The rest of the constraint table is filled with **S**.

In addition we have extended the constraint formulation proposed by [Gent et al., 2001] to deal with versions of *SM* in which preference lists contain ties [Brito and Meseguer, 2006b, Brito and Meseguer, 2006c]. For this problems, there are different definitions of stability: weak, strong and super. The type of stability affects the usage of **B**locked pair in the constraint C_{ij}. The definition given in Section is valid for weak stability. Considering strong/super stability, we replace **B** definition in [Gent et al., 2001] by,

- $C_{ij}[k,l] =$ **B**locked by the pair (m_i, w_j), when (a) m_i prefers w_j to w_k and w_j prefers to or is indifferent between m_i and m_l, or (b) m_i prefers to or is indifferent between w_j and w_k and w_j prefers m_i to m_l (strong stability).
- $C_{ij}[k,l] =$ **B**locked by the pair (m_i, w_j), when m_i prefers to or is indifferent between w_j and w_k, and w_j prefers to or is indifferent between m_i and m_l (super stability).

11.4 The Distributed Stable Marriage Problem

The *SM* problem, by its own nature, appears to be naturally distributed. Each person may desire to act as an independent agent. For obvious reasons, each person would like to keep his/her preference lists private. However, in the centralized case each person has to follow a rigid role, making public his/her preferences to achieve a solution. So this problem is very suitable to be treated by distributed techniques, trying to provide more autonomy to each person, and to keep preference lists private. In [Brito and Meseguer, 2005a, Brito and Meseguer, 2005b], we define the distributed problem corresponding to *SM*:

Definition 11.4.1. The Distributed Stable Marriage (*DisSM*) consists of n men, n women and a set of r agents. Each person has his/her own preference list, in which he/she ranks members of the opposite sex in strick order. The n men and n women are distributed among the agents: an agent owns some men and women and every person is owned by a single agent. An agent can access and modify all the information of its owned people, but it cannot access the information (i. e. preference lists) of people owned by other agents. As in the classical case, a solution is a stable matching (a matching between the men and women such that no blocking pair exists).

complete [Manlove, 1999]. Finding the matching of maximum cardinality is NP-hard.

For simplify description, it is assumed that each agent owns exactly one person (so, $r = 2 \times n$). Motivated by privacy requirements, we also introduce the following distributed problems [Brito and Meseguer, 2006b, Brito and Meseguer, 2006c]: the Distributed Stable Marriage with Incomplete lists problem (*DisSMI*), the Distributed Stable Marriage with Ties (*DisSMT*) and the Distributed Stable Marriage with Ties and Incomplete Lists (*DisSMTI*). Formally, these problems can be defined by Definition 11.4.1. Only, they differ from each other in the type of stability that is going to be considered (weak, strong or super) and the structure of preference lists (that is, if preference lists are complete or incomplete and if preference lists contain ties or not).

11.4.1 From Centralized to Distributed Algorithms

Regarding solving algorithms for the different versions of the distributed Stable Marriage problem, a first question is to see if the centralized algorithms can be extended to the distributed case keeping privacy. The Distributed Extended Gale/Shapley (*DisEGS*) algorithm is a distributed version of *EGS* that maintains privacy. It was used to solve the *DisSM* and *DisSMI* [Brito and Meseguer, 2005a, Brito and Meseguer, 2005b].

As in the classical case, the *DisEGS* algorithm consists of a sequence of proposals from agents which represent people of one sex to agents which represent people of the opposite sex. Assuming a man-oriented version of this algorithm, men agents propose marriage to women agents while women agents accept or reject the received proposals according to their preferences. During the execution of the algorithm, preference lists are reduced. Considering that preference lists may be incomplete, that is *SMI*, the algorithm finishes when each man agent is engaged or unmatched (because his preference lists is empty). The obtained engagement relations that results of the execution of the algorithm constitute a stable matching for the problem [Gale and Shapley, 1962].

In *DisEGS*, each agent can access to his/her own information only. For this reason there are two different procedures, one for men and one for women. In addition, actions performed by *EGS* on persons different from the current one are replaced by message sending. Thus, when m assigns woman w is replaced by `sendMsg`(propose,m,w); when w deletes herself from the list of p is replaced by `sendMsg`(delete,w,p). Since procedures exchange messages, operations of message reception are included accordingly. The code and other details of *DisEGS* appear in Appendix A.

DisEGS algorithm guarantees privacy in preferences and in the final assignment: each person knows the assigned person, and no person knows more than that. In this sense, it is a kind of ideal algorithm because it assures privacy in values and constraints. However, some information may be deduced by agents if after the execution of the algorithm the found stable matching is made public.

DisSMT and *DisSMTI* jointly with the three different types of stability produce six different decision problems plus one optimization problem: *DisSMT-weak*, *DisSMT-strong*, *DisSMT-super*, *DisSMTI-weak*, *DisSMTI-strong*, *DisSMTI-super* and *DisSMTI-weak-max*. Their resolutions by

DisSM problem	Centralized Algorithm	Extension to the distributed case, keeping privacy
DisSM	*EGS* [Irving, 1994]	*DisEGS* [Brito and Meseguer, 2005b]
DisSMI	*EGS* [Irving, 1994]	*DisEGS* [Brito and Meseguer, 2005b]
DisSMT-weak	break ties + *EGS* [Irving, 1994]	break ties arbitrary + *DisEGS* [Brito and Meseguer, 2005b]
DisSMT-strong	*STRONG* [Irving, 1994]	No extension (Appendix A)
DisSMT-super	*SUPER* [Irving, 1994]	Extension (see Appendix A)
DisSMTI-weak	break ties + *EGS* [Manlove, 1999]	break ties arbitrary + *DisEGS* [Brito and Meseguer, 2005b]
DisSMTI-strong	*STRONG2* [Manlove, 1999]	No extension (Appendix A)
DisSMTI-super	*SUPER2* [Manlove, 1999]	No extension (Appendix A)
DisSMTI-weak-max	break ties in all possible ways + *EGS* [Manlove, 1999]	Discussion (Appendix A)

Table 11.2: *DisSM*, *DisSMI*, *DisSMT-weak*, *DisSMT-super* and *DisSMTI-weak* can be solved by direct extensions of their corresponding centralized algorithms that keep privacy. However, for *DisSMT-strong*, *DisSMTI-strong* and *DisSMTI-super*, their centralized algorithms cannot be extended to the distributed case keeping preference lists private.

extending the specialized centralized algorithms (given in Section 11.3) to the distributed case have been studied in [Brito and Meseguer, 2006b, Brito and Meseguer, 2006c]). Here, Table 11.2 summaries of which of these algorithms can be extended to distributed while keeping preference lists private [Brito and Meseguer, 2006b, Brito and Meseguer, 2006c]). Details of this analysis also appear in Appendix A. From it, we conclude that only three out of the six decision problems can be solved by extending the centralized algorithms to the distributed case while keeping preferences private. About *DisSMTI-weak-max*, it requires to make public some extra information, although preference lists could remain private.

11.4.2 Distributed Constraint Formulation

For adapting the constraint formulation of Section 11.3 to the distributed case, we observe following: the constraint matrix $C_{i,j}$, that say which partial matching including man i and woman j is valid or forbidden, requires that i and j reveals their preferences. This requirement is incompatible with the formulation of distributed Stable Marriage problems in which people desire to keep their preferences private. Therefore, a method for solving distributed Stable Marriage problems needs that constraints are kept private during the search of a stable matching [Brito and Meseguer, 2005b, Brito and Meseguer, 2006c].

In Chapter 8, we presented $DisFC_1$ and $DisFC_2$, two approaches to privacy that differentiates between assignments and constraints. Briefly, privacy on assigments implies that agents are not aware of other agent values during the solving process and in the final solution. Regarding privacy on constraints, two models were considered in Chapter 8: *TKC* and *PKC* models. *TKC* assumes that when two agents i and j share a constraint C_{ij}, both know the constraint scope and at least one of them knows completely the relational part of the con-

$$C_{i(j)} = \begin{array}{c|ccccccc} & m_1 & \dots & & m_i & & \dots & m_n \\ w_{i_1} & 1 & \dots & 1 & 0 & 1 & \dots & 1 \\ & & \dots & & & & & \\ & 1 & \dots & 1 & 0 & 1 & \dots & 1 \\ w_j & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ w_{j'} & \# & \dots & \# & 0 & \# & \dots & \# \\ & ? & \dots & ? & 0 & ? & \dots & ? \\ & & \dots & & & & & \\ w_{i_n} & ? & \dots & ? & 0 & ? & \dots & ? \end{array}$$

Figure 11.5: Form of the partial matrix $C_{i(j)}$.

straint. However, as we said above, in the distributed Stable Marriage problems if an agent knows C_{ij}, it can deduce the preferences of the other agent because C_{ij} is building using the preference lists of i and j. Therefore, *TKC* is an unsuitable constraint model for solving distributed Stable Marriage problems.

Conversely, the *PKC* model assumes that when two agents i and j share a constraint C_{ij}, none of them knows completely the constraint. Each agent knows the part of the constraint that it is able to build, based on its own information. We say that agent i knows $C_{i(j)}$, and j knows $C_{j(i)}$. Note that, unlike the *TKC* model, the *PKC* model applied to distributed Stable Marriage problems does not need that any agent reveals his/her preference list.

In [Brito and Meseguer, 2005b], we describe how a *DisSMTI* instance can be formulated in the *PKC* model.[2] The first point is how the agent owning variable x_i can construct the partially known constraint matrix $C_{i(j)}$. This matrix is built from x_i, knowing its preference list but ignoring the preference list of y_j. Figure 11.5 illustrates the form of partial matrix $C_{i(j)}$, assuming lexicographical ordering in rows and columns.

Each element in the matrix $C_{i(j)}$ represents a partial matching in which the man m_i and the woman w_j are involved. An element with value 1 means that the partial matching it represents is stable. In the above example, all elements in rows above w_j are 1 (except m_i^{th} column that are 0). An element with value ? (undecided) means i cannot decide if the partial matching this element represents is stable or not. In the above example, all elements in rows below w_j may be 1 or 0, depending on the ordering of columns (except m_i^{th} column that are 0). Since x_i does not know the preference list of y_j, columns are ordered lexicographically, and the elements below w_j row are ? (undecided). If there is a tie between w_j and $w_{j'}$ (other ties may exist, only those with w_j are considered in $C_{i(j)}$) the $w_{j'}$ row has a tie with w_j row. Elements in $w_{j'}$ row are # (tie). Analogously, y_j build $C_{(i)j}$ using his/her own preference list but ignoring the preference list of x_i.

One interesting property of these constraints is that in $C_{i(j)}$ (conversely $C_{(i)j}$)

[2]This analysis is referred to *DisSMTI*, the most general Stable Marriage problem in which preference lists may be incomplete and may include ties. An instance of *DisSMTI* is an *SM* instance if preference lists neither are incomplete nor contain ties. An instance of *DisSMTI* is an *SMI* instance if some preference lists are incomplete but none of them contain ties. An instance of *DisSMTI* is an *SMT* instance if preference lists are complete but some of them contain ties.

all columns (rows) are equal, except the column (row) corresponding to x_i (y_j) [Brito and Meseguer, 2005b, Brito and Meseguer, 2006c].

Proposition 11.4.1. *In $C_{i(j)}$ (conversely $C_{(i)j}$) all columns (rows) are equal, except the column (row) corresponding to x_i (y_j).*

Proof. We have to prove that $C_{i(j)}[k,l] = C_{i(j)}[k,l'], l \neq i, l' \neq i, l \neq l'$. Effectively, if x_i prefers woman k to woman j, both values $C_{i(j)}[k,l]$ and $C_{i(j)}[k,l']$ are 1, corresponding to **S** (supported, see Section 11.3). If x_i prefers woman j to woman k, both values $C_{i(j)}[k,l]$ and $C_{i(j)}[k,l']$ are ? (undecided). Their exact value could be 1 or 0, depending on the preferences of y_j, information which is not available when constructing $C_{i(j)}$. Therefore, both are undecided in $C_{i(j)}$. If x_i is indifferent between woman j and woman k, both values $C_{i(j)}[k,l]$ and $C_{i(j)}[k,l']$ are # (ties). Analogous arguments hold for $C_{(i)j}$ rows. □

In addition to the above property, we observe that $C_{ij} = C_{i(j)} \diamond C_{(i)j}$, where $\diamond$ is an operator defined between the two elements that represent to a same partial matching in the two partial constraint matrixes. In general, this operator is different in for each distributed Stable Marriage problem and clearly depends on the type of stability that one want to guarantee. Regarding instances in which preference lists do not contain ties (i.e. *DisSM* and *DisSMI*), the $\diamond$ operator is defined by the next rules [Brito and Meseguer, 2005b],

$$\begin{array}{lll} 0 \diamond 0 = 0 & 1 \diamond 1 = 1 & 1 \diamond 0 = error \\ ? \diamond 1 = 1 & ? \diamond 0 = 0 & ? \diamond ? = 0 \end{array}$$

Rules including ? are quite intuitive. If a position in the constraint is decided (permitted/forbidden) in one constraint and undecided in the other, the result is the decided value. The last rule $? \diamond ? = 0$ is proved next.

Proposition 11.4.2. *If element $[k,l]$ is undecided in $C_{i(j)}[k,l] =?$ and $C_{(i)j}[k,l] =?$, then the element $[k,l]$ in $C_{ij}[k,l]$ is 0.*

Proof. From the construction of partially known constraint matrixes, we know that all undecided elements in $C_{i(j)}$ are related to values which are less preferred than j. If $C_{i(j)}[k,l] =?$, we infer that x_i prefers j to k. Conversely, if $C_{(i)j}[k,l] =?$, we deduce that y_j prefers i to l. Therefore, since x_i prefers j to k and y_j prefers i to l, the pair (i,j) is blocking pair to the pair (k,l) so $C_{ij}[k,l] = 0$. □

Regarding instances in which some preference lists contain ties (i.e. *DisSMT* and *DisSMTI*), the $\diamond$ operation depends on the stability type [Brito and Meseguer, 2006c]:

- *Weak Stability.* From the *weak blocking pair* definition results that no matched pair (m, w) will be blocked for any other pair (m', w'), such that either m is indifferent between w and w' or w is indifferent between m and m'. So, #'s are replaced by 1's in matrixes. Rules for the $\diamond$ operation are similar to those rules for *DisSM* and *DisSMI*, given above.

For *DisSMTI-weak*, not all stable matchings have the same length. One may desire to find a matching of maximum cardinality. With this aim, we consider the question 'Is there a *weakly stable* matching of size k?', where k starts with value n. If a weakly stable matching exits, it will be of maximum cardinality. Otherwise, the value k is decreased by one, and the problem is reconsidered. Modeling this idea with constraints, we add n variables, $u_1, u_2, \ldots, u_n$, plus an extra variable z, with the domains: $D(u_i) = \{0, 1\}, 1 \leq i \leq n$, $D(z) = \{k\}$. New constraints are: if $x_i < n + 1$ then $u_i = 1$ else $u_i = 0, 1 \leq i \leq n$ and $z = \sum_{i=1}^{n} u_i$. The agent that owns x_i also owns u_i. An extra agent owns z, located in the last position in the agent ordering.

- *Strong Stability.* The $\diamond$ operation is extended to deal with # (ties) as follows:
  - $\# \diamond \# = 1$. If element $[k, l]$ in both partial matrix $C_{i(j)}$ and $C_{(i)j}$ is #, then i is indifferent between j and k and j is indifferent between i and l. Therefore, according to the definition of strong blocking pair, the pair (i, j) cannot be a strongly blocking pair for (k, l). So $C_{ij}[k, l] = 1$ (permitted).
  - $1 \diamond \# = 1$. If element $[k, l]$ in matrix $C_{i(j)}$ is 1 and in $C_{(i)j}$ is #, then i prefers k to j and j is indifferent between i and l. Therefore, according to the definition of strong blocking, the pair (i, j) cannot be a strongly blocking pair for (k, l). So $C_{ij}[k, l] = 1$ (permitted).
  - $\#\diamond? = 0$. If element $[k, l]$ in matrix $C_{i(j)}$ is # and in $C_{(i)j}$ is undecided, then i is indifferent between j and k and j prefers i to l. Therefore, according to the definition of strong blocking, the pair (i, j) is a strongly blocking pair for (k, l). So $C_{ij}[k, l] = 0$ (forbidden).
  - $0 \diamond \# = 0$. If element $[k, l]$ in matrix $C_{i(j)}$ is 0 and in $C_{(i)j}$ is #, then i prefers j to k and j is indifferent between i and l. Therefore, according to the definition of strong blocking pair, the pair (i, j) is a strongly blocking pair for (k, l). So $C_{ij}[k, l] = 0$ (forbidden).
- *Super Stability.* The $\diamond$ operation has a single change from strong stability:
  - $\# \diamond \# = 0$. If element $[k, l]$ in both matrixes $C_{i(j)}$ and $C_{(i)j}$ is #, then i is indifferent between j and k and j is indifferent between i and l. Therefore, according to the definition of super blocking pair, the pair (i, j) is a super blocking pair for (k, l). So $C_{ij}[k, l] = 0$ (forbidden).

With the property expressed by Proposition 11.4.1 and the existent relationship between partial constraint matrixes expressed by the $\diamond$ operator, we have specialized the *DisFC*$_2$ algorithm to solve *DisSMTI* instances [Brito and Meseguer, 2006c]. We refer to this specialized algorithm as *DisFC*$_2$ for finding stable matchings (in short *DisFC-SM*).

In **phase I** of *DisFC*$_2$, a solution is computed with respect to $C_{i(j)}$ constraints; in **phase II**, this solution is verified with respect to $C_{(i)j}$ constraints.

If so, it is a true solution and the algorithm stops. Otherwise, it is not a true solution, and $DisFC_2$ resumes **phase I**. However, the special form of constraint matrix (Proposition 11.4.1) allows for an effective disambiguation by the low priority agent using the $\diamond$ operation, without accessing the information of the high priority agent, that is, the **phase II** of $DisFC_2$ is not necessary. Thus, *DisFC-SM* consists of only one phase.

Similar to $DisFC_2$, *DisFC-SM* requires a total order among agents. Here, it is assumed that men agents appear before women agents in the ordering. When the algorithm starts, each agent instantiates its variable and sends to lower priority agents the domains compatible with its assignment. A woman agent receives messages from every man agent, and assigns a value permitted by these n received domains. If no value is available, the woman agent performs backtracking. The process iterates until finding a solution or detecting an empty nogood, which means that the problem does not admit a stable matching for the type of stability that is being considered.

In the previous argument, something must be scrutinized in more detail. After assignment, what kind of compatible domain can a man agent send? If agent i assigns value k to x_i, it sends to j the row of $C_{i(j)}$ corresponding to value k. This row may contain 1's (permitted values for y_j), 0's (forbidden values for y_j), ? (undecided values for y_j) and # (undecided values for y_j but involved in ties with y_k). If the compatible domain has 1 or 0 values only, there is no problem and the y_j domain can be easily computed. But what happens when the domain contains entries with value ? or #? In this case, agent j can disambiguate the domain as follows. When agent j receives a compatible domain with ? (undecided) values, it performs the $\diamond$ operation with a row of $C_{(i)j}$ different from i. Since all rows in $C_{(i)j}$ are equal, except row corresponding to value i (see Proposition 11.4.1), all will give the same result. Performing the $\diamond$ operation, which depends of the type of stability that is considered, j will compute the corresponding row in the complete constraint C_{ij}, although j does not know to which value this row corresponds (in other words, j does not know the value assigned to x_i). After performing the $\diamond$ operator, the resulting domain will contain neither ? or # values, and the receiving agent can operate normally with it.

Some information may be leaked during the solving process. But this information does not allow one to deduce the relative preference order of any two persons in the list of another one. The only information that person j knows from person i (of the opposite sex) is the relative position between j and i's current partner, in the i's preference list. It is enough to revise the rows that j may receive from i:

- 1 ... 1 0 1 ... 1: j is more preferred than i's current partner
- 0 ... 0 1 0 ... 0: j is i's current partner
- #... #0 #... #: j is as preferred as i's current partner
- ? ... ? 0 ? ... ? : j is less preferred than i's current partner

Since j does not know i's current partner (unless it is j), privacy of preference lists is guaranteed. Unlike *DisEGS*, men agents in *DisFC-SM* may propose marriage to any woman agent, not necessarily to the most preferred one.

11.5 The Stable Roommates Problem

The Stable Roommates Problem (*SR*) [Gusfield and Irving, 1989] is a generalization of the Stable Marriage problem. *SR* consists of $2n$ participants, each ranks *all* other participants in strict order according to his/her preferences. A matching is a set of n disjointed participant pairs. Like in SM, a matching is *stable* if it contains no *blocking pair*. A pair (p_i, p_j) is considered a blocking pair in M if p_i prefers p_j to his/her current partner in M and p_j prefers p_i to his/her current partner in M. If this pair exists, this pair blocks M and, therefore, M is *unstable*. Unlike SM, there are instances of *SR* that admit no stable matching. Therefore, the goal is to determine whether a given *SR* instance is solvable, and if so, find a stable matching. Like SM, several *SR* versions exist [Gusfield and Irving, 1989]:

1. Stable Roommates with Incomplete Lists (*SRI*). Some participants may consider some of the other participants unacceptable, so they are not included in their preference lists. Preference lists may have less than $2n-1$ elements. A solution is a stable matching.

2. Stable Roommates with Ties (*SRT*). Some participants may consider some of the other participants equally acceptable. In such case, there is a tie among them. Preference lists have $2n-1$ elements, not strictly ranked. A solution is a stable matching. There exit three types of stability, weak, strong and super.

3. Stable Roommates with Ties and Incomplete Lists (*SRTI*). Some participants may consider some the other participants equally acceptable, while others are considered unacceptable. Preference lists may have less than $2n-1$ elements, not strictly ranked. A solution is a stable matching. As for *SRT*, there are three stability types, weak, strong and super (see Definition in Section 11.3).

Every *SR* versions may contain or not a stable matching. The solvability conditions, complexity and solving algorithms (centralized case) of each *SR* version appear in Table 11.3. Considering *SRTI-weak*, different solutions may exist with different lengths, so it is of interest to find the matching of maximum cardinality. This is *SRTI-weak-max*, an optimization problem that is NP-hard.

11.5.1 Algorithms for a Distributed Setting

The *SR* problem and its generalizations appear to be naturally distributed. In a centralized setting, each participant has to make his/her preference list public

SR version	Size	All solutions		Algorithm	Complexity
		Length	Partners		
SR	n	same	same	*Stable Roommates*	polynomial
SRI	$\leq n$	same	same	*Stable Roommates*	polynomial
SRT-weak	n	same	same	break ties in all possible ways + *Stable Roommates*, until finding a solution [Irving and Manlove, 2002]	NP-complete
SRT-strong	n	same	same	? [Irving and Manlove, 2002]	?
SRT-super	n	same	same	*SRT-super*	polynomial
SRTI-weak	$\leq n$	diff	diff	break ties in all possible ways + *Stable Roommates*, until finding a solution [Irving and Manlove, 2002]	NP-complete
SRTI-strong	$\leq n$	same	same	? [Irving and Manlove, 2002]	?
SRTI-super	$\leq n$	same	same	*SRTI-super* [Irving and Manlove, 2002]	polynomial
SRTI-weak-max	$\leq n$	same	diff	break ties in all possible ways + *Stable Roommates*	NP-hard

Table 11.3: Solvability conditions, solving algorithm [Irving and Manlove, 2002] (centralized case) and complexity for the different *SR* problems. Any instance of any *SR* version may be unsolvable. For *SR* and the three *SRT* versions, a solution has size n, while for others the solution size is $\leq n$. Given any instance, except of *SMTI-weak*, all its solutions have the same length. Given any instance, except of *SRTI-weak* and *SRTI-weak-max*, all its solutions involve the same partners. *SR*, *SRI*, *SRT-super* and *SRTI-super* are solved in polynomial time, *SRT-weak* and *SRTI-weak* are NP-complete, and *SRTI-weak-max* is NP-hard. Regarding *SRT-strong* and *SRTI-strong*, no polynomial algorithm is known and their complexity remains an open problem.

to achieve a solution. However, participants may desire to keep their preferences private during the search for a stable matching. This problem, like SM, is very suitable to be treated by distributed techniques.

In that sense, we present the Distributed Stable Roommates problem ($DisSR$) [Brito and Meseguer, 2005b], which consists of a number $2n$ of persons $\{p_1, p_2, ..., p_{2n}\}$ plus a set of r agents. For simplicity, we assume that each person is represented by an agent, so $r = 2n$. Likewise, we define the following problems [Brito and Meseguer, 2005b, Brito and Meseguer, 2006c]: the Distributed Stable Roommates problem with Incomplete Lists ($DisRMI$) , the Distributed Stable Roommates problem with Ties ($DisRMT$) and the Distributed Stable Roommates problem with Ties and Incomplete Lists ($DisRMTI$).

Like for *SM*, we investigate if centralized solving algorithms can be extended to the distributed case keeping privacy [Brito and Meseguer, 2005b, Brito and Meseguer, 2006c]. Their resolution is summarized in Table 11.4. From it, we conclude that only two decision problems can be solved by extending the centralized algorithms to the distributed case while keeping preferences private. Details of the extension of centralized algorithms to the distributed case appear in Appendix A.

Considering constraints, the formulation introduced in Section 11.3 is fully applicable to encode instances of every version of *SR*. The distributed constraint version, presented in Subsection 11.4.2 is also applicable to instances of every

DisSR problem	Centralized Algorithm	Extension to the distributed case, keeping privacy
DisSR	*Stable Roommates* [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRI	*Stable Roommates* [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRT-weak	break ties in all possible ways + *Stable Roommates*, until finding a solution [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRT-strong	? [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRT-super	*SRT-super* [Irving and Manlove, 2002]	Extension (Appendix A)
DisSRTI-weak	break ties in all possible ways + *Stable Roommates*, until finding a solution [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRTI-strong	? [Irving and Manlove, 2002]	No extension (Appendix A)
DisSRTI-super	*SRTI-super* [Irving and Manlove, 2002]	Extension (Appendix A)
DisSRTI-weak-max	break ties in all possible ways + *Stable Roommates* [Irving and Manlove, 2002]	No extension (Appendix A)

Table 11.4: *DisSRT-super* and *DisSRTI-super* can be solved by direct extensions of their corresponding centralized algorithms that keep privacy. For all the other versions, their centralized algorithms cannot be extended to the distributed case keeping privacy.

version of *DisSR*, with the following remark. In each setting, there are $\frac{2n*(2n-1)}{2}$ binary constraints; one for each different pair of persons. Constraint matrixes will have the form given in Figure 11.4 for *SR* versions and the form given in Figure 11.5 for *DisSR* versions. In every case, the definition of blocking pair depends on the type of stability that is being considered. Furthermore, we have to take into account that no person can match with himself/herself. To avoid that, we add unary constraints: $x_i \neq p_i$, for all i, $1 \leq i \leq n$.

Fortunately, none of the above changes affect the fulfilment of Proposition 11.4.1. That is, in every constraint table, except for one, all columns or rows are equal. This implies that one can use the *DisFC-SM* algorithm with *PKC* model, as described in Subsection 11.4.2, to solve instances of any of the versions of *DisSR* that appear in Table 11.4 [Brito and Meseguer, 2006c]. We remember that only one phase of the algorithm is required.

Regarding *DisSRTI-weak-max*, we consider the question 'Is there a *weakly stable* matching of size $k/2$?', where k starts with value $2n$. If a weakly stable matching exits, it will be of maximum cardinality. Otherwise, the value k is decreased by two, and the problem is reconsidered. A constraint formulation requires the addition of $2n$ variables $u_1, u_2, \ldots, u_{2n}$, plus an extra variable z, with the domains: $D(u_i) = \{0, 1\}, 1 \leq i \leq 2n$, $D(z) = \{k\}$. New constraints are: if $x_i < 2n + 1$ then $u_i = 1$ else $u_i = 0, 1 \leq i \leq 2n$ and $z = \sum_{i=1}^{2n} u_i$. The agent that owns x_i also owns u_i. An extra agent owns z. The discussion about privacy when solving *DisSM* versions at the end of Section 11.4 is fully applicable here.

11.6 Experimental Results

We have implemented the distributed versions of specialized algorithms that keep privacy: *DisEGS* (for *DisSM*, *DisSMI*, *DisSMT-weak* and *DisSMTI-weak*), *DisSUPER* (for *DisSMT-super*) and *DisSUPER-SR* (for *DisSRT-super* and *DisSRTI-super*). We report results of these algorithms plus *DisFC-SM* with the *PKC* model on random instances.

The generation of a random instances has been extended from the generation of random problems described in [Gent and Prosser, 2002]. Considering that preference lists may be incomplete and/or contain ties, a random class is defined by $\langle n, p_1, p_2 \rangle$, where for *DisSMTI* n is the number of men and for *DisSRTI* $2n$ is the number of participants; p_1 is the probability of incompleteness and p_2 the probability of ties. Random classes with $p_1 = p_2 = 0.0$ are instances of the original problems, that is, instances in which preference lists are complete and do not contain ties (*DisSM* or *DisSR* instances). Random classes with $p_1 \neq 0.0$ and $p_2 = 0.0$ are instances in which preference lists are incomplete but do not contain ties (*DisSMI* or *DisSRI* instances). Random classes with $p_1 = 0.0$ and $p_2 \neq 0.0$ are instances in which preference lists are complete and contain ties (*DisSMT* or *DisSRT* instances). Random classes with $p_1 \neq 0.0$ and $p_2 \neq 0.0$ are instances with ties and incomplete lists (*DisSMTI* or *DisSRTI* instances). In *DisSMTI* experiments $n = 5$, while for *DisSRTI* $2n = 12$; p_1 and p_2 take values 0.0, 0.4 and 0.8. For each class, results are averaged on 200 instances.

In *DisSMTI* instances, each agent represents a man or a woman, and execute *DisEGC* man or woman version [Brito and Meseguer, 2005b] (equivalent versions exist for *DisSUPER*). Alternatively, they run *DisFC-SR*. In instances of *DisSRTI* versions, each agent represents a participant. In distributed versions of centralized algorithms like *DisEGS*, *SUPER* and *DisSUPER-SR*, each agent only knows its preference list. In *DisFC-SM*, each agent only knows its preference lists and its partial constraint matrixes. In all algorithms, agent exchange different kind of messages to find a stable matching according to different kind of stability (weak, strong and super). *DisFC-SM* also requires a total ordering among agents. For *DisSMTI*, it is assumed that men agents have higher priority than women agents. For *DisSRTI*, person agents are lexicographically ordered.

Table 11.5 presents the *nccc* needed by algorithms for solving *DisSMTI* instances for each stability type, while Table 11.6 resumes the *msg* needed. Both tables are correlated. For weak stability, two problem types have been considered: finding a weak stable matching, and the one with maximum cardinality. In 6th column, the length of the largest matching appears between parenthesis. Considering weak stability, *DisEGS* is much faster than *DisFC-SM* for finding a weak stable matching: it is a specialized algorithm vs. the generic constraint formulation.

For *DisSMTI-weak-max* when it is possible to use *DisEGS* it is much faster. When preference list are incomplete ($p_1 = 0.4$, $p_1 = 0.8$) only *DisFC-SM* is able to find the optimal solution keeping preference lists private. Regarding their length, increasing p_2 causes larger matchings. Increasing p_1 instances become easier. Considering strong stability, *DisFC-SM* is the only algorithm that solves

		Weak Stability				Strong Stability	Super-Stability	
		Any		Max Card				
$p1$	$p2$	DisEGS	DisFC-SM	DisEGS	DisFC-SM	DisFC-SM	DisSUPER	DisFC-SM
0.0	0.0	133	994,237	133	921,137 (10)	994,237	133	994,237
0.0	0.4	133	1,115,142	133	1,025,018 (10)	1,087,487	195	758,883
0.0	0.8	133	1,637,439	133	88,897 (10)	1,249,360	320	33,874
0.4	0.0	81	169,442	–	270,446 (9.57)	169,442	–	169,442
0.4	0.4	81	168,956	–	216,534 (9.92)	166,463	–	90,122
0.4	0.8	81	16,958	–	68,429 (10)	189,495	–	10,623
0.8	0.0	28	2,758	–	15,141 (8.23)	2,758	–	2,758
0.8	0.4	28	1,843	–	19,599 (8.79)	3,548	–	2,436
0.8	0.8	28	920	–	30,674 (9.34)	3,849	–	1,647

Table 11.5: Computation cost of solving *DisSMTI* instances for the three stabilities. Entries "-" mean that the corresponding centralized algorithm cannot be extended to the distributed case while keeping preference lists private.

		Weak Stability				Strong Stability	Super-Stability	
		Any		Max Card				
$p1$	$p2$	DisEGS	DisFC-SM	DisEGS	DisFC-SM	DisFC-SM	DisSUPER	DisFC-SM
0.0	0.0	90	52,826	90	54,014 (10)	52,826	91	52,826
0.0	0.4	90	50,024	90	51,852 (10)	65,389	120	47,765
0.0	0.8	90	39,764	90	3,464 (10)	65,947	151	3,258
0.4	0.0	61	11,363	–	22,007 (9.57)	11,363	–	11,363
0.4	0.4	61	9,600	–	15,244 (9.92)	12,432	–	7,472
0.4	0.8	61	751	–	3,547 (10)	12,585	–	1,395
0.8	0.0	29	310	–	2,575 (8.23)	310	–	310
0.8	0.4	29	215	–	3,111 (8.79)	435	–	354
0.8	0.8	29	131	–	4,414 (9.34)	451	–	282

Table 11.6: Communication cost of solving *DisSMTI* instances for the three stabilities. Entries "-" mean that the corresponding centralized algorithm cannot be extended to the distributed case while keeping preference lists private.

		Weak Stability			Super-Stability	
		Any	Max Card	Strong Stability		
$p1$	$p2$	DisFC-SM	DisFC-SM	DisFC-SM	DisSUPER-SR	DisFC-SM
0.0	0.0	43,119 (176)	89,308 (10.56)	43,119 (176)	168 (176)	43,119 (176)
0.0	0.4	35,681 (199)	38,826 (11.94)	47,076 (46)	152 (22)	41,963 (22)
0.0	0.8	10,824 (200)	10,576 (12)	38,928 (69)	130 (1)	11,377 (1)
0.4	0.0	3,400 (165)	11,395 (8.67)	3,400 (165)	66 (165)	3,400 (165)
0.4	0.4	2,043 (198)	7,326 (11.15)	3,840 (29)	85 (12)	3,273 (12)
0.4	0.8	922 (200)	4,492 (11.87)	3,419 (61)	104 (0)	1,791 (0)
0.8	0.0	202 (197)	635 (9.67)	202 (197)	29 (197)	202 (197)
0.8	0.4	176 (198)	745 (9.97)	223 (75)	36 (57)	229 (57)
0.8	0.8	136 (200)	906 (10.30)	234 (39)	42 (11)	242 (11)

Table 11.7: Computation cost of solving *DisSRTI* instances for the three stabilities.

the problem keeping preference lists private. Larger p_2 values (more ties in preference lists), increase the cost of finding a strong stable matching. Considering super stability, *DisSUPER* can find a super stable matching when preference lists are complete ($p_1 = 0.0$) keeping preference lists private. In that setting, it is much faster than *DisFC-SM*. However, when lists are incomplete the dis-

		Weak Stability				
		Any	Max Card	Strong Stability	Super-Stability	
$p1$	$p2$	DisFC-SM	DisFC-SM	DisFC-SM	DisSUPER-SR	DisFC-SM
0.0	0.0	3,828 (176)	9,364 (10.56)	3,828 (176)	27 (176)	3,828 (176)
0.0	0.4	3,088 (199)	3,962 (11.94)	4,044 (46)	18 (22)	3,435 (22)
0.0	0.8	878 (200)	1,052 (12)	3,297 (69)	12 (0)	863 (1)
0.4	0.0	480 (165)	2,507 (8.67)	480 (165)	16 (165)	480 (165)
0.4	0.4	285 (198)	1,661 (11.15)	525 (29)	19 (12)	456 (12)
0.4	0.8	133 (200)	1,553 (11.87)	522 (61)	20 (0)	256 (0)
0.8	0.0	34 (197)	547 (9.67)	34 (197)	11 (197)	34 (197)
0.8	0.4	29 (198)	652 (9.97)	38 (75)	13 (57)	39 (57)
0.8	0.8	23 (200)	885 (10.30)	39 (39)	14 (11)	40 (11)

Table 11.8: Communication cost of solving *DisSRTI* instances for the three stabilities.

tributed version of $SUPER2$ cannot be applied without revealing information about the preference lists. That it is in the largest values of p_2, the instances are easier.

Table 11.7 and Table 11.8 resume, respectively, the *nccc* and *msg* needed by algorithms for solving *DisSRTI* instances for stability type. Both tables are correlated. Between parenthesis appear the length of the largest matching (4th column) and the number of solvable instances (3th, 5th, 6th and 7th columns). Considering super stability, *DisSUPER-SR* is much faster and sends less messages than *DisFC-SM*.

According to the results showed in the above four tables, we observe that a distributed version for a specialized algorithm, when applicable, is much better than *DisFC-SM*. This could be expected, since specialized algorithms takes advantage of the problem features. In contrast, *DisFC-SM* is based on $DisFC_2$, a generic algorithm that is applicable to any *DisCSP*. Regarding privacy, all those distributed algorithms guarantee that people keep their preference lists private during the search stable matchings.

11.7 Summary

In this chapter, we presented a distributed formulation for the Stable Marriage and the Stable Roommates problem. For these problems, some relaxed versions were also considered (i.e. preference lists may be incomplete and contain ties). All them appear to be naturally distributed and there is a clear motivation to keep their preference lists private during the solving process. On solving approaches, (1) we extend the specialized centralized algorithms to the distributed case, and (2) we provide a generic distributed constraint formulation. Keeping privacy, only a fraction of problem versions can be solved by the specialized distributed algorithms, while all can be solved by the generic distributed constraint formulation. When applicable, specialized algorithms are more efficient than generic ones. This relative inefficiency is the extra cost one has to pay to keep preference lists private. As future work, we would like to consider other constraint formulations for these problems.

Part V

Conclusions and Appendixes

Chapter 12

Conclusions

Distributed Constraint Satisfaction is a necessary framework for modeling and solving naturally distributed problems. Throughout this work, several complete search algorithms and heuristics for *DisCSP* have been presented. Proposed algorithms have been evaluated according to two issues: efficiency and privacy.

12.1 Conclusions

From our work, we can extract the following conclusions:

- Analogous to *CSP*, the use of variable reordering heuristics is fruitful for *DisCSP*. In this work, we have presented two approaches for variable reordering in *SCBJ*, a synchronous algorithm. Our results show that *SCBJ* with any of these heuristics always outperform the original algorithm in terms of computation effort and communication cost.
- The links that ABT adds between agents not sharing constraints are necessary for deleting obsolete information and thereby, to guarantee the algorithm termination. ABT_{kernel} is a basic kernel for grouping asynchronous backtracking algorithms that handling nogoods and links in a way such that termination is ensured. From ABT_{kernel}, we obtained four algorithms: ABT, ABT_{all}, ABT_{temp}, and ABT_{not}, the first algorithm that does not add links between agents not sharing constraints. Our experimental results show that the earlier links are added between agents not sharing constraints, the smaller number of messages the algorithm needs. On the other hand, the longer the duration of added links is, the greater number of messages the algorithm sends. Although ABT_{not} is the least economic algorithm, it exchanges much less information between agents not sharing constraints. ABT_{not} has to be selected only if some privacy policy justifies its use.
- Asynchronous algorithms are not always more efficient than synchronous ones. Furthermore, adding some synchronization points to asynchronous

algorithms improve their performance. In this work, we have presented ABT_{hyb}, a hybrid algorithm which combines synchronous and asynchronous elements. This algorithm avoids sending redundant messages after backtracking. ABT_{hyb} outperforms ABT regarding computation effort and communication cost.

- Although most of state-of-the-art algorithms for *DisCSP* assume binary constraints, they can be extended to handle constraints involving more than two variables. In this work we have extended one synchronous ($SCBJ$) and two asynchronous algorithms (ABT and ABT_{not}) to deal with non-binary constraints. Empirically, we have shown that adding constraint projections speeds up the search.

- In order to guarantee polynomial space on asynchronous/hybrid algorithms, the number of nogoods that agents may store has to also be polynomial. This implies that agents have to forget some obsolete nogoods that could be kept to avoid make same mistakes in future assignments. Empirically, we have shown that the heuristic of selecting the best nogood is a good strategy that improves practical performance.

- Part of this thesis have been concerned with privacy. In the context of asynchronous algorithms, especially in ABT, we have analyzed how privacy can be enhance without using costly cryptography tools. We have differentiate among three types of privacy: domain privacy, assignment privacy and constraint privacy. For each type of privacy we have discussed how privacy can be enforced.

  - *Domain privacy*: Assuming constraints are given implicity, ABT guarantees some domain privacy because the agent that holds a variable is the only one that knows the variable domain. However, ABT may be improved, for instances, by reducing the number of values that agents assign to their variables.
  - *Assignment privacy*: we presented *DisFC*, an asynchronous algorithm based on ABT, in which, instead of sending its assignment, each agent reveals to the other agents the set of values from their domains that is consistent with the agent's current assignment. *DisFC* achieves some assignment privacy, since every agent knows its value and no agent knows certainly other agent's value.
  - *Constraint privacy*: we have proposed the Partially Known Constraints model (PKC) to express interagent constraints. This model presumes that every agent involving in a constraint knows only a part of this constraint. Assuming *PKC*, we have studied in this thesis four distributed algorithms for *PKC*: $DisFC_2$, $DisFC_1$ and ABT_2, ABT_1. The former two algorithms are derived from *DisFC* and, thereby, they also preserve assignments. The other two methods are directly derived from ABT. Our algorithms are not perfect and leak

some information in the solving process. We have proposed a way to measure the constraint privacy of constraints that these algorithms offer.

Our experimental results for constraint privacy lead us to conclude the following points.

* ABT_2 and ABT_1 always offer higher constraint privacy than the original ABT.
* In general, $DisFC_2$ and $DisFC_1$ are less efficient and offer less constraint privacy than ABT versions for *PKC*. However, they also offer assignment privacy which are not provided by ABT algorithms. In the worst scenario, agents in $DisFC_2$ and $DisFC_1$ reveal as much information about their constraints as ABT agents. However, in order to construct the actual constraint known by any other agent, an agent in $DisFC_2$/ $DisFC_1$ must find all the solutions of a *CSP*, which is an NP-hard task.
* When allowing agents to lie about their constraints, constraint privacy is further enforced. We have presented $DisFC_{lies}$, an $DisFC_1$-like algorithm, in which agents can send false consistent domains with the only requirement that this false information must be amended within a finite time.
* Regarding efficiency, original algorithms are much better than their versions for *PKC*. This is the price one has to pay to achieve the required privacy.

- The presented algorithms in this thesis are fruitful for solving real problems with privacy requirements that can be viewed as *DisCSP*. Efficiency and privacy have been evaluated in the context of Meeting Scheduling [Freuder et al., 2001] and Stable Matching Problems [Gusfield and Irving, 1989]. These problems appear to be naturally distributed and there is a clear motivation to keep personal information private during the solving process. Regarding Meeting Scheduling problem, we have used two synchronous (*SCBJ* and *RR*, [Freuder et al., 2001]) and one asynchronous (ABT) algorithms to solve it. Experiments lead us to conclude that synchronous algorithms are more efficient and offer higher privacy than ABT for this problem. Regarding Stable Matching problems, we have studied two classes of problems: Stable Marriage and Stable Roommates problem. We have proposed a way to resolve these problems keeping their preference lists private. First, we model these problems using the constraint formulation of [Gent et al., 2001]. Second, we resolve problems in a simple way using the first phase of $DisFC_2$. For all problems, proposed solving methods offer more privacy than this that could be achieved by using the centralized approach.

12.2 Further Research

We consider as further research the following points.

- We have observed that heuristics for dynamic variables reordering improve synchronous search performance (see Chapter 4). In order to complement our work on dynamic variables reordering for *DisCSP* algorithms, and following the ideas given in [Zivan and Meisels, 2005b], the addition of dynamic variables reordering techniques to asynchronous searches could be part of further research.

- Asynchronous backtracking with temporary links, ABT_{temp}, appears as a good algorithm for asynchronous backtracking (see Chapter 5). However, the following question remains: how many **ok?** messages to allow through a temporary link? In our experiments, this parameter was adjusted manually after some trials. We think that it could be adjusted automatic and dynamically, customized for each agent. The automatic selection of this parameter is a direction for further research.

- The idea of adding synchronization points to ABT improves its performance because it avoids to send redundant messages. Identifying other cases of inefficiency in asynchronous algorithms may help to further improve their efficiency.

- For solving *DisCSP* with privacy requirements, one would like to use an algorithm that finds quickly a solution and offers a high privacy level for agents. In this thesis, privacy in *DisCSP* has been evaluated mainly from the perspective of asynchronous algorithms. Our experimental results demonstrate that for problem domains synchronous and hybrid algorithms could be more efficiency than asynchronous ones. The evaluation of synchronous and hybrid algorithms in terms of assignment and constraint privacy is still pending.

- The idea of allowing agents to lie about their actual constraints makes $DisFC_1$ algorithms to enhance constraint privacy. The use of lies in *DisCSP* algorithms should be considered to enforce other types of privacy.

- A common feature of the *DisCSP* algorithms in this work is that, in their presentations, we have assumed each agent holds only one variable. It is known that these algorithms can be applied to situations where one agent has multiple local variables by either: (1) each agent finding all the solutions to its local problem first and then all agents reformulating the resulting problem as distributed *CSP* or (2) creating multiple virtual agents, each of which corresponding to one local variable and simulating the activities of these virtual agents. Both methods are neither efficient nor scalable to large problems [Yokoo, 2001]. This is why the extensions of proposed algorithms to handle multi-variable agents could be part of future research.

Appendix A

Specialized Algorithms for Stable Matching Problems

This appendix analyzes the extensions of specialized algorithms for Stable Matching problems to the distributed case.

***DisSM*, *DisSMI*.** It is possible to extend the centralized Gale-Shapley [Gale and Shapley, 1962], in short *EGS*, to the distributed case while keeping preference lists private. We call this algorithm the Distributed Gale-Shapley (*DisEGS*). The man-oriented version of the *DisEGS* algorithm for solving *SM* and *SMI* instances appears in Figure A.1 (the woman-oriented is analogous, switching the roles man/woman). It is composed of two procedures, `Man` and `Woman`, which are executed on each man and woman, respectively. Execution is asynchronous. The following messages are exchanged (where m is the man that executes procedure `Man` and w the woman that executes procedure `Woman`),

- propose: m sends this message to w to propose engagement;
- accept: w sends this message to m after receiving a propose message to notify acceptance;
- delete: w sends this message to m to notify that w is not available for m; this occurs either (i) proposing m an engagement to w but w has a better partner or (ii) w accepted an engagement with other man more preferred than m;
- stop: this is an special message to notify that execution must end; it is sent by an special agent after detecting quiescence.

Procedure `Man`, after initialization, performs the following loop. If m is free and his list is not empty, he proposes to be engaged to w, the first woman in his list. Then, m waits for a message. If the message is accept and it comes from w, then m confirms the engagement (nothing is done in the algorithm). If the message

```
procedure Man()
m ← free;
end ← false;
while ¬end do
  if m = free and list(m) ≠ ∅ then
    w ← first(list(m));
    sendMsg(propose,m,w);
    m ← w;
  msg ← getMsg();
  switch msg.type
    accept  : do nothing;
    delete  : list(m) ← list(m) − msg.sender;
              if msg.sender = w then m ← free;
    stop    : end ← true;

procedure Woman()
w ← free;
end ← false;
while ¬end do
  msg ← getMsg();
  switch msg.type
    propose: m ← msg.sender;
             if m ∉ list(w) then
               sendMsg(delete,w,m);
             else
               sendMsg(accept,w,m);
               w ← m;
               for each p after m in list(w) do
                 sendMsg(delete,w,p);
                 list(w) ← list(w) − p;
    stop    : end ← true;
```

Figure A.1: The man-oriented version of the *DisEGS* algorithm.

is delete, then m deletes the sender from his list, and if the sender is w then m becomes free. The loop ends when receiving a stop message.

Procedure `Woman` is executed on woman w. After initialization, there is a message receiving loop. In the received message comes from a man m proposing engagement, w rejects the proposition if m is not in her list. Otherwise, w accepts. Then, any man p that appears after m in w list is asked to delete w from his list, while w removes p from hers. This includes a previous engagement m', that will be the last in her list. The loop ends when receiving a stop message.

DisSMT-weak. Each person may arbitrarily break his/her ties. From this point on, the problem becomes a *DisSM*, and any stable matching will be a weak stable matching for the original one. So the *DisEGC* algorithm is enough to solve it.

DisSMT-strong. The *STRONG* algorithm requires the computation of a deficient set for the bipartite graph defined by engagement between men and women. In the Stable Marriage context, a set X of s men is a *deficient set* if the men are collectively engaged to t women, for some s and $t < s$. The *deficiency* of X is defined by $\delta(X) = s - t$. In particular, *STRONG* requires, in some points during its execution, the computation of a deficient set with the maximum deficiency (also called critical set). The search of critical sets requires the exploration of the engagement graph. This forces to each person reveals his/her current partners which breaks the privacy requirement to solve the problem. So we conclude that *STRONG* cannot be extended to the distributed case while keeping privacy.

DisSMT-super. The *SUPER* algorithm [Irving, 1994] can be easily extended to the distributed setting without revealing the people's preference lists. *SUPER* consists of a two-step cycle. The first step is similar to that of *STRONG*. When the first step ends, every man can be engaged to 0, 1 or k women. If a man's preference list is empty then no super stable matching exists and the algorithm ends. In second step, all men tied at the tail of a woman's preference list are deleted and the woman from their preference lists. Such pairs must be deleted since they constitute super blocking pairs. This process continues until each man is either engaged or has an empty list. When the cycle ends, if a man's preference list is empty then no super stable matching exists. Otherwise, the matching given by the engaged pairs constitutes a super-stable matching. Similar to *EGS*, the first step can be extended to the distributed setting while keeping preference lists private. In the second step, each man who is deleted from a woman's preference list can only infer that the woman is indifferent between him and one or more men, but he will never know who is or who are those men. Therefore, *DisSUPER* keeps privacy of preference lists and can be used to solve *DisSMT-super*.

DisSMTI-weak. The same argument of *DisSMT-weak* applies here.

DisSMTI-strong. The *STRONG2* algorithm requires the computation of a maximum matching for a bipartite graph G (that represents the engagement relation among men and women). A matching is maximum if it is not properly contained in any other matching. A fundamental strategy for finding a maximum matching in a bipartite matching is using alternating paths. For a given graph G, a path P is an alternating path for the matching M of G if: (i) the two end points of P are unmatched by M and (ii) the edges of P alternate between edges matched by M and edges unmatched by M. A matching is maximum iff it admits no alternating path. The graph defined by the engagement relation among men and women have to be explore in order to find alternating paths. This forces to each person reveals his/her current partners. When multiples partners exits, all of them are equally preferred. Therefore, when a person reveals his/her partners it is also revealing a tie in his/her preference lists which breaks the privacy requirement to solve the problem. So *STRONG2* cannot be extended to the distributed case keeping privacy.

DisSMTI-super. The same argument for *DisSMTI-strong* applies here.

DisSMTI-weak-max. To compute the matching of maximum length, one has to break ties in all possible ways which will produce different instances of SMI, that can be solved by *DisEGC*. The solution with maximum length is recorded as the global solution. Extending this approach to the distributed case presents an issue: it is required a new agent, that synchronizes the possible ways in which agents can break their ties. This new agent has to know how many ties contain each agent. Although this approach makes public this information, preference list are still private. This agent records all solutions, to find the optimal one.

DisSR, DisSRI. These two problems are solved by the *Stable Roommates* algorithm [Irving and Manlove, 2002]. This algorithm consists of two phases. The first phase is an extended version of *EGS* algorithm where every person sends and receives matching proposals from the rest of people. *EGS* can be extended to the distributed case while keeping preference lists private. The distributed version of *EGS* is *DisEGS*.

The second phase of the *Stable Roommates* algorithm iterates reducing further the preference lists until all lists contain just one entry, in which case it constitutes a stable matching or until any preference list becomes empty, in which case no stable matching exists. This phase consists of two parts. Firstly, the algorithm builds a special sequence of matching pairs from the reduced preference lists. This sequence is called *rotation* and has the following form:

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{r-1}, y_{r-1})\},$$

such that y_i is the most preferred partner for x_i and y_{i+1} is the second most preferred partner for x_i for all i, $0 \leq i \leq r-1$, where $i+1$ is taken as module r. Secondly, the algorithm deletes from reduced preference lists the rotation obtained in the first part of this phase (see section 4.2 of [Gusfield and Irving, 1989] for more details).

Notice that the construction of a rotation requires that some agents reveal either their most or their second most preferred partners in their lists. In this sense, a distributed version of the centralized solving algorithm such that privacy is maintained, does not seem feasible in this case.

DisSRT-weak. The centralized approach uses the *Stable Roommates* algorithm, that cannot be extended to the distributed case without breaking privacy [Brito and Meseguer, 2005b]. Therefore, this approach cannot be extended to the distributed case.

DisSRT-strong. Up to our knowledge, no polynomial algorithm is known to solve this problem [Irving and Manlove, 2002]. So no extension can be done to the distributed case.

DisSRT-super. The *SRT-super* algorithm determines if a super-stable matching exits for a SRT instance. If so, the algorithm finds it. *SRT-super* consists of two phases. *EGS* is the kernel of the first phase, which is extended

to the distributed case by *DisEGS*, keeping privacy. The second phase can be implemented using the type of messages of *DisEGS* (propose and delete messages). So, *SRT-super* can be extended to the distributed case, keeping preferences private.

DisSRTI-weak. The same argument for *DisSRT-weak* applies here.

DisSRTI-strong. The same argument for *DisSRT-strong* applies here.

DisSRTI-super. The same argument for *DisSRT-super* applies here.

DisSRTI-weak-max. The same argument for *DisSRT-weak* applies here.

Bibliography

[Armstrong and Durfee, 1997] Armstrong, A. and Durfee, E. (1997). Dynamic priorization of complex agents in distributed constraint satisfaction problems. *Proceedings of the 15th IJCAI (IJCAI-97).*, pages 620–625.

[Atkinson et al., 2006] Atkinson, T., Bartak, R., Silaghi, M. C., Tuleu, E., and Zanker, M. (2006). Private and efficient stable marriages (matching) - a discsp benchmark. *Proceedings of the Fifth International Workshop on Distributed Constraint Reasoning (DCR), ECAI-06.*

[Bacchus and van Beek, 1998] Bacchus, F. and van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problem. *Proceedings of AAAI-98*, pages 310–318.

[Bartak, 1988] Bartak, R. (1988). Constraint satisfaction online tutorial. *http://kti.ms.mff.cuni.cz/∽bartak/constraints/stochastic.html.*

[Bessière et al., 2005] Bessière, C., Maestre, A., Brito, I., and Meseguer, P. (2005). Asynchronous backtracking without adding links: a new member to abt family. *Artificial Intelligence Special issue: Distributed Constraint Satisfaction.*, 161(1–2):7–24.

[Bessière et al., 2001] Bessière, C., Maestre, A., and Meseguer, P. (2001). Distributed dynamic backtracking. *Proceedings of the Workshop on Distributed Constraint, IJCAI-01.*, pages 9–16.

[Bessiere et al., 1999] Bessiere, C., Meseguer, P., Freuder, E. C., and Larrosa, J. (1999). On forward checking for non-binary constraint satisfaction. *Proceeding of the Fifth International Conference on Principles and Practice of Constraint Programming (CP-99). Lecture Notes in Computer Science.*, pages 88–102.

[Bitner and Reingold, 1975] Bitner, J. and Reingold, E. (1975). Backtrack programming techniques. *Communications of the ACM.*, 18(11):651–656.

[Box, 1978] Box, F. (1978). A heuristic technique for assignment frequencies to mobile radio nets. *IEEE Transactions on Vehicular Technology.*, 27(2):54–64.

[Brito, 2004] Brito, I. (2004). Synchronous, asynchronous and hybrid algorithms for discsp. *Proceeding of the Tenth International Conference on Principles and*

Practice of Constraint Programming, (CP-2004). Lecture Notes in Computer Science (Doctoral Programme of CP-2004)., 3258:791.

[Brito and Meseguer, 2003] Brito, I. and Meseguer, P. (2003). Distributed forward checking. *Proceeding of the Night International Conference on Principles and Practice of Constraint Programming (CP-2003). Lecture Notes in Computer Science.*, 2833:801–806.

[Brito and Meseguer, 2004] Brito, I. and Meseguer, P. (2004). Synchronous, asynchronous and hybrid algorithms for discsp. *Fifth International Workshop on Distributed Constraint Reasoning at the Tenth International Conference on Principles and Practice of Constraint Programming (CP-2004), http://www-2.cs.cmu.edu/∼pmodi/papers/DCR04-Proceedings.pdf.*

[Brito and Meseguer, 2005a] Brito, I. and Meseguer, P. (2005a). Distributed stable marriage problem. *Sixth International Workshop on Distributed Constraint Reasoning (DCR) at the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-2005), http://www.cs.bgu.ac.il/∼am/DCR-05/Papers/DisSM_18.pdf.*

[Brito and Meseguer, 2005b] Brito, I. and Meseguer, P. (2005b). Distributed stable matching problems. *Proceeding of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP-2005). Lecture Notes in Computer Science.*, 3709:152–166.

[Brito and Meseguer, 2006a] Brito, I. and Meseguer, P. (2006a). Asynchronous backtracking algorithms for non-binary discsp. *Workshop on Distributed Constraint Satisfaction Problems at the 17th European Conference on Artificial Intelligence (ECAI-2006).*

[Brito and Meseguer, 2006b] Brito, I. and Meseguer, P. (2006b). The distributed stable marriage problem with ties and incomplete lists. *Workshop on Distributed Constraint Satisfaction Problems at the 17th European Conference on Artificial Intelligence (ECAI-2006).*

[Brito and Meseguer, 2006c] Brito, I. and Meseguer, P. (2006c). Distributed stable matching problems with ties and incomplete lists. *Proceeding of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-2006). Lecture Notes in Computer Science.*, 4204:675–680.

[Chandy and Lamport, 1985] Chandy, K. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Computer Systems.*, 3(2):63–75.

[Dechter, 1990] Dechter, R. (1990). On the expresiveness of networks with hidden variables. *Proceedings of AAAI-90.*, pages 556–562.

[Dechter, 1999] Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85.

[Dechter and Dechter, 1988] Dechter, R. and Dechter, A. (1988). Belief maintenance in dynamic constraint satisfaction problem. *Proceedings of the Seventh National Conference on Artitifial Intelligence.*, pages 37–42.

[Dechter and Pearl, 1987] Dechter, R. and Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38.

[Dechter and Pearl, 1988] Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence.*, 34:1–38.

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

[Fernández et al., 2002] Fernández, C., Béjar, R., Krishnamachari, B., and Gomes, C. (2002). Communication and computation in distributed csp algorithms. *Lecture Notes in Computer Science (Proceedings of CP-2002).*, 2470:664–679.

[Franzin et al., 2004] Franzin, M. S., Rossi, F., Freuder, E. C., and Wallace, R. (2004). Multi-agent constraint systems with preferences: Efficiency, solution quality, and privacy loss. *Computational Intelligencee*, 20(2):264–286.

[Freuder, 1988] Freuder, E. (1988). Backtrack-free and backtrack-bounded search. *Search in Artificial Intelligence.*, pages 343–369.

[Freuder et al., 2001] Freuder, E., Minca, M., and Wallace, R. J. (2001). Privacy/efficiency trade-offs in distributed meeting scheduling by constraint-based agents. *Workshop on Distributed Constraint Reasoning (DCR, IJCAI-01)*, pages 63–71.

[Gale and Shapley, 1962] Gale, D. and Shapley, L. S. (1962). College admissions and the stability of the marriage. *American Mathematical Monthly.*, 69:9–15.

[Gale and Sotomayor, 1985] Gale, D. and Sotomayor, M. (1985). Some remarks on the stable matching problem. *Discrete Applied Mathematics.*, 11:223–232.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W.H.Freeman&Co.

[Gaschnig, 1979] Gaschnig, J. (1979). *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis.

[Geffner and Pearl, 1987] Geffner, H. and Pearl, J. (1987). An improved constraint-propagation algorithm for diagnosis. *Proceedings of IJCAI-87*, pages 1105–1111.

[Gent et al., 2001] Gent, I. P., Irving, R. W., Manlove, D. F., Prosser, P., and Smith, B. M. (2001). A constraint programming approach to the stable marriage problem. *Lecture Notes in Computer Science (Proceedings of CP-2001).*, 2239:225–239.

[Gent and Prosser, 2002] Gent, I. P. and Prosser, P. (2002). An empirical study of the stable marriage problem with ties and incomplete lists. *Proceedings of ECAI-2002.*, pages 141–145.

[Ginsberg, 1993] Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:24–46.

[Glover, 1986] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research.*, 13(5):533–549.

[Gusfield and Irving, 1989] Gusfield, D. and Irving, R. W. (1989). *The Stable Marriage Problem: Structure and Algorithms.* The MIT Press.

[Hamadi, 1999] Hamadi, Y. (1999). *Traitement des Problèmes de Satisfaction de Contraintes Distribués.* PhD thesis.

[Hamadi et al., 1998] Hamadi, Y., Bessière, C., and Quinqueton, J. (1998). Backtracking in distributed constraint networks. *Proceedings of the 13th ECAI (ECAI-98).*, pages 219–223.

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

[Irving, 1994] Irving, R. W. (1994). Stable marriage and indifference. *Discret Applied Mathematics*, 48:261–272.

[Irving and Manlove, 2002] Irving, R. W. and Manlove, D. F. (2002). The roommates problem with ties. *J. Algorithms*, 43(1):85–105.

[K. and Yokoo, 2000] K., K. H. and Yokoo, M. (2000). The effect of nogood learning in distributed constraint satisfaction. *Proceeding of Workshop on Distributed Constraint Reasoning (DCR-00)*, pages 169–177.

[Kautz and Selman, 1992] Kautz, H. and Selman, B. (1992). Planning as satisfiability. *Proceeding of the ECAI-92*, pages 360–363.

[Kirkpatrick and Hell, 1983] Kirkpatrick, D. G. and Hell, P. (1983). On the complexity of general graph factor problems. *SIAM Journal of Computing.*, 12(3):601–608.

[Lamport, 1978] Lamport, L. (1978). Time, clock, and the ordering of evens in a distributed system. *Communication of the ACM*, 21(7):558–565.

[Larrosa and Meseguer, 1998] Larrosa, J. and Meseguer, P. (1998). Adding constraint projections in n-ary csp. *Workshop on non-binary constraints of the 13th European Conference on Artificial Intelligence - ECAI98.*

[Lynch, 1997] Lynch, N. A. (1997). *Distributed Algorithms.* Morgan Kaufmann Series.

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

[Manlove, 1999] Manlove, D. F. (1999). Stable marriage with ties and unacceptable partners.

[Manlove et al., 2002] Manlove, D. F., Irving, R. W., Iwama, K., Miyazaki, S., and Morita, Y. (2002). Hard variants of stable marriage. *Theoretical Computer Science.*, 276(1–2):261–27.

[Meisels et al., 2002] Meisels, A., Razgon, I., Kaplansky, E., and Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms. *Proceedings of the 3th Workshop on Distributed Constraints Reasoning, DCR-02 (AAMAS-2002).*, pages 86–93.

[Meisels and Zivan, 2006] Meisels, A. and Zivan, R. (2006). Personal communication.

[Minton et al., 1990] Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1990). Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. *Proceedings of AAAI-90.*, pages 17–24.

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

[Morris, 1993] Morris, P. (1993). The breakout method for escaping from local minima. *Proceedings of AAAI-93.*, pages 40–45.

[Nissim and Zivan, 2005] Nissim, K. and Zivan, R. (2005). Secure discsp protocols - from centralized towards distributed solutions. *In Proc. 6th workshop on Distributed Constraints Reasoning, DCR-05.*

[Nwana and Ndumu, 1997] Nwana, H. S. and Ndumu, D. (1997). An introduction to agent technology. *In Software Agents and Soft Computing: Towards Enhancing Machine Intelligence, Springer-Verlag*, pages 3–26.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithm for the constraint satisfaction problem. *Computational Intelligence.*, 9:268–299.

[Roth et al., 2005] Roth, A. E., Snmez, T., and nver, M. U. (2005). Pairwise kidney exchange. *Journal of Economic Theory*, 125:151–188.

[Sabin and Freuder, 1994] Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. *In Proceedings of European Conference on Artificial Inteelligence, ECAI-94*, pages 125–129.

[Selman et al., 1992] Selman, B., Levesque, H., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. *In the Proceedings of AAAI-92*, pages 440–445.

[Silaghi and Mitra, 2004] Silaghi, M. and Mitra, D. (2004). Distributed constraint satisfaction and optimization with privacy enforcement. *In Proc. of the Third International Conference on Intelligence Agent Technology*, pages 531–535.

[Silaghi, 2002] Silaghi, M. C. (2002). *Asynchronously Solving Distributed Problems With Privacy Requirements.* PhD thesis.

[Silaghi, 2004a] Silaghi, M. C. (2004a). Incentive auctions and stable marriages problems solved with n/2-privacy of human preferences. *Technical report-CS-2004-11., Florida Ins. of Tech.*

[Silaghi, 2004b] Silaghi, M. C. (2004b). Meeting scheduling guaranteeing n/2-privacy and resistant to statistical analysis (applicable to any discsp). *In Proc. of the Third International Conference on Web Intelligence*, pages 711–715.

[Silaghi and Faltings, 2005] Silaghi, M. C. and Faltings, B. (2005). Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence Special issue: Distributed Constraint Satisfaction.*, 161(1–2):25–53.

[Silaghi et al., 2000] Silaghi, M. C., Sam-Haroud, D., and Faltings, B. (2000). Asynchronous search with aggregations. *In Proc. of the 17th. AAAI*, pages 917–922.

[Silaghi et al., 2001a] Silaghi, M. C., Sam-Haroud, D., and Faltings, B. (2001a). Consistency maintenance for abt. *Lecture Notes in Computer Science (CP-2000)*, 2239:271–285.

[Silaghi et al., 2001b] Silaghi, M. C., Sam-Haroud, D., and Faltings, B. (2001b). Hybridizing abt and awc into a polynomial space, complete protocol with reordering. *Tech. Report EPFL.*

[Silaghi et al., 2001c] Silaghi, M. C., Sam-Haroud, D., and Faltings, B. (2001c). Polynomial space and complete multiply asynchronous search with abstractions. *Notes of the IJCAI'01 Workshop on Distributed Constraint Reasoning (DCR).*

[Smith, 1994] Smith, B. M. (1994). Phase transition and the mushy region in constraint satisfaction problems. *Proc. of the 11th ECAI (ECAI-94)*, pages 100–104.

[Stallman and Sussman, 1977] Stallman, R. and Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking. *Artificial Intelligence.*, 9(2):135–196.

[Tang, 1993] Tang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press.

[Yokoo, 1995] Yokoo, M. (1995). Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. *Lecture Notes in Computer Science (CP-95)*, 976:88–102.

[Yokoo, 2000] Yokoo, M. (2000). Personal communication.

[Yokoo, 2001] Yokoo, M. (2001). *Distributed Constraint Satisfaction.* Springer.

[Yokoo et al., 1992] Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. *In Proc. of the 12th. DCS*, pages 614–621.

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

[Yokoo et al., 2002] Yokoo, M., Suzuki, K., and Hirayama, K. (2002). Secure distributed constraint satisfaction: Reaching agreement without revealing private information. *Lecture Notes in Computer Science (CP-02).*, 2470:387–401.

[Yokoo et al., 2005] Yokoo, M., Suzuki, K., and Hirayama, K. (2005). Secure distributed constraint satisfaction: Reaching agreement without revealing private information. *Artificial Intelligence*, 161(1-2):229–246.

[Zivan and Meisels, 2003] Zivan, R. and Meisels, A. (2003). Synchronous and asynchronous search on discsps. *Proceedings of EUMAS-2003.*

[Zivan and Meisels, 2004a] Zivan, R. and Meisels, A. (2004a). Concurrent backtrack search on discsps. *Proceedings of FLAIRS-2004.*

[Zivan and Meisels, 2004b] Zivan, R. and Meisels, A. (2004b). Concurrent dynamic backtracking for distributed csps. *Lecture Notes in Computer Science (CP-04).*, 3258:789.

[Zivan and Meisels, 2005a] Zivan, R. and Meisels, A. (2005a). Asynchronous backtracking for asymmetric discsps. *In Proc. 6th workshop on Distributed Constraints Reasoning, DCR-05.*

[Zivan and Meisels, 2005b] Zivan, R. and Meisels, A. (2005b). Dynamic ordering for asynchronous backtracking on discsps. *Lecture Notes in Computer Science (CP-05).*, 3709:32–46.

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1. J. Puyol, *MILORD II: A Language for Knowledge-Based Systems*

Num. 2. J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*

Num. 3. Ll. Vila, *On Temporal Representation and Reasoning in Knowledge-Based Systems*

Num. 4. M. Domingo, *An Expert System Architecture for Identification in Biology*

Num. 5. E. Armengol, *A Framework for Integrating Learning and Problem Solving*

Num. 6. J. Ll. Arcos, *The Noos Representation Language*

Num. 7. J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*

Num. 8. P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*

Num. 9. F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*

Num. 10. W. M. Schorlemmer, *On Specifying and Reasoning with Special Relations*

Num. 11. M. López-Sánchez, *Approaches to Map Generation by means of Collaborative Autonomous Robots*

Num. 12. D. Robertson, *Pragmatics in the Synthesis of Logic Programs*

Num. 13. P. Faratin, *Automated Service Negotiation between Autonomous Computational Agents*

Num. 14. J. A. Rodríguez, *On the Design and Construction of Agent-mediated Electronic Institutions*

Num. 15. T. Alsinet, *Logic Programming with Fuzzy Unication and Imprecise Constants: Possibilistic Semantics and Automated Deduction*

Num. 16. A. Zapico, *On Axiomatic Foundations for Qualitative Decision Theory - A Posibilistic Approach*

Num. 17. A. Valls, *ClusDM: A multiple criteria decision method for heterogeneous data sets*

Num. 18. D. Busquets, *A Multiagent Approach to Qualitative Navigation in Robotics*

Num. 19. M. Esteva, *Electronic Institutions: from specification to development*

Num. 20. J. Sabater, *Trust and reputation for agent societies*

Num. 21. J. Cerquides, *Improving Algorithms for Learning Bayesian Network Classifiers*
Num. 22. M. Villaret, *On Some Variants of Second-Order Unification*
Num. 23. M. Gmez, *Open, Reusable and Configurable Multi-Agent Systems: A Knowledge Modelling Approach*
Num. 24. S. Ramchurn, *Multi-Agent Negotiation Using Trust and Persuasion*
Num. 25. S. Ontaon, *Ensemble Case-Based Learning for Multi-Agent Systems*
Num. 26. M. Snchez, *Contributions to Search and Inference Algorithms for CSP and Weighted CSP*
Num. 27. C. Noguera, *Algebraic Study of Axiomatic Extensions of Triangular Norm Based Fuzzy Logics*
Num. 28. E. Marchioni, *Functional Definability Issues in Logics Based on Triangular Norms"*
Num. 29. M. Grachten, *Expressivity-Aware Tempo Transformations of Music Performances Using Case Based Reasoning*
Num. 30. I. Brito, *Distributed Constraint Satisfaction*
Num. 31. E. Altamirano, *On Non-clausal Horn-like Satisfiability Problems*